CUSTOMIZING WEB APPLICATIONS THROUGH ADAPTABLE COMPONENTS AND RECONFIGURABLE DISTRIBUTION

BY

PO-HAO CHANG

B.S., National Taiwan University, 1995

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Although platforms and execution scenarios are constantly evolving, Web applications today are tightly coupled with the specific platforms and execution scenarios that are envisioned at design time. Consequently, these Web applications do not adapt well to new clients and runtime execution contexts. The goal of this research is to develop methods and software to support Web applications that can be customized for diverse execution environments through *adaptable components* and *reconfigurable distribution.* Thinking in terms of *software product line engineering*, a Web application can be modelled as a product line, each instance of which is for a specific execution platform and context. To achieve such a product line, we identify two crucial requirements: universal accessibility and context-dependent component distribution.

We address both requirements by decomposing Web applications as a collection of active objects or *actors* and using *virtualization* on execution platforms. Virtualization provides a uniform programming environment for *actors* by encapsulating the incompatibilities that exist in different models and platforms. This supports a high-level of abstraction: programmers no longer need to deal with pages, HTTP requests, cookies and sessions, all of which are entities in specific application models and platforms; instead, they can focus on building components and defining their interaction, that is, on the application logic. Thus, a Web application in our virtual environment represents a product line characterized by its application logic, not a specific implementation which can only be deployed on a specific platform.

Following the principle of separation of concerns, the virtual environment is not only platform independent but also location agnostic: there is no notions about component location and component movement. However, specifications of location and mobility are required for an executable Web application. We design a specification system consisting of two parts: a *specification language* that enables developers write component distribution schemes, and *tools* which assist runtime environments to execute these schemes. Combining an application with a specification scheme, a Web application customized to a specific execution context is defined. Inspired by the success of Web Style Sheets, our specification system is based on the idea of using *transformations*: it converts a composition of actors into a composition of annotated actors. The annotation attributes are given by a specific runtime environment. For example, in a typical Web runtime environment, the location of an actor may be the *client* or the *server*. Component distribution can be specified by annotating each actor with the desired attribute value. Sophisticated loading policies can be defined in a similar way with specially devised attributes. We develop an algorithm to perform the transformation and prove the correctness.

One approach to support virtualization over heterogeneous platforms is to define and build a traditional middleware. This approach has the disadvantage of requiring the deployment of new software systems and protocols–something that is impractical given the sheer scale of the Internet and the heterogeneity of existing platforms. Instead, we adopt the paradigm of *generative programming* for customized application execution: before deployment, the actors defining a Web application undergo a generative process to obey the annotations given by the specification system. To host these generated actors, a light-weight runtime environment is required for each participating platform. In addition, the runtime environments of a Web application have to coordinate with each other to manage distributed actors across the Internet. We implement a generative application framework including runtime environments, coordination services and generators. Our experience suggests that it is feasible to facilitate reconfigurable component distribution using specification schemes to control component allocation and regulate loading patterns.

To Hsiao-Yen and Eeyore, who accompanied me all along the long journey.

Acknowledgements

I would like to thank my parents, Ban-Hsieung Chang and Mei-Chen Kuo. Without their persistence and eager expectations, I would not have had the courage and determination to take the less traveled road, and that made all the difference. I appreciate my doctoral committee–Prof. Gul Agha, Prof. Carl Gunter, Prof. Vikram Adve and Prof. AnHai Doan, who have been invaluable through their advice and comments. I am deeply indebted to my advisor Prof. Gul Agha, who taught me how to make my research much more easily accessible and offered numerous clarifications and improvements in the presentation style of this thesis.

I am grateful to my colleagues, Valery Berry, Nursalim Hadi and Kevin Kessler in the Office of Information Management at the University of Illinois at Urbana-Champaign, where I worked as a programmer during the transition of my research areas. Together we worked through several challenging and inspiring projects; the experiences I gained became the seeds of this theses. I also owe a particular gratitude to Dr. Wooyoung Kim, who helped me refine my ideas and his feedback on my early papers encouraged me into this research.

I would like to thank all members of the Open Systems Lab over the years, Prasanna Thati, Reza Ziaei, Nadeem Jamali, Koushik Sen, Predrag Tosic, Sandeep Uttamchandani, Myeong-Wuk Jang, YoungMin Kwon, Soham Mazumdar, Amr Ahmed, Kirill Mechitov, Sameer Sundresh, Liping Chen, MyungJoo Ham and others. They offered me insightful comments and critics for this research and made these years so memorable. Thanks also go to my Taiwanese fellow students in the University of Illinois at Urbana-Champaign, Ruei-Sung Lin, Wen-Tau Yih, Ying-Ping Chen, Kuang-Chih Lee, Chien-Wei Li, Le-Chun Wu, Ben-Chung Cheng and others. Their unlimited support helped me overcome my homesickness in the distant country.

Acknowledgements cannot be complete without thanking my wife Hsiao-Yen Yu. She gave up her bright and promising career in Taiwan and followed my adventure to the United States in 1998. This thesis is dedicated to her, for her love, sacrifice and patience.

Table of Contents

List of Tables . . . . **x**

List of Figures . . . . **xi**

Chapter 1 Introduction **1**
1.1 Background and Motivation 1
1.2 Application Scenarios 3
1.2.1 Location of Execution 3
1.2.2 Timing of Deployment 5
1.2.3 Component Distribution as a Separate Concern 6
1.3 Problem Analysis 9
1.4 Approach 11
1.4.1 Virtualization 11
1.4.2 Separation of Concerns 13
1.4.3 Generative Programming 13
1.5 Contributions 14
1.6 Research Overview 16
1.7 Thesis Outline 17

Chapter 2 Programming Model and Language **18**
2.1 Actors: A Model for Distributed Active Objects 18
2.1.1 Classic Actor Model 19
2.1.2 Model Extensions 19
2.2 `ActorScript` Basics 21
2.2.1 An Actor Scripting Language 22
2.2.2 Defining Actor Behavior 23
2.2.3 Communication in Actors 25
2.2.4 Actor Termination 28
2.3 Extra Features in `ActorScript` 28
2.3.1 Methods and Ports 29
2.3.2 Prototype Revisit 31
2.3.3 Concurrency 33
2.4 Actor Coordination 34
2.4.1 Continuation 34
2.4.2 Asynchronous Error Handling 37
2.4.3 Advanced Event System 39
2.5 Module System 41
2.5.1 Use: Referring to Other Prototypes 41
2.5.2 Import: Building New Prototypes from Existing Ones 41
2.5.3 Actor Inheritance 43

Chapter 3 Specification System . . . 45
3.1 Design of `ActorSpec` . . . 45
3.1.1 Assumptions and Design Principles . . . 46
3.1.2 Actor Annotations . . . 46
3.2 Selecting Actors . . . 47
3.2.1 Actors from a Concept of Design . . . 48
3.2.2 Actors of a Partial Genealogy . . . 49
3.3 Prototypes in `ActorSpec` . . . 50
3.3.1 Implementation of Prototype . . . 51
3.3.2 Reuse Patterns . . . 51
3.4 Specification Language . . . 53
3.4.1 Moving to XML . . . 53
3.4.2 More Modulization . . . 55
3.5 Resolving Specification Rules . . . 56
3.5.1 Problem Description . . . 59
3.5.2 Path-Splitting Algorithm . . . 61
3.5.3 Proof and Analysis . . . 63

Chapter 4 Generative Application Framework . . . 66
4.1 Architecture Overview . . . 66
4.1.1 An Actor Runtime Environment on the Web . . . 66
4.1.2 Attributes for Actor Distribution . . . 68
4.2 Podium Designs . . . 69
4.2.1 Browser: Message Loop . . . 70
4.2.2 Server: Thread Pool . . . 71
4.3 Framework Services . . . 71
4.3.1 Cross-Podium Message Delivery . . . 71
4.3.2 Remote Actor Management . . . 72
4.3.3 Remote Method Invocation . . . 73
4.3.4 Automatic Garbage Collection . . . 73
4.4 Customized Web Applications . . . 75
4.4.1 Prototype Implementations . . . 75
4.4.2 Prototype Linker and Application Loader . . . 77
4.5 Exploiting Locality . . . 78
4.5.1 Code Sharing . . . 79
4.5.2 Lazy-Copying of Aggregate Data Types . . . 79
4.5.3 Delayed Message Delivery . . . 81

Chapter 5 Applications and Experiments . . . 83
5.1 A Basic Arithmetic Calculator . . . 84
5.1.1 Application Components . . . 84
5.1.2 Deployment Schemes . . . 86
5.1.3 Calculating π . . . 87
5.2 Presenting Search Results . . . 90
5.2.1 Application Structure . . . 92
5.2.2 Effects of Computation Shift . . . 92
5.2.3 Loading Actors on Demand . . . 94
5.3 Content-Aware Web Applications . . . 96
5.3.1 Design Strategies . . . 98
5.3.2 Partial Plans . . . 98
5.3.3 Context Features . . . 99
5.3.4 Policy Design . . . 99

Chapter 6 Related Work **101**
6.1 Java and Virtual Machine 101
6.2 Transparent Distributed Computing 103
6.3 Metaprogramming 104
6.4 Component Redundancy 105
6.5 Web Services 106
6.6 Web Toolkits and Frameworks 108

Chapter 7 Discussion and Conclusion **111**
7.1 Virtual Programming Environment 111
7.2 Specification System 112
7.3 Generative Application Framework 113

References **115**

Author's Biography **121**

List of Tables

1.1 Market shares of different Web browsers, Web servers, and server-side scripting systems . . . 2

5.1 Performance results of the calculator application in different deployment schemes. 87
5.2 π calculation in different platforms. 89
5.3 Data transfer of the search application in different deployment schemes. 93
5.4 Response time of a query in different settings. 94
5.5 Data transfer of the search application with different loading policies. 96
5.6 Latency of the search application in different network settings. 96

List of Figures

1.1 FRB Mortgage Comparison Calculator is a representative Web application performing both computation and validation. 4
1.2 Preloading in batch is more responsive in a context where the bandwidth is not an issue. . . . 5
1.3 Loading on demand saves bandwidth at the cost of latency. 6
1.4 MythWeb, a Web-based front-end of a PVR system, exhibits a typical user interface with multi-layered presentation. 7
1.5 The classical server-based Web application model. 8
1.6 Computing locally shifts load to the client. 8
1.7 The document-based architecture is not suitable for reconfigurable interactive Web applications. 10
1.8 The virtual application environment enables platform independent and location agnostic application development. 12
1.9 The specification system transforms a composition of components into another composition of annotated components. 14
1.10 Generators convert annotated components into executable components in the target platforms. 15

2.1 Using different directives to extend a prototype. 44

3.1 The genealogy ordering is linear. 50
3.2 Specification rules only apply to physically existing runtime actors. 52
3.3 Using rules of extending relationship and creatorship may result in conflicts in specification. . 53
3.4 Using XML to represent specification rules. 54
3.5 A specification scheme may contain common building blocks. 55
3.6 A common block can be defined by a `Block` element. 56
3.7 A block can define several configurations. 57
3.8 The expanded specification of Figure 3.7. 58
3.9 A prototype dependency graph and specification rules. 59
3.10 The implementation linkage graph is derived from the prototype dependency graph and specification rules in Figure 3.9. 60
3.11 The implementation linkage graph after default implementations have been marked. 62
3.12 The implementation linkage graph after **Rule** 10 has been added. 62
3.13 The final implementation linkage graph. 63

4.1 Podiums and runtime services in the application framework 67
4.2 Using a proxy actor to represent the remote actor locally. 73
4.3 Recycling a proxy actor does not release the reference in the proxy table. 75
4.4 Reference cycles cannot be broken by local garbage collectors. 75
4.5 The **PrototypeLoad** attribute organizes client prototypes into loading batches. 78
4.6 Implementations of a prototype can share code. 80

5.1 A basic Web calculator. 84
5.2 The components of the Web calculator 85
5.3 The enhanced calculator. 88

5.4 The `PICalculator` actor. 88
5.5 A warning dialog box pops up when the browser stops responding for the long running π calculation. 90
5.6 A typical presentation layout of search applications. 91
5.7 The structure of the search application. 92
5.8 Lengthy and non-essential excerpts can be loaded on demand. 95
5.9 The architecture of the context management system for Web applications. 97

Chapter 1

Introduction

Evolving from its original mission of content delivery, the Web has become a gateway of assorted interactive applications: people access emails, shop online, trade equities, manage accounts and even remotely control home appliances using various Web applications. The popularity of Web applications is the consequence of the convenience and ease enjoyed by end users. In theory, users employing any Web client and platform to connect to the Internet can access all kinds of services around the world. In reality, the increasing heterogeneity and complexity of the Web is jeopardizing this ideal paradigm. Unlike its typical standalone and distributed peers, a Web application encounters a variety of execution environments and numerous, unpredictable circumstances in its deployment. Since the early days of the Web, developers have identified the need to differentiate execution contexts: it was common for a Web application to have one version with *HTML frames* and another without. Context differentiation has become more critical as the Web is evolving at a high speed: new kinds of context-specific versions such as *broadband*, *JavaScript-enabled*, *HTML only* and *low graphics* can be found in many Web applications as new Web-enabled devices, software systems and communication protocols keep emerging while old ones still survive. The diversity in actual execution environments results in a dilemma for Web application developers: to deliver universal service access, they strive hard to overcome the diversity; on the other hand, they are eager to exploit the opportunities offered by an execution environment to provide a certain level of QoS (Quality of Service). The thesis describes our research in tackling with this problem.

1.1 Background and Motivation

The universal accessibility of Web applications relies on two standards, the *lingua franca* HTML [19], and the ubiquitous HTTP protocol [37]. However, after a decade of evolution, the Web has become a smorgasbord of Web clients and servers which employ emerging standards and proprietary features (See Table 1.1). Moreover, the introduction of Web-enabled gadgets and appliances has further increased the degree of heterogeneity. In a sense, the Web has become fragmented, making it harder to ensure the compatibility of servers and

Web Browser	Share
Explorer	84.66%
Firefox	12.72%
Safari	1.79%
Opera	0.61%
Netscape	0.11%

June 2007
OneState.com [78]

Web Server	Share
Apache	53.76%
Microsoft	31.83%
Google	3.99%
Sun	1.86%
Others	8.56%

June 2007
Netcraft Ltd. [71]

Scripting System	Share
PHP	45.71%
ASP	18.04%
CGI	17.57%
PERL	6.34%
.NET	6.20%

April 2007
NetVention Inc. [60]

Table 1.1: Market shares of different Web browsers, Web servers, and server-side scripting systems

clients. For Web clients, we can classify their differences into three categories:

Naming: Browsers may adopt different names for scriptable components and built-in functions. For example the variable holding X-coordinate of a browser window could be *left* or *screenX*. Standards [21,39,73] exist but are rarely followed strictly.

Interpretation: It is common for a Web page to look different in two browsers because the browsers render HTML documents dissimilarly. Browsers may also expose different DOM (Document Object Model) [21] trees for an HTML document, making client script less portable.

Capability: Perhaps the most infamous example of a lack of uniform capability in early years of the Web was support for HTML *frame*. Content providers were forced to have two sets of pages to serve clients with HTML frames and the others without. Now the list of capabilities includes JavaScript [39], CSS (Cascading Style Sheets) [73], DOM Level 1, 2, 3 [21] and various proprietary plugins.

Discrepancies in *naming* and *interpretation* can be handled by browser detection code with the help of macro expansion and supporting libraries. But discrepancies in capability has profound implications on application design: for example, without JavaScript and CSS support, most user actions must be forwarded to the server for further handling. In order to maintain universal accessibility, the industry and practitioners had historically paid more attention to server-side technologies and ignored many rich features that had been available in Web browsers for years. Nevertheless, because of the inherent latency in communication channels, Web servers are not good at processing certain user actions that require immediate response – for example, actions such as *mouse move* and *drag-drop*. To address this problem, browser-centered Web applications using the *AJAX* (Asynchronous JavaScript and XML) [50] technique have been proposed and have won considerable acclaim for their responsiveness. However, use of the new crop of Web applications comes with a price: not all browsers can access these applications, either because they do not have the capabilities that are required in the browser by the application or because of the user's security settings. At this point Web developers face a dilemma: to provide universal access with a loss of efficiency or give up on universal accessibility.

Because a Web application is usually hosted in a single server or a pool of servers using the same platform, incompatibility among servers is not a common problem. Still, diversity in server technologies and systems ensures non-portable application code: when a service provider decides to switch to another server platform, or to upgrade to a newer version, a large portion of its Web applications must be re-designed and re-programmed.

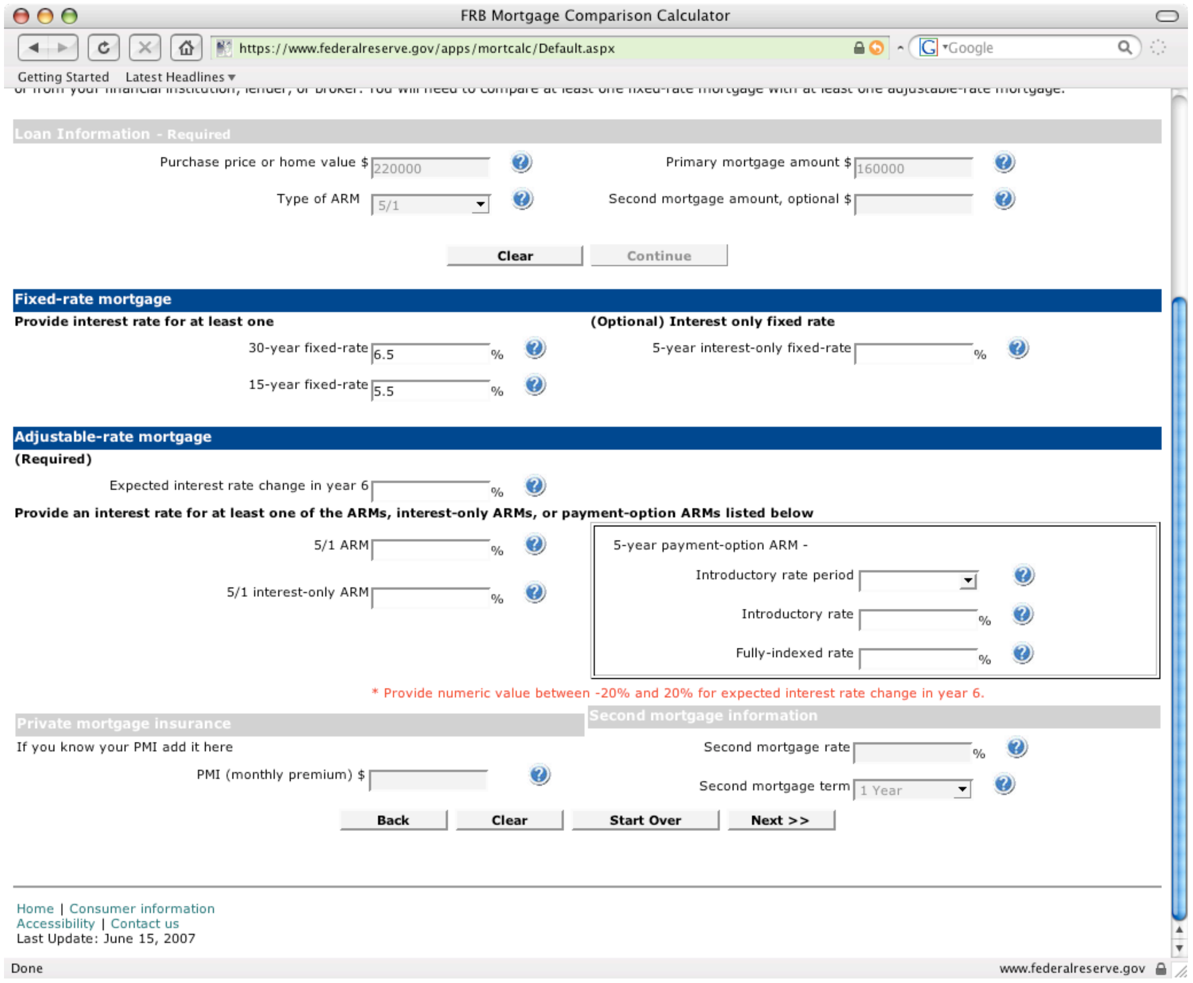
One could argue that it may not be a great sacrifice to support only the advanced browsers given the dominance of these browsers in the market. However, there is yet another conceptually related difficulty. Observe that Web applications are inherently distributed and the emergence of scripting-enabled browsers allows great flexibility in application deployment between a client and a server. The challenge to efficient and universally accessible Web applications comes from the incompatibility of execution platforms; but incompatibility also exists in a single instance because the client and the server are usually not compatible with each other. Viewing a Web application as a composition of distributed components, it becomes crucial to determine *where* and *when* to deploy components of interest.

1.2 Application Scenarios

To better illustrate how component distribution impacts on application runtime features, we introduce a sample set of application scenarios common in many Web applications. The motivating applications are divided into two categories corresponding the two questions we have raised: *where* to run a component and *when* to deploy a component. The results show an execution context may favor certain deployment plans and hence component distribution should be a separate concern orthogonal to the application logic.

1.2.1 Location of Execution

Validating user input is a common task in many Web applications. On the one hand, because local execution results in a faster response and reduces the server load, validation should be performed at the client if possible. On the other hand, if a service provider wants to conceal the validation logic, or the input has to be validated against confidential data, the validation component should be deployed at a secure place. Although such a requirement of confidentiality means that the validation component should normally be deployed at the server, in some cases, such as in an *Intranet* with a secure communication channel and a trusted client, the deployment could be at the client.


FRB Mortgage Comparison Calculator

https://www.federalreserve.gov/apps/mortcalc/Default.aspx

Getting Started   Latest Headlines

or from your financial institution, lender, or broker. You will need to compare at least one fixed-rate mortgage with at least one adjustable-rate mortgage.

Loan Information - Required

Purchase price or home value $	220000	Primary mortgage amount $	160000
Type of ARM	5/1	Second mortgage amount, optional $	

Clear   Continue

Fixed-rate mortgage

Provide interest rate for at least one   **(Optional) Interest only fixed rate**

30-year fixed-rate	6.5 %	5-year interest-only fixed-rate	%
15-year fixed-rate	5.5 %		

Adjustable-rate mortgage

(Required)

Expected interest rate change in year 6 %

Provide an interest rate for at least one of the ARMs, interest-only ARMs, or payment-option ARMs listed below

5/1 ARM	%	5-year payment-option ARM -	
5/1 interest-only ARM	%	Introductory rate period	
		Introductory rate	%
		Fully-indexed rate	%

* Provide numeric value between -20% and 20% for expected interest rate change in year 6.

Private mortgage insurance   **Second mortgage information**

If you know your PMI add it here		Second mortgage rate	%
PMI (monthly premium) $		Second mortgage term	1 Year

Back   Clear   Start Over   Next >>

Home | Consumer information
Accessibility | Contact us
Last Update: June 15, 2007

Done   www.federalreserve.gov

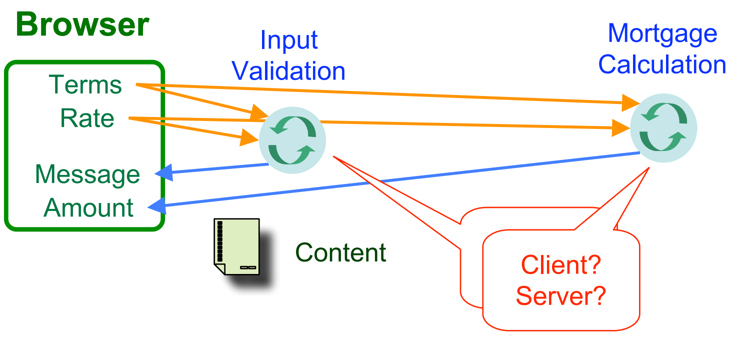


Figure 1.1: FRB Mortgage Comparison Calculator is a representative Web application performing both computation and validation.

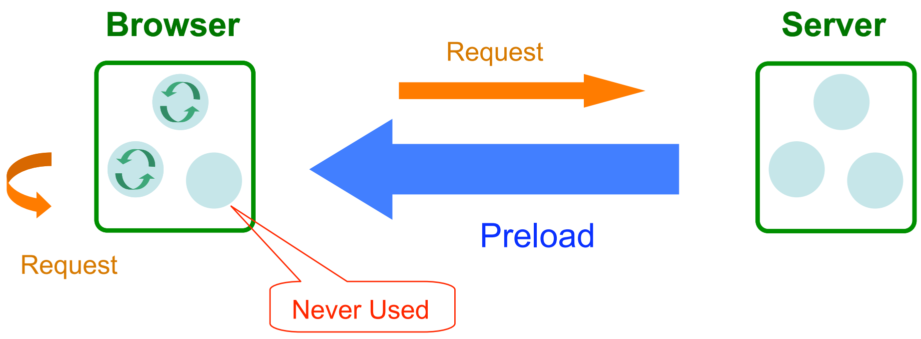


Figure 1.2: Preloading in batch is more responsive in a context where the bandwidth is not an issue.

For computing components which require no input from the server and consume few CPU cycles, such as a unit converter or a mortgage calculator, it is preferable to deploy them in the client both for a faster response and less work in the server. In other cases, the best location is not always clear: a CPU-intensive component which takes input from a back-end data source is usually better allocated in the server for two reasons–first, because JavaScript is not efficient at computing, and second, because bringing data across the Internet is a significant overhead; however, when a server has its CPU extremely busy but is less stressed in I/O and bandwidth, it is preferable to shift the component to the client. A case of this sort that we have seen in practice is a component which extracts excerpts of documents based on the user's query. The computation consumes significantly more CPU cycles than searching the documents satisfying the query. Although the computation normally finishes in milliseconds, it can take dozens of seconds when the server is overloaded with multiple tasks; in the latter context, deploying the component at the client is more appropriate.

Figure 1.1 shows a Web application[1] which computes the monthly payments for several kinds of mortgages. In addition to the number crunching mortgage calculation, the application contains components for various kinds of input validation. Relocating the components results in different performance characteristics of the application, such as initial latency, response time and system load, which would favor certain execution contexts.

1.2.2 Timing of Deployment

Many user interface controls, such as layered menus, list boxes and detailed information panels, require multiple levels of presentation. These components can be *preloaded* with their containers. A common technique for implementing multiple levels of presentation is to embed all components in a single page–whether such components are immediately ready for use or not. The advantage of this deployment scheme

[1]Federal Reserve Board Mortgage Comparison Calculator. https://www.federalreserve.gov/apps/mortcalc/

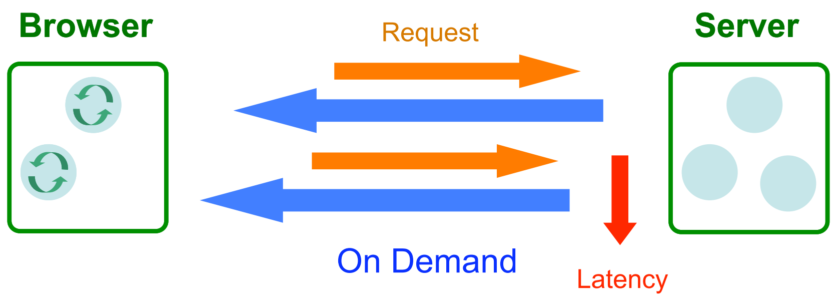


Figure 1.3: Loading on demand saves bandwidth at the cost of latency.

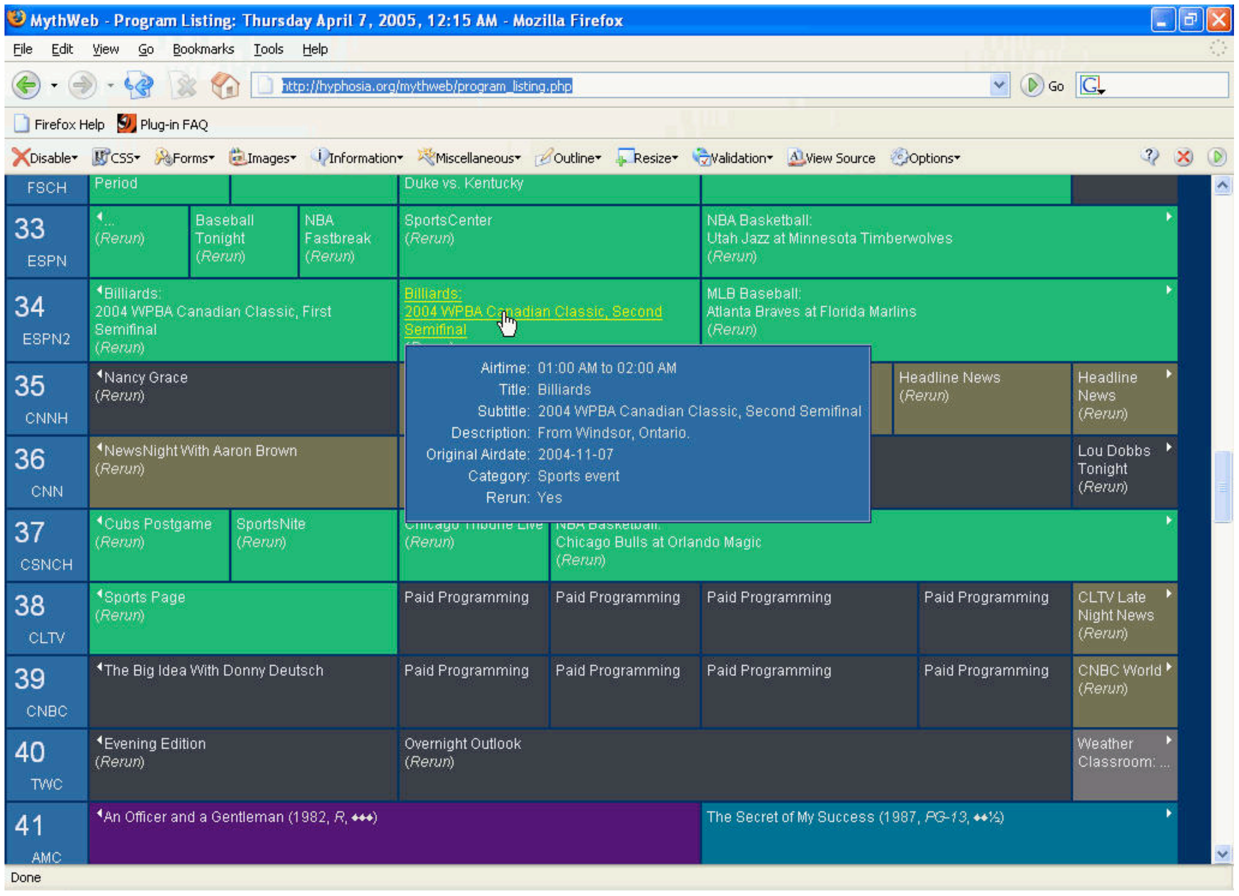
is that once the page has been loaded, the application responds to a user action instantly: preloading client components results in a shorter response time. However, preloading may waste bandwidth as some preloaded components will never be used (see Figure 1.2). Consequently the deployment scheme is not feasible in an execution context where the available bandwidth is scarce. Examples of such execution contexts include a busy site or a mobile network connection. Alternatively, components can be loaded *on demand* when a user action explicitly asks for a component (see Figure 1.3). On-demand loading suffers from the inherent latency in the Internet but uses precious bandwidth more efficiently. A smarter solution is to exploit the user's profile: if certain preferences or patterns can be identified, the application preloads only frequently used components. In this case, the preferable distribution depends on the current network utilization and the identifiable usage patterns.

Figure 1.4 is a screenshot of a Web-based front-end of MythTV,[2] a PVR (personal video recorder) application on linux. When the mouse hovers over a TV program title, a panel showing the program to detail appears. In the specific case depicted, the full information of all TV programs and supporting script code accounts for more than 65% of the total download. It is common for users to browse only a few titles (out of hundreds) before issuing a record command and leaving the page. Loading the full information only when a title is selected reduces the amount of data transmitted at the cost of a pause after each selection.

1.2.3 Component Distribution as a Separate Concern

In the past the Web server was solely responsible for two key tasks: generating dynamic contents and accessing data sources (see Figure 1.5); now advanced browsers are able to consume raw data and then create HTML fragments accordingly, thus sharing the responsibility of content generation (see Figure 1.6). For example a classic Webmail application retrieves emails from a local database or an external source

[2]MythTV. http://www.mythtv.org/


MythWeb - Program Listing: Thursday April 7, 2005, 12:15 AM - Mozilla Firefox

http://hyphosia.org/mythweb/program_listing.php

FSCH	Period		Duke vs. Kentucky			
33 ESPN	... (Rerun)	Baseball Tonight (Rerun)	NBA Fastbreak (Rerun)	SportsCenter (Rerun)	NBA Basketball: Utah Jazz at Minnesota Timberwolves (Rerun)	
34 ESPN2	Billiards: 2004 WPBA Canadian Classic, First Semifinal (Rerun)		Billiards: 2004 WPBA Canadian Classic, Second Semifinal	MLB Baseball: Atlanta Braves at Florida Marlins (Rerun)		
35 CNNH	Nancy Grace (Rerun)			Headline News (Rerun)	Headline News (Rerun)	
36 CNN	NewsNight With Aaron Brown (Rerun)				Lou Dobbs Tonight (Rerun)	
37 CSNCH	Cubs Postgame (Rerun)	SportsNite (Rerun)	Chicago Tribune Live (Rerun)	NBA Basketball: Chicago Bulls at Orlando Magic (Rerun)		
38 CLTV	Sports Page (Rerun)	Paid Programming	Paid Programming	Paid Programming	Paid Programming	CLTV Late Night News (Rerun)
39 CNBC	The Big Idea With Donny Deutsch	Paid Programming	Paid Programming	Paid Programming	Paid Programming	CNBC World (Rerun)
40 TWC	Evening Edition (Rerun)	Overnight Outlook (Rerun)				Weather Classroom: ...
41 AMC	An Officer and a Gentleman (1982, *R*, ♦♦♦)			The Secret of My Success (1987, *PG-13*, ♦♦½)		

Airtime: 01:00 AM to 02:00 AM
Title: Billiards
Subtitle: 2004 WPBA Canadian Classic, Second Semifinal
Description: From Windsor, Ontario.
Original Airdate: 2004-11-07
Category: Sports event
Rerun: Yes

Done

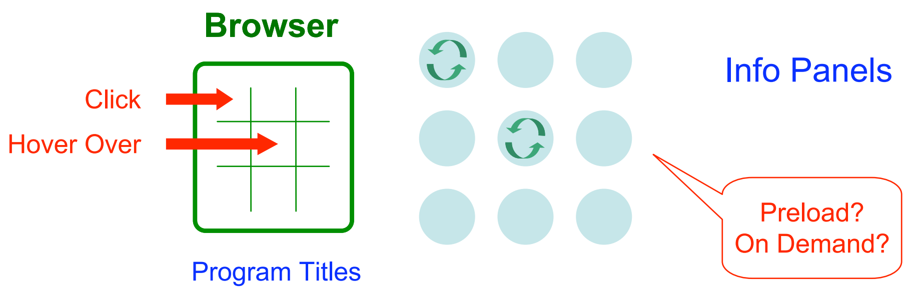


Figure 1.4: MythWeb, a Web-based front-end of a PVR system, exhibits a typical user interface with multi-layered presentation.

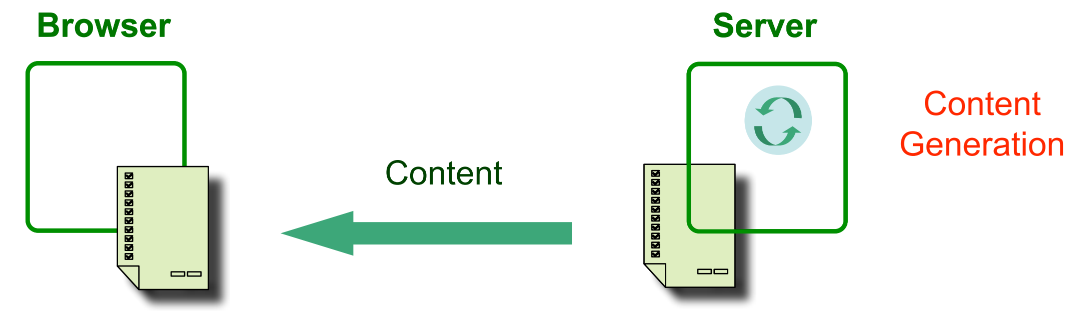


Figure 1.5: The classical server-based Web application model.

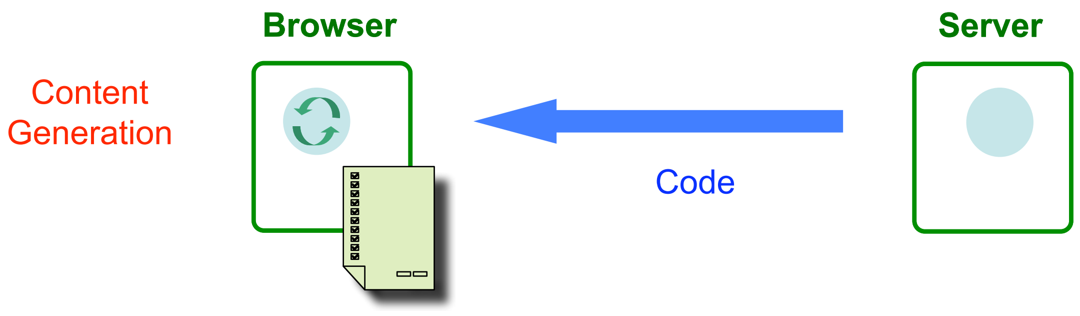


Figure 1.6: Computing locally shifts load to the client.

through the POP3 protocol, and then prepares formatted HTML documents for the client. Nowadays many *live* Webmails delegate the task of presentation to the client, demoting the server's role to a gateway to the data source. Processing data locally usually results in greater responsiveness but may consume more bandwidth if processing the data reduces what has to be communicated. Relocating the computation logic is obviously non-trivial: the client and the server rarely speak the same language and their models in handling presentation (HTML/XML documents) are considerably dissimilar. The discrepancy between the client and the server is far more than that among the clients.

A Web application consists not only of the data generated at runtime, but also of many pieces of static code. Loading both the code and the data piecewise saves bandwidth but hurts responsiveness, while loading in big batches clogs up the connection under certain conditions. In other words, different execution contexts have their own preferences in loading policies. Changing the loading policy of application components is no easier than redistributing them. The Web client refers to an Internet resource through its URL (Universal Resource Locator). Different loading policies mean both different sets of URLs and specialized loading code for the client as well as the server.

As our analysis in Section 1.1 showed, the real challenge of diversified Web platforms is in the *capability* of key technologies. Abstractly it is also a problem of component distribution: less capable clients imply fewer choices in deployment. For example, a client without a scripting environment needs the server to do all the computation; the lack of AJAX enabling features suggests limited local processing capabilities such as the ability to support on-demand loading.

The discussion above illustrates how the preferred application distribution is often strongly dependent on the execution context; relevant variables of such a context include network properties, security constraints, server load and even the user's behavior, all of which are hardly foreseeable in advance. Following the principle of *separation of concerns* [32], *aspects* orthogonal to the application logic should be decoupled from logical components. In a traditional distributed environment, where a certain uniformity is taken for granted, decisions on these aspects can be postponed until the deployment time with little adverse implication. In contrast, Web application developers have to make the right decisions early in the design phase: reverting them later could cost dearly because of the diversity of programming paradigms on different platforms. This means that Web developers face the second challenge: how to find a universally good distribution scheme for various execution contexts?

1.3 Problem Analysis

Instead of searching for a universally good plan in a rapidly evolving Internet age, we believe it is more practical to have different component allocations and loading policies for different execution contexts. Unfortunately in practice Web applications are not structured to provide such versatility. Our study identifies two obstacles: Web applications are rigid in structure (hard to reconfigure) because of the lack of a uniform and symmetric programming environment. While some methods to enable components to be relocated and loading policies to be changed have become available, support for describing and executing application distribution schemes is still missing.

Although the HTTP protocol and the HTML document format brought in ubiquitous Web applications, ironically they are the culprits that are responsible for the rigid Web application structure:

- The HTTP protocol enables simple and efficient document retrieval, but its *asymmetry* and *statelessness* make it less attractive for interactive applications. One-way initiation requires some twists to realize server-to-client communication, and many Web server platforms resort to ad hoc techniques to maintain *session state*. Programming artifices are often platform-dependent and thus not portable.

- The Web application is in a loosely-coupled client-server model. This dichotomy has allowed each side

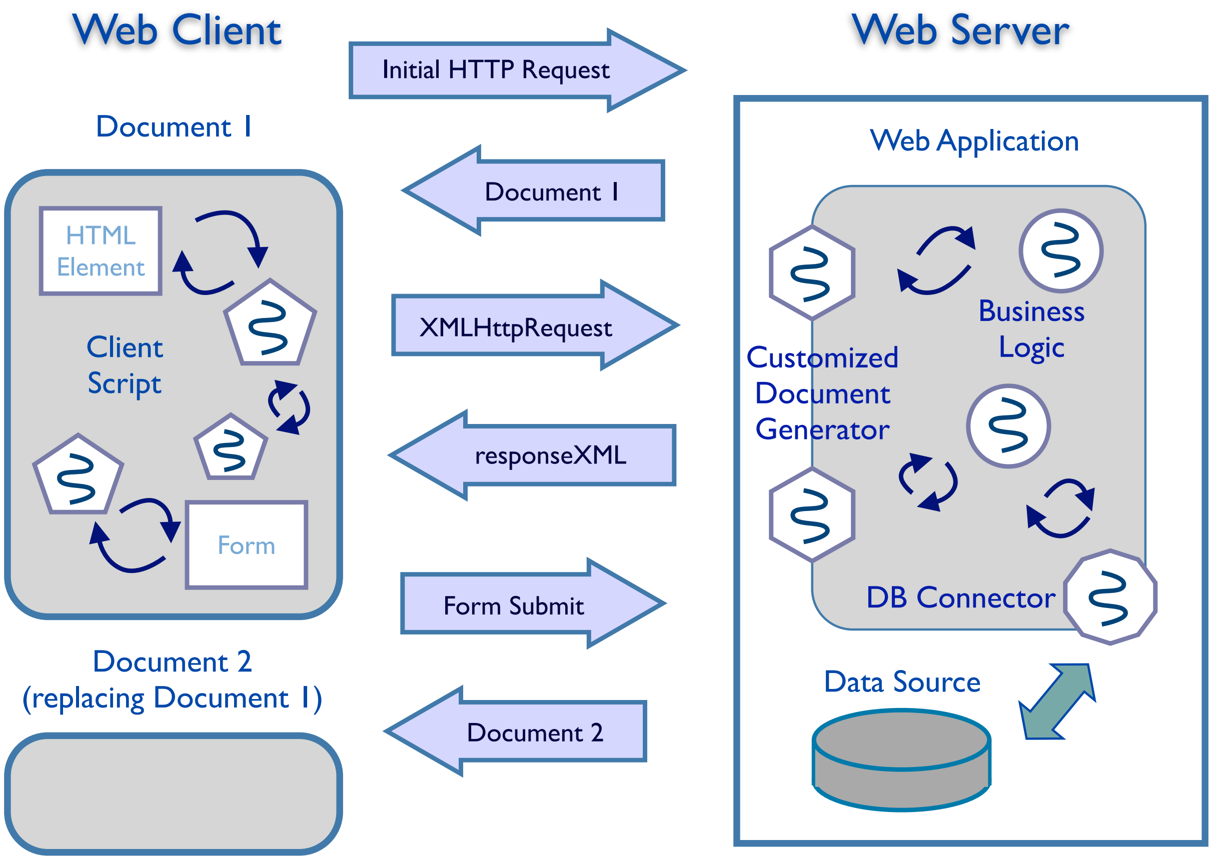


Figure 1.7: The document-based architecture is not suitable for reconfigurable interactive Web applications.

to undergo independent evolution: for the client, the focus of development has been to provide more powerful systems to manipulate local contents and to access remote resources, while for the server, the focus of development has been better application state management and content customization. The independent evolutionary paths with different foci have resulted in mutually incompatible object models and technologies at the two ends.

We believe the main cause of the problem is that the Web architecture is *document-based* (Figure 1.7), which is well suited for the original purpose (document delivery) but not suitable for interactive Web applications. However, the inertia of such a gigantic system has inhibited all attempts to significantly reshape the Web. Instead, the emergence of new technologies and protocols keeps fortifying the paradigm even more.

Because the development of Web clients and that of Web servers have had different goals, it is not surprising that components for the same purpose can be very different in the client and in the server. Consequently, application components can not be freely relocated for heterogeneous and asymmetric platforms. Another implication of the document-based architecture is its plain loading mechanism: everything from the server, no matter whether it is a fragment of presentation, a piece of code or a part of the runtime state, is loaded from the server as a document and then processed by local document handlers. Changing the loading policy not only means changing server content generators and static resources, but also the corresponding client processing code. The *document* is not a good metaphor for application code and runtime state, for these a more supportive loading mechanism is needed.

The emphasis on documents also impedes the development of a general specification system for application distribution on the Web. Specifically, there is little work on specifying and enforcing Web application distribution schemes. In contrast, *Web Style Sheets* [23] are widely used to describe how documents are presented so that presentation can be distinct from data content. Using CSS, a document can have customized *look-and-feel* in a specific environment; on the other hand, XSL (eXtensible Stylesheet Language) [49] is able to convert a document into other formats so that it can be viewed in various styles.

In summary, incompatible application models underlie a rigid application structure, and the plain document loading mechanism entangles loading strategies with the application logic: both lead to non-reconfigurable component distribution of Web applications. Furthermore, there is little attention on expressing and enforcing Web application distribution schemes, including those for component allocation and loading policies.

1.4 Approach

The goal of this research is to enable customizable Web applications which can adapt to different execution platforms and reconfigure component distribution according to the execution conditions [12]. In terms of *software product line engineering* [84], a customizable Web application is a product line, each instance of which serves a specific execution context. There are two critical requirements for a Web application product line: universal accessibility and context-dependent component distribution.

1.4.1 Virtualization

We address both requirements–universal accessibility and context-dependent distribution–by using *virtualization* on execution platforms to provide a uniform programming environment for components. Note that if we can model a Web application as a composition of distributed objects in a uniform environment, the two challenges collapse into one: a less capable client means fewer options in component distribution and

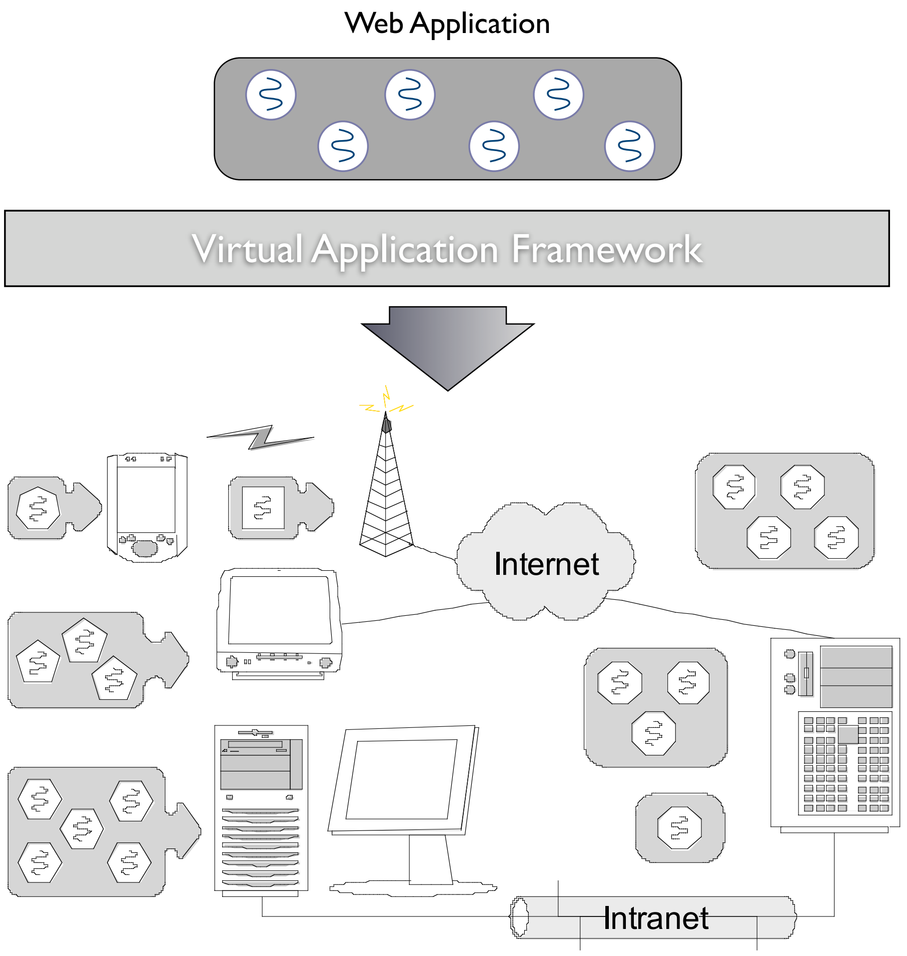


Figure 1.8: The virtual application environment enables platform independent and location agnostic application development.

thus both challenges can be addressed. The virtualization must be a high-level abstraction: programmers no longer need to deal with pages, HTTP requests, cookies, sessions, which are entities in specific application models and platforms; instead, they can focus on building components and defining their interaction, that is, on the *application logic*. The core of the virtualization has to be simple enough that it can be implemented in existing platforms, including thin clients and simple servers; in addition, it must be extensible: if a special feature is available in a platform in use, an extension can be designed to express that feature.

Specifically, we design a virtual programming environment (see Fig. 1.8) which hides the incompatibilities in many platforms and models. A Web application of the virtual environment represents a product line characterized by its application logic, not a specific implementation deployed on an actual execution platform.

1.4.2 Separation of Concerns

Because one of our goals is to customize the Web application's component allocation and loading policies, these concepts should not be entangled with the application logic. Following the principle of *separation of concerns*, our virtual programming environment is not only platform independent but also *location agnostic*: there is no notions about component location and component movement. Ideally, an application of the virtual environment can be deployed on the Web, in a standalone computer, or even in a *grid* environment [41, 94], as long as the respective framework implementation and extra details such as component distribution are provided.

In any case, the extra details are required to build an executable Web application. We design a specification system [13, 14] to complement the virtual programming environment. The system consists of two parts: a specification language with which developers write component distribution schemes, and tools, which assist runtime environments to execute these schemes. Combining an application with a specification scheme, a Web application customized to a specific execution context is defined. Inspired by the success of Web Style Sheets, our specification system is also based on the idea of using *transformations* (see Fig. 1.9): namely, to convert a composition of components (precisely speaking, component designs) into another composition of annotated components. The annotation attributes are given by a specific framework implementation. For example, in a typical Web framework the location of a component may be the *client* or the *server*. Specifying component allocation can be achieved by annotating each component with the desired attribute value. We will show that sophisticated loading policies can be defined in a similar way with specially devised attributes.

Specification through transformation has another advantage: in addition to the application model, the specification system is also independent of the runtime environment. The clear separation implies that the specification system not only works with the implementation of the specific runtime environment described in this thesis, but is compatible with all runtime environments that expose control attributes (specification interfaces). Less coupling between the specification system and the runtime environment allows greater flexibility in runtime environments and more interexchangeability in specification systems.

1.4.3 Generative Programming

One approach to support virtualization over heterogeneous platforms is by defining and building a *middleware* [2, 3]. This approach has the disadvantage of requiring the deployment of new software systems and protocols, which is impractical for the sheer scale of the Internet and the heterogeneity of existing platforms. Instead, we adopt the paradigm of generative programming [26] for customized application execution: before deployment, the composing components of a Web application undergo a generative process to obey the

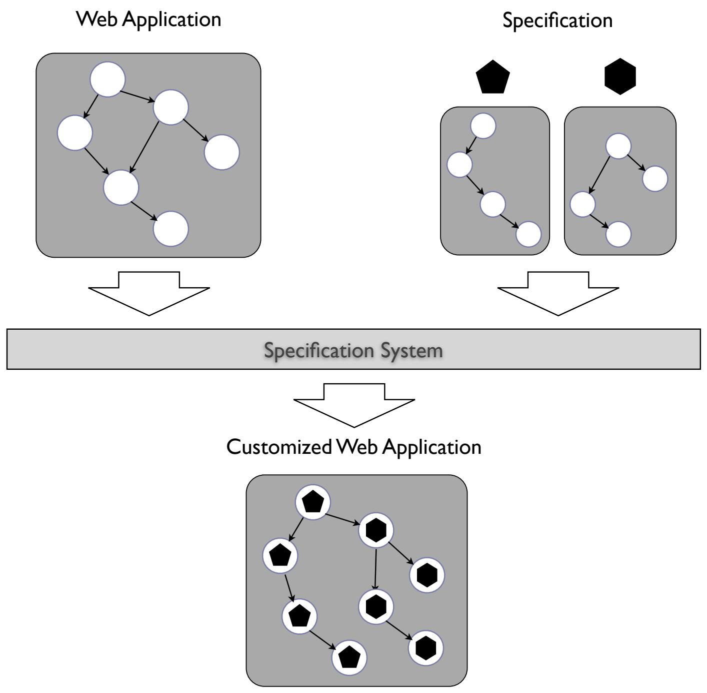


Figure 1.9: The specification system transforms a composition of components into another composition of annotated components.

annotations specified by the specification system (see Fig. 1.10). Using the terminology of Aspect-Oriented Programming (AOP) [35,62], aspects relevant to object distribution and target platforms are woven into the code. To host these generated components, a light-weight runtime environment is required for each participating platform. In addition, the runtime environments of a Web application have to coordinate with each other to manage distributed components across the HTTP channel. We implement a generative application framework including runtime environments, coordination services and generators. The framework implementation supports reconfigurable component distribution through a specification interface, specifically an attribute to control component allocation and another to regulate loading patterns.

The generative approach offers adaptability to address universal accessibility: developers can take advantage of latest technologies as soon as their supporting generators and framework modules become available. At the same time, the approach provides flexibility needed to dynamically optimize the distribution of objects depending on the given configuration and requirements.

1.5 Contributions

There are four main contributions in this thesis:

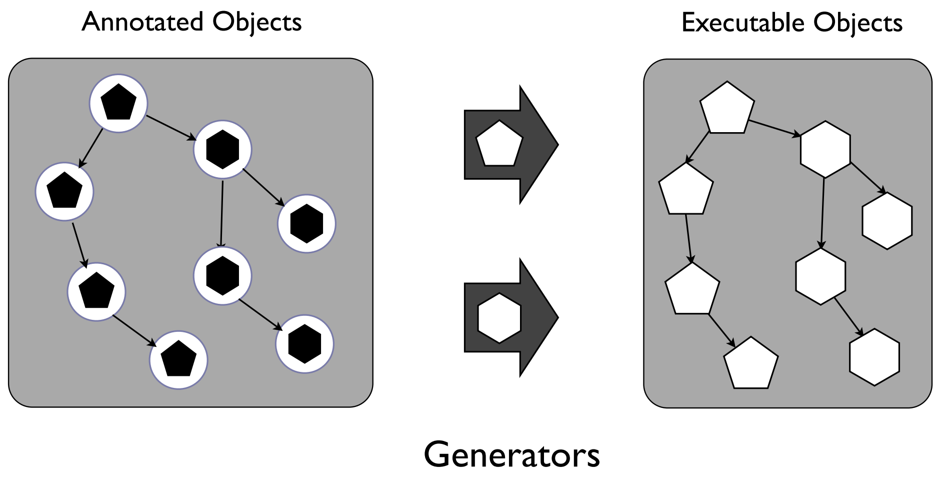


Figure 1.10: Generators convert annotated components into executable components in the target platforms.

1. *Empirical evidence of link between component distribution and performance.*

   We conduct experiments on a set of representative elements of Web applications with assorted component distributions. Our results show the existence of distinct runtime characteristics and thus varying performance on different execution contexts.

2. *An application model for location-agnostic and platform-transparent Web applications.*

   We leverage the classic actor model to fit the paradigm of component-based distributed software, which includes Web applications. The model's focus on communication (in contrast to sharing) enables location-agnostic development while its simplicity and extensibility guarantees the application's portability over heterogeneous platforms.

3. *A generic specification system for expressing and enforcing orthogonal concerns.*

   We design a scheme to express concerns by annotating components and a language to structure these annotations systematically. To resolve annotations on individual components, we develop an algorithm to transform specification rules. A merit in our design is the clean separation among concerns, component logic and implementation; this loose coupling provides reusability by enabling the development of a software systems as a collection of building blocks.

4. *An extensible generative framework supporting customizable Web applications.*

   We implement an execution framework and demonstrate its effectiveness in customizing Web applications by combining annotations with component designs. The framework is generative, allowing it to serve numerous contexts features and be extensible for future platforms.

1.6 Research Overview

The research described in this thesis can be divided into three parts:

- *A programming model and a scripting language* to describe Web applications. The model provides a virtual environment for component interaction and the language is used to express the behavior of individual components.
- *A specification language* to specify special requirements on components, and an algorithm to transform a generic application design into a customized version which embeds the requirements.
- *A mechanism to execute a customized Web application* according to its embedded requirements. The mechanism includes an application framework implementation which consists of a set of static libraries and runtime systems, and generators translating component designs into specific component implementations of target platforms.

We choose an active object model, specifically an extended *actor model*, for the virtual programming environment. The actor model is simple: *actors* (active objects) interact with each other only through message passing; the simple semantics makes it possible to support the model in an unsophisticated execution platform such as a Web browser. The model is also expressive and extensible: as we will illustrate, many sophisticated features such as advanced event models and coordination constructs can be implemented with message exchange protocols.

We design an actor-oriented scripting language, `ActorScript`, to describe an actor's behavior. The scripting language borrows the control structure and most of its data types from *JavaScript*. Note that most programming languages would serve for this purpose, as long as they can interact with the virtual actor environment. The reason to have our own model and language is that existing ones tend to have special features of their own, which may not be efficiently implementable in others. The purpose of the model and language is to describe the application logic, which is not intended to be executed directly, but to be used to generate customized executable applications. An application of the virtual environment is represented by a set of actor designs and referral relations between them.

Our specification system, `ActorSpec`, is based on *actor annotations*. The basic building block is the specification rule, which associates a set of actors with a specified attribute. To enable modular specification rules, we design an XML-based language so that complex specification rules can be composed from simple ones in a structural form. As indicated, `ActorSpec` transforms an application into an annotated application following the supplied specification. Since an application is represented by a set of actor designs and referral relations, the result is another set of actor designs with attributes and their referral relations. We develop an algorithm to perform the transformation and prove its correctness. Note `ActorSpec` only *annotates* actor designs with attributes, it does not *interpret* these attributes. The interpretation and enforcement of attributes is left to the execution environment.

We implement a generative application framework for Web applications, `ActorWeb`, which supports adaptive components and customizable distribution. The core of `ActorWeb` is the *Podium*, which provides a runtime environment for actors in a specific platform. We have a client podium hosting JavaScript actors in the Web browser, and a server podium for actors in both JavaScript and Java in the Web server. To support actor management between podiums such as remote actor creation, remote method invocation and garbage collection, a set of runtime systems are designed to coordinate podiums. `ActorWeb` realizes adaptive components through generators. Before being deployed, an actor design in `ActorScript` goes through a generator for the target podium. For example, the generator for the client podium generates an actor implementation in JavaScript, with linkage to the client podium's libraries. With control attributes exported, `ActorWeb` enables reconfigurable actor distribution via a linker and a loader. The linker resolves the correct linkage of actor designs by their *Podium* attribute, so that when an actor is created, it will be deployed in the right podium as specified. The loader loads actors and actor designs accoding to their loading attribute, either when the ones referring to them are being loaded or when they are actually demanded.

1.7 Thesis Outline

The rest of the thesis is structured as follows. Chapter 2 starts with a comprehensive description of the virtual programming environment: specifically, we define the extended actor model and `ActorScript`, followed by examples of expressing advanced features with message protocols, and a module system for reusing actor designs. Chapter 3 describes `ActorSpec`, including the specification rules, the specification language, the algorithm for transformation and a proof for its correctness. We present `ActorWeb` in Chapter 4, which begins with the design of the generative application framework and the control attributes the framework exposes to the specification system; explanation of each module then follows. Using this understanding of framework composites, the mechanism reconfiguring actor distribution and generating adaptive actors is delineated. We also list several optimization techniques useful in certain podium implementations. Chapter 5 reviews some related work and Chapter 6 illustrates a couple of representative applications and reports the experiments. The final chapter concludes the thesis with a discussion and provides pointers to future work.

Chapter 2

Programming Model and Language

There are two motivations in our programming model and language. First, to provide a virtualization over the heterogeneous distributed Web platforms for platform independent and location agnostic application development. Second, we want to support component-based Web applications because most commonly used Web application platforms, including the ones we target at, are component-based systems. Given this motivation it is natural to view a Web application as a composition of distributed active objects, or *actors*. Being distributed implies knowledge is encapsulated within objects; being active means the objects are able to decouple method execution from synchronous method invocation [69], and thus exploit concurrency. We believe the distributed active object model is especially suitable for location agnostic application development in a dynamic distributed environment such as the Web.

In this chapter we describe our virtual programming environment, which includes a programming model– an extended actor model, and a scripting language `ActorScript`. The programming model defines the ways in which components interact with each other; the scripting language is used for component design. Note the programming model does not rely on `ActorScript`: any language can be used to express a component's behavior as long as the component follows the programming model's semantics in communication.

2.1 Actors: A Model for Distributed Active Objects

To support location agnostic Web application development, we assume that distributed objects do not implicitly share information (such as through classes or shared variables) but interact with each other only through message passing. Such lack of sharing enables objects to be placed freely. Distributedness also suggests the potential of concurrency: components in different platforms can execute independently. Moreover, the details and intricacies of communication protocols are platform dependent and have implications on object location, and thus have to be abstracted out . All these lead us to the defining characteristics of the actor model [1, 55].

2.1.1 Classic Actor Model

In the classic actor model, *actors* are autonomous components which operate asynchronously. Each actor encapsulates a state and a set of methods that manipulate the state with a thread of control. In response to a message, an actor can *create* new actors, *send* messages to other actors, and *change* its behavior for the next message; correspondingly, the actor model supports the following three primitives:

- **create** takes a behavior description and creates an actor. It may take additional parameters for initialization. The newly created actor has the specified behavior and its reference is returned to the creator.

- **sendto** takes the receiver's reference and sends the message to it asynchronously: the sender continues its computation without waiting the response from the receiver.

- **become** takes a new behavior description which is used to handle the next message. The rest of the computation is carried out by an anonymous actor.

In the classic actor model, actors communicate with others only through asynchronous messages; other message-passing paradigms, such as synchronous communication, must be built in terms of asynchronous messages. In addition, actors are *synchronization blocks*: actors process one message at one time, or at least exhibits this semantics. Thus the actor serves as the basic unit of encapsulation and distribution in concurrency.

2.1.2 Model Extensions

The classic actor model is simple–it only has three primitives, and extensible–through message exchange many other language constructs can be modelled properly. For example, shared memory can be implemented as an actor which accepts *read* and *write* messages embedding *memory addresses*. Synchronous interaction can be realized through *Continuation Passing Style (CPS)*: instead of returning to the sender, the receiver takes an additional *continuation* argument which is meant to receive the return value and control from the computation after it is done.

In spite of the completeness of the actor model's asynchronous semantics, our aim is not to design a new system based on the paradigm; rather, the aim is to use the model to facilitate Web application development on existing systems. Most programmers are used to synchronous semantics in communication because of its similarity to method invocation, and *set and get* operations on object properties, which are natively supported in our target platforms. Neither is it straightforward to perform CPS transformation on these synchronous language constructs nor are first-class continuations necessarily supported or implementable in these systems. Thus, it is more convenient to represent synchronous messages directly. The *become* primitive is also uncommon in typical component-based systems; since an actor can aggregate all its possible behaviors at initialization and select the desired one for the next message by changing its local state, the *become* primitive is often omitted in actor system implementations. Furthermore, in the classic actor model automatic garbage collection is assumed; given the difficulty of distributed garbage collection [95,97], having a primitive to release an actor from the system is more common in practice.

To support our generative approach, we separate the programming environment into two levels. The core is a virtual actor framework, which supports three basic primitives for actor management and message exchange.

- **create** takes an actor behavior's name with additional arguments if the behavior is parameterized, and returns a reference of the newly created actor. It implies that the actor framework can bind an actor behavior with a name. Note the actor framework does not understand the semantics of the behavior of hosted actors: it just creates a referential entity associated with the designated behavior.

- **send** takes three arguments: in addition to the receiver's reference and an unformatted binary stream as the message body, a flag is required to indicate whether its delivery is synchronous or asynchronous. When a message is being dispatched synchronously, the sender is blocked until the receiver finishes processing the message, and then the response to the message is returned to the sender. On the other hand, sending an asynchronous message allows the sender to continue its computation immediately, and the response is consumed by the actor framework.

- **destroy** takes an actor's reference and removes the actor from the framework. Further messages sent to the destroyed actor are ignored and an error code is returned to the sender if the message is synchronous. Note that we do not presume the existence of an automatic garbage collector: the *destroy* primitive just means that the framework stops delivering messages to the destroyed actor by invalidating its reference. The physical space of the dead actor is not necessarily released.

In our actor framework, we do not make assumptions on actor behavior: an actor is a *blackbox* from the outside. The actor framework can host an actor as long as the actor receives and responds to messages. For example, the following actor behavioral features are not allowed in the classic actor model but compatible to our actor framework if their internal behavior can be described properly:

Multitasking: an actor can handle more than one message at the same time.

Re-entrance: an actor can send a synchronous message to itself.

These features are only pertinent to the actor's internal behavior: they have no implications on the virtual actor framework, which guarantees the delivery of messages, but does not dictate when and how messages are delivered. For example, an actor may be able to handle multiple messages according to its behavior description, but in a framework implementation on a restricted platform which can only handle one message at a time, it turns out the actor behaves as if it were in the classic actor model. Section 4.2 describes a framework implementation for the Web browser using JavaScript and a message loop for message delivery. Most Web browsers support scripting Java classes in an applet through an API such as *LiveConnect* [43]; therefore the framework, written in JavaScript, can host a Java object which employs several threads inside to accept and process multiple messages simultaneously. However, since the browser's scripting environment is single-threaded, when a message is being delivered, the message loop is blocked in waiting for the thread control–the actor capable of multitasking can never receive more than one message at one time.

At the server end, most Web server platforms support multithreading and thus enable a framework implementation which can deliver as many messages as the number of available threads at the same time. Nevertheless the flow control mechanism existing in many network devices and software regulates the number of active HTTP connections from a client: the maximal number of synchronous messages being transmitted across the Internet is limited as well.

In summary, the core of our programming environment, an extended actor model, supports actor creation, message delivery and actor destruction. It does not interpret or execute the actor's behavior. To model an application properly we need a programming language which can define an actor's behavior, support high-level communication protocols on top of the unformatted message body, and register an actor's behavior to the framework with a unique name.

2.2 `ActorScript` Basics

We design `ActorScript`, an actor scripting language, to facilitate the development of Web applications. `ActorScript` borrows the control structure and most of its data types from JavaScript [39, 61], whose near omnipresence on the Web lessen the work in building translators. However, there are several distinctive features differ `ActorScript` from JavaScript:

- *An actor is self-contained*: For an actor, passing parameters in method invocation, throwing out exceptions and returning values all mean exporting part of its internal state, and thus require (deep) copying to ensure the strict data encapsulation requirement. In JavaScript, *functions* are also data but the semantic meaning of cloning a function is unclear. Instead, we introduce a new data type *port* for the reference to a method of a specific actor. A *port* can be exported and be called as a method.

- *An actor can invoke a method asynchronously*: The caller continues its execution without waiting for the return from the callee. If the caller and the callee are on different platforms, or on a multithreaded platform, the invocation of an asynchronous method may result in concurrent execution. The event can be viewed as a form of asynchronous method invocation.

- *An actor is created by a prototype*: The creation prototype defines an actor's initial layout and behavior. We avoid the term *class* because in many object systems, a class is not only a "concept of design" but also an "implementation of design." An implementation is a concrete entity at runtime and thus *class variables* or *static variables* become common knowledge shared between objects of the same class. Moreover, *class* implies *type*. In `ActorScript` actors are *typeless* and can change behavior at runtime.

Note that, because of the strict data encapsulation requirement, objects in other object systems must be modified somewhat for use in our model. Obviously, the reverse is more straightforward: it is simple to implement our model on top of other object systems which do not enforce such an encapsulation discipline.

2.2.1 An Actor Scripting Language

We need a programming language to express an actor's behavior, or strictly speaking, the logic of an actor's response to messages. Theoretically any high-level programming language serves for this purpose because an actors is a blackbox with respect to the rest of the system. However, the description must be executable when an actor is brought into a runtime environment. Since most Web platforms support one or more scripting languages, it is natural to adopt an existing scripting language. A scripting language is a popular programming tool for its simplicity and ease of use [80]. Programmers use a scripting language as *glue* to bind assorted components together into applications. Two features make scripting languages particularly suitable in Web application development: first, scripting languages are simple and thus easy to learn–an important factor for Web developers with less technical training; and second, they are dynamic and therefore they are suitable for incremental application development. Besides, many scripting languages share common features and implementation techniques, such as the use of *associative arrays* with *function references* to support object-oriented programming, this makes it feasible to translate an application from one scripting language to another [17].

However, most existing scripting languages share two critical choices in semantics that make them inadequate for location transparent distributed application development. To be specific they are sequential and implicitly assume a shared memory space. Being sequential means a scripting language must rely on the host system to support interactive applications; this implies some platform dependent code is scattered into an application. A shared memory space makes the it difficult to redistribute an application on distributed platforms: local communication is inherently different from remote communication and the discrepancy cannot be masked.

With our virtual actor framework, the solution is clear: *actorizing* a scripting language. An actor is an isolated scope for a collection of script thus changing an actor's state can be described in a scripting language. For cross-actor communication, an actor uses the primitives provided by the actor framework. We choose *JavaScript* [39] (officially ECMAScript [61]) as the base language of `ActorScript`. The most critical factor for the choice is its near omnipresence: almost all script-enabled browsers support a version of JavaScript and many server scripting platforms also execute JavaScript code in default or with an add-on module. Such omnipresence means less work in implementing our generative application framework (there are tools [44, 90] to translate JavaScript into other programming languages) without loss of universal availability.

`ActorScript` borrows the control structure and most of its data types from JavaScript, including branch, loop, exception mechanism and all the basic data types. Since we use the actor framework to support components, the *object* is demoted to a pure associative array, which we name *structure*. *Structures* are neither created by a *constructor* function nor associated with method invocation; they are just static data repository. The **new** keyword and **(.)** operator are exclusively reserved for *actors*. As a result, to use arrays and structures, the programmer has to use the non-object syntax:

```
// create empty array, cannot use ‘‘new Array();’’ as in JavaScript
var array1 = [];
// create a structure
var structure1 = {a:1, b:2};
// (.) is reserved for actors
var a = structure1["a"];
structure1["b"] = structure1["a"] + 1;
```

2.2.2 Defining Actor Behavior

Two key requirements in supporting actor creation are expressing the actor's behavior and identifying that behavior. For example, in C++ and Java, the *class* declaration defines its instances' behavior and the class name is bound to the definition. In JavaScript, the behavior is described in a constructor function whose name serves as an identification. An `ActorScript` actor's behavior is a state and a set of methods: the state is represented by a set of variables in the scope of the actor (as in JavaScript, `ActorScript` supports local variables in functions). We adopt a behavior declaration syntax similar to Java, for example:

```
prototype counter {
    var n;
    function counter(i) {
        if (i == undefined)
            n = 0;
        else
            n = i;
    }
    function inc() {
        return ++n;
    }
    function dec() {
        return --n;
    }
}
```

The `counter` prototype declaration defines the initial behavior of `counter` actors. There is a variable n storing the current counter value, and a constructor setting up the initial value. A `counter` actor supports two methods: $inc()$ increases the counter value by one and returns the updated value; $dec()$ decreases the value instead.

There are several distinctive differences in `ActorScript`:

- As explained in the beginning, we avoid the term `class`; the keyword `prototype` emphasizes it is a description of initial behavior.
- There are no static variables because a prototype is just a description of behavior: a prototype is an abstract concept and does not have a concrete existence at runtime.
- Currently access control is not supported: every variable can be accessed as a property of the actor, and every function can be invoked as a method of the actor.
- There is no type declaration: similar to JavaScript, `ActorScript` is dynamically typed, and supports no other modifiers in Java.

The **new** keyword is reserved for actor creation. For example, two `counter` actors with the behavior defined above, are created as follows:

```
// initial value is 0 because i == undefined
var c0 = new counter();
// initial value is 10
var c1 = new counter(10);
```

2.2.3 Communication in Actors

An actor can communicate with other actors through:

- Invoking synchronous and asynchronous methods.
- Throwing and catching exceptions.
- Setting and getting actor properties.

The syntax is similar to that of JavaScript. Invoking an actor's method synchronously is equivalent to calling an object's method in JavaScript:

```
// initial value is 5
var c2 = new counter(5);
// v is 6
var v = c2.inc();
// v is 5
v = c2.dec();
```

JavaScript has no counterpart of invoking an actor's method asynchronously in ActorScript. To invoke a method asynchronously, we use the symbol `$` after the method's name:

```
// c2 is increased by 1 after the method finishes
c2.inc$();
// v can be 6 or 7
v = c2.inc();
```

Note an asynchronous method invocation does not return a value to the caller, even if the method definition indicates one. Besides, there is no defined order in invoking multiple methods asynchronously: a synchronous method invocation can be processed earlier or later than a previous asynchronous one, depending on the runtime environment implementation; thus in the above example v can be 6 or 7. The choice of the symbol `$` does not have special significance: we decide to have a postfix symbol for the method name (the rationale is discussed in the next section) and other symbols either cause ambiguity in the grammar or are not allowed in JavaScript.

A major difference between the `ActorScript` semantics and JavaScript is that parameters and return values are *passed by value*. Because each actor has its own address space, a reference in an actor is meaningless in other actors (the actor reference is maintained by the actor framework and thus valid in the whole application). Pass-by-value also implies instances of the *function* data type in `ActorScript` are not exportable.

An actor can invoke its own methods in two different ways: either with the `(.)` operator as a normal cross-actor method invocation or directly through its local functions. For example, we can add a function `inc2()` which double increases the counter by invoking the `inc()` method twice:

```
function inc2() {
    this.inc();
    return this.inc();
}
```

Or the actor can directly call the `inc()` function twice:

```
function inc2() {
    inc();
    return inc();
}
```

In this example the two cases (invoking the method and calling the function) happen to have the same behavior but they can be different in other situations. Parameters and return values in method invocation are always passed by value, even if the caller and the callee are actually the same actor. However, parameters and return values in local function calls are following the JavaScript semantics; those of aggregate data types (array and structure) are passed by reference. Another interesting question is: is the first implementation correct? In the classic actor model, an actor handles only one message at one time: when an actor invokes its own method synchronously, the actor blocks itself and thus a deadlock occurs. `ActorScript` allows such kind of reentrance as one of its distinctive features.

As in JavaScript, a **throw** statement in `ActorScript` can throw anything as an exception; however, an exception can be thrown out of the actor where it is defined. Therefore in `ActorScript`, the **throw** statement copies the exception first and throws the copy instead of the exception evaluated in the expression. If the exception is non-exportable, such as a function, the **throw** statement throws an **undefined** exception: at least the escape control flow can be caught.

An actor variable is a property of the actor. Other actors can *set* and *get* the value in the property through the **(.)** operator. Similar to a method invocation, the operands of these operations are passed by value. The following example illustrates the difference:

```
prototype foo {
    // we can also initialize variables in declaration
    var n = [1, 2, 3];
    var f = new foo();
}


var f = new foo();
// c3 is 2
var c3 = f.n[1];
// n1 = [1, 2, 3] but is a copy of f.n
var n1 = f.n;
// n1 = [3, 2, 3] but f.n = [1, 2, 3]
n1[0] = 3;
// f.n[0] is still 1; it is not changed
c3 = f.n[0]
// set 3 at index 0 of an anonymous array
f.n[0] = 3;
// c3 is 3; f.f evaluates to an actor's reference
c3 = f.f[2]
// f.f.n = 1; it's ok because (f.f).n is a property set operation
f.f.n = 1;
```

It is straightforward to implement these operations using the **send** primitive. The cross-actor exception can be handled similarly: the exception mechanism is a form of synchronous communication (asynchronous error handling is discussed in Section 2.4) in that in addition to normal return, the caller can handle abnormal return with a catch block or upward propagation. The response to the original **send** is tagged with a flag indicating it is an exception instead of a normal return value so that the actor framework can deliver it to the right place.

2.2.4 Actor Termination

Recall that automatic garbage collection is not a built-in feature in the virtual actor framework. Instead, the programmer can *kill* an actor with the **kill** operator:

```
// kill actor1, it returns true if actor1 is valid before the expression
kill actor1;
// kill a dead actor, it returns false but no exception raised
kill actor1;
```

After the **kill** expression, `actor1` no longer exists in the application. Trying to access its properties or invoke its methods causes a runtime error. The **kill** operator is directly built on the **destroy** primitive in the actor framework. Consequently, it only kills the actor in its operand: if the actor contains other actors, they are not killed automatically. Instead, `ActorScript` supports the *finalizer* as in Java:

```
funalize() {
    // clean-up code here ...
}
```

If an actor has a finalizer defined, it must be executed before the **destroy** primitive. A final note: although automatic garbage collection is not a built-in feature, having an automatic garbage collector does not conflict with the built-in manual actor destruction mechanism. In Section 4.3 we demonstrate how both schemes may co-exist in the runtime environment.

2.3 Extra Features in `ActorScript`

There are several distinctive features in `ActorScript` which are not required in the actor model that we use and thus not supported in the virtual actor framework. These features define special semantics of the actors in `ActorScript` and shed light on the implementation. Actors implemented in other languages can interact with those in `ActorScript` transparently but may act differently.

Specifically, `ActorScript` employs the *port* to represent an interface of the actor. Both method invocation and property access can be described in the port communication protocol with different port operations.

As the port unifies the communication interface of the actor in `ActorScript`, it also suggests a unified internal representation. We review the anatomy of the defining *prototype* and map it to the object structure in JavaScript. Concurrency within an actor is intentionally omitted in our actor model; instead we pragmatically define a property of synchronization block in `ActorScript`: at any instant an actor processes one message atomically, except a new synchronous message coming from the actor itself.

2.3.1 Methods and Ports

In many programming languages, *functions* or *methods*, representing a piece of code, can be passed as data. For example, in the C standard library `qsort()` takes a pointer of a comparison function. In JavaScript a function can be assigned to an object's property so that calling the function can be achieved through the object's method. In C++, a member function's pointer can be passed just as a function's pointer can be passed in C while through Java's *reflection* API, the reference of an object's method can be acquired.

As mentioned in Section 2.2, a function cannot be exported out of its defining actor. Nevertheless, it is desirable to have a data type representing a piece of code, such as a function pointer and a method reference. Otherwise, a new language construct is required to support *delegation*. Note that some forms of delegation[1] are crucial for the goal that `ActorScript` should be as close as JavaScript as possible. Without delegation, the way to script standard components of the Web user interface in `ActorScript` could be very different from that in JavaScript.

Note that the forms of delegation used above are not feasible in actors: actors are self-contained and distributed. An internal representation of an actor's method is meaningless to other actors, which may reside in a different address space. On the contrary, in the languages highlighted above, functions or methods are presented uniformly and operable globally in the whole application. Allowing actors in `ActorScript` to share some privileged knowledge, such as the internal representation of code, violates the design principle that an actor is a blackbox to the rest of the system. Consequently, we reach the following conclusions:

- If an actor exports a "reference" of its internal function, the "reference" must include the actor's reference as well so that its issuer can be identified.
- When another actor intends to use the "reference," it must tender the "reference" to the issuing actor for interpretation; it cannot operate on it directly.

The analysis leads us to the concept of *port*. Roughly speaking, a port is a message entry point of an actor which is connected to an internal function. When an actor owns a port of another actor, it can call

[1]We use the term *delegation* as more popularly used today rather than its more rigorous definition in earlier actor languages [67].

the actual function through the port as it calls a local function, except that the arguments and return value are passed by value since the port operation is cross-actor communication.

```
// p is a port of some actor
// call p as if p is a function reference
var r = p();
// store p into a variable
var p1 = p;
// the variable is evaluated to the port
r = p1;
// now r holds a port not a value and thus can be called
r();
```

The port `p` can be acquired through the **(.)** operator. Using the `counter` example:

```
var c2 = new counter();
// c2.n is 0; it is a get operation on the variable n
var n0 = c2.n;
// c2.inc is a port to the function inc()
var p = c2.inc;
// n1 is 1; it is equivalent to c2.inc()
var n1 = p();
// port can also be called asynchronously; it is equivalent to c2.inc$()
p$();
```

Note in JavaScript, the following statements are equivalent:

```
function func() { ... };
var func = function() { ... };
```

Both statements assign a function to the variable `func` and the function can be read from the variable. However in `ActorScript`, when an actor's property contains a function, performing a **get** operation on that property returns a *port* of the function. The syntax of calling a port is similar to invoking a method: the following example shows they are consistent:

```
// invoke inc() of c2
```

```
c2.inc();
// c2.inc is evaluated to a port
// calling the port is equivalent to invoking the method
(c2.inc)();
```

If an actor intends to export an internal function `f()`, for example as a return value, it must use the expression `this.f`, which is evaluated to a port to the function `f()`: `f` yields to a function reference only valid inside the actor and thus it causes an error in exporting. Using the port, delegation in `ActorScript` is almost identical to that in JavaScript, with the exception of the pass-by-value semantics:

```
// create a button
b = new Button("Click me to increase the counter!");
// delegate b.onClick to the inc() of c2
b.onClick = c2.inc;
```

2.3.2 Prototype Revisit

In `ActorScript`, an actor's property can hold a data value as well as a function, and the property can be reassigned freely. Since a port is created by the **(.)** operator when one of its operands, an actor's property, holds a function, it is worth asking what if a port is called after its corresponding property has been reassigned, possibly to a data value?

In `ActorScript`, a data value is viewed as a constant function. Thus in the previous example, if `c2.inc` has been reassigned to a number, invoking `onClick` method returns that number. Precisely speaking, a port is a message entry point connected to an actor's property, not a function, and applying **(.)** on an actor reference and a property name yields a port, no matter what the property holds. The effect of calling a port depends on the current content of the property, not that at its creation. In fact, `ActorScript` actors interact with the actor framework solely through ports. Several operations are defined on the port to achieve different types of actor communication. Currently `ActorScript` supports three operations on the port: **get**, **set** and **apply**.

There are three actor communication types enumerated in Section 2.2: an actor can communicate with others through *method invocation*, *exceptions* and *property access*. Since the exception can be viewed as an abnormal return of a synchronous method invocation, we only consider the other cases in the port protocol, namely method invocation and property access. When a port is at the left-hand side of an assignment (setting an actor's property), it means sending a message to the port using the **set** operation with the value evaluated from the right-hand side expression as the message body. When a port is in an expression and not followed by an argument list (getting an actor's property), it means sending a message to the port using the **get** operation. If a port is followed by an argument list (invoking an actor's method), it means sending a message to the port using the **apply** operation with the argument list as the message body.

On the other side, when an `ActorScript` actor receives a message with the **set** operation, it assigns the message body to the variable indicated by the port. When the message is a **get** operation, the actor has two choices: if the variable indicated by the port holds a data value, it returns a copy of it; if the port holds a function instead, it returns a reference to this port: conceptually the reference is just a *structure* containing `this`, the actor's reference, and the variable name, the property. When the message is an **apply** operation, the actor applies the function held in the variable: note a data value is viewed as a constant function.

The port protocol of `ActorScript` suggests that besides method invocation, property access can be asynchronous since they are both message exchanges as for the protocol is concerned. That is the reason we use a postfix symbol for the method name, actually a port, to invoke the method asynchronously. The symbol `$` stands for a special flag in the message, indicating the use of the **send** primitive in the asynchronous mode. In practice, since the actor framework does not support a mechanism to collect the return value of an asynchronous **send**, performing an asynchronous **get** operation is meaningless. However, we can set an actor's property asynchronously:

```
// set n to 10 asynchronously
c2.n$ = 10;
```

This is especially useful in a high-latency communication link such as the Web, given that no causal dependency is required. A port can be invalidated: in JavaScript an object's property can be removed by the `delete` operator, the same is true in `ActorScript`. Sending a message to a deleted port causes a runtime error.

In `ActorScript`, an actor's interface is characterized by a set of ports. Modern component frameworks such as Enterprise JavaBean [89] and .NET framework [83] tend to use *methods* instead. To access a component's property in these frameworks, corresponding **getter** and **setter** methods must be implemented. We believe the port is a better abstraction because the property is a common concept shared by its **getter** and **setter** methods; the concept is lost in using unrelated method names. In addition, using the port protocol provides better extensibility: new operations can be defined on the port. For example, to disable a port temporarily, that is, to block requests to the port, a **lock** operation can be added.

The port protocol provides another perspective on the actor prototype: conceptually a prototype defines an actor's behavior, that is, the ways it responds to messages; operationally a prototype is a map from ports to actions. Internally the map can be the top-level scope (as in other scripting languages) of an actor; externally it defines the interface of an actor. The port protocol suggests a clear implementation strategy for `ActorScript`.

2.3.3 Concurrency

In the classic actor model, an actor processes one message at one time and thus can be viewed as a *synchronized object* or a *critical section*. In general `ActorScript` actors follow this semantics and concurrency is achieved by asynchronous method invocation. An actor can invoke methods of several actors asynchronously: when the runtime environment can simultaneously deliver messages to multiple actors, these actors work concurrently.

In most cases the semantics provides a simple and convenient method for synchronization; however, it prohibits recursive synchronous method invocation. We have shown that according to the semantics invoking a synchronous method on the caller itself (**this**) immediately results in a deadlock.

However, there are plenty of algorithms described with "synchronous" recursive calls and one of our design goal for `ActorScript` is to make use of existing code with limited modification. Specifically, if an actor only uses synchronous method invocation in its behavior, we expect it could act rather similarly to an object in JavaScript; conversely, an object in JavaScript should be able to be *actorized* to fit our actor framework. To facilitate this, "synchronous" recursive calls must be allowed.

We adopt a pragmatic solution which relaxes the actor semantics a little bit: an active (processing a message) `ActorScript` actor can accept a request of synchronous method invocation if it is from the actor itself (directly or indirectly). The definition is clear and the rule immediately allows synchronous recursive method invocation. The relaxed semantics is simple to understand and has two advantages:

- Although the `ActorScript` actor can be reentrant, there is still at most one thread working on an actor: when an active actor accepts a request of synchronous method invocation, the original thread working for the actor either has suspended (blocked by the previous synchronous method invocation to another actor) or can be switched to serve the incoming one (requested by itself). It does not require a new mechanism to guard the critical section in an actor.

- The relaxed semantics is consistent with the language constructs for synchronization in most threading models. As a synchronization unit an actor can be associated with a *lock* or a similar language construct protecting a critical section, which by definition is reentrant to the thread owning the privilege. Since the original working thread is blocked, it can be reused for the new request of method invocation and keeps the privilege of the critical section.

2.4 Actor Coordination

Our actor model is simple, but it lacks built-in primitives for actor coordination. Many language constructs such as **signal**, **wait**, **semaphore** and **join** are widely available in concurrent systems for thread or process synchronization. Although these language constructs are compatible to the actor model: an actor is an object encapsulating a thread of control, we decide not to include them in `ActorScript`. The decision is based on the fact that we make few assumptions about the runtime environment, and threading is not one of them. In a single-threaded or non-preemptive system, using thread-level synchronization can result in undesirable outcomes such as frequent deadlocks. Instead, we show that auxiliary actors can be defined for actor synchronization and thus these language constructs are not necessary in most cases.

Exception handling mechanism coordinates actors when errors occur. However, the `try/catch` block does not handle exceptions from asynchronous method invocation. We discuss two solutions for asynchronous error handling. The *event* model is extensively used in coordinating asynchronous processes. Asynchronous method invocation in `ActorScript` directly supports a basic event model; however, modern interactive applications usually require a more complex event model. We demonstrate these advanced features can be expressed in `ActorScript`.

2.4.1 Continuation

A problem in asynchronous method invocation is that after the method finishes its task, the continuation disappears: the control flow terminates and the return value is lost if there is one. It is desirable to support first-class continuations so that a continuation can be passed around and reclaimed: in the classic actor model *continuation passing style* is used extensively. Although `ActorScript` does not support first-class continuations, we observe that in most cases two special forms of continuations are sufficient, and that these can be implemented with auxiliary actors.

Consider the following scenario: an actor asynchronously invokes a method which sorts a set of data; when the sorting done, the sender displays the sorted data. However, the actor has no knowledge about when the sort method finishes. In many event-driven systems, the function intended to be called asynchronously such as an event handler has a special parameter holding a *callback function*. When the function finishes, instead of returning the control to its caller, it calls the callback function with the return value. In the data sorting example, the caller can assign the display function as the callback function: when the sorting is done, the sorted data can be displayed in a timely manner.

It is easy to implement the callback function with the *port*. The method intended to be invoked asynchronously can have an extra parameter for the *callback port*. For example:

```
// original method definition
function sort(data) {
    // sorting data
    return data;
}

// new method definition with callback function
function sort(data, callback) {
    // sorting data
    callback$(data);
}
```

Note that the callback function must be called asynchronously. One reason is that the actor should be released as soon as possible: if the callback function is called synchronously, the actor cannot accept another message before the callback function finishes. Second reason for the asynchronous callback function is related to exception handling: the actor has little knowledge about the callback function's logic and thus should not catch the exceptions that the callback function may throw.

Now the caller invokes the sort method with a callback port `view.display`:

```
sortingActor.sort$(orders, view.display);
```

When `sortingActor` finishes sorting, its original return value, the sorted data is forwarded to the port `view.display`: the new method nicely solves the synchronization problem. However, the original `sort` method can be invoked synchronously as well: using an extra parameter for the callback port results in a different version of the `sort` method. It is not sensible to have two methods for the same purpose just because of the two styles of invocation. Taking a closer look, using a callback function creates a *sequential block* containing the asynchronously invoked method and the callback function. The two tasks in the block are executed sequentially because the callback function requires the return value as its parameter, and the block is initiated asynchronously so that it is a new execution path forked from the caller's path. The concept can be generalized to that of a block that can contain any number of tasks and each task consumes the return value of the previous one: these tasks form a *continuation chain*. It is feasible to design an actor to coordinate actors in a continuation chain; the prototype `ChainCoordinator` can be defined as follows:

```
prototype ChainCoordinator {
    function chain() {
        // invoke the first method in the chain and store the return value
        var ret = arguments[0](arguments[1]);
        // invoke the rest methods in the chain iteratively
        // the previous return value becomes the next argument
        for (var i=2; i<arguments["length"]; i++)
            ret = arguments[i](ret);
    }
}
```

Using a `ChainCoordinator` actor, the caller can invokes the original `sort` method asynchronously without losing the return value:

```
var c = new ChainCoordinator();
c.chain$(sorter.sort, orders, view.display);
```

The `ChainCoordinator` actor chains a sequence of continuations, but at the end the continuation is still lost. Another useful form of continuations is the *rendezvous* of multiple continuations. For example, a data analyzer can issue multiple concurrent data crawlers; after all crawlers finish their jobs, the data analyzer starts to analyze the crawled data. We can use an actor to join the concurrent tasks: the prototype `JoinCoordinator`[2] can be defined as follows:

```
prototype JoinCoordinator {
    var counter;
    var chain;
    var cont;
    function JoinCoordinator() {
        counter = arguments["length"] / 2;
        cont = arguments[arguments["length"]] - 1;
        for (var i=0; i<arguments["length"]-1; i+=2) {
            chain = new ChainCoordinator();
            chain.chain$(arguments[i], arguments[i+1], this.finish);
```

[2]We prefer to use the term *join coordinator* rather than *join continuation* used previously in the actor literature [68, 96]. This is because of the familiarity of the favored term to thread programming.

```
        }
    }
    function finish() {
        counter--;
        if (counter == 0)
            cont();
    }
}
```

The basic idea is similar: an auxiliary actor can be designed to manipulate continuations, even though they are not first-class in `ActorScript`. The data analyzer can use a `JoinCoordinator` as follows:

```
new JoinCoordinator((new Crawler()).crawl, site1,
                    (new Crawler()).crawl, site2, this.process)
```

After the two crawlers finish their `crawl` method, the actor can start its processing. Note `JoinCoordinator` is implemented in a more elegant style: since the constructor invokes the `chain` method asynchronously, it returns immediately and thus can serve for task creation and activation. `ChainCoordinator` requires a separate method for asynchronous activation because the actor cannot be created asynchronously by our definition.

2.4.2 Asynchronous Error Handling

The exception is another control flow which is lost in asynchronous method invocation. The `try/catch` block is not designed for asynchronous method invocation: when an exception is thrown, the invoking actor–the exception catcher, may have already left the block, or even ceased its existence. Keeping the control flow in the `try/catch` block until all methods invoked in the block have finished their tasks is not feasible: such synchronization severely limits the potential for concurrency.

Some distributed systems [4, 29] supporting asynchronous method invocation adopt a mechanism similar to the callback function: using an extra parameter to store the *error handler* which is notified when an error occurs. As the call back function, the native form results in two versions of a method, and we can use an auxiliary actor to support the *error handler port*:

```
prototype AsyncExceptionCatch {
    function invoke(port, args, error) {
            try {
                port(args);
            }
            catch (e) {
                error(e);
            }
        }
    }
```

The prototype can be naturally merged with `ChainCoordinator` such that in each stage of the chain, an error port can be assigned to catch exceptions. It does the job, however not in an elegant style: an error port must be manually specified in each method invocation while in a `try/catch` block, the catch clause automatically catches all exceptions in the block.

In asynchronous method invocation, the significance of the uncaught exception is different from that of the lost return value. The return value is lost intentionally because that is expected; otherwise the method would be invoked synchronously. However, an exception means something unexpected happened: neither exiting on nor ignoring all uncaught exceptions is feasible.

From the perspective of the system architecture, all messages are sent by the **send** primitive provided by the actor framework, which therefore can be viewed as the top-level caller. It is reasonable to assume that the top-level caller has a chance to catch the uncaught exceptions. In our generative application framework (see Section 4.1), the *application actor* owns the top-level environment and delivers asynchronous messages on behalf of their actual senders: it is feasible to have certain control at the application level. For this purpose, a special port `onError` is defined in the application actor; the runtime environment forwards all uncaught exceptions, including those from an asynchronous method invocation to that port. The mechanism cannot be simply written in `ActorScript` because it requires runtime support; however, an implementation is straightforward since there must be a top-level environment where all unprocessed errors can be intercepted.

Instead of centralizing all error handlers in a function attached to the application port, we can distribute them following the principle of *modular programming* (as is done using the `try/catch` block for regular exceptions). Such modularity can be achieved by a protocol for asynchronous error handling: giving the callee a chance to forward an exception before throwing it to the application actor.

We reserve a special port `onError`, the same port name as the one defined in the application actor. The pun is intentional on purpose. When the application actor receives an exception from asynchronous method invocation, it checks if the throwing actor has the `onError` port. If the port exists, the exception is forwarded to the port; otherwise the exception is just ignored or the application exits depending on the type of the exception. This mechanism can be implemented at the application level: note the `onError` port of the application actor catches all uncaught exceptions and thus can be wired to a special designed function which manages and dispatches these exceptions. In the implementation, however, the function is not necessarily attached to the physical application actor, which may be remote to the exception throwing actors; instead, each execution platform should implement the function in its top-level runtime environment.

If a caller intends to handle the exceptions from an asynchronous method invocation, it can delegate the `onError` port of the callee actor to the exception handler port (not necessarily its own port) before invoking the method. For example:

```
// set up the error processing port
callee.onError = this.processException;
// when an exception thrown, it is forwarded to processException
callee.doSomethingAsync$();
```

Using the `onError` port has one disadvantage: if the callee actor is shared by multiple actors, its `onError` port may be changed frequently by these actors for their own purposes. The problem is that there is no guarantee that setting up the delegation and invoking the asynchronous method can be done atomically. Between the time a caller actor assigns the `onError` port and then invokes a method asynchronously, it is possible that another caller actor reassigns the port and thus overwrites the delegation for the first actor. When a *race condition* happens, it is possible that the first actor cannot catch its exceptions and the second catches the wrong ones. A *transaction* actor can be implemented to eliminate the race condition, or an actor to manage multiple error handlers so that they do not conflict with each other. However, the complexity and overhead of these auxiliary actors kills the benefit of the `onError` port's simple design. For that reason we believe it is more sensible to leave the decision to the programmer.[3]

2.4.3 Advanced Event System

Modern component frameworks tend to have one *event system* to complement the synchronous method invocation mechanism. An event is sent asynchronously to a target component, which has to implement one or more *event handlers* to process the event, and the sender can continue its work without waiting for the response. In the actor model, asynchronous method invocation is a defining feature and sending an event to an actor can be thought as asynchronously invoking a method of that actor. Changing an event handler is

[3]In other contexts, a different solution may be reasonable. See, for example, the transactor model [36].

as simple as rewiring the port to the desired behavior: in Section 2.3, we showed that a button's **click** event handler can be dynamically set to any port at runtime. The basic event model is also the most popular one used in Web applications.

A criticism of the basic model is that the model mixes the *event* and the *event handler*: sending *an event* does not necessarily mean invoking *an event handler*. Multiple components may be involved with an event: user interface components are usually composed of other user interface components; in this context, both the containing and the contained have the same interest in an event sent to one of them. For example, a *button* may be a visible component of a *user form*: when the user clicks the button, an event is sent to it. However, the action can be interpreted as the user clicks the user form as well–in addition to the event's target, the containing form should have a chance to handle it. Conversely, a component may require multiple handlers in response to an event: the target component sometimes takes different actions on an event according to its state. For example, a common technique to implement the action of *dragging* a visual component is to employ different event handlers on mouse events in different stages of dragging. These features can be included in a specially designed method for each case, such as a button's `onClick` method which notifies the containing actor before or after its processing. However, attaching the port to another event handler which does not follow the protocol voids the mechanism.

In a more complex event system such as the one in *DOM Level 2* and *Java AWT*, the event and the event handler are separate entities so that an event can propagate through multiple components in a pre-defined order while an event handler can be attached and detached dynamically at runtime. Both features can be implemented in `ActorScript` with an auxiliary event library. To support the separation of the event and the event handler, a generic *event actor* or an *event structure* with a static event library (see Section 2.5) can be defined. Using `Button` as an example: a button's `Click` port is attached to an *event actor* instead of an event handler, which is a function connected to another port for invocations. The propagation logic is embedded in the **dispatch** method of the event actor: sending an event is implemented by asynchronously invoking the **dispatch** method, not a specific event handler. The dispatch method then synchronously or asynchronously triggers its containing and contained actors' `Click` event actors, and invokes its real handlers in a pre-defined pattern. The registered handlers for an event are maintained in a structure together with the methods to support dynamic attachment and detachment at runtime.

2.5 Module System

A module system is a mechanism to assist developers structuring and composing programs. Like many other scripting languages, a JavaScript program reuses other pieces of code by loading the script files in plain text or the formats understandable to the specific interpreter into the global scope. In this way, a module is defined by a file containing related code.

A package is a named collection of prototypes, and prototypes in a package must be placed together so that the execution framework can locate them properly. `ActorScript` uses a Java-style packaging system. A prototype declares the package it belongs to using the `package` directive:

```
package Geometry;

prototype Triangle {
// details of the prototype definition
}
```

We distinguish two different reuse patterns: a prototype can declare other prototypes it refers to, and a prototype can be built on top of another one.

2.5.1 Use: Referring to Other Prototypes

When a prototype requires another prototype to work with, it has to indicate its package by the `use` directive. For example, a prototype requiring `Triangle` has to declare:

```
use Geometry;
```

Because actors communicate with others only through message exchange and they are typeless in `ActorScript`, the `use` directive only means the prototype may create an actor based on a prototype in that package. The directive provides information to create the *prototype dependency graph* of an application (see Section 3.5).

2.5.2 Import: Building New Prototypes from Existing Ones

In many object-oriented systems such as C++, Java and .NET, the module system based on the concept of class has an additional benefit: a *class* has a concrete existence at runtime; thus a class's properties and methods can be shared by all its instances. Some concepts not suitable to be modelled in objects are aggregated into classes, for example the `Math` class in Java. These classes are not intended to be instantiated, but are provided as static shared libraries. This feature is not available in `ActorScript`: prototypes are just concepts of design, not concrete implementations. Actors based on a prototype may be dispersed across several address spaces: there may be multiple implementations of the prototype and thus no shared repository available. It is difficult to express concepts such as a `Math` library in actors. A single `Math` actor in an application is not feasible because it implies a concrete `Math` implementation in one physical platform, not all platforms. Therefore an actor requiring the `Math` library needs to contact the specific platform no matter where the actor resides. Instead, a `Math` library should be available in most platforms, but the concept of platform, which implies that of location, does not exist in `ActorScript` for location agnostic development.

Since each actor is considered a standalone entity, it is possible that an actor is the only one in an execution platform. Using a `Math` library in an actor can be viewed as having the library and the actor co-exist in a same address space: the library is part of the actor's behavior. To support this pattern of reuse, `ActorScript` provides another directive called `import`: it *imports* another prototype into a design. For example:

```
prototype Math {
    function sin(t) { ... }
    function cos(t) { ... }
    function tan(t) { ... }
    var PI = 3.14159;
    ...
}

import Math;
prototype Foo {
    function foo(r) {
        // functions and variables in Math are imported
        var h = r * sin(PI/4);
        ...
    }
}
```

An interesting question is whether or not the `sin` port of an actor of `Foo` can be dynamically reassigned. The answer is yes because it is an `ActorScript` actor: no matter how the imported prototype is described and implemented, possibly through a built-in library, the `import` directive merely makes its composites accessible to ActorScript code, that is, wiring them into an actor's variables and thus ports.

There is an extra benefit in using import: as we mentioned, actors do not share information; however, this just means that there is no shared entities at runtime through which several actors can exchange information directly. Actors of a prototype share initial values; for example in the case above, all actors have the same initial value in PI. This gives us a convenient property: actors can share common actor references defined in their prototypes. In a Web application, many actors need the reference of the root actor–the application actor, which can therefore be achieved by preparing a customized prototype implementation for the runtime environment.

2.5.3 Actor Inheritance

There is no built-in inheritance in ActorScript. Developers can build their own inheritance mechanisms through prototype preprocessing. Delegation is an obvious approach: to design a prototype extending from an existing one, we can embed an actor of the extended prototype, and delegate the ports which are intended to inherit from the embedded actor. This approach has a disadvantage: because the super actor and the base actor use different address spaces, possibly remotely separated, the semantics of method invocation is different in the prototype's own methods and the inherited ones.

Using import to bring the extended prototype into a new one is another viable approach. Since the methods defined in the extended prototype are merged into the new prototype, there is no semantic discrepancy in methods and variables. However, because extended methods share the same scope with the extending actor, once an inherited method is overridden, it cannot be accessed anymore–the binding has been lost forever. There is nothing similar to the super keyword in Java, which binds a physically existing super object whose variables and methods are always referable.

We introduce another directive extends to solve the problem, although not in a neat form. The example below uses extends to inherit the counter prototype:

```
prototype double_counter extends counter {
    // n is imported from counter; no need to declare it twice
    function double_counter(i) {
        // call the counter constructor
        super["counter"](i);
    }
    function inc() {
        // call the inc method of counter
```

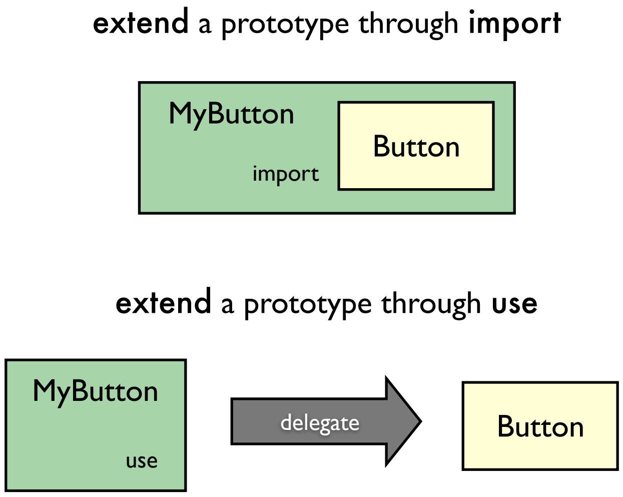

extend a prototype through import
MyButton
import
Button
extend a prototype through use
MyButton
use
delegate
Button

Figure 2.1: Using different directives to extend a prototype.

```
        super["inc"]();
        return super["inc"]();
    }
    function dec() {
        return n-=2;
    }
}
```

The `extends` directive works in a manner similar to that of the `import` directive, but in addition to importing variables and methods into the extending prototype, `extends` declares a structure `super` to store the methods in the extended prototype. Therefore an extended method can be invoked through `super["method_name"]`.

Note special functions and ports such as constructor, finalizer and `onError` are *not* automatically chained along the line of inheritance. Programmers have to write their own code for that purpose.

Chapter 3

Specification System

The specification system enables the programmer to write different specification schemes on an application product line, thus creating customized applications. For example, we use our specification system to manage actor distribution in different execution contexts: if the browser is relatively less secure or less capable, most actors are allocated in the server; if the connection network has limited bandwidth, uncommonly used actors are loaded on demand.

We design `ActorSpec` [13, 14], a specification system for a generalized actor model: it is not limited to the specific model described in this thesis, but all actor models with the defining features. `ActorSpec` includes basic specification rules, a modular specification language and tools that connect specifications to applications. The basic idea of `ActorSpec` is annotating actors with desired attributes.

3.1 Design of `ActorSpec`

The specification system is a mechanism which supports *separation of concerns.* Our target concern is application distribution, which is related to composing runtime actors and code of prototypes. To allow reconfigurable application distribution, the runtime environment must support the concepts of *location* and *movement.* The concept of location can be about *object execution* and *object creation*, while that of movement is a generalization of various timings of component distribution. To enable customizable applications, the specification system needs to bridge these concepts from the runtime environment into applications of the programming environment. In particular, we observe two requirements in designing our specification system:

- How to express concerns in a way that they impact the behavior of actors in the runtime environment? Specifically, *location* and *movement* are abstract concepts; the challenge is to express these concepts in a way that the declarative meaning of a concern corresponds to the desired behavior. This requirement implies that the specification system has to work with the runtime environment closely where these concerns are supported and executable.

- How to connect concerns to the application logic? As we express the concepts in the specification system, how do they bind with an application in a way that enables us to give a precise execution semantics which results in the desired runtime behavior? This requirement suggests the specification system needs to understand the programming environment in detail, which defines the structure of applications.

3.1.1 Assumptions and Design Principles

Two important principles guide the design of *ActorSpec*. First, we want a clean separation between an application logic and its specification so that each can be reused independently of the other. Note that although the programmer may be able to produce a more efficient application by understanding a particular specification system, such hardwiring of the specification would compromise the portability and flexibility of the application. Moreover, there is already a huge code base of Web applications: we would like `ActorSpec` can work with these libraries and applications without requiring significant modification to adapt to different runtime environments. Second, because we do not make assumptions on the actor runtime environment in which the application will be deployed, the specification system does not rely on the specific features in a runtime environment. Explicitly, runtime information which is dynamic or specific to a particular runtime environment, is not available to the specification system.

In summary, we want to decouple the three modules (programming environment, specification system and runtime environment) so that their inter-dependence can be minimized; however the goal contradicts the requirements we just stated. We adopt different strategies in designing our specification system: for connecting to an application logic, `ActorSpec` only uses the most common structures of the application, not its detailed behavior; for expressing concerns, `ActorSpec` defines a *protocol* to communicate with the runtime environment.

3.1.2 Actor Annotations

Observe that many non-functional concerns, including those that we are particularly interested in, can be specified by annotating actors. For example, in order to specify the static execution location of an actor, an attribute **Location** can be defined, and we can allocate an actor to a desired location by annotating it with a *Location Name*. Specifying the dynamic locations is more complicated: dynamic movement may be dependent on runtime information about the dynamic moving patterns. If we constrain the concept of movement in an asymmetric platform configuration such as the Web with a server and a client, it can be reduced to the loading policy. We can design an attribute for the loading policy on actors to control their distribution, collectively the application distribution. If the runtime environment supports multiple programming languages, an attribute **Language** can be defined and each actor will use the language specified by the attribute at runtime.

Taking a closer look, the annotation approach defines a *protocol* between the runtime environment and the specification system. The runtime environment can export an attribute for a sophisticated feature unknown to the specification system, which then can work with the feature through the attribute as one for an intuitive feature. Consider a runtime environment supports actor migration: through annotation the specification system can control an actor's runtime moving policy just by annotating the actor with the moving policy's name. For example, consider a moving policy that dictates: after the actor receives three messages from a remote site, it migrates to that site. The runtime environment assigns the policy a name **Hit3AndRun** and exports it to the specification system, which annotates those actors expected to use the policy with an attribute **Hit3AndRun**. It can take a further step by defining a parameterized attribute **HitAndRun** which takes a positive integer: an actor moves after the number of messages from a remote site reaches the attribute parameter. Thus, without runtime information and understanding of the runtime environment, the specification system can still effectively express an abstract concern through actor annotation.

3.2 Selecting Actors

It is straightforward to design rules for actor annotation: each specification rule has two parts, *identity* and *annotation*, which bind an actor with the desired attribute in a value:

`[actor identity]` : **attribute** = *value* ;

However, since the specification system only takes input from the application logic design, the information it can acquire only contains individual prototype designs and the composition structure of these prototypes. Because the actor is a runtime entity of an application while the prototype designs and their composition structure are static entities, it turns out that identifying a specific actor is not generally possible without the actor's unique runtime identification such as its reference or pointer, which is intentionally not available to the specification system. As a compromise, we relax the rule so that it applies to a set of actors instead of a specific one:

`[actor feature]` : **attribute** = *value* ;

The rule annotates all actors having the designated feature, which then serves as an *actor selector*: that is, the feature selects a set of actors.

3.2.1 Actors from a Concept of Design

Given the source code of an application, we can reason about all information related to its static properties. To achieve the greatest expressiveness, the specification system has to map the runtime properties from the static properties as closely as possible. One of the key properties in an application's static structure is the creation statements in its source code: without runtime information, it is hard to separate the actors created by the same statement. However, we believe this approach is not feasible because the source code of a prototype may not be available in `ActorScript`: we do not rule out other programming languages to be used in our programming environment and a prototype can be merely an adaptor to external components, for example the system library of our runtime environment is just a wrapper of that in the underlying platform, not implemented in `ActorScript`. Besides, a slight change on a prototype design would void the annotations on statements in the source code. More reasonably, what the specification system can expect is that the set of composing prototypes' names which, unlike the source code, represent their concepts in the application design and not the details of their behavior.

We apply a rule to the set of all the actors that are created using a prototype with the prototype's name, an actor feature shared by the actors being selected. This turns out to be reasonable for the applications we have looked at and the implementation strategy is intuitive: using the right implementation of a prototype, which produces the actors with desired properties. The syntax for specifying that all actors created with prototype X have the same *value* of **attribute A** is as follows:

`[prototype X]` : [**attribute A**] = *`value`* ;

For example, suppose the runtime environment exposes an attribute **Location** as described before, a specification rule allocating all `DateValidators` to the client can be written as follows:

`DateValidator` : **Location** = *`Client`* ;

In general, the selector selects a set of actors from a concept of design. In `ActorScript` it is represented by the *prototype*; in an object-oriented language, it can be the *class*. The class-based annotation is widely used in transparent distributed computing (see Section 6.2) with a severe limitation: because the scope of a rule is the whole application, rules at the class level are suitable for large or unique components in an application but not for small ones which are used for different purposes, and thus should have different distribution strategies. For example, the `Button` is a common component in Web applications: we expect buttons of different purposes have different attributes.

3.2.2 Actors of a Partial Genealogy

At first glance the composition structure does not provide more information then composing prototypes' names because a prototype's name uniquely determines its role in the whole composition structure: if a special referring structure can properly identify a set of target actors, the name(s) of their prototype(s) can as well. However with the prototype's name, the referral links do provide extra resolution to distinguish the runtime actors of that prototype in a way that these links have some implication on runtime features. If a prototype is referred by two or more prototypes, the actors of this prototype can be classified to by their creators' prototypes. This motivates the second rule for our specification system. In order to specify that the actors of Y created by an actor of X share the same *value* of **attribute A**, we write:

`prototype X > prototype Y :` **attribute A** `=` *`value`* `;`

For example, we can allocate different `Button` actors to different locations, based on their creators' prototypes:

`OrderForm > Button :` **Location** `=` *`Server`* `;`

`InventoryForm > Button :` **Location** `=` *`Client`* `;`

Note that the idea of this specification has implications for the reuse pattern of "has-a" relationship [48] which says that an object is composed of other objects, and the former (called a *container* object) is a natural context of the contained (composing) objects. However, the relation is not always unique and static: an actor can be contained in more than one actor and its ownership may be transferred at runtime. On the other hand, the creator of an actor is unique and non-transferable. Our observation suggests that, in practice, the primary owner of an object is usually its creator. This justifies the use of creatorship to approximate ownership for the purpose of actor selection.

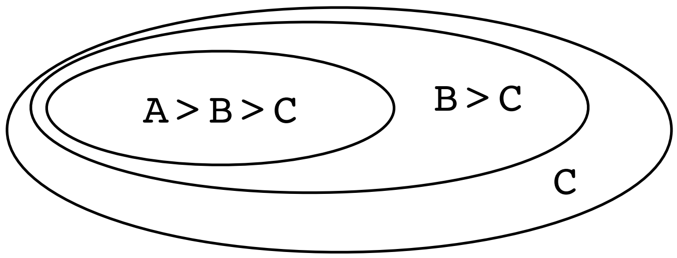


Figure 3.1: The genealogy ordering is linear.

A generalization of the rule form based on direct creatorship is to specify an actor by its partial *genealogy*–extending its creatorship to more generations. Note that the full genealogy is determined at the time of creation and remains invariant permanently for an actor. For example, the following rule says the attribute of the actors of prototype X_i is decided by examining its genealogy up to i generations.

`[prototype X`$_0$`] > ... > [prototype X`$_i$`] :` **[attribute A]** `=` *`value`* `;`

Obviously, allowing rules associated with more than one generation can lead to conflicting rules on an attribute. The actors selected by the selector $X > Y$ are by definition instances of Y: both of the two rules are applicable to the target actors. We adopt the principle that a more specific rule overrides a less specific one. This resolves all the conflicts because the genealogy ordering is *linear*: if there are two rules applicable to an actor, one of the rule selectors must decide a subset of that by the other (see Figure 3.1) and thus be more specific.

Extending direct creatorship to genealogy is useful because some actors have similar composition structures; to distinguish the actors within these structures, examining the creatorship for more generations is required. For example, a `ListItem` is always created by a `ListControl` in our framework library: using `ListControl` only is not enough to differentiate the contexts of different `ListItem` actors.

3.3 Prototypes in `ActorSpec`

Besides actors, prototypes are components in our programming environment. Strictly speaking, an application is a composition of actors at runtime but statically an application executable is a composition of prototype implementations. When we contemplate component distribution, prototypes should be taken into consideration: in addition to runtime actors, it is crucial to control the loading policy of application static code, which is composed of prototype implementations. In this section, we demonstrate specification rules can be used to specify prototype requirements as well, and sort out some ramifications on issues of reusing prototypes in specification.

3.3.1 Implementation of Prototype

Annotating a set of actors of a prototype with a specific attribute implies that these actors use a specialized implementation of that prototype which produces the actors obeying the given specification. Note in our virtual programming environment, a prototype is a *concept of design* instead of an *implementation of design*. A prototype in the programming environment may be realized in more than one prototype implementation in the runtime environment. Consider the example in Section 3.2. The `Buttons` created by an `OrderForm` will be deployed in the server and those created by an `InventoryForm` in the client: there are two implementations of the `Button` prototype in the runtime environment, and each of the selectors in these rules not only selects the actors it intends to specify but also the specific prototype implementation of these actors. From this prospective, a specification rule annotates a prototype implementation as well as a set of runtime actors. For example, the runtime environment may expose an attribute controlling the loading policy of prototypes **PrototypeLoad** with two possible values *PreLoad* and *OnDemand* as follows:

```
SubMenu : PrototypeLoad = OnDemand;
PricePanel > GridControl : PrototypeLoad = PreLoad;
CalcPanel > GridControl : PrototypeLoad = OnDemand;
```

The specification rules say the code (prototype implementations) of `SubMenu` and that of `GridControl` in a `CalcPanel` should be loaded *on demand* while the code of `GridControl` in a `PricePanel` should be *preloaded*. These specification rules are intended to apply to the prototype implementations, not to the runtime actors created by the prototype implementations.

3.3.2 Reuse Patterns

In an object-oriented system, an instance of a subclass is also an instance of the superclass and thus owns all the properties defined in the superclass. It is natural to ask whether a rule based on an extended prototype applies to all its *extending* prototypes in `ActorScript`. For example, when we extend the `Button` prototype

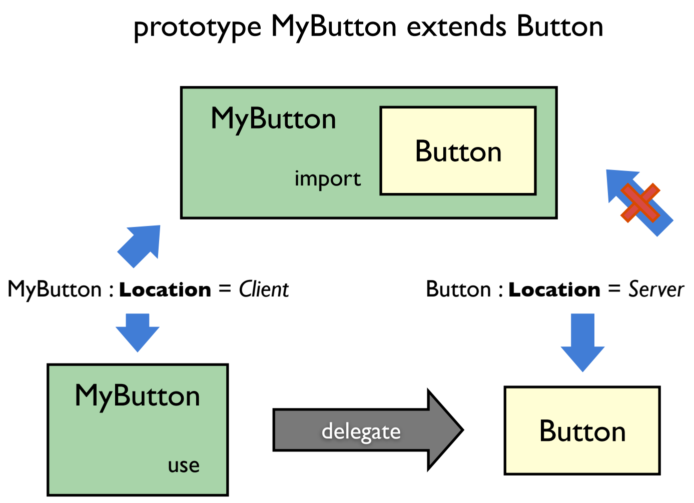

prototype MyButton extends Button
MyButton
Button
import
MyButton : Location = Client
Button : Location = Server
MyButton
use
delegate
Button

Figure 3.2: Specification rules only apply to physically existing runtime actors.

to build a new prototype `MyButton`, does a rule specifying `Button` also apply to the actors created by `MyButton`?

The short answer is no. Because *actors are typeless*, there is no type hierarchy in `ActorScript`. As we described in Section 2.5, the extending prototype uses the extended prototype to describe the design through the `import` directive. The usage is invisible in the virtual programming environment: other actors do not have the information about how a prototype is constructed and extended.

To explain in detail, the specification system only specifies actors physically existing in the runtime environment. The inability in specifying a rule to the extending actors is merely a result of our choice in using the `import` directive to build the inheritance mechanism on prototypes; in this case there is no extended actors physically existing in the runtime environment. If we had adopted *delegation* instead, that is, created an extended actor on behalf of the extending one and forwarded all messages to it, a specification rule on the extended prototype should have applied to a "part" of the actor: namely, to the extended actor (Figure 3.2).

Although actors are typeless, it still desirable to exploit the design patterns of extending in actors, which are design features and thus should have certain implication in actor deployment. Originally we did include a rule form to utilize the pattern of extending but we dropped this rule form after further consideration. In a programming model which supports multiple inheritance, such kinds of rules result in potential conflicts and establishing rule precedence severely complicates the specification schemes. Even without the existence of multiple inheritance, rules based on extending patterns can easily conflict with rules on genealogies. For

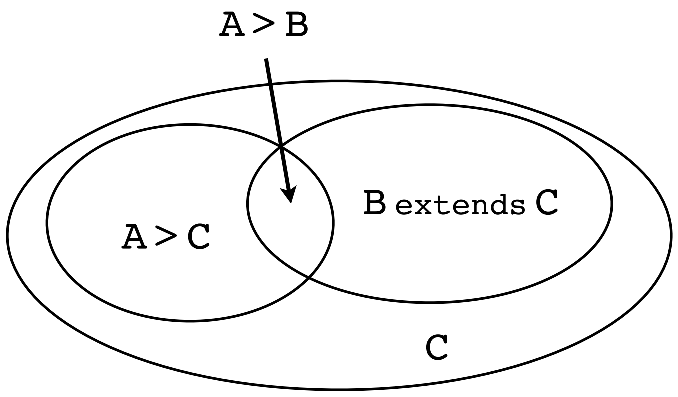


Figure 3.3: Using rules of extending relationship and creatorship may result in conflicts in specification.

example in Figure 3.3, `B` extends `C` and two rules are defined with the selectors `A > C` and `B`. Here we see the actors in the set of `A > B`, which is the intersection of `A > C` and `B`, have two applicable specification rules: it is hard to determine which one is more specific and thus should apply to them.

3.4 Specification Language

It turns out that using the specification rules described above directly is verbose and error-prone for human developers. Two attribute annotations may be combined in a rule if the target genealogy (the selector) is the same, but two genealogies must be written in two rules even when they differ in only one generation. Moreover, not all attributes are available for a prototype and some attribute values are in conflict with others. For example, apparently the attribute **PrototypeLoad** is only applicable to a client implementation.

It is also difficult to manage and reuse individual specification rules. In a specification scheme containing a large number of complex rules, there is a greater chance that there are common building blocks that are reusable. It is desirable to construct specification schemes modularly.

3.4.1 Moving to XML

We use XML (eXtensible Markup Language) [25] to organize specification rules: an XML element represents a prototype and multiple rules can be expressed in a tree structure. For example, the XML fragment and rules in Figure 3.4 are equivalent.

Using XML helps the developer to structure specification rules. Although it is legal to have a rule of a genealogy starting with a sub-component such as a `SubMenuItem`, such kind of rules make little sense in

```
<OrderForm Location="Client">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <TaxCalculator Location="Client"/>
    <AddressValidator Location="Server"/>
</OrderForm>


OrderForm : Location =  Client;
OrderForm > ListControl : Location = Client;
OrderForm > ListControl > ListItem : Location = Client;
OrderForm > ListControl > ListItem : PrototypeLoad = OnDemand;
OrderForm > TaxCalculator : Location = Client;
OrderForm > AddressValidator : Location = Server;
```

Figure 3.4: Using XML to represent specification rules.

practice. Instead, a set of specifications usually start from a major functional component which contains several components, such as an `OrderForm` in our example. Another advantage of using XML is the existence of XML schema validation tools which can check validity and consistency of the specification rules. Note that adopting XML does not sacrifice expressiveness: any specification rule can be expressed in one XML fragment where every node has at most one child. For example the following rule:

$\texttt{Prototype}_0$ > $\texttt{Prototype}_1$ > ... > $\texttt{Prototype}_i$: **attribute** = *value* ;

can be written in

$\texttt{<Prototype}_0\texttt{>}$

    $\texttt{<Prototype}_1\texttt{>}$

        ...

        $\texttt{<Prototype}_i$ `attribute="value"/>`

        ...

    $\texttt{</Prototype}_1\texttt{>}$

$\texttt{</Prototype}_0\texttt{>}$

```
<OrderForm Location="Client">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <TaxCalculator Location="Client"/>
    <AddressValidator Location="Server"/>
</OrderForm>

<ProfileForm Location="Client">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <PhoneValidator Location="Client"/>
    <AddressValidator Location="Server"/>
</ProfileForm>
```

Figure 3.5: A specification scheme may contain common building blocks.

3.4.2 More Modulization

Consider a specification scheme in Figure 3.5. The specifications on `OrderForm` and `ProfileForm` have a common building block highlighted in the grey areas. The observation immediately leads us to a shorthand representation in Figure 3.6. The specification scheme defines an XML `Block` element for common blocks with an attribute *name*, which then can be used to refer a building block in specification. The idea behind this is to make use of XML's tree structure: a node can readily refer to a set of subtrees in a modular representation.

However, using an element to represent a fixed set of subtrees is not as useful as it seems to be. If there is no other rule in Figure 3.5, it is not necessary to define a `Block` for `ListControl` and `AddressValidator` because `OrderForm` and `ProfileForm` have the same specification on these prototypes; top-level specifications on `ListControl` and `AddressValidator` are sufficient: `OrderForm` and `ProfileForm` will follow. It turns out that the `Block` tag only helps *in* different specification schemes, such as those for different execution contexts, but not *within* a specification scheme. For this purpose a reusable block must be parameterized: it does not represent a set of rules (with fixed attribute values), but a group of *selectors*; the actual attribute values assigned by these selectors, the *configuration of the block*, can be controlled through a parameter.

```
<Block name="block1">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <AddressValidator Location="Server"/>
</Block>


<OrderForm Location="Client">
    <TaxCalculator Location="Client"/>
    <block1/>
</OrderForm>


<ProfileForm Location="Client">
    <PhoneValidator Location="Client"/>
    <block1/>
</ProfileForm>
```

Figure 3.6: A common block can be defined by a `Block` element.

We introduce another tag `Configuration` for different configurations in a `Block`. Each `Configuration` element in a `Block` must define the *name* attribute and contains an XML fragment representing the configuration. To reuse a block, we can set the *configuration* attribute to choose the desired configuration in the block. Figure 3.8 shows the expanded specification from Figure 3.7. Blocks and configurations can be constructed recursively: a `Configuration` element can contain other blocks. In addition, a `Block` element can contain elements other than `Configuration`: these elements will be included in the block replacement no matter which configuration is selected. (See the `AddressValidator` specification rule in `block1`.)

3.5 Resolving Specification Rules

It is straightforward to expand a specification scheme in the XML language described into specification rules: `ActorSpec` parses the scheme into a tree structure, and then replaces all elements whose tag names are not in the prototype list with the corresponding block definitions. Given an application and specification rules, the runtime environment by our definition is able to execute the application obeying the rules.

However, the runtime environment exposes *control attributes* of actors and prototypes: it does not expect

```
<Block name="block1">
      <AddressValidator Location="Server"/>
      <Configuration name="conf1">
             <ListControl Location="Client">
                   <ListItem Location="Client" PrototypeLoad="OnDemand"/>
             </ListControl>
      </Configuration>
      <Configuration name="conf2">
             <ListControl Location="Client">
                   <ListItem Location="Client" PrototypeLoad="PreLoad"/>
             </ListControl>
      </Configuration>
</Block1>

<OrderForm Location="Client">
      <TaxCalculator Location="Client"/>
      <block1 configuration="conf1"/>
</OrderForm>

<ProfileForm Location="Client">
      <PhoneValidator Location="Client"/>
      <block1 configuration="conf1"/>
</ProfileForm>

<FriendsList Location="Client">
      <EmailValidator Location="Client"/>
      <block1 configuration="conf2"/>
</FriendsList>
```

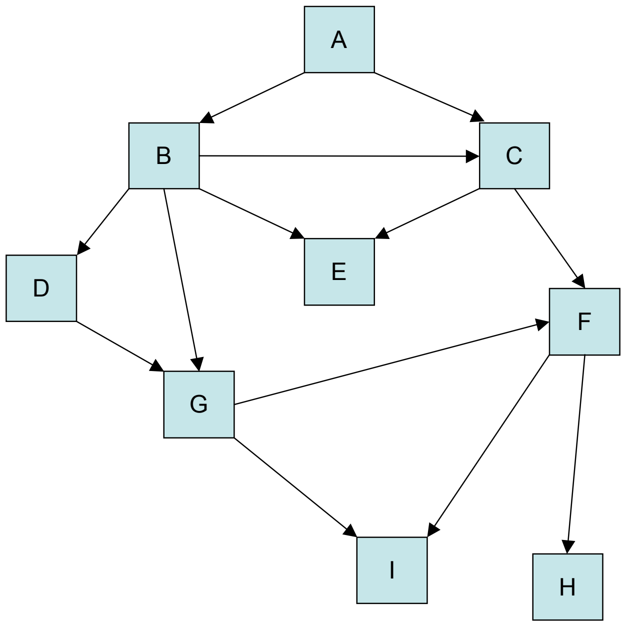
Figure 3.7: A block can define several configurations.

```
<OrderForm Location="Client">
      <TaxCalculator Location="Client"/>
      <AddressValidator Location="Server"/>
      <ListControl Location="Client">
             <ListItem Location="Client" PrototypeLoad="OnDemand"/>
      </ListControl>
</OrderForm>

<ProfileForm Location="Client">
      <PhoneValidator Location="Client"/>
      <AddressValidator Location="Server"/>
      <ListControl Location="Client">
             <ListItem Location="Client" PrototypeLoad="OnDemand"/>
      </ListControl>
</ProfileForm>

<FriendsList Location="Client">
      <EmailValidator Location="Client"/>
      <AddressValidator Location="Server"/>
      <ListControl Location="Client">
             <ListItem Location="Client" PrototypeLoad="PreLoad"/>
      </ListControl>
</FriendsList>
```

Figure 3.8: The expanded specification of Figure 3.7.



```
1. A : Attr = Blk;
2. B : Attr = Wht;
3. C : Attr = Blk;
4. D : Attr = Wht;
5. E : Attr = Wht;
6. F : Attr = Blk;
7. G : Attr = Wht;
8. H : Attr = Blk;
9. I : Attr = Blk;
10. C > E : Attr = Blk;
11. D > G > I : Attr = Wht;
```

Figure 3.9: A prototype dependency graph and specification rules.

the input of specification rules. Instead, the runtime system anticipates to have a composition of actors (strictly speaking, actor prototypes) *annotated with attributes* as a customized executable application. To fill the gap, we design methods to resolve specification rules and accordingly transform an application in the programming environment into a customized one in the runtime environment.

3.5.1 Problem Description

Statically, an application in our virtual programming environment is a composition of actor prototypes; each prototype may depend on other prototypes and itself (one prototype can have multiple implementations in the runtime environment so we need to preserve self-dependency information). Because it is not necessary to know the interface to invoke a method of an actor (actors are *typeless* and method invocation is realized through message exchange), that prototype A depends on prototype B just means an actor of A may create an actor of B. We can define *Prototype Dependency Graphs* as follows:

Definition 1. ***A Prototype Dependency Graph*** *is a directed graph where a vertex represents a prototype and an arc denotes the dependency of two prototypes.*

A prototype dependency graph can be derived by inspecting the composition structure of the application. Figure 3.9 depicts a prototype dependency graph. Note that the graph may contain cycles and

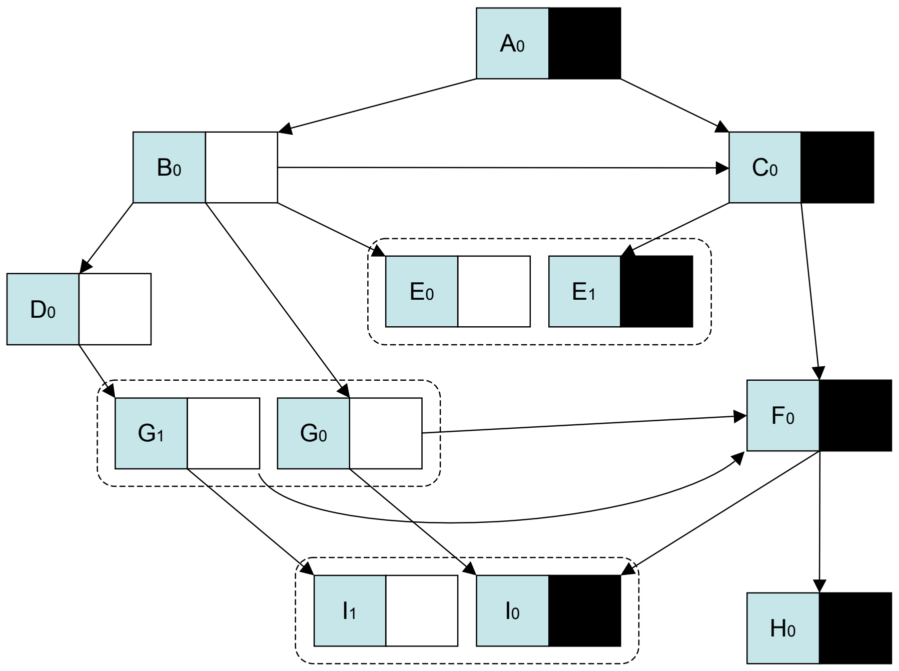


Figure 3.10: The implementation linkage graph is derived from the prototype dependency graph and specification rules in Figure 3.9.

self-loops. Obviously the prototype dependency graph is unique for a given application. On the other hand, an executable application image is a composition of prototype implementations which can vary in different execution contexts and runtime environments. Similarly we define *Implementation Linkage Graphs* to capture the linkage structure of prototype implementations.

Definition 2. ***An Implementation Linkage Graph*** *is a directed graph where a vertex represents a prototype implementation and an arc denotes the linkage relation of two prototype implementations. A vertex also includes one or more attributes which characterize the implementation.*

An implementation linkage graph contains information about a specific version of an application. One application may have several implementation linkage graphs with different requirements. In our case, the requirement is written in specification rules expanded from a specification scheme in the XML language. Customization is achieved by transforming the composition of *concepts of design* into a composition of *implementations of design*. Figure 3.10 shows the implementation linkage graph derived from the prototype dependency graph and specification rules in Figure 3.9. From the graph we can see prototype E, G and I have two different implementations: the implementation linkage graph may not have the same topology of the prototype dependency graph.

With the implementation linkage graph, our generative application framework, that is, the runtime environment, can generate or find the desired implementations and link them correctly; the customized application will obey all the specification rules. Thus the problem of resolving specification rules can be described as follows: given a prototype dependency graph with a set of specification rules, how can we generate an implementation linkage graph consistent with these rules?

3.5.2 Path-Splitting Algorithm

We develop an algorithm for the transformation. First we make the following assumptions to simplify the algorithm description:

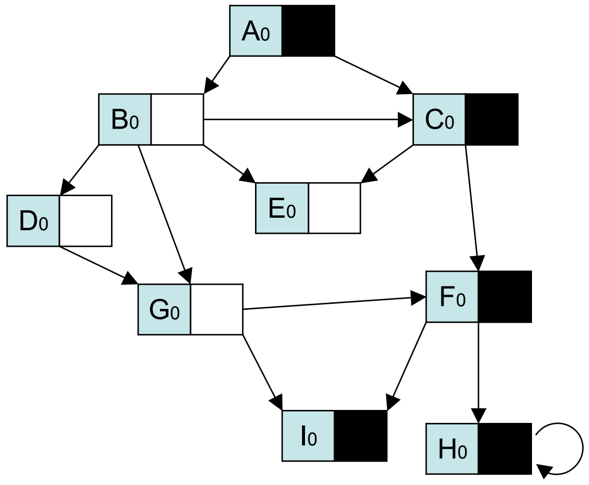
1. There is only one attribute **Attr** in the specification rules. Multiple attributes can be combined into a new one. For example, if there are two attributes $\mathbf{Attr}_A$ and $\mathbf{Attr}_B$ with possible value sets $\{a_1, a_2\}$ and $\{b_1, b_2\}$, we can design a new attribute $\mathbf{Attr}_{AB}$ with the possible value set $\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}$.

2. For each prototype X, we have a rule `X:`**`Attr`**` = `*`Value`*`;`, that is, there is a default implementation of each prototype. Since each prototype must have an implementation, the assumption is valid. The most compact form of a rule can be easily computed, or if the specification scheme is complete, we can randomly assign the default implementation because there must be a more specific one overriding it.

3. There are no duplicate prototypes in any specification rule's selector. Because a selector corresponds to a *path* in the prototype dependency graph, it means that the path contains no *cycles* and *self-loops*. We have not found any serious application requiring duplicate prototypes in a specification rule, but since our algorithm does work in their existence, we do not rule out the possibility. The assumption is just for convenience of the algorithm description and proof.

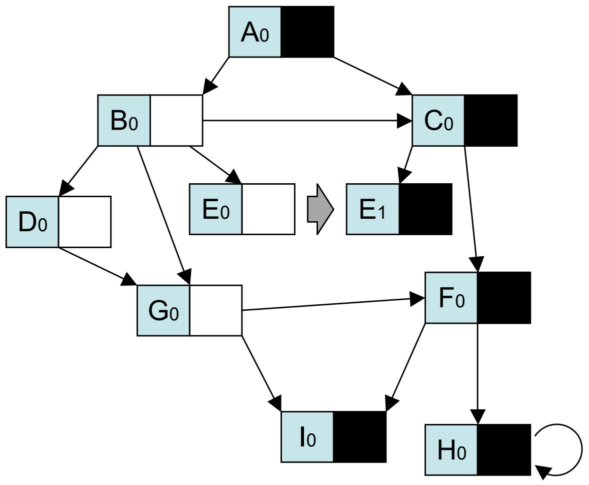
We use Figure 3.9 and Figure 3.10 as an example of the transformation: the process is shown as follows: **Rule** $1-9$ are default implementations, so we build the initial implementation linkage graph by marking each vertex of the prototype dependency graph with *Implementation 0* and its specified value (see Figure 3.11). With **Rule** 10, the specified value of E from C conflicts with E_0, so we create a new implementation vertex of E, namely E_1, with the correct value. Then we have to remove the arc from C_0 to E_0 and add an arc from C_0 to E_1 because the rule specifies the path from C to E (see Figure 3.12). Adding **Rule** 11 is more sophisticated: in addition to I, G requires a new implementation G_1 although the rule does not specify G (note the new vertex has the same value as the original one). Without G_1, G_0 would have a non-deterministic choice of I_0 and I_1, and thus the information given by the rule would be lost (see Figure 3.13).

From the example, we can generalize the algorithm in Algorithm 1.



```
1. A : Attr = Blk;
2. B : Attr = Wht;
3. C : Attr = Blk;
4. D : Attr = Wht;
5. E : Attr = Wht;
6. F : Attr = Blk;
7. G : Attr = Wht;
8. H : Attr = Blk;
9. I : Attr = Blk;
```

Figure 3.11: The implementation linkage graph after default implementations have been marked.

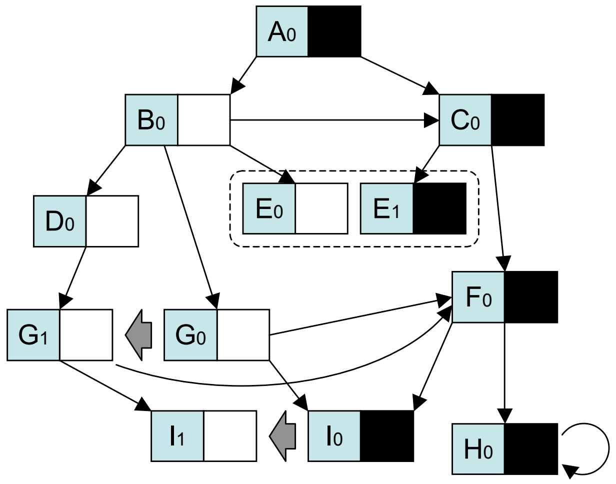


```
10. C > E : Attr = Blk;
```

Figure 3.12: The implementation linkage graph after **Rule** 10 has been added.

Algorithm 1 Generate Implementation Linkage Graph

```
create the initial graph with all single prototype rules $R = (p, value)$
for all genealogy rules R in an ascending order of $length(R)$ do
    add $R = ((p_0, p_1, \ldots, p_n), value)$
end for
```



```
11. D > G > I : Attr = Wht;
```

Figure 3.13: The final implementation linkage graph.

Note that we separate the rules into those with a single prototype $(p, value)$ as selector and those with a genealogy $((p_0, p_1, \ldots, p_n), value)$. We define the length of a rule as the number of prototypes in the rule. Adding the rules in an ascending order of $length(R)$ means the more specific rules are added later and thus override the less specific ones. To add a genealogy rule R, we have a subroutine shown in Algorithm 2. Note in line 8, when the arc (v_0, v_1) is removed, it is possible that no implementation vertices have an arc to v_1. It means the rule is overspecified but that does not result in an error: since the implementation v_1 is no longer reachable from the root (the application itself), it is not part of the final graph.

Algorithm 2 Add a genealogy rule $R = ((p_0, p_1, \ldots, p_n), value)$

```
1: for all path $(v_0, v_1, \ldots, v_n)$ such that $v_i$ is an implementation of $p_i$ do
2:   for $i = 1$ to $n$ do
3:     generate a new implementation $v'_i$ of $p_i$
4:     copy $v_i$'s value and arcs to $v'_i$ except the arc $(v_i, v_{i+1})$
5:     add the arc $(v'_{i-1}, v'_i)$
6:   end for
7:   set the value of $v'_n$ to $value$
8:   remove the arc $(v_0, v_1)$
9: end for
```

3.5.3 Proof and Analysis

For an application composed of n prototypes and m specification rules, it is obvious to see the algorithm finishes in $O(n + m \log m + m \cdot max(length(R)))$. Since $m \geq n$ and $max(length(R))$ is a constant in practice, the time complexity is $O(m \log m)$. To prove the correctness of the algorithm, we need to show:

1. The implementation linkage graph G' is a valid implementation of the prototype dependency graph G.

2. It correctly follows the specification rules R.

For the first claim, we have to prove that for each path $p = (v_0, v_1, \ldots, v_n)$ in G starting from the root (the application), there exists a unique path $p' = (v'_0, v'_1, \ldots, v'_n)$ in G' where for all $i \in [0, n]$, v'_i is an implementation of v_i. It means if an actor can be created under a genealogy in the design of G, it can be created under the same genealogy with corresponding implementations in G'. It can be proved by mathematical induction on the number of rules added: the initial case is trivial since the two graphs are identical in topology. We can show that the claim holds after a new rule $R = ((p_0, p_1, \ldots, p_n), \mathit{value})$ has been added:

If the path p does not contain a segment corresponding to $(p_0, p_1, \ldots, p_i)$ for any $i \in [1, n]$, adding the rule does not change p'. If p does contain, we can write:

$$p = (v_0, v_1, \ldots, v_j, p_0, p_1, \ldots, p_i, v_{j+i+2}, \ldots, v_n) \tag{3.1}$$

$$p' = (v'_0, v'_1, \ldots, v'_j, v'_{j+1}, v'_{j+2} \ldots, v'_{j+i+1}, v'_{j+i+2}, \ldots, v'_n) \tag{3.2}$$

Algorithm 2 creates correct implementation v''_k for p_k for all $k \in [1, n]$, and connects v'_{j+1} to v''_1 and v''_i to v'_{j+i+2}. Therefore,

$$p'' = (v'_0, v'_1, \ldots, v'_j, v'_{j+1}, v''_1 \ldots, v''_i, v'_{j+i+2}, \ldots, v'_n) \tag{3.3}$$

is the new path corresponding to p after the new rule R added. It is unique: because the algorithm does not change the out-degree of any vertex, there is no non-determinism.

Note that the converse direction of this claim is also true but it is of no significance: if there were no p for a p', the creation path p' would have never happened in the runtime trace.

For the second claim, we have to show that for each path $p' = (v'_0, v'_1, \ldots, v'_n)$ in G', the *value* of v'_n is the value specified by the most specific rule applicable to v_n in G. Similarly we use mathematical induction on the number of rules added to construct the proof. The initial case is trivial because all vertices follow the default implementations. Assume we already have each path p' satisfying the claim in G' and then apply Algorithm 2 on a rule to get p'' in G''. If the path p does not end with the selector of the rule, that is, it is irrelevant to the rule, according to the previous proof p'' ends with v'_n: thus the value of the attribute does not change. If the path p does end with $(p_0, p_1, \ldots, p_n)$, then p'' ends with the new vertex created by Algorithm 2, which has the correct value specified by the rule. Therefore, the *value* of v'_n is specified by the latest added rule applicable to p. Since the algorithm adds rules from the less specific ones to the more, the proof is complete.

Chapter 4

Generative Application Framework

In this chapter, we describe the process of generating and executing a customized Web application according to an actor distribution specification. Such customization creates two requirements for the runtime environment. First, as might be expected, the runtime environment must expose some parameters–**attributes** in the specification language–which can be used to specify reconfigurable actor distribution. Second, the runtime environment must provide a mechanism to execute the components of the Web application following the specification attributes.

4.1 Architecture Overview

In this section, we describe `ActorWeb`, a generative application framework for Web applications. `ActorWeb` includes an actor runtime environment which accommodates specialized actors and exposes control attributes for actor distribution, and a set of prototype implementation generators which generate customized prototype implementations for the target platforms.

4.1.1 An Actor Runtime Environment on the Web

An actor runtime environment enables actor creation, delivers messages and empowers actors to respond to messages. Based on our past experience [16], we design a virtual framework for Web applications. The framework is virtual in two senses: first, the framework provides a *virtualization* over heterogeneous distributed platforms; and second, instead of a native environment for actors written in `ActorScript`, the framework is composed of a set of libraries, runtime systems (such as a garbage collector), and generators which accommodate `ActorScript` actors on the host platforms.

A *Podium* is an abstraction of a physical execution platform: it is built on top of a host platform, and serves as an actor virtual machine. An actor runtime environment includes one or more *Podiums.* In the context of Web applications, we need two kinds of podiums:

Client Podium: The client podium provides an abstraction of the browser. The client podium is im-

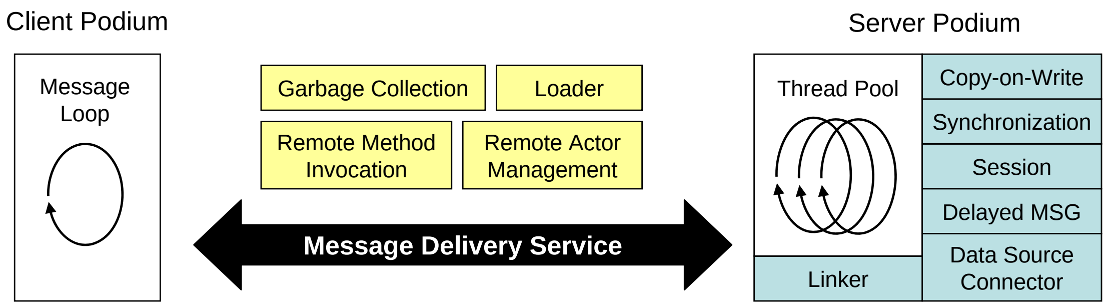


Figure 4.1: Podiums and runtime services in the application framework

plemented as a set of JavaScript libraries, which are downloaded in the browser as the application initializes. Because scripting environments in current browsers are single-threaded, the basic structure of the client podium is a message loop. Delivering an asynchronous message to a local actor is realized by placing a request in a message queue; when the client podium becomes idle, it picks a pending message from the queue and dispatches it.

Server Podium: The server podium is a more powerful virtual machine which can service multiple asynchronous messages concurrently. For this reason, support for synchronization is essential in the server podium. Session control is crucial in maintaining application-wide resources and keeping track of stateless HTTP requests. In the context of Web applications, the server has another role in application deployment: supplying actors, prototypes and static resources to the client podium. The *Linker* and *Loader* are designed for this purpose. The framework exposes a specification interface to control these modules for reconfiguration of actor distribution. The server podium is implemented with *Rhino* [44], which executes JavaScript in Java, embedded in the *Apache Tomcat* [46] Java application server.

The architecture of the actor runtime environment is depicted in Figure 4.1. The *Message Delivery Service* supports both synchronous and asynchronous message delivery between podiums. In most cases a piggyback technique described in [16] can handle the asymmetry of the HTTP protocol. Recently some developers have advocated using persistent HTTP connections for streaming services: the technique neatly solves the bi-directional communication problem but at the cost of sacrificing of server scalability (sockets cannot be recycled in a session). The *Remote Actor Management* supports transparent remote actors. When an actor's reference is exported, the runtime service creates a proxy actor acting on behalf of the actor in the destination podium. *Remote Method Invocation* implements message serialization and deserialization and handles remote exception propagation as well.

The *Garbage Collection* service currently relies on the local garbage collectors provided by the host platforms, which cannot handle cross-podium cycles and obsolete proxy actors. A variant *mark-and-sweep* algorithm is employed to break these links. Following the message exchange semantics literally turns out to be inefficient in the setting of distributed environment: we implement two runtime services which exploit locality for performance improvement.

4.1.2 Attributes for Actor Distribution

To control actor distribution, we consider the following questions:

1. Which podium hosts the actor?
2. Which podium creates the actor? Note that an actor may be created in one podium and deployed in another.
3. When does the actor move? For example, an actor may migrate immediately after its creation, or it may migrate on demand.
4. When is the prototype implementation loaded? An implementation may be loaded with another prototype implementation linking to it, or it may be loaded on demand.

While the second and third question suggest the possibility of *actor migration*, currently our framework implementation does not support actor migration completely. The decision is based on the fact that such migration requires complex runtime support and the cost cannot be justified in *lightweight* actors. However, we do consider a more limited form of mobility: an actor can be created in one podium where it is more efficient to access the resources needed for the creation, and then the actor can be sent to another podium where it has more local interaction. For example, although an information panel should be deployed in the client because the user can interact with it locally, the server with direct access to the relevant data source is a better place to create the personalized information panel.

Asymmetric data movement exists in the Web applications and platforms: the user has direct contact with the client while the data source has optimized connection to the server. In most cases the amount of the user input is not on the same scale as that of the needed data source and the cost of loading a mobile actor into the client podium is much greater than that into the server podium. Therefore, `ActorWeb` only supports one-way mobility: a mobile actor is created in the server podium and then migrates to the client podium, but not vice versa. We observe two different timing strategies for the loading behavior of a mobile actor to the client podium: an actor may be deployed to the client podium as soon as a client actor has its reference, or the actor may wait in the server podium until it is actually needed by another actor. The two loading strategies–eager deployment and lazy deployment–also apply to client prototype implementations in the obvious way. However, our implementation ignores the much less significant cost in loading prototypes into the server podium: all needed server prototypes are loaded when the application launches.

As the discussion suggests, the application framework exposes three attributes on actor distribution:

```
1. Podium = {Client | Server | Mobile}
2. ActorLoad = {PreLoad | OnDemand}     (valid on Podium = Mobile)
3. PrototypeLoad = {PreLoad | OnDemand} (valid on Podium = Client)
```

Conventional wisdom suggests that **PrototypeLoad** should be applicable to actor prototypes whose **Podium** is set to *Mobile* because such prototypes are needed in the client podium as well. However, we argue that this feature is not useful because such prototypes can be loaded along with the first actors belonging to them. Using `ActorSpec` with these attributes, the developer can have customized component distribution of a Web application for a specific execution context by supplying an appropriate actor distribution scheme.

4.2 Podium Designs

A podium is an actor runtime environment in a specific platform. There are different strategies to implement an actor runtime environment. Since we are building the podium on top of a component-based system, it is natural to follow the classic actor definition: an actor is an object encapsulating a thread of control. This strategy has two shortfalls: it implements message exchange using thread communication, which is not efficient in synchronous method invocation; besides, thread programming is not available in existing Web clients.

We adopt a different strategy: namely, we directly use the native mechanism of (synchronous) method invocation in the host platform; for asynchronous method invocation, the caller places a request in a message queue, and in turn the podium's message dispatcher invokes the request synchronously (using the native mechanism) on behalf of the caller.

In accordance with our extended actor model, a podium needs to support features corresponding to three primitives: `create`, `send` and `destroy`. In this section, we discuss the implementation of these features in podiums.

4.2.1 Browser: Message Loop

The scripting environment in the browser is quite restrictive: essentially it is a single-threaded interpreter. Although lately a few features which suggest possible concurrency, for example *tabs* and *inline frames*, have been added, the programming interfaces of these features are extremely browser dependent and their semantics are far from formalized. From another perspective, however, the limitation of single threadedness simplifies the implementation of the client podium. For example, synchronization is not an issue in the client podium.

To create an actor, the client podium assumes a constructor function is available: an `ActorScript` prototype must be translated into a JavaScript constructor function–the prototype implementation. However, an actor in the client podium may create a server actor or a client actor whose prototype has not been loaded yet. Consequently we need a wrapper function `_create()`, which dispatches a **create** request to the right place through a framework service.

Similarly, to delete a local actor, the podium provides a function `_destroy()` which does the cleanup. If the target actor resides locally, the function just calls its finalizer (if defined) and then uses JavaScript's **delete** operator to remove all entries in that actor. If the target actor is in the server podium, the function redirects the request as the `_create()` function.

Message exchange is the definitive feature of the actor model: the podium has to implement the **send** primitive to support `ActorScript` communication. From Section 2.3, the implementation function should have the following interface:

```
_send(actor, sync, operation, port, arglist)
```

For the **set** operation the `_send` function directly uses the JavaScript object assignment:

```
actor.port = _export(arglist);
```

`_export()` is a function supporting the *pass-by-value* semantics, which clones `arglist` if necessary. The **get** operation is more complicated: if `actor.port` is a function, the `_send()` function returns a port object in JavaScript, otherwise it returns `_export(actor.port)` .

The **apply** operation has two modes, *sync* and *async*. In the synchronous mode, the function calls the method directly in JavaScript with the exported `arglist`. In the asynchronous mode, the function places a request embedding the port and argument list on a message queue. To dispatch a message in the message queue asynchronously, the client podium installs a JavaScript timer function in initialization which calls the port with the argument list in the message.

4.2.2 Server: Thread Pool

The design of the server podium is similar to that of the client. The podium functions are written in Java with the same design logic. However, given that the server podium can dispatch multiple messages using multiple Java threads, the timer-based message dispatcher becomes a thread pool of dispatchers. In addition, actors must be generated in synchronized Java objects so that no more than one thread at a time can execute on an actor. Note that this is consistent with the semantics described in Section 2.3: an active (processing a message) `ActorScript` actor can accept a request of synchronous method invocation if it is from the actor itself.

Given the nature of non-preemptive multitasking, sharing a limited number of threads among actors increases the likelihood of deadlocks; however, deadlocks are in any case unavoidable in the presence of synchronous message exchange. We believe at this time deadlocks are not an issue in the context of Web applications because the Web infrastructure is not intended to be entirely reliable. The existing time-out mechanism automatically breaks deadlocks and programmers should focus on error recovery instead.

4.3 Framework Services

Although the podiums provide virtual execution environments in the browser and server, *cross-podium services* are indispensable in coordinating actors in different podiums. For example, to service a request for remote actor creation, a special message is sent from the local podium to the remote one which creates the actor and then returns a reference. Similarly remote method invocation requires a framework service to handle serialization and deserialization. Furthermore, because individual podiums have no access to the global actor snapshots at runtime, it requires cross-podium coordination to recycle obsolete actors. All of these services can be implemented with a special message exchange protocol between the podiums. To support transparent message exchange, a cross-podium message delivery service is designed as the core of framework services.

4.3.1 Cross-Podium Message Delivery

The HTTP protocol is a message delivery system in nature. The challenge in building a cross-podium message delivery service on top of that comes from the fact that the HTTP protocol is asymmetric. It is straightforward to encode a message from the client to the server in an HTTP request, both in synchronous and asynchronous mode. But the other direction is not obvious.

In our early framework design [16], we used a *piggyback* technique. The technique is based on the following observation: without special server components, all events are triggered from the browser (by the user). Under this assumption, when the server needs to send a message to the client, it must be in a context of processing another client's request. Note that the client is always waiting for a response from the server: in the asynchronous mode the callback function of the request can resume control asynchronously. Instead of sending the expected response to the client, the server encapsulates its request message with a special note which instructs the client to deliver the message and return the control to the server by sending a new request. The *piggyback* technique is efficient because the connection is not persistent–it only exists when needed. However, as the current server podium supports multiple messages delivery at the same time, a client's request may result in multiple execution threads. This complicates the management of *piggyback* messages.

Recently the use of persistent HTTP connections [87] for bi-directional communication has gained more acceptance. The technique does not rely on polling the server (for example, refreshing the page) to get updated information; instead, the server maintains a persistent HTTP channel through which it can push data to the client continuously. The technique readily solves the problem of asymmetry but at a cost: the connection must be persistent in a session. Maintaining a live HTTP connection for each active client consumes more server resources and thus harms scalability.

4.3.2 Remote Actor Management

The main responsibility of the remote actor management service is to support transparent remote actors. Specifically, the service enables a remote actor to be accessed in the same way as a local actor. The basic idea is *masquerading*: a runtime system assists the podium to create a local proxy actor, and the proxy acts on behalf of the remote one through message forwarding.

The service maintains a *proxy table* in each podium. When an actor's reference is being exported out, the service decides whether it is a local reference or a remote one. If it is local, the service adds a table entry which points to the actor, and sends a special message containing the entry identification to the destination podium, which in turn creates a proxy actor accordingly.

One disadvantage of this scheme is that each import of an actor instantiates a new proxy actor for the remote actor, that is, there can be many proxy actors representing one remote actor. To save memory space, the podiums can have another table which tracks the proxy actors by their identifications. The service checks the table in the podium to determine whether the actor has been imported before, and reuses the existing proxy actor when available. Sharing the proxy actors comes with a cost: the local garbage collector cannot automatically release the proxy actors because the table always holds their references.

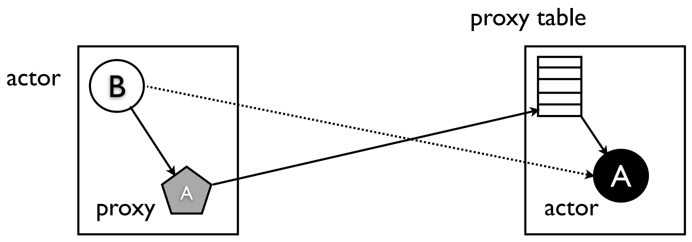


Figure 4.2: Using a proxy actor to represent the remote actor locally.

If the reference being exported is remote, that is, it is in fact a reference of a local proxy actor, the service just sends the identification embedded in the proxy actor to the original podium, where the actual actor's reference could be found in the proxy table (because it was exported before). To create and kill a remote actor, the requesting podium sends a message including the command and the actor's reference (its identification).

4.3.3 Remote Method Invocation

Given the cross-podium message delivery and remote actor management services, supporting remote method invocation is trivial. Note that the proxy actor is just a message forwarder: the proxy forwards messages to the actual recipient through the cross-podium message delivery service and thus allows local actors to act with it transparently (see Figure 4.2). To forward a message to the remote podium, the proxy actor uses the **send** primitive provided by the message delivery service. The remote actor management service then locates the recipient actor and delivers the message. The primary role of the remote method invocation service is serialization and deserialization of the payload in a message. Since both podiums run JavaScript, the serialization format can be plainly the JavaScript literal representation so that the `eval()` function can deserialize. For arrays and structures which can contain cyclic references, *pointer swizzling* [75] is required in serialization.

4.3.4 Automatic Garbage Collection

`ActorScript` does not assume the existence of an automatic garbage collector; instead, it provides a manual actor deallocation operator and a finalizer mechanism. The decision is based on the fact that automatic garbage collection is not implementable in all potential platforms, and when it is implementable in a combination of platforms, the execution framework can provide the service transparently.

Automatic garbage collection across podiums emerge as a non-technical challenge in the design of `ActorWeb`. Unlike distributed active object systems [95, 97] which require more complex garbage collection mechanisms, in our case the two most widely-used garbage collection algorithms, *reference count* and *mark-and-sweep* [64], can be applied with ease. Because the prototype implementations are generated, it is easy to insert code to manage the reference counts. Our podium designs employ message loops to dispatch messages: temporarily suspending actor messages demotes active actors into static objects and a mark-and-sweep algorithm is ready to use.

However, two issues have to be considered. First, our target platforms have their built-in garbage collectors, which can clean up those actors that are referenced only by the local actors. A new garbage collector is redundant and not cost-effective: only a small portion of actors cannot be released by the local garbage collectors, and many Web applications are not intended to run continuously–when the session ends, all actors are released. Second, the browser's JavaScript environment is not intended to support system programming: JavaScript has no programming interface to report the usage of memory in the scripting environment.

Consequently it is better to make the service optional, which means that it can be deployed on demand. The *mark-and-sweep* algorithm is more suitable given this requirement: when a pre-defined condition has been met, the application initiates the garbage collection service and freezes all actors by stopping message delivery.

It turns out that there are only two cases the local garbage collectors cannot handle. The first happens in the server actors exported to the client: the server podium has no knowledge about whether a client proxy actor is released by the client's garbage collector, and thus its proxy table must permanently keep the references of exported actors, which cannot be recycled (see Figure 4.3). The lack of the finalizer mechanism in JavaScript contributes to this problem. In the server podium, the proxy actors are implemented in Java so finalizers can be added to notify the client podium when they are released. The second case is the classical cyclic reference problem: local garbage collectors cannot detect a reference cycle across multiple podiums without certain protocols (see Figure 4.4).

Observe in both cases the presence of proxy actors is a necessary condition; we implement a *mark-and-sweep* algorithm to break the links involving proxy actors. The garbage collection service identifies obsolete proxy actors and releases their entries in the proxy tables. This observation also gives us a hint about the timing of garbage collection: when the number of proxy actors grows beyond a threshold, the application initiates the garbage collection service.

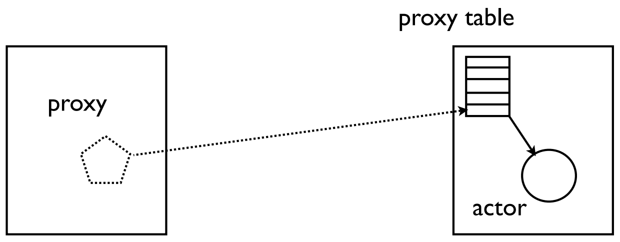


Figure 4.3: Recycling a proxy actor does not release the reference in the proxy table.

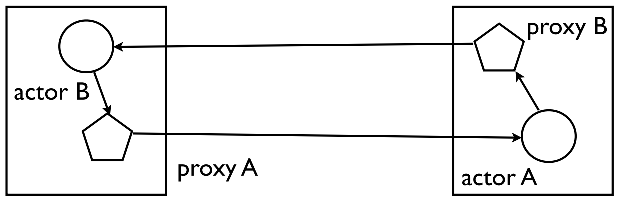


Figure 4.4: Reference cycles cannot be broken by local garbage collectors.

4.4 Customized Web Applications

A Web application can be customized through adaptive components and reconfigurable distribution. The adaptive components are `ActorScript` prototypes, which undergo a generative process to fit to the target podiums. The reconfigurable distribution is achieved by the annotations on prototypes; these annotations guild the prototype linker and application loader to distribute components as desired.

4.4.1 Prototype Implementations

Given the *Implementation Linkage Graph*, it is straightforward to generate a customized application executable. First the generator constructs prototype implementations if they are not pre-built. For a prototype written purely in `ActorScript`, there are three possible implementations:

Server: The implementation works in the server podium and creates actors there; it is generated in the server language with interfaces to the server podium.

Mobile: The implementation has two parts: the server part creates actors and the client part includes the methods of these actors. The newly created actors are tagged with their **ActorLoad** attribute values.

Client: The implementation works in the client podium and creates actors there. The **PrototypeLoad** attribute controls the linkage, not the code of the implementation.

Since both the server and the client podiums run JavaScript, we just need to generate a JavaScript constructor function for each prototype implementation. *Rhino* (the JavaScript execution engine used in the server podium) can run Java objects smoothly with JavaScript, and thus it is also doable to generate a Java class for a prototype implementation which supposedly enjoys better performance. However, Java objects are not dynamic: their methods cannot be reassigned and thus not fully compatible with `ActorScript`. *Rhino* provides a JavaScript to Java compiler but it still takes JavaScript code as input.

Converting an `ActorScript` prototype into a JavaScript constructor is painless because their control structures and object (actor) layouts are very similar. The `counter` prototype in Section 2.2 is generated to the following JavaScript:

```
function counter(i) {
        if (i == undefined)
            this.n = 0;
        else
            this.n = i;
}
// var n is not initialized, declaration is not needed
counter.prototype.inc = function() {
    return ++this.n;
}
counter.prototype.dec = function() {
    return --this.n;
}
```

The major difference is in actor related code, specifically the statements and expressions involved with the `(.)` operator. They are linked to the specified podium library which provides the actor features. For example a prototype implementation in the client podium needs to use the JavaScript library provided by the podium:

```
// var c2 = new counter(5);
var c2 = _create("counter", 5);
// var v = c2.inc();
```

```
var v = _send(c2, true, "apply", "inc");
// v = c2.dec();
v = _send(c2, true, "apply", "dec");
// c2.inc$();
_send(c2, false, "apply", "inc");
// c2.n = 0;
_send(c2, true, "set", "n", 0);
// v = c2.n;
v = _send(c2, true, "get", "n");
```

Not all prototype implementations are generated: prototypes provided by external libraries undergo a different process. For example we can design a database connector for the browser using the Ajax technique [50]. In this case the two implementations are provided separately.

The generator also has to provide the correct linkage for each prototype implementation. For example in Figure 3.13, the implementation G_1 has the references to F_0 and I_1. Conceptually the statements in G_0

```
var actor = new F();
var actor = new I();
```

will be replaced with

```
var actor = new F0();
var actor = new I1();
```

4.4.2 Prototype Linker and Application Loader

Recall that the server podium loads all server prototypes (including the server part of mobile prototypes) as soon as the application starts: the linkage to these prototypes is also resolved at that time. When a server prototype is used for the first time, the server podium tries to resolve the unresolved client linkage. Here the **PrototypeLoad** attribute plays its role: if the attribute value is *PreLoad*, the client podium loads the prototype at this time; if the value is *OnDemand*, the client podium loads a stub of the actual prototype. Either way, the server prototype resolves all its linkage.

We have a different linking strategy for client prototypes. A client prototype tries to resolve its linkage to other prototypes when the loader is ready to load it. As a result, loading a client prototype may trigger a chain of prototype loads through the linkage. In Figure 4.5 we can see the attribute **PrototypeLoad**

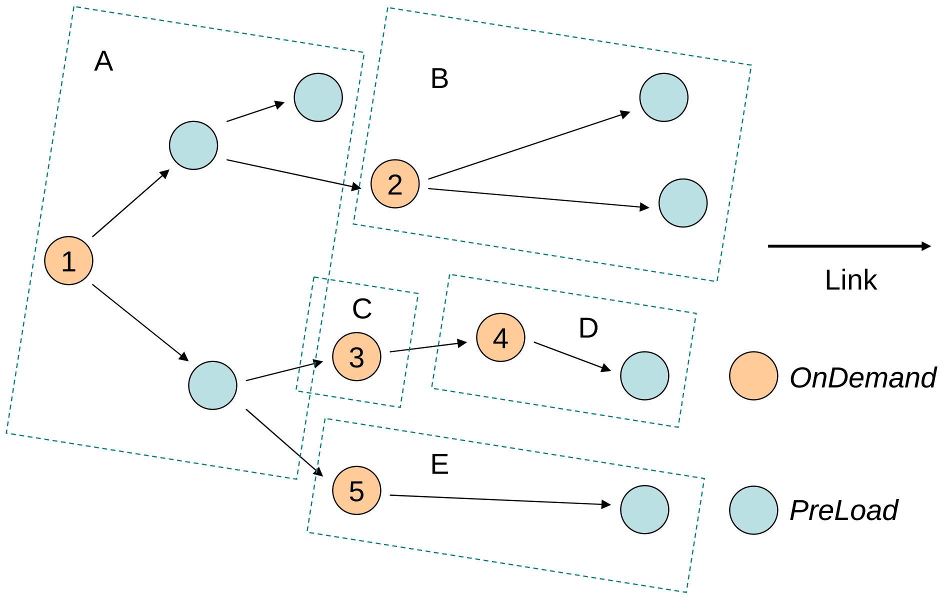


Figure 4.5: The **PrototypeLoad** attribute organizes client prototypes into loading batches.

organizes the client prototypes into several loading batches: when a prototype needs to resolve a client prototype which is the root of a loading batch, all prototypes in the batch will be loaded.

The **ActorLoad** attribute of mobile actors works in a similar way. When a mobile actor is going to be sent to the client podium, the loader inspects the references to other mobile actors the actor holds: those tagged with *PreLoad* will be loaded with the actor together while the references to the rest will be filled with stubs. These actors that are loaded simultaneously also form a loading batch as the client prototypes. Note loading batches are not statically determined: there can be more than one actor which has references to a mobile actor so that it is not foreseeable which path will be taken at runtime. In the example shown in Figure 4.5, if a prototype other than 1 in A is loaded first through a referential link not depicted, the loading sequence will become very different.

4.5 Exploiting Locality

One major criticism of location agnostic development is that the application cannot exploit *locality*: to enable redistribution, interaction between components are assumed to be remote. However, there must be some components colocated in the same address space: these components still have to follow the semantics of remote components. In this section we introduce several techniques which can reduce the overhead.

Code Sharing allows different prototype implementations sharing their common code. *Lazy-Copying* is a technique for local method invocation where deep copying aggregate data structures often incurs unnecessary overhead. *Delayed Message Delivery* postpones the delivery of certain synchronous messages whose return values are not required immediately; this helps performance by reducing the latency in synchronization.

4.5.1 Code Sharing

Conceptually each actor carries its own behavior but in reality actors created by a prototype implementation share the code of behavior. Using the path-splitting algorithm described in Section 3.5, a new prototype implementation is created even with a very slight difference. For example, in Figure 3.13, G_0 and G_1 have the same attribute but different linkages. Another possibility is that two client prototype implementations have the same behavior and linkage but different loading policies.

Implementations of a prototype should be able to share code in a podium, assuming they are using one address space. From the above cases we can see two opportunities in sharing: the linkage and the loading policy. These properties are supposedly hardwired in a prototype implementation by the generator; however, they can be parameterized. The linkage of a prototype implementation is a set of prototype implementation names, which can be stored in variables. Note in JavaScript, an `ActorScript` implementation is just a constructor function, which itself is an object and thus can have its own variables. To enable code sharing, we can exploit the `prototype` property in JavaScript (note it is different from the prototype in `ActorScript`). When a JavaScript object cannot find a property request, the object in the `prototype` property is consulted. In this way we can make the shared code as a standalone constructor function, and each implementation has its `prototype` pointing to that function, with separate variables for its linkage and loading policy.

The shared code does not affect the loading policies of these sharing prototype implementations. In Figure 4.6, assume both implementations are for the client podium and $Implementation_0$ is *PreLoad* while $Implementation_1$ is *OnDemand*. When $Implementation0$ is first loaded, the shared code is loaded along with it. Later as a request to $Implementation1$ triggers its loading, the loader will just load the non-shared part with the knowledge that the shared code has been loaded.

4.5.2 Lazy-Copying of Aggregate Data Types

Recall that an actor is self-contained: its internal state is not shared with other actors. Instead, actors exchange information through message passing. This property lead us to the semantics of remote procedure

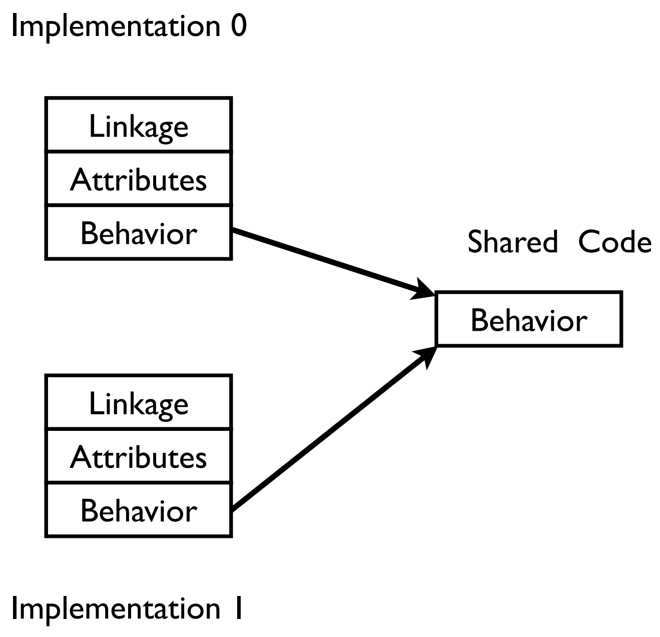


Figure 4.6: Implementations of a prototype can share code.

calls: a value is exported through a copy because the caller and the callee may reside in different podiums.

The *pass-by-value* semantics looks harmful. It implies a method invocation may involve several *copy* operations, even if the caller and the callee are running in the same podium. However, except *array* and *structure*, most data types in JavaScript and thus in `ActorScript` are *immutable*–that is, there is no syntax, method, or property that allows the programmer to change the content of an instance of these data types. For example, a variable can be assigned to different numbers, but there is no way to change the value of a number. The immutability implies that in the implementation, there is no tangible difference in the parameter (message) passing semantics. In our implementation, the export functions for these data types just return the input parameters literally. The actual copy operation is performed by the remote method invocation service at the time of serialization.

For an array or a structure, however, the export function performs a deep cloning because the receiving actor may change the content. Observing the fact that many arrays and structures are passed *read-only*, we employ a *lazy-copying* strategy to avoid unnecessary copy operations. When the export function takes an array, for example, it does not clone the array right away; instead, a shadow copy is returned. The shadow copy contains no entries but a link the the original array. The *read* and *write* operations of the shadow array are defined as follows:

Read: reads the the value from the original array through the link.

Write: copies all items from the original array and removes its *shadow* status and then writes the value.

Because actors can work concurrently, the original array may be changed in the presence of its shadow copies. It is essential to ensure the shadow copies always read the unchanged content. For that reason, the *write* operation of the original array has to be modified as well. The modified *write* operation forces all its shadow arrays copying the full content before writing the target item. The chance of such content flush is not high in practice: most value exports happen in parameter passing of synchronized method invocation. If the callee actor does not save the parameter (a shadow copy) in its state, when the control returns to the caller, the shadow copy has been recycled.

Further modifications around this concept can be applied. For example, the export function can decide whether or not to adopt lazy-copying depending on the size of the input array: if the size is small, it may be more efficient to make the copy immediately. A shadow *read* can also cache the item which has been read as incremental content copying.

4.5.3 Delayed Message Delivery

When a distributed application is designed with the full knowledge of the cost of communication and its component distribution, the behavior of composing objects tends to exploit the understanding. Consider an HTML form: when the user clicks a submit button, all the input values in the form are transmitted to the server, in addition to the submit event. The reason is that in response to a click action, the event handler at the server has to access the form's content. Without bringing these input values with the event, the server needs follow-up requests to the client. On account of the inherent latency in synchronous message exchange, this could result in an unbearable delay.

Invoking a method asynchronously does not suffer the latency in synchronization. But in many cases, synchronous method invocation is more convenient and intuitive: in the example above, to access the form's content is a series of synchronous property get operations because the rest of the computation cannot start before the property values become ready. However, these return values may not be used immediately and the rest of the computation should be able to continue as far along as possible until they are actually needed. The idea is championed by some concurrent systems [8,102] through the concept of *future* objects.[1] A future object represents the return value of an asynchronous procedure call. The caller continues its computation until the access to the future object: synchronization occurs at that point.

The concept of future objects is compatible with our actor model: we can specify that invoking a method asynchronously will receive a future object representing the return value. However, we decide not to include

[1]See concurrent call/return communication with a join continuation passing transformation as an alternative [68].

the feature in `ActorScript` at this time. An obvious reason is that it requires substantial runtime system support in synchronization, which conflicts with our generative approach using minimal core libraries and is not feasible especially in a simple platform such as a browser. Formulating the delicate semantics of future objects is another obstacle. For example, in addition to when the value is needed, should the synchronization occur at the time when a future object is being exported?

In principle, synchronization should be as late as possible for maximal concurrency; this implies that there could be more than one future object waiting for the same value because `ActorScript` uses the *pass-by-value* semantics. Multiple future objects waiting on a value further complicates the message exchange mechanism in the context of a distributed environment: a message response has to be delivered to multiple platforms.

Instead of using future objects, we design a runtime system exploiting the idea in a restricted but transparent form: the runtime system postpones the delivery of certain synchronous messages until the return values are needed or another remote synchronous message is being sent. Therefore, multiple synchronous messages are sent in bulk and also the synchronization latency is reduced. This scheme is based on an observation that in the context of Web applications, communication results in more latency then computation. Implementation can be done in a strategy similar to lazy-copying: instead of sending a synchronous request to the remote method invocation service, the podium holds it and returns a shadow structure. When a message is being sent to the structure (the caller is accessing the value), the podium releases the request (with all pending requests if any) and then replaces the structure with the return value before returning the control. The implementation is not complex on the basis of that `ActorScript` is dynamically typed [2] and an object's methods can be changed at runtime.

There is one concern about the delayed message delivery service: the service must ensure that the delay will not violate the semantics in the concurrency model. In our current configuration, there are only two podiums working in tandem and thus the consistency can be preserved as long as the delayed messages are sent in the original order. At this time, the service is exclusively available in the server podium and only for the *set* and *get* operations on actor properties.

[2]Note that for purposes of this typing, as objects in JavaScript, all actors belong to a single actor type.

Chapter 5

Applications and Experiments

We illustrate the performance impact of adaptable components and reconfigurable distribution by two typical Web applications. As discussed in Section 1.2, the location of execution can affect the distribution of computation, the amount of data transmitted in a session and the response time of an application. We demonstrate these effects with a Web-based arithmetic calculator which performs computation based on the user input. Variants of this application are ubiquitous on the Web, including unit converters, mortgage calculators and currency converters. A calculator does not transmit a significant amount of data over the Internet: to emphasize the trade-off between computation load and network bandwidth, we employ the second example, a Web application presenting search results in the browser. The application represents a generic component in many Web-based information retrieval systems which supply information relevant to the user input. The example also highlights the influence of the timing of deployment: our experiments show that the application's runtime features in bandwidth and latency change along with the loading policy of the detailed information of search results.

We use three different types of computers in the experiments. The server machine has two 2.4GHz Pentium 4 CPUs with the hyperthreading feature enabled and 4GB RAM; it runs Fedora Core 5. The client computer has a 1.5GHz PowerPC G4 CPU and 1.25GB RAM; the operating system is Mac OS X 10.4 and the testing browser is Firefox 2.0. A PDA is used for some extreme cases in the experiments. It has a 667MHz CPU based on the ARM1176 core but other technical specifications are not clear for the closed system. The experiments are conducted in three different network settings. In the LAN setting the client and the server are connected with a 100Mbps Ethernet. In the WAN setting the client connects to the Internet with DSL technology. The ping time to the server is 20ms and the sustainable bandwidth from the server to the client is 291.87KBps. The PDA connects to the Internet through an EDGE (Enhanced Data rates for GSM Evolution) mobile network, whose speed is reported in a wide range: 40Mbps to 200Mbps.

At the end of this chapter we present the design of a context management system as a high-level application of this research. The system supports context-aware Web applications through adaptation policies for specification plans.

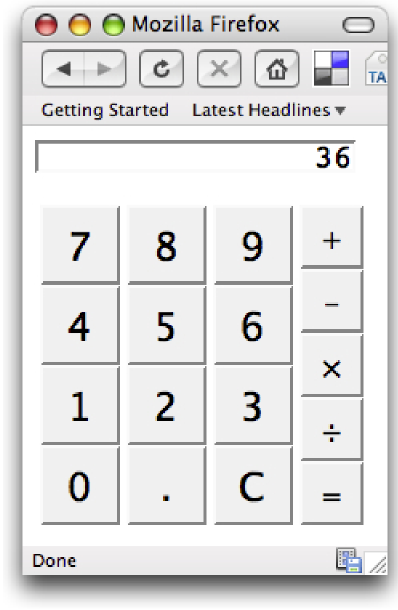


Figure 5.1: A basic Web calculator.

5.1 A Basic Arithmetic Calculator

Figure 5.1 shows a basic arithmetic calculator, which accepts input from the buttons, performs computation and then displays the result. In fact it represents a generic form of Web applications: applications for very different purposes such as authentication, spell checking and searching documents have a similar data flow to that of the calculator. However, there are two features distinct to the application: the application does not require a data source and it generates a negligible amount of data.

5.1.1 Application Components

Figure 5.2 shows the structural constituents of the application. The `Accumulator` stores the calculator value in its `value` property and displays it properly. To keep track of the user input, it exposes the `append` method to the `NumberButtons`. Note a `NumberButton` just delivers a user action intending to append the calculator value: the **dot** button is `NumberButton`, too. The `ComputingEngine` is the core of the application; it stores the operator and operand since the last calculation and exports the `command` method to the `CommandButtons`. The **clear** button resets the calculator's state and the **command** button instructs the `ComputingEngine` to carry out the calculation and update the calculator's value.

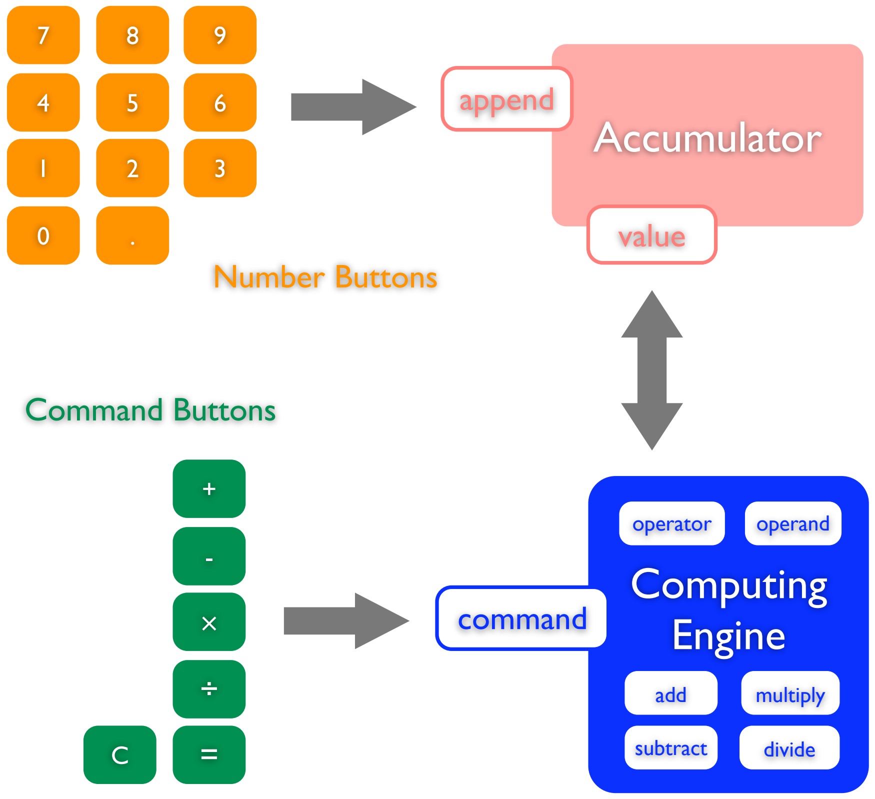


Figure 5.2: The components of the Web calculator

5.1.2 Deployment Schemes

Since there are four prototypes in the application and each of them has five options in specification attributes (see Section 4.1), in theory we can have 4^5 possible deployment schemes; in reality, nonetheless, only a few make sense. For example, assigning the `NumbereButtons` to the server and the `Accumulator` to the client only creates unnecessary data movement without resulting in any benefit. We choose three commonly used deployment schemes for the experiment:

- **Scheme 1:** All components are assigned to the server. The scheme is expected to result in the smallest initial load because the client does not need any code for computing. However, for each user action the server has to ship almost the same amount of data as the initial load. This scheme is suitable when the application is rarely used, for example a mortgage calculator alongside a real estate advertisement; otherwise it is not attractive given the extra load and latency in communication that the scheme results in.

- **Scheme 2:** The `ComputingEngine` works in the server and all the rest in the client. The scheme should have a medium initial load because the client does not need the code for the `Computing Engine`. The scheme is expected to be a good one for optimizing the bandwidth used when the application is assumed to be used only once: the latency is not terrible because the amount of data transmitted (only numbers) is negligible.

- **Scheme 3:** All components are allocated to the client. The scheme is expected to have the largest initial load because the client needs all the prototypes; however, it is also the only load: after loading application, the client has no need for the server and because of that, there is no communication latency in operating the application.

The experiment results are shown in Table 5.1. Using the table the response time and the amount of data transmitted can be estimated. In **Scheme 1**, the client makes a request to the server on each user action: to calculate the math expression $34 + 2$, the user has to click buttons five times (three digits, one operator and one command) and thus the total amount of data transmitted is 8.5KB. In **Scheme 2**, only the `CommandButtons` invoke the remote `ComputingEngine`. However, the `ComputingEngine` has to read the value of the `Accumulator` to serve the requests–it requires four HTTP requests for the math expression[1]: the accumulated latency in the WAN setting is 136ms.

The results conform to our expectation: **Scheme 2** does have smaller initial load than **Scheme 3**, but the difference is not significant. The reason is that the `ComputingEngine` is comparably simpler, and thus

[1]This is the cost of location agnostic development: refer Section 4.5 for details.

Deployment Scheme	Load Size		Latency	
	Initial	Subsequent	LAN	WAN
Scheme 1	1.7KB	1.7KB	4ms	48ms
Scheme 2	3.0KB	negligible	3ms	34ms
Scheme 3	3.2KB	none	none	none

Table 5.1: Performance results of the calculator application in different deployment schemes.

uses less code than the `Accumulator` because all the operators are built-in in JavaScript and `ActorScript`. In addition, making the `ComputingEngine` remote to other client actors requires some code for cross-podium communication, which is not needed in **Scheme 3**. The results also suggests that it is not feasible to load the `ComputingEngine` on demand: the supporting code's size is larger than what the on-demand loading scheme can save in this case.

The experiment suggests that deploying all components to the client is generally better: it has the shortest response time and in most cases saves the bandwidth. However, there are still plenty of applications of this kind which use the server approach for other reasons such as *robustness* and *data gathering* [11, 101]. In fact, this application is a very special case of Web applications because the computation logic can be directly represented by built-in functions in the language: the server's computing power is not useful and the size of the client code is tiny.

5.1.3 Calculating π

To better illustrate the server's role in the application, we enhance the calculator with a new feature: calculating π to the number of digits specified by the user (see Figure 5.3). To implement the feature, we have to define the arbitrary-precision decimal number and its arithmetic operations in `ActorScript`. These operations involve a more complex computation and the size of code is not negligible in transmission.

To support the new feature, the display of the `Accumulator` is changed to accommodate variable length of number representation and a new actor prototype `PICalculator` is designed to carry out the computation (see Figure 5.4). The new feature can be implemented as a method of `ComputingEngine`; but to be able to specify it separately, the feature must also be implemented as an actor.

We generate the enhanced calculator with **Scheme 2** and **Scheme 3** and conduct an experiment on the two versions. Because of the new prototype, the initial load size in **Scheme 3** surges to 6.9KB while that in **Scheme 2** just slightly increases to 3.2KB: the difference becomes noticeable with the new feature. However, the most substantial change is the response time, which includes computation time and network latency (see Table 5.2).

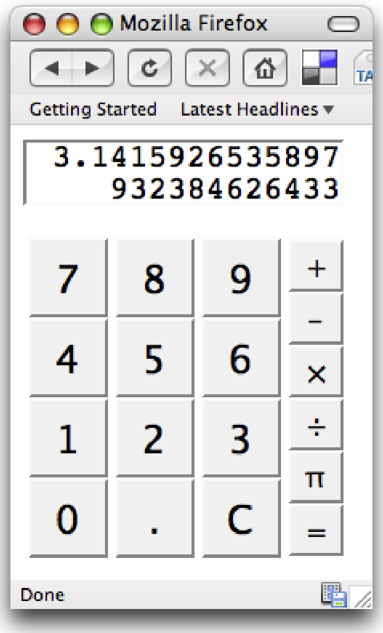


Figure 5.3: The enhanced calculator.

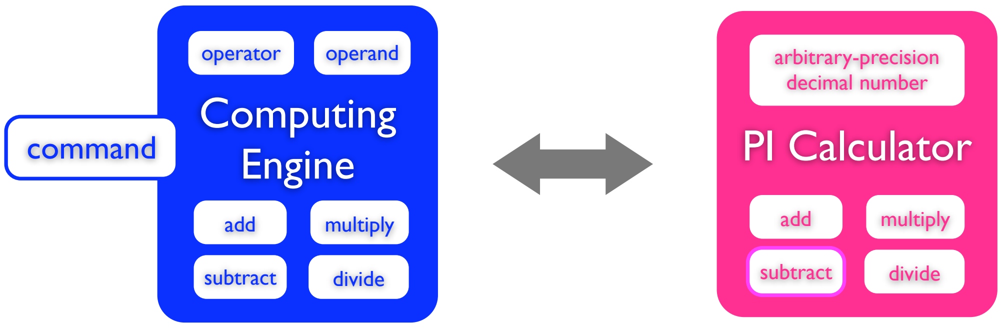


Figure 5.4: The `PICalculator` actor.

π	Computation Time				Latency	
Digits	Client	Server	PDA	Java	WAN	LAN
10	4ms	20ms	83ms	<1ms	34ms	3ms
50	14ms	35ms	289ms	4ms	34ms	4ms
100	63ms	65ms	848ms	6ms	34ms	4ms
500	985ms	487ms	13703ms	68ms	44ms	4ms
1000	3659ms	1839ms	52718ms	145ms	45ms	4ms
1500	8829ms	4015ms	n/a	209ms	45ms	4ms
2000	15924ms*	7116ms	n/a	276ms	44ms	4ms

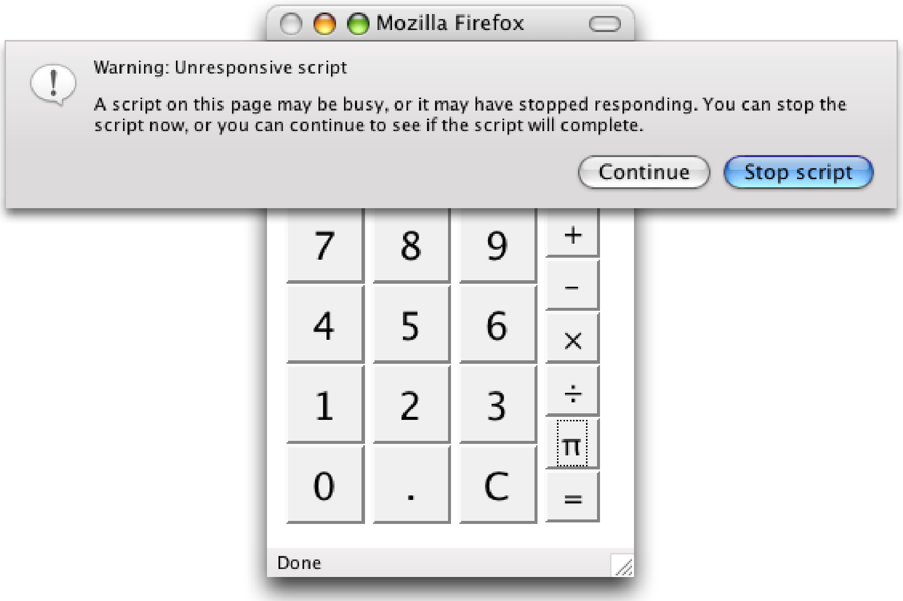
*Including the time to dismiss the warning dialog box.

Table 5.2: π calculation in different platforms.

The results show a faster computer (the server) can perform complex computation in a shorter time: when the number of digits is greater than 500, deploying the `PICalculator` to the server significantly reduces the response time. Shifting the computation to the server does increase the amount of data transmitted; for example, calculating 2000 digits in the server adds 2K in the payload, which prolongs the latency (44ms in the WAN setting) as well. However, with reduced computation time and less initial load size, the overhead can be justified in most cases.

Although the server version is more efficient, the response time of the client version looks not that bad and somehow tolerable. There are two additional issues in the client version. First, its response time is heavily related to the available computing power in the client, which is difficult (if not impossible) to detect in the browser. The third column in Table 5.2 shows the response time in the PDA we use. The performance is not acceptable for 1000 digits and the application freezes the whole system with more digits requested. Second, because current browsers do not support a multi-threaded scripting environment, a long execution client script can lock the browser or even the whole system. To enable the user to prevent such prolonged locking, many browsers pop up a warning dialog box (see Figure 5.5) when a client script continuously runs for a certain time. The interruption is not desirable in a Web application.

It is hard to believe that a modern computer which can execute millions of instructions in a millisecond, calculates π at such a slow speed. The problem is in `ActorScript`. `ActorScript`, as well as JavaScript, cannot express a complex data structure efficiently and treats all numbers as floating numbers; doing so hurts the performance of arithmetic operations on arbitrary-precision decimal numbers. Java has a built-in class, `java.math.BigDecimal`, to support arbitrary-precision signed decimal numbers. We implement a Java version of `PICalculator` using the class: note in `ActorWeb`, a prototype implementation can be implemented in a Java class separately. The fourth column in Table 5.2 shows the response time using the implementation. The results suggest that computation intensive actors should be implemented with suitable


Mozilla Firefox

Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, or you can continue to see if the script will complete.

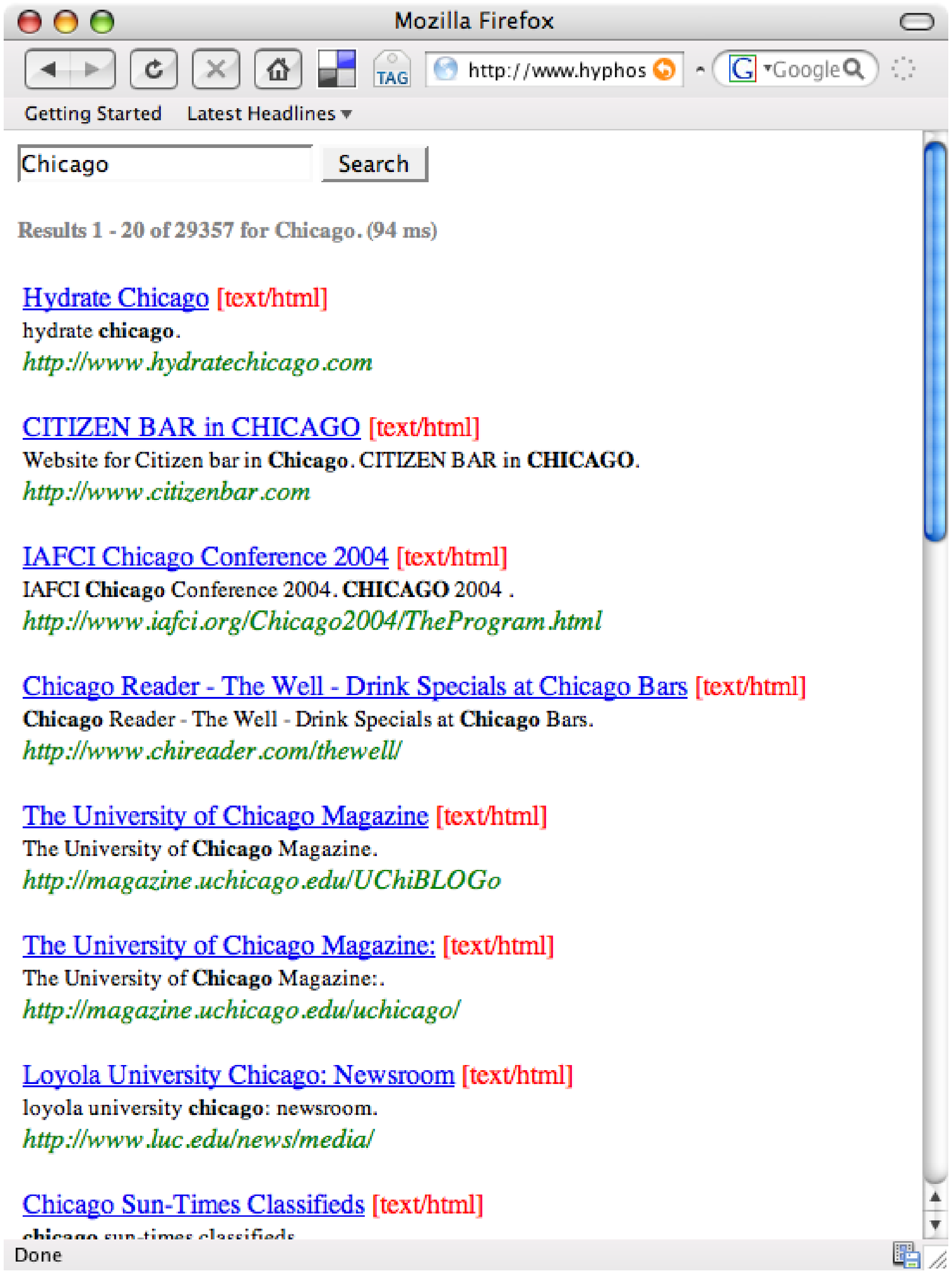
Figure 5.5: A warning dialog box pops up when the browser stops responding for the long running π calculation.

technologies instead of generated from `ActorScript`.

5.2 Presenting Search Results

Since the amount of data transfer is negligible, the calculator example does not address the performance impact on bandwidth and the need of on-demand loading. And because the basic operations are so simple, aggregate server performance is hard to evaluate. The π calculation is computation-intensive; however it is not a typical Web application which can attract numerous requests at the same time–even if it were, the computed results could have been easily cached or saved for future use.

To complement the calculator example, we implement a search application (see Figure 5.6) to illustrate the performance impacts on aggregate server load and bandwidth. We do not focus on the application's search function, which requires special software and is not practical to implement in `ActorScript`. Instead, the core logic of the application is in post-processing the search results, which is an essential component of a wide range of Web applications which present information from an external source according to the user input.



Results 1 - 20 of 29357 for Chicago. (94 ms)

Hydrate Chicago [text/html]
hydrate **chicago**.
http://www.hydratechicago.com

CITIZEN BAR in CHICAGO [text/html]
Website for Citizen bar in **Chicago**. CITIZEN BAR in **CHICAGO**.
http://www.citizenbar.com

IAFCI Chicago Conference 2004 [text/html]
IAFCI **Chicago** Conference 2004. **CHICAGO** 2004 .
http://www.iafci.org/Chicago2004/TheProgram.html

Chicago Reader - The Well - Drink Specials at Chicago Bars [text/html]
Chicago Reader - The Well - Drink Specials at **Chicago** Bars.
http://www.chireader.com/thewell/

The University of Chicago Magazine [text/html]
The University of **Chicago** Magazine.
http://magazine.uchicago.edu/UChiBLOGo

The University of Chicago Magazine: [text/html]
The University of **Chicago** Magazine:.
http://magazine.uchicago.edu/uchicago/

Loyola University Chicago: Newsroom [text/html]
loyola university **chicago**: newsroom.
http://www.luc.edu/news/media/

Chicago Sun-Times Classifieds [text/html]

Figure 5.6: A typical presentation layout of search applications.

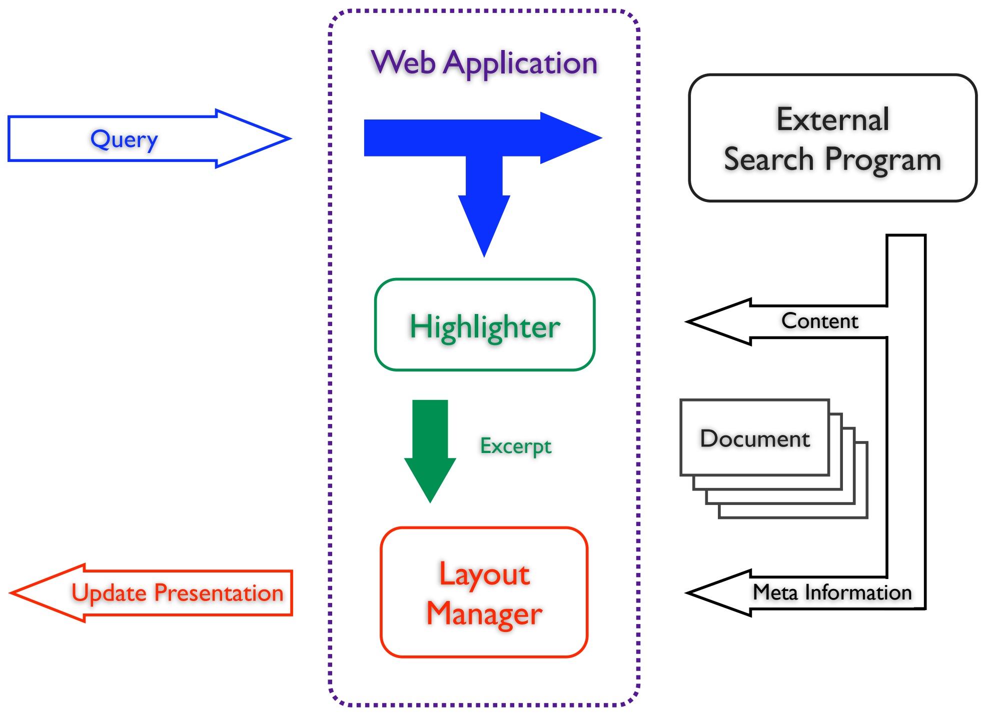


Figure 5.7: The structure of the search application.

5.2.1 Application Structure

The structure of our search application is depicted in Figure 5.7. The application forwards the query to an external search program, which performs a search and returns search results, including contents and meta information such as document sources and types. We use *Lucene* [45, 54], an open source text search engine library written in Java, to construct the backend search program. The `Highlighter` extracts the most relevant part of content to the query and generates an excerpt accordingly. The `LayoutManager` organizes meta information and excerpts to produce a presentation of the search results to the user. The presentation is composed of user interface actors such as the query input box and those representing document titles, which are not of interest in the experiment and omitted in Figure 5.7.

5.2.2 Effects of Computation Shift

The `Highlighter` is the most important component in the application. The processing logic is not trivially simple and the amount of flow-in data is generally much more than the flow-out. Therefore, changing the location of the `Highlighter` not only shifts the computation load but also the bandwidth requirement. When the `Highlighter` is deployed in the client, the `LayoutManager` must be placed in the client to avoid unnecessary data transfer. In this case, contents and meta information of search results have to be sent

Results per Query	Highlighter Location	
	Client	Server
10	1.4KB	1.5KB
25	3.7KB	3.9KB
50	19.8KB	12.4KB
100	61.4KB	31.5KB
200	170.4KB	79.6KB

Table 5.3: Data transfer of the search application in different deployment schemes.

through the Internet. The `LayoutManager` has more choices in location when the `Highlighter` is in the server; nevertheless both choices demand less bandwidth, which is proportional to the size of excerpts instead of to the size of contents. The trade-off is that the response time can be long when the server handles numerous request at the same time: although the computation cost here is not as high as that in the π calculation, the server load can be huge when the number of concurrent queries grows.

To observe the performance impact of the `Highlighter`'s location, we fix the `LayoutManager` to the client in the experiment. The experiment is based on the query of "Chicago" to a repository containing 339000 Web pages in different settings. The search returns 29357 results and the amount of data transfer is listed in Table 5.3.

The numbers are confusing in small numbers of results per query: when the `Highlighter` is in the server, the amount of data transmitted should be less. The reason is that the scoring function we use is based on the density of the query in a document; it turns out the top documents are relatively short (see Figure 5.6) and thus the excerpts are just the full contents with occurrences of the query marked. As the number of results per query increases, the difference in data transfer conforms to our expectation.

The response time varies in different network settings. We choose two extreme combinations to stress the effect of computation shift. Note the response time is the sum of computation time and network latency. Therefore, the best combination should be the server `Highlighter` and the LAN setting while the worst should be the client and the WAN. The experimental results are summarized in Table 5.4.

From the table we can see when the server is dedicated to a single request, the results are consistent with our expectation: the Server/LAN combination beats the Client/WAN in all cases although the margin is not obvious in less results per query though. Because these documents are short (as we mentioned earlier), there is not much work to do and the advantages in both computing power and network latency wane. However, when the number of concurrent queries grows, that is, the server is serving more requests at the same time, the response time of the Server/LAN combination quickly becomes slower than that of the Client/WAN. The server deployment turns out to be unacceptable under the load of 500 concurrent queries. On the contrary,

Concurrent Queries	Experiment Setting	Results per Query				
		10	25	50	100	200
1	Client/WAN	45ms	67ms	173ms	326ms	810ms
	Server/LAN	42ms	64ms	101ms	284ms	586ms
10	Client/WAN	56ms	68ms	151ms	344ms	807ms
	Server/LAN	61ms	91ms	271ms	554ms	1058ms
50	Client/WAN	42ms	67ms	153ms	333ms	802ms
	Server/LAN	307ms	463ms	999ms	2378ms	5278ms
100	Client/WAN	42ms	68ms	159ms	324ms	825ms
	Server/LAN	632ms	933ms	2027ms	4769ms	10620ms
500	Client/WAN	44ms	72ms	165ms	329ms	819ms
	Server/LAN	3222ms	4969ms	10284ms	25077ms	55205ms

Table 5.4: Response time of a query in different settings.

the response time of the client deployment is almost unchanged up to 500 concurrent queries. The results show that shifting computation to the client decentralizes the server load and thus enjoys better scalability without loss of performance. The conclusion matches our previous research [92] on distributed Web crawling.

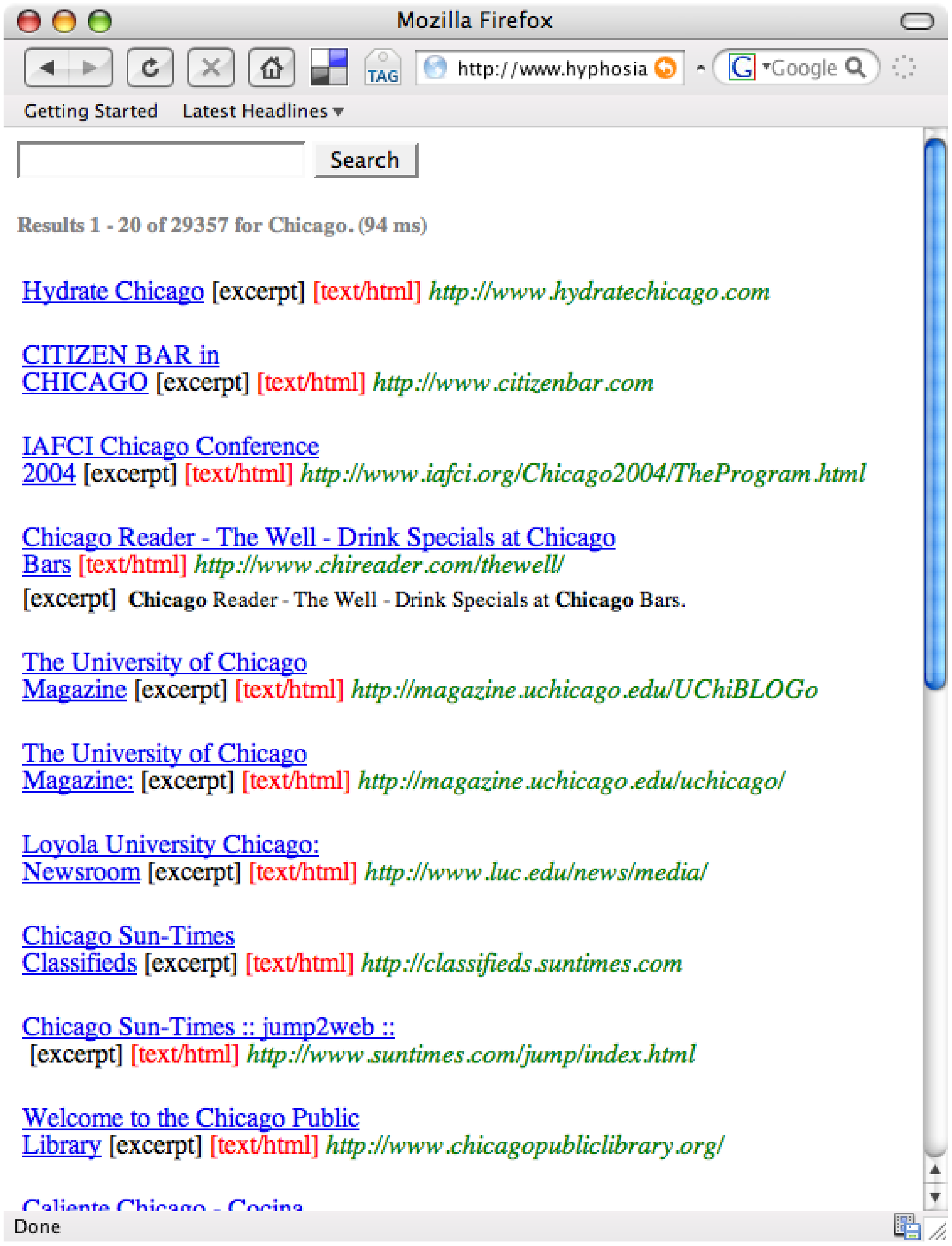
5.2.3 Loading Actors on Demand

The search application requires substantial outgoing network bandwidth. For example, in our experiment returning 200 search results requires 170.4KB for full contents and 79.6KB for excerpts only; with 500 queries the server outputs 85.2MB and 39.8MB, respectively. It is unlikely that the user will read through all the excerpts, probably just a few of them. In the case the bandwidth utilization is inefficient and thus the components which are not immediately (or ever) needed should be downloaded on demand. This scenario exists in a wide range of Web applications including the MythTV example in Section 1.2.

To support loading excerpt actors on demand, we have to change the layout a little bit (see Figure 5.8) so that they are not required at the time when the results are shown. The application only displays the meta information of the search results: an excerpt actor is downloaded and becomes visible only when the user clicks the tag representing it.

To emphasize the difference between the two loading policies, we conduct an experiment on the search application which returns 200 results for a query. Excerpts are highlighted in the server, and the layout manager which creates and arranges excerpt actors, is moved to the server to create *mobile* actors. Table 5.5 summarizes the amount of data transmitted in different cases.

The results of the experiment show that loading excerpts on demand significantly improves the utilization of bandwidth. Even when the user reads as many as 25 excerpts out of 200 (probably not that common), the saving is still almost 50%. However, if all the excerpts must be eventually moved to the client, the



Results 1 - 20 of 29357 for Chicago. (94 ms)

Hydrate Chicago [excerpt] [text/html] *http://www.hydratechicago.com*

CITIZEN BAR in CHICAGO [excerpt] [text/html] *http://www.citizenbar.com*

IAFCI Chicago Conference 2004 [excerpt] [text/html] *http://www.iafci.org/Chicago2004/TheProgram.html*

Chicago Reader - The Well - Drink Specials at Chicago Bars [text/html] *http://www.chireader.com/thewell/*
[excerpt] **Chicago** Reader - The Well - Drink Specials at **Chicago** Bars.

The University of Chicago Magazine [excerpt] [text/html] *http://magazine.uchicago.edu/UChiBLOGo*

The University of Chicago Magazine: [excerpt] [text/html] *http://magazine.uchicago.edu/uchicago/*

Loyola University Chicago: Newsroom [excerpt] [text/html] *http://www.luc.edu/news/media/*

Chicago Sun-Times Classifieds [excerpt] [text/html] *http://classifieds.suntimes.com*

Chicago Sun-Times :: jump2web :: [excerpt] [text/html] *http://www.suntimes.com/jump/index.html*

Welcome to the Chicago Public Library [excerpt] [text/html] *http://www.chicagopubliclibrary.org/*

Caliente Chicago - Cocina

Figure 5.8: Lengthy and non-essential excerpts can be loaded on demand.

Data Transfer	PreLoad 200	OnDemand 5	10	25	50	200
Initial	79.6KB	32.4KB				
Additional	none	1.6KB	3.4KB	8.3KB	16.9KB	66.8KB
Total	79.6KB	34.0KB	35.8KB	40.7KB	49.3KB	99.2KB

Table 5.5: Data transfer of the search application with different loading policies.

Latency		LAN	WAN	Mobile
PreLoad		11ms	495ms	7379ms
OnDemand	Initial	6ms	288ms	5439ms
	Additional	4ms	43ms	1607ms

Table 5.6: Latency of the search application in different network settings.

OnDemand policy requires 25% more data transfer than the *PreLoad* policy because of the overhead to support loading actors on demand. The saving comes with another cost: when an excerpt actor is requested, the user could experience a delay in loading the actor. The latency varies under different network conditions; Table 5.6 lists the latency measured in the three network settings we use. The results show in the LAN and WAN settings, additional latency is negligible: in fact the *OnDemand* policy enables the application to respond faster because of reduced data transfer. However, in the mobile network, where the bandwidth is supposed to be the most precious resource, the *OnDemand* policy yields unsatisfactory results: the latency is so huge that even reading only a few excerpts causes the user to suffer an unacceptable experience.

5.3 Content-Aware Web Applications

The above experiments support the notion that it is desirable to make Web applications context-aware. Specifically, these applications require:

- The ability to *adapt* themselves to specific contexts of their deployment.
- The potential to *evolve* under widespread change in both execution environments and patterns of usage.

Context-aware software and service adaptation have been extensively studied and gained success in pervasive computing [72] and multimedia QoS [38] adaptation; however, several observations in the case of adaptive component distribution in Web applications suggest different assumptions and design strategies:

- It is acceptable to have a few bad deployments since there are several factors in the Internet which

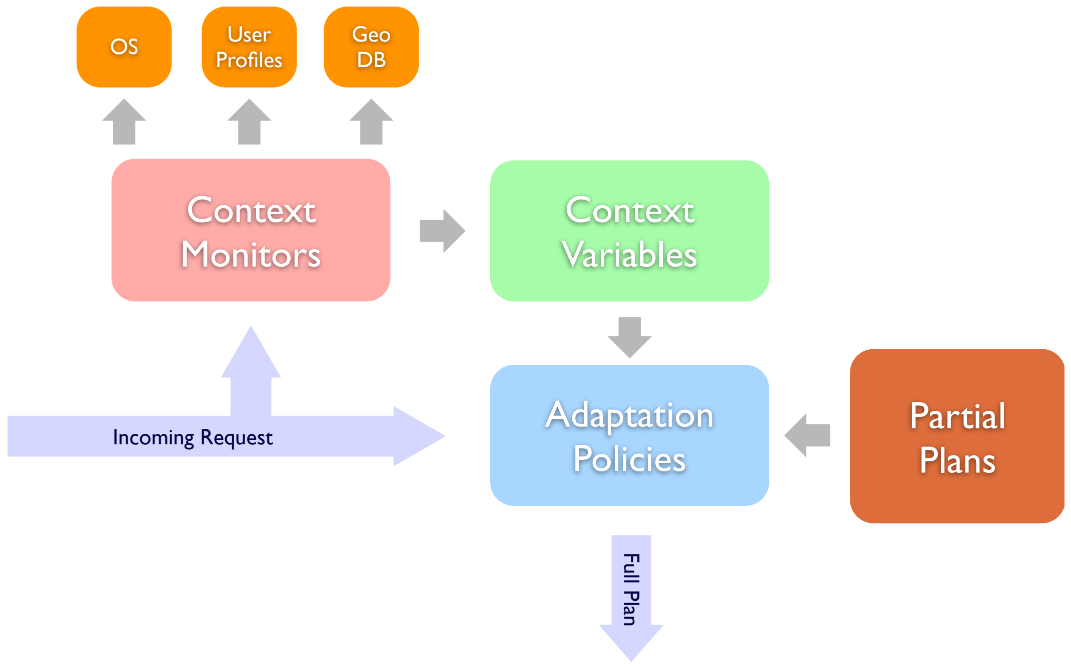


Figure 5.9: The architecture of the context management system for Web applications.

cannot be observed and predicted, such as the actual network condition and the client's stability. It is more important to ensure overall efficiency instead of optimal allocation in each execution.

- Resource consumption in a single execution is usually not demanding and the service duration is comparatively short. The pressure on the server system comes from numerous concurrent sessions, not individual ones. This has two implications:
  - Complex decision-making processes such as negotiation may kill any benefit gained through adaptation.
  - The ability to re-adapt (under context change) during a session is not crucial.
- The Web is an *open system* composed of standards and protocols. A solution requiring extra features in all participating platforms is unlikely to win wide acceptance.

We design an extensible context management system for Web applications [15] based on the foundation of this research. The system (see Figure 5.9) includes three modules: *context monitors* actively collect context information and store context features in *context variables*, which are used by *adaptation policies* to generate full deployment plans.

5.3.1 Design Strategies

The core elements of this research support customizable Web applications, that is, for each execution context, a specification on component distribution can be defined for a special version of the application. However, this approach is not scalable in building context-aware Web applications, which may need to adapt to multiple orthogonal context features. Consider the following scenario. There are two context features of interest, one has m variations and the other has n: in combination there are mn contexts and thus specification plans. In other words, the number of specification plans is *exponential* to that of the context features of interest.

The structured specification language described in Section 3.4 has eased the pains in writing numerous specification plans by exploiting their composing structures; even so, the service provider has to supply these specification plans manually for the context-aware Web application. There is a missing layer: a truly context-aware application should adapt its configuration to the *context*, not to the *deployment plan*. An ideal scenario for the missing layer would be totally skipping the deployment plans: an intelligent mechanism automatically annotates actors according to the context features of interest. Instead of a total solution, we demonstrate that the missing layer can be filled with a couple of add-on modules, which select partial configuration plans based on user-defined rules on the context.

There are two requirements of context-aware Web applications: *adaptability* and *extensibility*. We follow the principle of *separation of concerns* in design to ensure adaptability, and adopt the paradigm of *generative programming* in implementation to guarantee extensibility. The architecture is *layered*: the bottom layer is the core elements of this research, where component distribution is separated from application logic and thus can be reconfigured according to separate specifications. A generative framework allows new distribution features to be added in the future. The structured specification language can be leveraged to support the middle layer, where features related to higher-level concepts are abstracted from component distribution rules and new features can be exploited using new transformation processes. In the top layer, context features are rendered into context variables which are used in defining deployment policies. New context features can be imported through new context variables with modules to collect them.

5.3.2 Partial Plans

The key to our context management system is the role of the *configurable block* in the structured specification language. It is sufficiently expressive to represent a specification plan's composing structure, as described in Section 3.4; it can also organize cross-cutting concerns: logically unrelated components sharing common properties can be aggregated into a specification block. For example, a developer can identify those objects whose deployment has great impact on a certain resource (hence share a common property), such as CPU cycles and bandwidth, and define a specification block accordingly. The block then serves as a *partial plan*, which is a parameterized function of the condition of the specific resource. The context management system can reuse partial plans to create a deployment plan for a new identified context preference. This approach also provides extensibility: as new resources are taken into consideration, new specification blocks and new configurations can be designed independently without making drastic changes to existing ones.

5.3.3 Context Features

The concept of *context* is abstract and the available *features* of a context are evolving. For example, in the past service providers had little access to information about the client's *geolocation*, which is widely exploited nowadays for better service and resource allocation. Nonetheless, a context feature can be utilized only if it is quantitative and measurable. In our context management system, a context feature is represented by a *context variable*, and the introduction of a new context variable must come with a variable monitor which estimates the up-to-date value and maintains the variable.

Monitors can be implemented in a variety of forms as long as they update their variables in a timely manner. For example, the system status monitor for the context variables `System.CPU` and `System.Bandwidth` is implemented with OS system calls; the monitor for client capabilities is implemented with JavaScript detection code; and the user preference monitor reports related variable values by consulting the user profile database.

5.3.4 Policy Design

Defining an adaptation policy is straightforward: a policy is a pair consisting of a condition on context variables and a set of partial plans. When the context management system receives an incoming context, it collects the partial plans in the policies whose conditions evaluate to *true* and generates the full deployment plan.

Consider the first example in Section 1.2: we want to deploy the *Mortgage Calculator* to the client only under a special context where the CPU cycles are much more precious than the bandwidth. The policy can be written as follows:

```
(System.CPU*[Cost_cpu] > System.Bandwidth*[Cost_bandwidth])
    => <MortgageCalculator Podium="Client"/>
```

A cost function can be a constant for the normalization factor or a function of other context variables. In the second example, we first define a specification block **TVListing** with a configuration *FastResponse* which specifies all client components as *PreLoad*. The following policy selects the partial plan for fast response when the client's connection has long latency and the available bandwidth is not in stress.

```
(Client.Latency*[Cost_latency] > System.Bandwidth*[Cost_bandwidth])
    => <TVListing configuration="FastResponse"/>
```

Chapter 6

Related Work

We present a comprehensive survey of the work related to and giving inspiration for our research. Java provides a platform independent programming and execution environment on various physical devices, including those on the Web. Transparent distributed computing allows location agnostic application development such that component distribution can be modified freely. Our specification system transforms an application according to specification schemes; in a broad sense writing specifications is *metaprogramming.* A product line of an application can be viewed as a "redundant group", through which differentiated services are provided to different execution contexts. Web services share the same infrastructure with Web applications although they are aimed for different purposes and represent different paradigms in software architecture. Finally there are several Web libraries and frameworks which hide the details of Web platforms and enable programmers to develop more portable code.

6.1 Java and Virtual Machine

Started as one of the earliest solutions to execute programs in Web browser, *Java technology* [88] has become omnipresent nowadays. A key feature is its promise of *Write Once, Run Anywhere*: Java applications can be developed on any platform, compiled into a standard binary format (Java bytecode) and run on any device with a Java Virtual Machine (JVM). JVM provides a uniform virtual platform on a variety of physical devices from cellphones at the low end to enterprise servers at the high end. Ideally using Java, the problem of heterogeneity in the Web does not exist anymore. Stimulated by Java's popularity, Microsoft has introduced its own solution to the virtual execution platform with *Common Language Infrastructure* (CLI) in *Microsoft .NET Framework* [83]. Ironically, although Java has earned a great success in many application domains, it has not been widely adopted as a solution on the Web, even though the Web is where Java made its debut. There are many technical (for example, its mediocre performance in the server) and non-technical (for example, the redundancy in its features and those available in modern browsers) for its failure: in any case, Java is by no means an ideal candidate for seamless cross-tier Web application development:

- Although Java provides a rich set of libraries for distributed computing, it does not provide uniform object access over distributed JVMs. Java objects are not location transparent: local interactions have different syntax and semantics from remote ones. Reconfiguring a distributed Java application requires the programmer to modify a great deal of code.

- The discrepancy in different Java application models (editions) results in non-trivial component redistribution in objects using model-dependent libraries. Moreover, these libraries usually expose too many details: developers have to deal with the HTTP protocol and session control instead of straightforward method invocation and event handling.

There are good reasons to distinguish local interactions (that is, interactions in the same address space) from remote interactions [76,100], and to expose the details of the underlying system. Most of these reasons are based on the conventional wisdom: the efficiency and robustness of computer programs cannot be overemphasized. In particular, local and distributed computing have fundamental distinctions in:

Latency: The difference between local and remote access is at least several orders of magnitude.

Semantics: Local access is through direct references while proxies are required for remote one.

Failure: Distributed computing needs to ensure consistency of state after a partial failure.

Concurrency: Local computing enjoys more control over the order of object invocation.

Since these factors have great impact on performance and robustness, component distribution should be taken into consideration at the first stage, that is, built into the computation logic. We agree that using a unified object model across multiple address spaces does not guarantee efficiency; instead, performance could be very bad. However, no matter when the decision on object placement should be made, it is desirable to have the flexibility to change that with limited effort. As indicated, in different execution contexts, the optimal component distributions of a Web application might not be the same. And for any distributed system as complex as the Web, the number of distinguishable contexts is countless. Detaching the aspect of object placement from computation logic provides a certain degree of flexibility. Besides, although our virtual environment assumes that all objects interact as if they were remote (for example, parameters are passed by value), it does not mean a runtime environment cannot exploit the fact that some objects are deployed in the same address space at runtime. As shown in Section 4.5, several optimization techniques to reduce the inefficiency that would result from a direct or naive implementation of the semantics can be applied on podium design. Last but not the least, Web applications often have different priorities in performance metrics: response time is usually more important than bandwidth and infrequent failures can often be tolerated in exchange for the ability to do quick modification.

6.2 Transparent Distributed Computing

For better control of object deployment in distributed Java programs, researchers have proposed several Java extensions for a transparent distributed Java platform. *JavaParty* [82] introduces a new `remote` class modifier to annotate Java classes whose instances should be distributed across different hosts, and hides the details of remote communication from programmers. The JavaParty runtime system performs object distribution and migration using a *distributor* object, which can allocate objects into desired nodes specified in its state, where a node is identified by its index in current system configuration. Deployment changes can be achieved by specifying different strategies in the distributor. One restriction is that the distributor only works at the class level because it does not have runtime information to identify an object. *Addistant* [91] is a Java bytecode translator that partitions a standalone Java application into a distributed one through a separate user-specified distribution policy. It also requires the distribution policy to be specified at the class level. We have already shown that specifying at the class level does not provide enough resolving power in object distribution and could have specification conflicts in the presence of inheritance.

Besides, JavaParty supports assigning an object's initial location directly in the creation statement with an annotation. In this way the distribution strategy can be at the *statement* level: objects created at the same statement have the same initial location. *Doorastha* [27] takes the annotation-based approach a step further. In addition to annotation of classes whose instances are intended to be distributed, Doorastha enables the programmer to annotate the parameter passing semantics of method invocation to improve performance. However, when the object distribution policies are dispersed in the source code as part of object creation statements and class definitions, the adaptability is severely impaired: an application needs to be re-annotated and recompiled to adopt a different placement policy.

ProActive [8] (formerly *Java//* [9]) is a distributed programming language based on concurrent objects; the language supports the Java syntax and high-level transparent inter-object synchronization though the *future* object. It has built-in abstraction for the physical location: the *node* object. To create an object the target node must be specified; therefore the distribution strategy has to be resolved at the object level. This is accomplished through a *coordinator* object, with which each object consults at its creation time. Moreover, the distribution strategy can be changed without modifying the source code: the coordinator can read a strategy from an external source. Several Java-based actor systems, such as *Actor Foundry* [4], *SALSA* [96] and *Adaptive Actor Architecture* [63], have similar capabilities. These languages and systems support reconfigurable distribution strategies at various levels; however, there are two limitations:

- They are implemented with Java class libraries: these "relocatable" objects gain the extra features through subclassing a specially designed class. The overhead is high for small components and local communication is not fully exploited.

- The distribution strategy only applies to objects: their initial nodes can be assigned and their migration strategies can be specified. There is little control on code (class objects) loading. These systems either assume shared code repositories or require pre-installation on all nodes. Conceptually reconfigurable code distribution can be achieved through customized Java class loaders.

`ActorSpec` has the potential to be used with these systems. For example, the coordinator can use the prototype linkage graph and runtime information, such as the context of the request object, to assign initial nodes and migration strategies, or just statically generate annotations in source code. In general, although these systems provide transparent component distribution for Java applications, they are not ready for Web applications. The uniformity and symmetry of these distributed platforms is achieved by using Java, whose availability is not a valid assumption in all Web platforms. Moreover, all of them use their own protocols to support remote communication and migration over the network: non-standard protocols are not feasible given the ubiquitous existence of firewalls.

6.3 Metaprogramming

Metaprograms are programs that manipulate other programs (or themselves) as their data and writing such programs is called metaprogramming [5]. Our specification system uses specification schemes (metaprograms) to modify applications–it is a metaprogramming system. There are many applications of metaprogramming. The *Meta Object Protocol* (MOP) [66] and Aspect-Oriented Programming (AOP) [35,62] are two of them with a common goal in providing a mechanism to separate functional and non-functional concerns. A comparison of these techniques and their relation to computational reflection can be found in [33].

The basic idea of the MOP is using *meta objects* to control *base objects*: an MOP is a protocol about expressing and executing meta objects. To put our research into the context, the loader and linker of the application framework are meta objects which govern actors' placement and movement: the attributes associated with each actor are part of the metaprogram of a distribution scheme. Several two-level systems [3, 98] had been built and demonstrated the feasibility to use meta objects to achieve separation of non-functional concerns such as backup policies, multimedia QoS and synchronization from the application. Although there is no strict definition, many MOP-enabled systems support runtime meta objects. Runtime meta objects provide more flexibility and extensibility at the cost of overhead in runtime system support.

AOP takes a more static approach: instead of having meta objects working at runtime, an AOP tool weaves non-functional aspects back to functional components. The generators in our application framework have a similar role as an AOP tool: it consumes annotations to generate customized prototype implementations. In general AOP can result in more efficient applications than an MOP system because most non-functional concerns are resolved statically at compile time. However, the *join point models* which dictate the places where aspects and applications meet have more limitations on aspect design and reconfigurability. Several researchers applied these principles on Web systems to separate concerns of component distributions [65, 93].

The MOP and AOP mainly focus on a modular framework to structure concerns. They allow programmers to customize an application to a very high degree but require careful analysis in an early phase to avoid an intrusive design. On the other hand, our design leaves the burden to the runtime environment designer, who decides the attributes to be exported and implements a mechanism to interpret the specifications through runtime systems and code generators. Instead of implementing customized aspects or doing metaprogramming, programmers annotate their applications. Our approach is less powerful because a pre-designed framework only allows limited customization. We argue that the trade-off can be justified: it keeps Web development simple and the ability to reconfigure object distribution is sufficient for this problem domain. A different runtime environment can be designed for a different domain when needed. In addition, most of these tools and systems assume a homogeneous system, which is not valid on the Web as indicated.

6.4 Component Redundancy

Component redundancy has been widely used in computer systems to provide reliability [34] and availability [51]. In the domain of Web applications, developers exploit the technique for different purposes: adapting to clients and ensuring QoS under constraints. Some Web applications require the user to select the right versions, for example, flash or HTML only; others automate the process by embedding client script to detect the browser's capabilities. The technique partly solves the problem of diversified clients and provides a certain level of QoS. However, there is no unified component model over the Web platforms. As a consequence a component does not have a clear boundary with the rest of the application. Lacking well-defined interfaces, it is difficult to develop and maintain multiple implementations of a component; this explains why most Web applications with component redundancy provide only two choices. Another issue is the way to choose the right version upon request. Manual selection by the user is undesirable and detection code only checks for compatibility and capabilities.

Microsoft's ASP.NET [74] supports limited platform-independent development and customized deployment of Web applications. HTML form controls are packaged into HTML server controls and it allows developers to design their own server-side controls. A copy of the document layout in the client is kept in the server where all events for server controls are processed; event handlers directly operate on the copy without recreating a fresh one for the client. Specially designed server-side controls can be employed to support customized deployment. An ASP.NET server-side control detects the capabilities of a requesting client and metamorphoses itself into different implementations: it can be pure server code or a mix of code of both sides. However, the development cost is not negligible: it requires a great deal of careful programming on server controls. Developers have to write the detection code and manually arrange the deployment. For this reason many third party server controls emit identical code for all browsers without exploiting this feature.

Ada Diaconescu et al. propose a component framework [30,31] on the server and have an implementation in J2EE (Java 2 platform, Enterprise Edision) [89]. Using their framework, developers implement multiple Enterprise JavaBeans (EJBs) in different strategies to form a redundancy group of a component, and the server can choose the suitable EJB according to the execution context. The ability to replace components is granted by the server framework and thus cannot be extended to the client. Besides, developers have to implement multiple EJBs for a component: redundancy is manually provided instead of automatically generated. In our generative application framework the uniformity is cross-platform and thus redundancy is supported in both the client and the server. In addition, no only can redundancy be manually provided with multiple prototype implementations but also be automatically generated with different specification schemes applying to the component. In the latter case, the customization is restricted in component distribution as opposed to that with unlimited freedom to change its data structures and algorithms in the former case.

6.5 Web Services

Web applications are usually mistaken for *Web services* [10,22] because the literal meaning of both suggests software on the Web. In fact, they represent two very different concepts and the only common element between them is the Web infrastructure, specifically the network communication protocols and the content description languages. Web applications are intended for human use: human users can easily find a Web application with their real world experience and recognize the semantic meaning of its presentation composed in HTML. Nevertheless, Web applications are not suitable for automated use, such as by a software agent. For example, all major airlines have a Web application for flight booking and status checking in their Web site and a traveller has little difficulty in finding and booking the best fare by using these Web applications. However, it is not trivial to implement a *vertical* or *meta* airfare search engine, a software agent, for the task: there are three obstacles:

- How to locate the application in a Web site?
- How to find the available services in an application?
- How to operate an application to acquire the desired service?

These obstacles do not hinder human users, who can effectively understand the visual hints given in the Web pages, and are very resilient to various formats for a concept (such as a date or a phone number). For automated use, the practice is to employ a site-specific software agent for each case: the required knowledge is embedded in the agent's logic. This approach is not scalable; a major problem is that Webmasters tend to frequently update their Web sites, and thus the applications, to add new features and new looks. A human user can adapt to the new application immediately but a site-specific software agent becomes obsolete under the change.

It is the consensus of many experts that the use of the Web will shift from *human-computer* to being increasingly *computer-computer* [58, 99]. The concept of *Web services* emerged from the need to enable computers to provision and acquire services effectively through the Web infrastructure. For this purpose, several standards have been developed, including the following three core specifications:

- UDDI [40] is a protocol for publishing and discovering Web services. Using UDDI, a software agent can locate the Web services of interest.
- WSDL [18, 24] is an XML format for describing a service's interfaces, including the details of their bindings to specific protocols.
- SOAP [20] is a protocol for exchanging XML-based messages over the Internet. SOAP provides a basic message exchange framework for high-level communication patterns.

Additional specifications are available but the core ones largely overcome the obstacles and thus are sufficient in most cases. The standards for Web services are based on XML for the content description languages and primarily rely on HTTP for the communication protocol. Therefore a Web service is ready to be implemented and deployed in a Web server and our research can be applied to it, for example, writing the service logic in `ActorScript` and using `ActorSpec` to distribute the implementation in multiple platforms. However, it is not always the best approach to building Web services: the most important assumption in this research is the heterogeneity of Web platforms and the incompatibility of application models of them, but with Web services standards, components of a Web service speak the same language in a uniform model. At the other end, a Web service can be viewed as an important component of a Web application and thus a Web client should be able to access the service directly and even execute part of the service locally. In practice, current Web clients rule out this possibility for security concerns: with few exceptions such as pictures and multimedia files, the client script can only download resources from the original site that it comes from. Therefore, the only applicable case is that the Web application and the required Web services are hosted in the same site although this is not a norm in the paradigm of Web services. A likely configuration of utilizing Web services in a Web application is making the application server as a gateway to consolidate required services and to support that, a new server podium with the capability to make use of Web services is needed.

6.6 Web Toolkits and Frameworks

As the Web browser has become an important role in Web applications, many Web toolkits and frameworks have been developed to facilitate browser and *AJAX* programming. These programming tools provide a high-level abstraction of the browser through libraries and compilers to address one or more of the following difficulties:

- Browsers may have different programming interfaces for a common feature, or define different semantic meanings of a common interface.
- The interfaces to manipulate Web documents are quite primitive: the programmer has to deal with basic elements instead of high-level controls or widgets.
- Communication support is limited: to achieve effective server-to-client interaction, the programmer must design a special protocol and implement marshalling and unmarshalling.

These programming tools lessen the problem of cross-browser incompatibilities and greatly simplify communication programming in the multi-tier Web architecture.

Prototype.js [81] is a very lightweight JavaScript library which provides client script programmers a better interface to local Web documents and supports *AJAX* programming through a set of special functions and JavaScript prototypes. Although the application model is still document-based, the library hides major incompatibilities among browsers. *Dojo* [42] is another JavaScript library for the browser. Instead of exposing raw HTML/XML elements, Dojo implements a graphical user interface toolkit supporting high-level *widgets*, which are better abstraction of components in Web applications. Prototype.js and Dojo are primarily for client programming; their applications can work with any Web server as long as the server code follows their communication protocols.

Google Web Toolkit (GWT) and *Echo2* [77] take a different approach: instead of using JavaScript on Web documents directly, programmers develop Web applications in Java with a high-level user interface library. Similar to our approach, before the application deployed, a generative process translates part of its source code into JavaScript for the client. A major difference between GWT and Echo2 is the distribution of their generated code. GWT is a real browser-centered solution: most of the code is executed in the browser and thus the application state is maintained there as well. On the contrary, Echo2 places most code in the server, and only uses the client in certain components requiring fast response. Using Java has a significant advantage: applications can be fully tested and debugged in a JVM before release, given that there are plenty of Java programming tools available. However, not all Java objects can be implemented in this way: obviously multithreading is not supported in current browsers. Besides, at this time GWT and Echo2 are tightly coupled with Java Servlet Technology [59], which has not been a dominant platform on the Web.

Rather than using a library to support uniform access to the Web user interface, *XML11* [86] defines an abstract windowing protocol inspired by *X Window System* [52]. Developers can use any library built on top of the XML11 protocol to achieve device independent presentation. The example implementation is based on Java and supports code migration through XMLVM [85], a Java virtual machine in JavaScript, which executes Java class files. Because it works at the JVM instruction level, location transparency is realized by using a *distributed shared memory model*. With shared memory and bytecode emulation, code migration in XML11 does not involve code transformation. Although using low-level emulation is not as efficient as using high-level translation, in most cases Web applications are not computation-intensive for a single execution. Instead, transportation overhead can be huge: the JVM instructions are encoded in XML–a simple Java class must be represented with numerous instructions and thus requires a big payload in communication. At this time XML11 provides no specification mechanism for object distribution: the programmers have to manually annotate Java classes for their objects' execution location.

In general, all of these Web toolkits and frameworks are more interested in providing higher level abstraction over the underlying Web platforms and less in customizing application distribution. Their applications can accommodate a variety of browsers, and in the case of Prototype.js and Dojo generic Web servers. However, the supported platforms must meet their assumptions on platform features and capabilities. The discrepancy and barrier between the client and the server still exist: these programming tools do not support an interface to control object distribution of an application. Customizability is limited: customization is based solely on the capabilities and features of the execution platforms and developers have little control over it. Nonetheless, these toolkits and frameworks can help in implementing podiums of our application framework, especially a better user interface library for our application framework.

Chapter 7

Discussion and Conclusion

The thesis describes a new approach to build customizable Web applications through adaptive components and reconfigurable distribution with a virtual programming environment, a specification system and a generative application framework. The virtual programming environment provides a location agnostic and platform independent abstraction over heterogeneous Web platforms. The specification system facilitates writing and enforcing reconfigurable component distribution through application structure transformation and component annotation. To execute a customized application on specific platforms, we implement a generative application framework including a light-weight execution environment for each participating platform, and a generator adapting actors according to their annotations.

7.1 Virtual Programming Environment

Our virtual environment is based on the actor model. Two important requirements for a virtual environment are being expressive enough for application description and being simple enough for implementation. The actor model is simple because communication is only through a single form of message exchange. Function calls and events, the most widely used language constructs for component interaction in existing systems, can be easily expressed. We demonstrate that many advanced features can be expressed with auxiliary actors, which can be viewed as *metaactors*. Researchers have also reported that more sophisticated system services, such as *transaction* [36], *replication* and *synchronization* [47], can be properly described in the actor model. Although any programming language can be used in our model as long as it can interact with the actor framework, we design `ActorScript` using most common features in scripting languages so that actor prototypes can be translated into other languages with reasonable effort.

A problem in `ActorScript` is that an actor handles one message at one time, as a *synchronization block*, and there is no built-in feature to suspend a request already in process. Consequently even if the request in process is blocked, the actor cannot accept another one except that from itself: the actor is blocked as a whole. The restriction makes it difficult to express certain components with synchronized method interfaces.

For example in a *bounded buffer*, when a **put** request realizes the buffer is full, it has already entered the actor and no other **get** requests can get in before it exits. Such behavior can be implemented with asynchronous interfaces, which require a different paradigm from the currently used one, or busy-waiting, which is not efficient, however. Some actor systems [102] have a language construct similar to the *Monitor* [56], which solves the problem neatly. However, we intentionally omit the feature because it is not implementable in current browser environments. Besides, since `ActorScript` allows changing an actor's layout at runtime, suspending a request can result in inconsistency. For example, when an actor suspends a request and becomes available, other actors can change the actor's composition, including deleting or changing some properties and methods. Before the suspended request resumes, the actor may have become a totally different one. Our preliminary solution is to add a pair of new operations on ports: **lock/unlock**. When a port is locked, the actor framework stops delivering messages to it. In our client podium, such operations would immediately cause a deadlock, but semantic inconsistency is avoided. A possible solution is restricting the actors which allow **lock/unlock** operations to the podiums supporting multithreading.

A possible extension to `ActorScript` is a type system for actors. JavaScript 2.0 (ECMAScript Edition 4) [57] will include a type system in its object model. Most scripting languages do not have a type system for their data structures and objects because of the nature of dynamicity. For example, in `ActorScript` an actor's port can be freely rewired to other ports or even removed at runtime. In addition, there is no prototype hierarchy in `ActorScript` and strictly enforcing type checking makes programming difficult. We are evaluating different ways to define "types" on actors so that safety and flexibility can be balanced.

7.2 Specification System

Our specification system `ActorSpec` is flexible. The language and methods work for all kinds of specifications based on object annotation, including those of a concern unrelated to object distribution. For example to customize security settings, a security attribute can be defined. Our system can identify those objects requiring special treatment on security, that is, special implementations. However, one limitation of `ActorSpec` is that it resolves rules statically and does not work with dynamic features. The transformation approach makes it possible to statically check invalid or conflicting specification rules, but it is non-trivial to find *bad* rules since "badness" is usually related to runtime features. For example, a bad distribution scheme such as interleaving server and client actors along an execution path results in multiple hops in the Internet. Because the specification system does not understand the semantics of actors, it cannot reason about the execution path.

A long-term goal of the specification system is to automatically generate rules according to abstract features, such as runtime system load, usage patterns and application structures. To achieve this some kind of logic is required in the specification language. For example it can greatly reduce the number of rules if we can express: `if` *A*`'s` **Location** `is` *Server*`, all actors which can create` *A* `have the same` **Location** `setting`. The challenge is how to resolve conflicting rules in this form. Currently if two rules apply to an actor, the more specific wins: such a resolution does not work with logic rules (at least without some sophisticated A.I. ability to reason about specificity. We need to design a rule precedence on logic rules.

Another problem of the object annotation approach is that some properties shared by a set of objects cannot be specified directly. For example if we want to express a synchronization concern of a group of objects, it is not possible to select the *group* and attach a desired attribute to it. The concern must be specified in an indirect way through a specially designed attribute, such as *synchronization group number*: objects of a group are assigned the same number. We are investigating the applicability of the *relational model* [28] to our specification system. Note that a rule specifying an attribute on an object can be viewed as a relation of this attribute containing a *2-tuple* of that rule. In this form, attributes applicable to an object group of size n can be represented with *(n+1)-ary relations.*

7.3 Generative Application Framework

Our framework design is extensible because it strictly follows the programming model: component communicates only through message exchange and thus podiums are loosely-coupled. It is straightforward to develop other server podiums to work with our client podium and vice versa. Moreover, the design is based on common properties of clients and servers: it is easy to port the implementation to other platforms. An exception is the requirement on multithreading for server platforms; some popular server scripting platforms such as *PHP* [70] and *Perl/CGI* [6] have no built-in support for concurrency. In these cases, the framework has to issue multiple HTTP requests to initiate threads. A future direction of our work is to extend the framework to include more podiums in different roles. For example, some Internet intermediary platforms [7,53,79] allow partial processing on cached data or work as surrogates for very simple clients such as cellphones and PDAs. The extension can promote the *two-tier* architecture of our framework to *n-tier* and provide availability to a wider range of applications.

Our user interface module is directly built on top of the standard DOM library, and thus closely represents the native interface of the browser. It allows maximal customizability but is not easy to use: programmers manipulate HTML elements and attributes instead of widget actors and properties. It is feasible to build a user interface toolkit similar to Dojo, Echo2 and GWT. Since these toolkits are also built on top of HTML and CSS, they can co-exist with the standard DOM library with limited modification.

Actor migration is a possible extension to the framework. We do not support full actor mobility at runtime because we have not found enough evidence of its usefulness to justify the overhead; however, the implementation of mobility is straightforward for an actor in `ActorScript`: an actor's state can be stored in a structure and sent to another podium as we implement the *mobile* actors in the framework. But to support full mobility, that is, to support the functionality that enables actors to migrate at any time, another runtime service is required. In the current implementation, an actor can only move once before any actor sends a message to it; that means no other actors have its temporary reference. When an actor can move out of a podium at any time, it is possible that some actors hold the old reference. Therefore, we need to set up a proxy actor for that reference: when these actors send messages to the migrated actor, the proxy actor forwards it to the right receiver.

The attribute exposed by the framework to control the location of an actor is expressive enough but the attribute to specify the loading policies is restrictive. The problem is that actors annotated with *PreLoad* are not preloaded strictly: they can only be loaded after others have created them. If a programmer's coding style is to create actors on demand, the attribute becomes useless. For prototypes, which are static, we can design a new value for the attribute to instruct the loader to preload at the time an application launches; however, because an actor's creation may involve runtime variables and result in side effects, the framework cannot speculatively create and load an actor. We are considering a new value *Speculative* which hints some actors can be created at any time without violating the correctness of the application. The new value will increase the expressiveness of the loading policy but it also requires additional runtime support.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Gul Agha. Introduction. *Communications of the ACM, SPECIAL ISSUE: Adaptive middleware*, 45(6):30–32, 2002.

[3] Mark Astley, Daniel Sturman, and Gul Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5):99–107, 2001.

[4] Mark Christopher Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[5] Jonathan Bartlett. The art of metaprogramming. *IBM DeveloperWorks*, 2005-2006.

[6] Gunther Birznieks, Scott Guelich, and Shishir Gundavaram. *CGI Programming with Perl.* O'Reilly Media, 2000.

[7] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.

[8] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components.* Springer, 2005.

[9] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998.

[10] Ethan Cerami. *Web Services Essentials.* O'Reilly Media, Inc., 2002.

[11] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data.* Morgan Kaufmann, 2002.

[12] Po-Hao Chang. An adaptive distributed object framework for the Web. In *Proceedings of the Combined 14th Workshop for PhD Students in Object-Oriented Systems and Doctoral Symposium*, 2004.

[13] Po-Hao Chang and Gul Agha. Supporting reconfigurable object distribution for customizable Web applications. *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, LNCS Volume 4278, 2006.

[14] Po-Hao Chang and Gul Agha. Supporting reconfigurable object distribution for customizable Web applications. In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC '07)*, pages 1286–1292, New York, NY, USA, 2007. ACM Press.

[15] Po-Hao Chang and Gul Agha. Towards context-aware Web applications. *Distributed Applications and Interoperable Systems: 7th IFIP WG 6.1 International Conference, DAIS 2007, Paphos, Cyprus, June 6-8, 2007. Proceedings*, LNCS Volume 4531, 2007.

[16] Po-Hao Chang, Wooyoung Kim, and Gul Agha. An adaptive programming framework for Web applications. In *Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT '04)*, 2004.

[17] Pinhong Chen, Kurt Keutzer, and Desmond A. Kirkpatrick. Scripting for EDA tools: A case study. In *ISQED '01: Proceedings of the 2nd International Symposium on Quality Electronic Design*, page 87, Washington, DC, USA, 2001. IEEE Computer Society.

[18] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*, March 2001. W3C Note 15. http://www.w3c.org/TR/wsdl.

[19] World Wide Web Consortium. HTML 4.01 Specification. http://www.w3.org/TR/html4/.

[20] World Wide Web Consortium. SOAP Specifications. http://www.w3.org/TR/soap/.

[21] World Wide Web Consortium. W3C Document Object Model. http://www.w3.org/DOM/.

[22] World Wide Web Consortium. Web Services Activity. http://www.w3.org/2002/ws/.

[23] World Wide Web Consortium. Web Style Sheets. http://www.w3.org/Style/.

[24] World Wide Web Consortium. *Web Services Description Language (WSDL) Version 1.2*, March 2003. W3C Working Draft. http://www.w3c.org/TR/wsdl12/.

[25] World Wide Web Consortium. Extensible Markup Language (XML) 1.1, February 2004. W3C Recommendation.
http://www.w3.org/TR/2004/REC-xml11-20040204/.

[26] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[27] M. Dahm. Doorastha: a step towards distribution transparency. In *Proceedings of the Net.Object Days 2000*, 2000.

[28] C. J. Date and Hugh Darwen. *Foundation for Future Database Systems: The Third Manifesto (2nd Edition)*. Addison-Wesley Professional, 2000.

[29] M. Deshpande, D. Schmidt, C. Ryan, and D. Brunsch. Design and performance of asynchronous method handling for CORBA. *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE: Confederated International Conferences CoopIS, DOA, and ODBASE 2002. Proceedings*, LNCS Volume 2519, 2002.

[30] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software system. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, May 2004.

[31] Ada Diaconescu and John Murphy. A framework for using component redundancy for self-optimising and self-healing component based systems. In *Proceedings of the CSE 2003 Workshop on Software Architectures for Dependable Systems (WADS)*, May 2003.

[32] Edsger W. Dijkstra. *A Principle of Programming*. Prentice Hall, 1997.

[33] Bill Donkervoet and Gul Agha. Reflecting on adaptive distributed monitoring. *Formal Methods for Components and Objects, 6th International Symposium (FMCO), Amsterdam, The Netherlands*, LNCS to appear, 2007.

[34] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John J. P. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions Software Engineering*, 17(7):692–702, 1991.

[35] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), 2001.

[36] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 195–208, New York, NY, USA, 2005. ACM Press.

[37] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Network Working Group RFC 2616. http://www.ietf.org/rfc/rfc2616.txt.

[38] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 128, Washington, DC, USA, 1998. IEEE Computer Society.

[39] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2006.

[40] Organization for the Advancement of Structured Information Standards. Introduction to UDDI: Important features and functional concepts, October 2004.

[41] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[42] Dojo Foundation. Dojo, the Javascript Toolkit. http://dojotoolkit.org/.

[43] Mozilla Foundation. LiveConnect. http://developer.mozilla.org/en/docs/LiveConnect.

[44] Mozilla Foundation. Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[45] The Apache Software Foundation. Apache Lucene. http://lucene.apache.org/java/docs/index.html.

[46] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[47] Svend Frolund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[49] John Robert Gardner and Zarella L. Rendon. *XSLT and XPATH: A Guide to XML Transformations*. Prentice Hall, 2002.

[50] Jesse James Garrett. AJAX: A New Approach to Web Applications, February 2005.

[51] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, 1991.

[52] The Open Group. X Window System (X11R6) protocol, 1999.

[53] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of IEEE Pervasive Computing and Communication (PerCom)*, 2003.

[54] Erik Hatcher and Otis Gospodnetic. *Lucene in Action*. Manning Publications, 2004.

[55] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[56] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

[57] Waldemar Horwat. ECMAScript 4 netscape proposal.
http://www.mozilla.org/js/language/es4/index.html.

[58] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[59] Jason Hunter. *Java Servlet Programming, 2nd Edition.* O'Reilly Media, 2001.

[60] NetVention Inc. Web Statistics, Feb 2006. http://www.netvention.com/webstats.php.

[61] ECMA International. *ECMAScript Language Specification*, December 1999.

[62] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley Professional, 2004.

[63] Myeong-Wuk Jang. *Efficient Communication and Coordination for Large-Scale Multi-Agent Systems.* PhD thesis, University of Illinois at Urbana-Champaign, 2006.

[64] Richard Jones and Rafael D Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[65] Mik Kersten and Gail C. Murphy. Atlas: a case study in building a Web-based learning environment using aspect-oriented programming. *ACM SIGPLAN Notices*, 34(10):340–352, 1999.

[66] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[67] Wooyoung Kim. *ThAL: an Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[68] WooYoung Kim, Rajendra Panwar, and Gul Agha. Efficient compilation of call/return communication for actor-based programming languages. In *Proceedings of the Third International Conference on High Performance Computing, pp 62-67, Trivendarum, India, IEEE Computer Society*, 1996.

[69] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Pattern languages of program design 2*, pages 483–499, 1996.

[70] Rasmus Lerdorf and Kevin Tatroe. *Programming PHP.* O'Reilly Media, 2006.

[71] Netcraft Ltd. August 2006 Web Server Survey.
http://news.netcraft.com/archives/2006/08/01/august_2006_web_server_survey.html.

[72] Wai Yip Lum and Francis C. M. Lau. A context-aware decision engine for content adaptation. *IEEE Pervasive Computing*, 1(3):41–49, 2002.

[73] Eric Meyer. *Cascading Style Sheets: The Definitive Guide.* O'Reilly, 2000.

[74] Microsoft Corporation. Microsoft ASP.net. http://www.asp.net/.

[75] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18:657–673, August 1992.

[76] Sape Mullender, editor. *Distributed Systems.* Addison-Wesley and ACM Press, 2 edition, 1993.

[77] NextApp, Inc. Echo2. http://www.nextapp.com/platform/echo2/echo/.

[78] OneState.com. Global usage share mozilla firefox has increased according to onestat.com, July 2006. http://www.onestat.com/html/aboutus_pressbox44-mozilla-firefox-has-slightly-increased.html.

[79] Oracle Corporation and Akamai Technologies Inc. Edge Side Includes (ESI) Overview, May 2001. http://www.akamai.com/en/resources/pdf/esioverview.pdf.

[80] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[81] Sergio Pereira. Developer notes for prototype.js - covers version 1.4.0, 2006.

[82] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[83] David S. Plat. *Introducing Microsoft .Net, Third Edition* . Microsoft Press, 2003.

[84] Klaus Pohl, Gnter Bckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, 2005.

[85] Arno Puder. A code migration framework for AJAX applications. *Distributed Applications and Interoperable Systems, 6th IFIP WG 6.1 International Conference, DAIS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, LNCS Volume 4025:138–151, 2006.

[86] Arno Puder. XML11 - an abstract windowing protocol. *Science of Computer Programming, SPECIAL ISSUE: Principles and Practices of Programming in Java (PPPJ 2004)*, 59(1-2):97–108, 2006.

[87] Alex Russell. After AJAX: Low-latency data to (and from) the browser, March 2006. O'Reilly Emerging Technology Conference.

[88] Sun Microsystems, Inc. Java Technology. http://java.sun.com/.

[89] Sun Microsystems, Inc. Simplified guide to the Java 2 Platform, Enterprise Edition, 1999.

[90] Henri T. J4P5: Javascript 4(for) Php 5. http://j4p5.sourceforge.net/.

[91] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. In *Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001)*, pages 236–255, June 2001.

[92] Prasanna Thati, Po-Hao Chang, and Gul Agha. Crawlets: Agents for high performance Web search engines. *Mobile Agents : 5th International Conference, MA 2001 Atlanta, GA, USA, December 2-4, 2001. Proceedings*, LNCS Volume 2240, 2001.

[93] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference.* IEEE Press, October 2003.

[94] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Services Infrastructure (OGSI) Version 1.0, 2003. Global Grid Forum.

[95] Abhay Vardhan and Gul Agha. Using passive garbage collection algorithms for garbage collection of active objects. In *Proceedings of the 6th International Symposium for Memory Management, pp 106-113, Berlin, June 20-21*, 2002.

[96] Carlos A. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[97] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. *Memory Management: International Workshop IWMM 92 St. Malo, France, September 1719, 1992 Proceedings*, LNCS Volume 637, 1992.

[98] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. A metaobject framework for QoS-based distributed resource management. In *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE '99)*, December 1999.

[99] Steve Vinoski. Web Services interaction models, part 1: Current practice. *IEEE Internet Computing*, 6(3):89–91, 2002.

[100] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.

[101] Wen-Tau Yih, Po-Hao Chang, and WooYoung Kim. Mining online deal forums for hot deals. In *Proceedings of 2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI0́4)*, 2004.

[102] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268, New York, NY, USA, 1986. ACM Press.

Author's Biography

Po-Hao Chang was born in Taipei, Taiwan on May 28, 1973. Being a son of a computer system engineer, he got his first computer, an Apple II, at the age of 11; since then the computer had become indispensable to his life. He participated the founding of the first computer club in Taipei Municipal Chien Kuo High School, where he graduated in 1991. Po-Hao enrolled in the Department of Computer Science and Information Engineering at National Taiwan University in the fall of 1991. He joined the Open System Integration Laboratory led by Prof. Jie-Yong Juang in his junior year, and participated the design and implementation of system software for an early PDA. He received his Bachelor of Science Degree in 1995.

After serving his military duty in the Taiwan Army, Po-Hao enrolled in the Ph.D. program in the Department of Computer Science at the University of Illinois at Urbana-Champaign. Initially he was doing research on coprocessors and processor arrays under Prof. Benjamin Wah's supervision. In 1999, Po-Hao joined the Open Systems Laboratory directed by Prof. Gul Agha, where he did research on software systems and architectures related to the fascinating World Wide Web, including Web engineering, data gathering and information retrieval. In 2005, he co-founded *dealocus.com*, a commercial site providing real-time deals information by automatically gathering, aggregating and analyzing information on the Web. In 2006, Po-Hao joined RiverGlass, Inc., a start-up company specialized in analytic tools for a wide range of data sources, including the Web.