Hot Swapping and State Persistence

A Framework for State Transfer

By

Gerardo Reynaga A.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

8 September 2004

© Copyright
2004, Gerardo Reynaga A.

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97457-X

Our file Notre référence

ISBN: 0-612-97457-X

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Acknowledgments

I read in many thesis the way people thank other people for keeping them sane during the process of writing their thesis. I was not sure why was so important to recognize people that keeps you sane during this process, even when you do not know that you are going insane. Now, I understand the importance of it, and I follow suit...

I would like to thank sincerely my supervisor, Dr. Tony White for all his help, advice and guidance. He has given me the opportunity to continue and complete this work. His camaraderie, support and ideas are very much appreciated. A sincere thanks goes to you.

Some say mountain climbing is an extreme sport; well, finishing a degree in parallel with your partner it is more extreme than going to Mount McKinley, believe me I have done both and I can compare. Regardless of how much we pushed each other to the edge, we were tied up to the same rope eventually stopping each other from falling into the crevasses along the way. Kamilla, *kærar þakkir*, Ég elska þig. I thank you deeply.

I want also thank Miguel Vargas and Juan Pablo Zamora for their support, ideas and excellent talks ranging from technology to politics. A special thanks goes to Javier Govea for his stimulating discussions on the subject. Thanks to Ivan Araujo for his feedback on the pattern. Furthermore thank all of you "members" of the Little Hispanic SCS community: Luis Rueda, Monica Caniupán, Loreto Bravo, and Leo Bertossi. I would like to thank my friends at the School of Computer Science. For helping me with the administrative issues students have to endure, thanks to Nicki and Sandy. I wish to thank Lisette Lopez for her unorthodox ways of support.

Finally, I would like to dedicate this thesis to God and my Parents, for teaching me what a University cannot. My love and a very special thanks goes to them. Muchas gracias Papás con mucho cariño para ustedes.

Gerardo B. Reynaga Anguiano

I always liked the following poem and it reflects, somehow, what one goes through when writing a thesis.

Ithaca

As you set out for Ithaca hope your road is a long one, full of adventure, full of discovery. Laistrygonians, Cyclops, angry Poseidon - don't be afraid of them: you'll never find things like that on your way as long as you keep your thoughts raised high, as long as a rare excitement stirs your spirit and your body. Laistrygonians, Cyclops, wild Poseidon - you won't encounter them unless you bring them along inside your soul, unless your soul sets them up in front of you.

Hope your road is a long one. May there be many summer mornings when, with what pleasure, what joy, you enter harbours you're seeing for the first time; may you stop at Phoenician trading stations to buy fine things, mother of pearl and coral, amber and ebony, sensual perfume of every kind - as many sensual perfumes as you can; and may you visit many Egyptian cities to learn and go on learning from their scholars. Keep Ithaca always in your mind. Arriving there is what you're destined for. But don't hurry the journey at all. Better if it lasts for years, so you're old by the time you reach the island, wealthy with all you've gained on the way, not expecting Ithaca to make you rich.

Ithaca gave you the marvelous journey. Without her you wouldn't have set out. She has nothing left to give you now. And if you find her poor, Ithaca won't have fooled you. Wise as you will have become, so full of experience, you'll have understood by then what these Ithakas mean.

Konstantin Kavafis

Contents

Acknowledgments	i
1 Introduction	1
1.1 Context and Motivation	2
1.2 Problem description	3
1.3 Expected Contributions	3
1.4 Overview	4
2 Literature Review	5
2.1 Overview of Hot Swapping	5
2.2 Overview of Theoretical Work	11
2.3 Abstract Representation of Hot Swapping Frameworks	12
2.4 Application Layer	14
2.4.1 CONIC	15
2.4.2 PODUS	17
2.4.3 DYMOS	20
2.4.4 POLYLITH	23
2.4.5 Argus	26
2.4.6 Durra	29
2.4.7 On-Line Software Version Change	31
2.5 Middleware Layer	32
2.5.1 Dynamic Software Updating. (DSU)	32
2.5.2 Dynamic Classes for C++	36
2.5.3 Dynamic Java	38

2.5.4	Dynamic Virtual Machine	39
2.5.5	Java S-Modules	40
2.5.6	Hot-Swapping Technique for Object-Oriented Paradigm	41
2.5.7	SwapBox: a Hot-Swapping Framework for Swappable Java Beans	42
2.6	Operating Systems Layer	44
2.6.1	DAS	44
2.6.2	K42	46
2.6.3	Kea	48
2.7	Summary	50
3	A Framework for State Transfer (FST)	53
3.1	State Transfer	53
3.2	Proposed Guidelines for State Transfer	54
3.2.1	State Loading Policies	57
3.3	State Transfer Pattern	57
3.4	Proposed Design for State Transfer	61
3.4.1	Design Assumptions and Boundaries	61
3.4.2	Goals	62
3.4.3	Approach	63
3.4.4	Storing the State	66
3.5	Summary	70
4	HTTP Servers	71
4.1	An HTTP Server	71
4.2	Apache and its Hot Swapping Features	73
4.3	The Anatomy of Apache HTTP Server	74
4.3.1	Apache's APR	75
4.3.2	Apache 2.0	76
4.3.3	The Module's Anatomy	81
4.3.4	Request Processing	86
5	Implementing State Transfer in Apache	89
5.1	Overview	89

5.2	FST Implementation	91
5.2.1	Loading and Saving a Module's State	92
5.2.2	The State Machine of a Child Server	96
5.3	Parsers	97
5.4	Code Samples	98
5.5	Performance Evaluation	103
6	Conclusions	109
6.1	Summary	109
6.2	Contributions of the Thesis	110
6.3	Future Work	110
	Appendices	112
A	The Java Virtual Machine	112
B	A Bit of History on Apache	114
C	Apache's web server restart cycle	116
D	XML Schema	119
E	Testing Environment	126
	Bibliography	128

List of Tables

2.1	A general comparison of the reviewed systems	7
2.2	General architectural features of some of the reviewed systems	8
2.3	State transfer features on relevant hot swapping techniques.	52
4.1	Similitude between Argus and Apache when replacing a component. .	74
5.1	Child servers initialisation and death under different conditions. . . .	92
5.2	Simple static test for mod_server and mod_server_s	105
5.3	Automatic static test for mod-A and mod-A' . Reply rate.	108
5.4	Automatic static test for mod-A and mod-A'	108
5.5	Average time comparison for SAX and Gdome implementations. . . .	108
B.1	Netcraft survey comparing the top four most popular HTTP servers [39].	115

List of Figures

2.1	Abstract architecture of reviewed hot swapping technologies.	13
2.2	Hot swapping characterisation by their domain approach.	14
2.3	The architecture for Conic.	16
2.4	Calling a procedure using an interprocedure (proxy).	19
2.5	DYMOS runtime environment.	21
2.6	Polyolith’s software bus.	26
2.7	Durra run-time environment.	30
2.8	An example of code relinking [25].	33
2.9	An example of type renaming [25].	34
2.10	Dynamic Software Updating.	36
2.11	Two-level mapping in dynamic C++ classes [27].	37
2.12	DAS address space change caused by calling an object [20].	45
2.13	Stages in replacing an instance A of object j by instance B [4].	47
2.14	Portal invocation and portal remapping in Kea [53].	49
3.1	A component is formed of one or more modules.	63
3.2	A Hot-swapping Application	64
3.3	State machine for a module with state transfer functionality	65
3.4	Sequence diagram for a module with state transfer functionality	66
3.5	Example of a saved state in XML data representation.	69
4.1	Browser-HTTP server interaction sequence diagram.	72
4.2	Apache implementation of the leader/follower pattern.	79
4.3	Class diagram of a module’s structure.	84
4.4	Overview of a module’s anatomy.	84

4.5	Hooks available to modules at different stages of the child life cycle. .	86
4.6	Request-response cycle in Apache [21]	87
5.1	Loading a state after an <code>httpd -k start</code>	93
5.2	Saving a state after an <code>httpd -k restart</code>	94
5.3	Loading and saving a state after an <code>httpd -k graceful restart</code>	95
5.4	Child Server State Machine.	96
5.5	Standard Apache Module Structure.	98
5.6	Available hooks for modules.	99
5.7	Child server initialiser including <code>get_state()</code> and <code>x_child_exit</code>	99
5.8	Including directives as policies for state transfer in the <code>httpd.conf</code> file.	100
5.9	Partial listing of <code>get_state()</code> function.	101
5.10	Partial listing of <code>trace_state()</code> function.	102
5.11	Partial listing of <code>x_child_exit()</code> function.	104
5.12	Example of a saved state in XML data representation.	105
5.13	Automatic static testing comparison for <code>server-s</code> and <code>server-status</code> .	107
A.1	The Java Virtual Machine.	113
C.1	Apache's restart cycle [21].	118
E.1	Child server configuration parameters.	127

Chapter 1

Introduction

With the evolution of computational power the expectations on software applications are increasing. There is, at times, a growing tendency to have complexity and functionality not always wanted or needed by users. Thus, having an architectural design that gives room for changes is becoming more needed. At the same time change, evolution and reconfiguration are one of the core problems of software development.

A common approach to cope with software changes is to shut-down, update, and then bring back on-line the application. By using this approach we, first, lose the *state* of the application when the application is shut down. Second, on-going services and processing in progress at the time of the shutdown is interrupted. In some cases this is acceptable, however currently this is less and less the case. The loss of state and service interruption may translate into compromised safety and economical losses, just to name some potential effects. Non-stop computer programs, such as aviation control systems, financial and banking transaction applications, telephone switches, telecommunications, are applications that require the capability of extension, replacement and change while they are running. This capacity can improve performance, security, availability, fault tolerance, reliability and evolution on a system, among other advantages. One way to provide this ability to grow is by allowing the system to be hot swappable while the application is still running.

Software hot swapping is often found in the literature as dynamic software updating, dynamic (software) reconfiguration, dynamic code replacement, on-line software version change, dynamic configuration, run-time reconfiguration, on-the-fly program

modification, runtime software evolution. In the context of the present work the terms, software hot swapping, dynamic software update and dynamic configuration are equivalents. The concept implies that the system must be able to add, delete or replace components without affecting other components of the application. Software hot swapping avoids halting the running system, recompilation of the whole system, loading the compiled code and losing its state. A system which supports dynamic updating is often seen as one in which the software version can be changed while the system continues to provide service. In other words, parts of the code of the executing software can be changed while the system is operating.

1.1 Context and Motivation

Change is for software (in most cases) as death is for humans: certain and unavoidable. Despite technology development software programs are, in general, prone to errors and they do not satisfy every anticipated need. It is well known that a computer program undergo a number of changes during their life cycle. Some of these changes are done to correct bugs, some others to add functionality. The present work is mainly motivated by the demand of software updates, bug fixes, change on requirements, extensions, etc. in a few words: software change.

Dynamic software configuration is not a new problem, as we will discuss the work that has been done in Chapter 2, there are many existing solutions, frameworks and architectures. By doing an introductory survey we can bring together the underlying principles and characteristics of hot swappable applications. We believe that we can apply these principles and concepts to existing systems that support loadable modules.

A good number of dynamic configuration approaches come from the realm of distributed systems and fault-tolerant systems. Our work will focus on hot swapping systems running on a single computer. The test case we will be working on provides a set of characteristics that can facilitate the implementation of dynamically updating the system's modules.

1.2 Problem description

A running program that is *correct* is made up of a collection of elements, *i.e.*, components and modules. We want to replace one or more of these elements at run-time leaving the system in the same *correct* state as it was before the replacement. Replacing a module should preserve, in case needed, as much of the original state as possible. That is, the process behaves as if it had been executing the new version of the module since the beginning from its initial state.

In order to define program correctness we borrow the term *correct* used by Gupta [11]. A program, Π , is correct if it can execute properly and produce expected results for inputs provided to it. That is, the behaviour of Π is well understood for its input domain.

1.3 Expected Contributions

Although much research has been done on dynamic configurable systems there are still issues to be worked on. This thesis focuses on state transfer between versions of the same module when a hot swap (replacement) is done.

In this work, we aim to develop a framework for hot swapping modules on a computer program as it runs, based on principles of previous successful approaches to dynamic configuration. We aim to show that computer programs with key essential characteristics for dynamic configuration can support hot swapping if our framework is applied properly. To substantiate this claim, we have reviewed existing literature, carefully analyzing techniques used in prior work. We will apply the developed framework to a large-scale application case study.

In synthesis the expected contributions are as follows:

- Present a comprehensive review of past and recent dynamic software updating techniques and mechanisms.
- Characterise the hot swapping problem across a number of different dimensions.
- Present an abstract representation of hot swapping architectures
- Propose an architectural framework for describing hot swapping research.

- Introduce a State Transfer pattern.
- Develop a technique for transferring state from the old version of a module to the new version.
- Implement this framework and prove its concept and usages in real computer programs.
- Present a performance assessment, in order to demonstrate the easiness of implementation of the approach and minimal intrusion.

1.4 Overview

In Chapter 2 we start by laying out the essential characteristics of hot swapping applications. We then describe some of the past and present approaches to dynamic reconfiguration. At the end of the chapter we analyse and identify techniques that are useful for our research. Chapter 3 describes State Transfer; its implications and its significance. We outline our approach in this chapter. The remainder of the chapter presents a state transfer design pattern and proposes the use of XML schema for state persistence. In order to introduce our implementation Chapter 4 describes the concepts behind an HTTP Server and the underlying model which Apache provides for hot swapping. In Chapter 5 we introduce the test case environment, we will describe how our approach is applied to a real application, this chapter ends with a summary of our findings. Contributions, conclusions, future research and open problems will be discussed in Chapter 6.

Chapter 2

Literature Review

One way to provide the ability of modification and extension of a system while it is running is by allowing the system to have dynamic configuration. This chapter reviews the relevant work that has been developed in the field of dynamic configuration, providing an understanding of hot swapping strategies, their advantages and disadvantages. We begin by presenting an overview of the reviewed systems. Next, we review existing techniques utilized to approach software hot swapping. The chapter finishes with a section on work carried out in the hot swapping field of programming languages and operating systems, followed by a chapter summary of meaningful features in dynamic configuration.

2.1 Overview of Hot Swapping

In general, computer systems are constructed from a set of component parts. A component is typically the unit of configuration. These units have a mapping to one or various pieces of code in a programming language. Most of the systems are initialised with a predefined component configuration or they are built in ahead of time with all possible component configurations. Moreover, these systems are expected to have a long productive life and run for several years. However it is not uncommon for systems to be subject of changes and maintenance activities in order to achieve the expected productive life.

The concept of hot swapping refers to a computer application that is able to add

or replace components without affecting other components of the application. Hot swapping avoids halting the running system, rebuilding the whole application, and losing state when the application is restarted. It should introduce as little impact as possible, or no impact at all, on an executing system.

In response to system maintenance, growth, and requirements for long life techniques, architectures and frameworks have emerged which provide systems with the ability to sustain changes in a friendlier way. Kramer and Magee [29] point out that system configuration can be done *statically* or *dynamically*.

The *static configuration* approach to designing a system from its configuration specification is to have a library of components that will be included at building time *e.g.*, compiling then loaded into the system. They define this procedure as complete since all components of the system are configured at the same time. If changes are required, the complete system must be stopped and rebuilt.

Dynamic configuration provides an incremental dynamic updating that allows unpredicted modification and extension without rebuilding the entire system. The changes are done without stopping the unaffected parts of the system. Changes are initiated by a set of specifications such as the introduction of new components, modification of existing ones, and provision of different interconnection patterns. These specifications are validated and a configuration manager generates the required operating system calls to load the required code, delete and create components, and remove and set up connections.

The review in this chapter focuses mainly on the work done in the field of *dynamic configuration*. Table 2.1 summarises past work on hot swapping approaches. Each approach we review is presented, in this table, in approximate chronological order, the top being the oldest, and in the following sections it is broken down by layer. The systems, mechanisms and approaches are the following: DAS [20], DYMOS [33], Conic [29], Durra [38, 6], Argus [8], PODUS [46], Polyolith [41], On-line Software Version Change (OSVC) [13], VINO [47], Dynamic C++ classes (DC++) [27], Dynamic Java (Djava) [2], S-Module Design for Software Hot swapping (JS-Modules) [15], Runtime support for type-safe dynamic Java classes (DVM) [35], Software Hot-swapping Techniques for Upgrading Mission Critical Applications on the Fly (S-Modules Framework) [3], SwapBox [50], Dynamic Software Updating (DSU) [26], K42 [4].

System	Type	Layer	OS	Language	Portability	OO
DAS	OS	OS	DAS	Procedural	PDP-11	No
DYMOS	Integrated environment	App	Unix	StarMod	Med	No
Conic	Language	App	Distributed	Pascal	No	No
Durra	System Language System	App	Unix	Not spec.	No	No
Argus	Language-System	App	Distributed	Clu	Limited	Legacy code
PODUS	Procedural Framework	App	Unix	Procedural	Yes	No
Polyolith	System	App	Distributed	C	No	No
OSVC	Programming	App	Unix	C	No	No Provisions
Kea	OS Kernel	OS	Kea	C	Intel-based	No
VINO	Extensible OS	OS	VINO	C++	Limited	Yes
DC++	Programming Environment	MW	Unix	C++	Yes	Yes
DJava	VM	MW	Any OS	Java	Changes to VM to any VM	Yes
JS-Modules	VM	MW	Any OS	Java	Changes to VM to any VM	Yes
DVM	VM	MW	Any OS	Java	Changes to VM to any VM	Yes
S-Modules Framework	VM	MW	Any OS	Java	Changes to VM to any VM	Yes
SwapBoxJBeans		MW	Any OS	Java	to any VM	Yes
DSU	Programming Environment	MW	Unix	Popcorn	Yes	No
K42	OS Kernel	OS	K42 Linux	C++	Yes	Yes

Table 2.1: A general comparison of the reviewed systems

We fill in the table with the following criteria, six criteria are used to categorise the systems. The *System* column contains the identifying names of the systems, mechanisms and/or approach to hot swapping. The *Type* column classifies the systems with regard to the approach these take to hot swapping. The *Layer* column is subdivided into three major classifications: Operating System, Middleware (MW), and Application (App) and represents where the technique approaches hot swapping. For the purpose of this table we differ from the classic middleware definition, we refer to middleware as the mechanism that facilitates the development of an application. The application class comprises a larger set of mechanisms to hot swapping. The *OS* column contains the operating system on which the application runs. The *Language* column holds the type of language used in a particular approach. Some mechanisms are fully portable (Virtual Machines) others are subject to compilers or even hardware, the *Portability* column depicts the flexibility in terms of portability. The *OO* column describes if the mechanism is structured in an object-oriented fashion. These criteria are used to provide a summary road map of hot swapping approaches explained in the sections that follow.

	Validity	Component State	Changing Interface [API]
Conic	Checks change specifications	Quiescent	No changes
Podus	Checks syntactic and semantic dependencies	Non-active proc.	Supported
DYMOS	Integrated environment	Idle	Supported
PolyLith	Aid by some programmer intervention	Quiescent	Supported
Argus	Set of correctness conditions	Quiescent	No changes
Durra	Executives monitor reconfiguration conditions	Quiescent	–
DSU	TAL and user	Active	Supported
OSVC	State transfer	Quiescent	Supported
IBM's K42	Mediator	Quiet state	Best common format

Table 2.2: General architectural features of some of the reviewed systems

Table 2.2 lists some of the reviewed systems describing the general architecture characteristics.

Validity. Validity describes the method used to ensure that the introduced new version or component does not affect the stability of the system. In the case of Conic the configuration manager validates change specifications against the system descriptor file and produces an updated descriptor. DYMOS has an integrated approach which provides flexibility and intimate knowledge of the running program, it is type-checked. Dynamic Software Updating (DSU) works with Popcorn, a prototype compiler from a safe-C language and TAL (typed assembly language). In On-line Software Version Change (OSVC) the system checks whether a state transfer can be done or not, if it can be done then it is considered a valid swap. A mediator is used during the hot swapping process in K42. This mediator performs the replacement and brings the object to a quiescent state.

Component State. Most of the techniques require the component to reach certain state before hot swapping. A component can be in quiescent or active state in order to do a swap. Quiescent, idle, non-active, and passive state are all used as synonyms across the literature. A component in quiescent state is not performing transactions and any transaction it has already initiated have terminated. For example, as part of the command to update a procedure DYMOS provides a *when-condition* directive in order to carry on the update. In Polylith, reconfiguration is done at reconfiguration points. In Argus, if a quiet state is not reached then the subsystem is forced to reach it. DSU does not require a particular state to be reached since the problem of timing during the update is coded within the component. The framework proposed in Chapter 3 performs the hot swapping after the component has finished its transactions and does not initiate new transactions.

Changing Interface. Some systems provide means to support changing the interface others do not allow interface changes. DYMOS provides a wide range of changes: interface type, function and data definitions, and loop bodies. Polylith has a software bus (mediator) that takes care of changes on interfaces.

In summary, the essential characteristics of hot swapping applications are as follows:

- *System availability.* The application provides continuous and uninterrupted service.
- *Modular design.* Hot swapping applications are designed in a modular or component-based architecture based on the concept of separation of concerns. That is, components and interfaces comprising the application are well defined in order to facilitate hot swapping. The modular structure also provides data structures local to the component. Well-defined interfaces are necessary for interposition (*e.g.*, proxy, mediator, broker, etc.) and replacement of components.
- *State Transfer.* An updating system is able to transfer the state from the old module being swapped-out to the new version.
- *Program correctness.* The hot swapping application must preserve program correctness at all times.
- *Flexibility and maintainability.* The hot swapping application offers means to replace a module with a new one that is implemented differently. The replacement takes place at runtime.
- *Extensibility.* A hot swapping system must evolve to take on tasks not anticipated in their original design.
- *Correct timing.* Applications provide mechanisms to ensure well-timedness and that the update does not result in inconsistent state due to race conditions or dead locks.
- *Rollback-enabled.* Some updating systems offer roll back capabilities in case a flaw or a buggy update is detected following a hot swap.
- *Hardware independent.* The application should not require special-purpose hardware or depend on hardware exploits.

- *Operating System independent.* Although this is a difficult goal to achieve some of the existing hot swapping applications provide OS flexibility.

2.2 Overview of Theoretical Work

Gupta and Jalote [11] describe a formal framework for hot swapping. In this framework the authors work with programs, processes and process state. The definition of a program pertains to the program model used. For example, in procedural languages a program maybe modelled as a sequence of statements. A process is a program in execution. A process has two components; the code being executed and a state. The code of the program tends to remain constant throughout the lifetime of the process and the state changes according to a state transition function which specifies the next state depending on the current state and the program being executed. This function forms the execution model. s_{π_0} is defined as a unique initial state for a program Π , this is the state of a process executing Π at the very beginning of its life. A state is a complete characterisation of a point in the lifetime of a process.

Definition 1. A state is said to be reachable state of a program Π if and only if a process executing Π from its initial state s_{π_0} can reach s at some time for some inputs.

There is no restriction on which state must the process start its execution. If a process executing the program Π is started from a reachable state of Π , then the behaviour of the process is identical to the behaviour if the state was reached by the process starting from s_{π_0} .

The author's formal definition of hot swapping is as follows:

Definition 2. A hot swap change from program Π to Π' at time t using state mapping S , in a process P (executing Π) is equivalent to the following sequence of steps:

- 1.- P is stopped at time t in state s
- 2.- The code of P (until now Π) is replaced by the program Π' , its state is mapped by S and P is then continued from state $S(s)$ and with code of Π' .

This definition applies to an instantaneous hot swap rather than an incremental process. The authors believe it is a cleaner and more general formalization approach to updates. Incremental changes can be modelled as a series of instantaneous hot swaps.

Definition 3. A hot swap in the process P from Π to Π' at time t , in state s , and using state mapping S is valid if after the change, P is guaranteed to reach a reachable state of Π' in a finite amount of time.

Thus, a hot swap is valid, if after certain transition period after the change, the process starts behaving as if it had been executing the newer version of the program since the beginning from its initial state. The authors maintain that in order to have a meaningful state mapping the developer has to define the state mapping function S using his or her semantic knowledge of the two versions of the program. For example, variable x in Π is renamed to y in Π' , this information is available only to the developer.

The authors continue their theory work to show that the problem of correct timing (validity) is, in general, undecidable. Thus, leaving the task of identifying correct timing conditions for a hot swap to programmers.

Malabarba *et al.* [35] present a formal model for object oriented paradigms. In this work the authors formalize the notion of class, interfaces, inheritance, composition and dependencies among classes in Java.

2.3 Abstract Representation of Hot Swapping Frameworks

The abstract architecture of hot swapping techniques presented below is the result of the analysis done of the reviewed dynamic updating techniques. Figure 2.1 depicts the abstract architecture for most of the reviewed hot swapping technologies. The infrastructures usually consists of a *Configuration Manager* and one or more *Components*. Each component is comprised of one or more modules, and often a *proxy* element. Proxies provide: means to communicate with other components, services

Figure 2.1: Abstract architecture of reviewed hot swapping technologies.

for interface changes, and they solve the referential problem; as discussed by Gang [3] and Lei [50]. The referential problem can be explained with the following example. Client *A* uses services from component *B*. If we replace component *B* with *B'*, *A* has to change the reference from *B* to *B'*. Client *A* must know the change. If client *A* has passed *B*'s handle to client *C*, then *C* will retain the reference to *B* even if *A* has changed the reference to *B'*, unless *A* informs *B* about the change. The proxy hides the handle from its clients and it deals with those clients on behalf of component *B*. When the update is done, the change is transparent for *A* and *C* since service requests are done through the proxy. Component and proxies communicate through interfaces. Each interface that a component provides to the outside world looks like a “lollipop” (a circle on a stick) in the diagram. Each interface that the component requires from some other component in the system looks like a half-circle on a stick [22].

The right hand side of Figure 2.1 depicts the architecture after an update is carried on. Component *B* is removed, and module *A* changed into *A'*, and module *C* changed into *C'*. A new communication link is established between component *A* and *C*. The elements enclosed in dashed boxes (*ST*) apply to certain technologies. Some hot swapping techniques have an *ST* element within the component to take care of the state transfer.

Figure 2.2: Hot swapping characterisation by their domain approach.

The following sections present a detailed description of the systems discussed in §2.1. These are classified according to the layer each technique approaches hot swapping. Figure 2.2 characterises the layers for the reviewed techniques. The dimension criteria followed in this figure correspond to the column *Layer* described for Table 2.1.

2.4 Application Layer

Techniques grouped in this layer provide some of the following features: fully integrated environment, proprietary languages, or even strong hardware dependencies. Although all the techniques are research projects, we considered this layer would be more difficult to adapt to a more widely and common use.

2.4.1 CONIC

CONIC was developed by J. Kramer, J. Magee, M. Sloman, A. Lister, and colleagues at Imperial College and it was designed with the objective of supporting the construction of dynamic reconfigurable distributed systems. It offers a configuration language which helps describing systems with interconnected modules; it specifies, as well, a language for programming module types and a distributed operating system to support the execution of CONIC systems *i.e.*, a run-time environment [29]. CONIC separates structure from component behaviour.

CONIC programming language is based on Pascal with message passing added. The message transactions are unidirectional and bidirectional request-response type. Modules have interfaces defined as entry and exit ports in which messages can be received or sent, respectively. Entry and exit ports correspond to the required and provided interfaces of our abstract architecture respectively, see Figure 2.1. Running modules only communicate by message passing. Modules can be compiled independently since they do not make references to outside objects; module types are configuration independent.

Two of the configuration features that CONIC has are: a system configuration specification and a configuration change specification. Module types, from which the application will be constructed, are identified by a system configuration specification. This specification details the interconnections of the instances of the module types. Change specifications and configuration specifications are done separately in order to facilitate dynamic reconfiguration. That is, changes are done by editing a configuration specification and then rebuilding the system.

One of CONIC's tools is a Configuration Manager. This configuration manager receives requests to change the system and then translates them into commands for the distributed run-time system to enforce the reconfiguration. It allows building a distributed program by defining a set of modules and their port interconnections.

In a dynamic change the following steps are taken: A stop action is done on all modules that must be locked, leaving them in a passive state. The configuration manager removes the module, and unlinks the connections (exit and entry ports). Then it creates the new module, introduces the new node (waits for the node to be passive), links and copies the state, and at the end it starts the passive set of

Figure 2.3: The architecture for Conic.

modules [30]. A passive set is the set of modules that must be stopped according to: preconditions of the link, unlink and remove statements. The passive and quiescent states, connection initialization and finalization code aid the application to preserve application consistency [31]. Note that the state mentioned here is the state from the OS point of view: active or passive. A component in active state can initiate, accept and respond to transactions. A passive state accepts and responds to transactions but it does not initiate new transactions. Figure 2.3 depicts the architecture of Conic. System Specification and Configuration Changes are fed to the Configuration Manager and the Validation elements, respectively. The Configuration Manager translates the instructions into commands and passes them to the OS, producing a New system descriptor.

Despite its strong and flexible environment for dynamic configuration, Segal and Frieder [46] find two potential problems with CONIC:

- It has its own programming language, compiler and run-time environment. The learning curve can be steep.
- It does not impose a client-server relationship on a distributed program, but it forces communication through a fixed set of ports to a fixed set of module. This prevents a module from choosing where to send a request at run-time.

We add to these:

- Although Conic has a full set of instructions to manipulate modules and its communication links, the system does not provide tools or mechanisms to save and load the state of a module being replaced. In some cases if the state is not preserved the program may not produce the expected output.

2.4.2 PODUS

PODUS was conceived and developed by Ophir Frieder and Mark E. Segal at the University of Michigan and later on at Bellcore [17]. Frieder and Segal undertook the problem of dynamic reconfiguration and updating programs written in procedural languages. Their approach requires that the programs are written in a top-down design methodology.

In their approach a procedure is considered a unit for replacement. After replacing all old procedures with new procedures it is considered that the program has been updated [17, 46]. Their approach is as follows:

- Before the start of an update the procedure is compiled, linked and loaded on the computer
- Once the new version has been loaded an update command is required by the user to initiate the update
- The updating system halts the program and checks the state of the procedures on the run-time stack
- The criteria for updating a procedure is based on system-computed syntactic criteria and user-provided procedural semantic dependencies
- Only procedures that are syntactically and semantically non-active can be updated

Frieder and Segal [17] describe dependencies as follows. A *syntactic dependency* can be illustrated with the following example. Lets say that a procedure P_1 can call a set of procedures $[A, \dots, D]$. This set of procedures is called the syntactic dependency function. All procedures that are on the run-time stack and/or can be called by the new procedure P_{new} are considered syntactically active. *Semantic dependencies* are

the links that are not detectable from the program's syntax and these are defined by the programmer as a non-syntactic dependency. For example, a group of procedures A , B and C that control different parts of a system may not call each other, but still depend on each other's actions. There are no guidelines for this user's semantic specification. If a procedure A is active and is semantically related to procedure B then B is considered semantically active. Semantic dependencies allow concurrent update of procedures since a procedure P can only be updated if it is inactive and all procedures that might be called from P are inactive.

If a procedure cannot be updated then the system waits until it reaches the non-active state. An old version of a procedure P_i can call P_j or P'_j , depending on whether an update of P_j has already occurred. On the other hand, a new version of P_i , P'_i , can call only the new version of procedure P_j . PODUS determines active procedures by examining the state of the program's runtime stack. After all procedures are updated then the program runs using the latest version.

If there are changes in the interfaces of the procedures these are handled with what is described as *interprocedures*. Interprocedures translate arguments and return values required by the new procedure specification to talk to the old procedure. In other words the interprocedure is informed of both -old and new- specifications. Thus, the interprocedures are cognizant of the existence of multiple versions of a procedure running while individual procedures are not. These interprocedures are user-defined [46, 17].

Mapper procedures are -as well- user-written procedures, defined by Frieder and Segal as *mprocedures*. Similar to interprocedures these *mprocedures* translate static data into a format suitable for the new version of the procedure. Distinct from interprocedures, which are invoked with each procedure call, mprocedures are called only at update time for each procedure that requires it; that is, when the new version of the procedure is invoked for the first time [17]. Mapper procedures are equivalent to state transfer (ST) element in our abstract architecture.

Interprocedures and mapper procedures help retaining consistency during the update. In order to support distributed environments PODUS uses remote procedure calls [46]. Figure 2.4 depicts the use of interprocedures when invoking new procedures from procedures that have not been updated. Interprocedures are what we call proxy

Figure 2.4: Calling a procedure using an interprocedure (proxy).

element in our abstract architecture (see §2.3); however, in PODUS interprocedures are per procedure instead of per component.

The updating system is implemented by overloading a segmented virtual memory infrastructure. The address-space information contains a version-ID field (among other information) allowing multiple versions of the program to be in separate segments of the program. A binding table exists within each version space for that particular program. Only the procedures in that version space use the binding table. Interprocedures and mprocedures can map calls from one segment to another *i.e.*, cross version space. Addressing is done by mapping the components of these large addresses onto existing registers of conventional virtual memory and data structures [46, 17]. However the use of segmented virtual memory provides flexibility at times at the cost of performance.

PODUS loses some flexibility by doing the updates only on inactive procedures thus caution has to be taken when designing and programming procedures that are long running, long procedural loops, or programs with one main large procedure. This is the reason why the authors suggest following a close top-down design model. A good advantage of PODUS is that it does not force the user to a specific set of development tools. However it imposes a class of languages and a programming style.

State Transfer. State Transfer is accomplished by the use of *mprocedures* and abstract data types. Static data used by a procedure is converted by *mprocedures* into a format suitable for the new version of the procedure. Data accessed by several different procedures (such as global variables or file data) is access through abstract

data types.

2.4.3 DYMOS

Dynamic Modification System (DYMOS) is one of the early systems developed that supports dynamic configuration. DYMOS was developed by Insup Lee [33], it is a fully integrated hot swapping system which includes a file editor, a compiler, a shell (command interpreter), a source code management system and a run-time environment for updating computer programs, written in StarMod language. StarMod is a concurrent language similar to Modula, Mesa or Ada. StarMod supports the concept of module and abstract data types. The updating in DYMOS is done by replacing individual procedures. DYMOS' runtime environment is tantamount to the configuration manager described in the abstract architecture.

As an integrated approach, DYMOS provides a detailed knowledge of the running program. The programmer has control over the updating conditions, ensuring updates are done at the correct time. However this integrated nature of the system brings some drawbacks, mentioned at the end of this section.

There are three general steps to update a module using DYMOS. Step 1 calls up the editor and gets a copy of the module from the source program manager. Step 2 starts the recompilation of the module. Step 3 uploads the object code into memory and modifies the current code image so new calls are handled by the latest version. However the old object code is destroyed when it can no longer be referenced. After the update command is requested the system checks whether the program in memory is consistent to the executable code. If not, the update command is not performed. Figure 2.5 depicts the overall structure of the system.

Coexistence of modules can occur in memory. However, in order to solve inconsistencies the programmer is allowed to specify when a program can be modified by specifying the procedures that should not be running at the time of the update. The specification consists of giving the list of procedures or modules that must be idle (not executed by any process). For example, the update command:

```
update A when B,C idle
```

This command updates A when procedures B and C are not executing.

Figure 2.5: DYMOS runtime environment.

DYMOS allows simultaneous updates of procedures and modules. A procedure is DYMOS basic unit for code abstraction, and a module is the basic unit for data abstraction; a module groups a set of procedures. Procedures and modules can be added or deleted. By editing a module, its specification can be changed, an old data representation can be converted into a new data representation at a desired time; that is, the programmer specifies when the changes take place. The command that follows is an example of several operations under DYMOS:

```
update M,A,B delete C when A,C,M.P idle
```

This command has module *M* and procedures *A*, *B*, and *C*. Procedures *A* & *B* and module *M* are replaced and procedure *C* is deleted when *A*, *C* and *M.P* are not executing. *M.P* is procedure *P* of module *M*. Deleted procedures or modules can no longer be referenced. This deletion depends entirely on the programmer since the operation is indivisible. Apache HTTP server provides a similar functionality by introducing a graceful restart. Processes are not replaced until these finish their activity.

Similarly to the interprocedures used in PODUS (or the proxy described in the abstract architecture §2.3), Lee suggests the use of a parameter-convert routine that

dynamically transforms parameters of the old version into those required by the new version. This routine is part of the new procedure and is only executed when it receives calls for the old version.

One of the main goals of DYMOS is to provide the means to maintain consistency of a program when an update is done.

Lee defined an algorithm for partitioning an update into various smaller changes. Partitioning the change provides the following advantages. First, changes are done incrementally, which is less complex. Second, the fewer procedures and modules are being updated the longer they are available. Third, it might reduce the completion time to update since the so called *when-condition* contains less clauses. The algorithm consists of decomposing the set S of procedures and modules into a sequence of subsets that can be updated separately. He also describes a series of algorithms to find an update precedence between two procedures and modules, to find the sequence of subsets that can be updated separately, and to subdivide existing subsets. He gives, as well, a method for checking consistency after an update command.

The fact that DYMOS is a fully integrated system provides advantages and disadvantages. Among the advantages we note is the detailed knowledge of the running program. It allows changes to module definitions (including type), function, and data definitions. Ensuring update correctness is among the advantages of this hot swapping system. Another advantage is the algorithm for partitioning updates.

Among the disadvantages we note is that DYMOS is language specific (StarMod) with alterations to support dynamic updating, which eventually leads to portability problems. Nevertheless, Lee points out that it would be better to design a new language providing a uniform type conversion feature for the procedure's parameters and local variables, module's variables, files, and exported type variables than having custom solutions.

Portability of the whole system might prove to be a hard task. For example, adapting an editor other than "vi" or "ed" could be difficult. There is no mentioning of how to support distributed programs. Lee noted that a roll back mechanism and a fault tolerant scheme can be embedded into the system. He also realizes that there is a need to measure the overhead and execution time of his dynamic modification system.

State Transfer. The intimate knowledge this approach has of the running program provides a considerable benefit when performing state transfer. The module to be updated can replace: a) the symbol table definitions; b) the symbol table definitions and the object code; or c) the symbol table definitions, object code, and the data. This replacement is done at compile time, in which the symbol table definitions and the new object code are generated. The system allows as well conversion routines to be defined with the new version. The convert routine initialises the variables of the new version. If the convert routine exists then it is called instead of the initialisation code when the module is updated. This approach is similar to the approach we propose in §3, in which the initialisation and conversion are done in the initialisation code.

2.4.4 POLYLITH

Polyolith was first implemented in 1985 [28, 41, 42]. It is a distributed programming system that provides support for system reconfiguration. Polyolith aims to overcome problems that arise from having modules in different languages executing on heterogeneous architectures.

One of the main objectives of Polyolith is to provide an abstract decoupling agent, a *software bus* that wraps communication and data transformation activities. This helps decoupling *interfacing* requirements from functional requirements. The programmer does not pay constant attention to constraints imposed by the architecture, language processor or communication media. In Polyolith these constraints are not ignored but they are isolated for independent handling.

Polyolith supports three general types of reconfiguration:

Geometric This is equivalent to process migration.

Modular Changes to individual modules, implementing different algorithms within the module, for example.

Structural System's logical structure also called modular structure or topology. The programmer does not have to adapt the source programs in order to alter the

interfacing mechanisms. Bindings between modules may change, new modules maybe added, while other modules removed.

In this approach, a module is referred to as a collection of data declarations and individual program units, *i.e.*, processes. A module is equivalent to components described in §2.3. Bindings, or communication channels, connect the interfaces of the modules in an application. Communication channels are given as part of the configuration information, these are not part of the module implementation.

The Polyolith system provides a module interconnection language (MIL) to describe the application's modular structure. MIL is based on a graph model of interconnection, the nodes correspond to modules, and the arcs between modules are the bindings between module interfaces. The graph structure grants the designer a structure for manipulation and analysis.

A change on the communication channel is accomplished by variations on the *software bus*. The *software bus* takes care of the run-time interfaces for an application, it has agents that isolate and encapsulate the interface details. In order to change interface properties the programmer changes the bus not the application modules. All modules in Polyolith communicate via the bus.

A bus is any agent that encapsulates interfacing details for a software application. The bus offers data representation coercion in a heterogeneous system. The bus manages the data transformation during normal communication and this coercion instrument serves to capture the state of an executing process and move it to a reasonable data structure, making easier the relocation to another host.

Purtilo and Hofmeister provide two operations, *encode/decode*, that help capturing and restoring the stack data structure. In these operations the execution thread is captured implicitly by controlling when the state is captured. PolyLith has an automatic transformation to generate code to restore the stack when a program is restarted. Capturing data from the heap and files requires some programmer intervention instead of automatic capturing. An automatic transformation is used to generate code to restore the stack when the program is restarted. However, there is a drawback, if the structure and the semantics of the stack change in the new version of a process's program then the automatically-generated code will have to be modified by hand. Our approach provides `save_state()/get_state()` operations which are

tantamount to Polyolith's encode/decode operations. As with our approach, Polyolith restores the state when the component is restarted.

In terms of performance, Purtilo and Hofmeister [42] mention that there is a cost in capturing, transferring and restoring process state dependent on the size and complexity of that state. However there is no specific performance data available about it.

In her dissertation C. Hofmeister [28] adds primitives to support a) adding/ removing modules and bindings, b) accessing messages in transit, and c) synchronizing reconfiguration activities. She also defines an approach for capturing and restoring persistent state that is supported by the reconfiguration primitives, providing support for module participation during dynamic reconfiguration. We take this approach, module participation, to provide support for capturing and restoring state.

Module replacement is done at reconfiguration points placed in advance by the programmer. Flags are tested at regular intervals. The frequency of this testing depends on how many reconfiguration points are specified and where. Some degree of optimization can be attained by having these reconfiguration points outside computationally intensive loops or procedures since the run-time cost is merely that of testing the flags installed for reconfiguration. Reconfiguration can be initiated by any module or by a third party.

In summary, *state transfer* is achieved by the encode/decode operations. The source code is modified to include: a) delay reconfiguration until appropriate, b) package up and send state, b) install the state, c) resume execution. The compiler is used to perform the restore and capture of the activation record stack. Figure 2.6 portrays Polyolith's design. A geometric change is depicted as Proc. C which is running on a different processor and communicating through the software bus. A modular change requires process-level participation to perform state capture and restore. Modular structure is defined using module interconnection language. All this is bound using the configuration manager (Polyolith).

A XML data representation for state transfer would provide great flexibility, mainly in this technique which supports geometric reconfiguration on heterogeneous architectures.

Figure 2.6: Polyolith's software bus.

2.4.5 Argus

Argus is a system and programming language designed to support implementation and execution of reliable distributed systems developed by Barbara Liskov at MIT [34]. Argus is based on the Clu programming language.

Although Argus was not designed to support dynamic reconfiguration, many of its constructs that support distributed computing (for example, explicit interfaces, and clear separation of interface from implementation) provide a useful basis for dynamic reconfiguration [8], *i.e.*, adding and removing components, or by moving a component from one node to another. In Argus, a distributed application is composed of a set of processes called *guardians*, which implement a collection of operations (procedures and functions). Guardians are tantamount to modules in our abstract architecture (see Figure 2.1).

Guardians encapsulate one or more resources. Guardians cannot be accessed directly; however, access to them can be done through special procedures called *handlers*, a similar concept to a remote procedure call interface. Guardians contain data objects that save the state of its resource. Communication among them is done using a message-based communications procedure. Each guardian resides on a single node but they can be moved to other nodes.

Guardians have stable and volatile objects. The former ones maintain their state by saving it to (for example) disk so it can be recovered after a crash. After a crash and after recovering the stable objects, Argus creates a special recovery process which initialises the volatile objects. A crash destroys all volatile objects of a guardian. Volatile objects are used to store redundant information or information that can be discarded. The process of restoring the state from persistent storage is similar to DYMOS (see §2.4.3) and our proposed solution Section 3, in which the recovery process may involve providing procedures to compute the new state from old state.

Recovery is done by using versions. The state of an unlocked (not being accessed for reading or writing) object is stored in a base version. Changes to the object are not done to the base version but to a copy residing in memory. If the action succeeds then the copy becomes the base version and is saved to persistent storage if the object is stable.

In her dissertation Bloom [7] designs mechanisms to support dynamic updating with no changes to Argus. The unit of replacement Bloom uses are guardians, as the minimal unit, and multi-guardian implementations called *subsystems*. Similar to DSU (see §2.5.1), subsystems can be replaced simultaneously. In DSU Hicks explains it as follows: if only single modules/guardians could be replaced, functions used by other modules/guardians in the program could not change type. A subsystem can have long-term running state that is resilient to crashes.

For single guardian replacement the user installs the new implementation and external binding information in the library. The new code is loaded; the existing stable state is associated with the new instance; a mapping is done from handlers of the replaced guardian to the handlers of the new guardian. Then the replacement mechanism converts the running instances to the new implementation. The conversion requires the following to occur: the new code is loaded, the existing stable state is with the new instance, and a mapping is defined from the old guardian to the new guardian. At the end the old instance is removed. The replacer checks that the handlers and state are correct before proceeding with the replacement. Finally the new guardian is made available to the clients.

Similar to single guardian replacement; in subsystem replacement only the interface must remain unchanged across a replacement. All implementation details

should be modifiable. The state information in the new subsystem may be arranged (redistributed among components) differently than in the old version.

Bloom and Day develop a set of correctness conditions for reconfiguration in Argus [8]. A correct reconfiguration requires that the new implementation and state only generate accepted behaviour which would have been permitted by the replaced abstraction in the state in which the replacement occurs. Argus takes two approaches to provide support for reconfiguration. *System-supported replacement* (SSR), which is the replacement of any active guardian or subsystem subject to certain type correctness conditions. The user selects the guardians and subsystems to be replaced and quiesces them. Quiescing can be either forcing the active transactions to abort or waiting for these to finish. Services to clients may seem as they were temporarily interrupted and then restored again. *Application-level replacement* (ALR), in this approach the programmer provides special operations that extract the abstract state from the guardian and create a new guardian from this state. The rebinding of handlers occurred as clients tried to use handlers of a replaced guardian and received exceptions; then the client looks in the Argus catalog to try to find a service again. In order to get the state of an object, the guardian to be replaced is crashed, the code is replaced and the guardian restarted. Then the extracting handler is activated, returning the state of the guardian. This approach requires more work by the guardian. Apache restarts components by sending a restart signal to all of its components, which is the same approach as crashing guardians.

The disadvantage of this system is that it is tied to the Argus system, similar to Conic, DAS or Dymos, that are linked to a specific not widely used language, architecture or OS sometimes not widely used. However Argus provides several of basic constructs to work with fault-tolerant applications which ensure consistent state and in turn are easily applied to dynamic updates. The work done around Argus provides a good insight into state transfer. This system is one of few techniques that try to ensure consistency after restarts of the unit of replacement.

State Transfer is done when the guardian is crashed. State is saved only for stable objects. The programmer decides what portion of the guardian state is made stable. As with our proposed framework, the state is saved at the time the module is “crashed”. Argus, and our approach, provide special operations to extract the state

and retrieve it.

2.4.6 Durra

Durra is a language designed to support the development of distributed applications running on heterogeneous machine networks. An application description in Durra has a set of task descriptions and type declarations that are used to manage network resources. Barbacci *et al.* [38] explain that a task description is a compilable template that defines interface, behaviour, attributes, and internal structure. Durra is a non-procedural language, the programmer specifies the application structure and the resources allocated to the components separately from the coding of the component [38, 6]. Similar to Conic, in Durra the system structure of an application is defined separately from its behaviour, providing more flexibility over the evolution of the application.

The Durra run-time environment has two active components: the application modules (called processes) and the Durra configuration managers (called executives). As depicted in Figure 2.7 an instance of an executive runs on each processor, and processes are distributed across the processors in the system. The executives interpret the resource allocation commands produced by the Durra compiler, monitor reconfiguration conditions, and implement the changes in the application structure. The application processes are instances of independent tasks (programs). These component processes can be programmed on any language with a valid Durra proxy (called interface) for that language. The component processes use the interface to communicate with other processes and the Durra executives.

Durra allows specification of conditions for system reconfiguration. The changes may include removing processes and queues, instantiating new processes to different queues, or instantiating new queues; all this without affecting other components. These changes may be triggered by internal or external occurrences.

Durra takes the same approach as most of the hot swapping techniques in order to replace a component and maintain its state: it makes processes quiescent before reconfiguring. That is, when a process is not participating in a communication with other processes, it will not initiate a new communication or receive new communication that requires response from this process. Since processes are seen as black boxes

Figure 2.7: Durra run-time environment.

the only way to know the state of a process is if it notifies the master executive.

In Durra, interconnections between tasks are done through channels. These are considered links in which data is passed from one process to another. Interface information is denoted by unidirectional ports. Barbacci and Lichota classify ports in two types: *in* and *out*. An *in* port type takes data from the channel and an *out* port type is used to send data to the channel. A port declaration specifies direction and type. In & Out ports are tantamount to required and provided interfaces in our abstract representation (see §2.3).

Although much of the motivation behind the work on Durra is to support fault-tolerance, Durra has features that allow dynamic reconfiguration. However much of this dynamic reconfiguration has to be anticipated and specified in advance.

The strategy employed for *state transfer* is as follows. A new process, *i.e.*, re-instantiated task, is initialised with the state of the old process at the point of failure. The programmer decides how often and what needs to be saved. The authors suggest dedicating a port to write (periodically) the state and one port to read (at process initialisation). In this process, the developer specifies, in Durra, that these two ports are connected through a regular Durra queue, and that, if the process fails, it must be replaced by another instance of the same task, retaining the state-saving queue and reconnecting it to the replacement process.

2.4.7 On-Line Software Version Change

Gupta and Jalote [23, 13] present a framework for specifying on-line software updates. In succeeding work [11, 12] Gupta *et al.* establish formal grounds and definitions for working on hot swapping software in general (see §2.2). On-line Software Version Change was developed at the Indian Institute of Technology.

In their framework the authors define an on-line change as a series of activities that involve stopping a running process, modifying its code, mapping its state and then starting the new code with the existing state. Gupta uses interprocedures, similar to PODUS and DSU, for procedures that change interface between versions, and uses a state transformer function for the global state. Interprocedures are tantamount to our proxy element in the abstract architecture, §2.3.

Gupta defines an on-line change to be *valid* if some time after the change, the process reaches a reachable state of the new program version. Hence, “validity ensures that following a change, the process starts behaving like the new version of the program after a transition period” [12].

An important contribution of Gupta’s work is the formal framework for hot swapping([11] §2.2). As well as a proof that determining update validity is, in general, undecidable. This means that we cannot develop a general purpose algorithm which, given programs P, P’, a state mapping S, and a reachable state s of P, will determine -with these parameters- if the change is valid. Hence, the programmer must identify correct timing conditions for each update.

The system provides an interactive tool called *modification shell*, it is the user interface to the on-line replacement system. The modification shell is comparable to a configuration manager (§2.3). The *replace* command, with the executable file of the new version as a parameter, provides the change functionality. The programmer develops, compiles, links and debugs both versions of the programs.

The programs are ran as child processes of the *modification shell* and ptrace¹ system call is used to inspect and modify the child’s process address space and machine registers. The state of the running process is inspected to determine if state transfer

¹ptrace system call allows observing and controlling the execution of another process, and examine and change its core image and registers.

can be done consistently and to actually execute the state transfer. Apache implements hot swapping following the same principle: http servers run as child processes of a master server (modification shell/configuration manager); thus there is a direct control over these child servers.

Although the theoretical work on validity is an important contribution, it is not easily extendable to realistic languages. The approach Gupta takes is closely implemented in applications like the Apache HTTP server (see §4) in the sense of replacing child processes.

State Transfer. To transfer state, the process running the first version is made to call a routine which stores the limits of data and stack addresses in variables whose names are known to the replacement system. The system reads these values and writes them onto the corresponding variables of the process running the new version. The new process is then made to call a routine which expands the data and stack space to the limits specified by themselves. The replace module then copies the data and stack space of the first process onto the second. The machine registers are copied next.

2.5 Middleware Layer

Techniques grouped in this class provide frameworks and general-purpose methodology for hot swapping. Some of the techniques provide this support through minor modifications to compiler or virtual machines. In general, techniques in this layer facilitate the development of hot swapping applications; that is, the mechanisms are not applications on its own.

2.5.1 Dynamic Software Updating. (DSU)

Hicks *et al.* [26] present a general-purpose framework for updating programs as they run. The authors make special emphasis on flexibility, robustness, efficiency and easy of use. Their framework allows changes to functions, types or data. Functions and data may change type, and named types may change definition. Changes are not restricted as to when the updates should be done, DSU allows changes to active

Figure 2.8: An example of code relinking [25].

code; as opposed to other approaches like Conic, PODUS, Durra and Argus. Code is verified before implementing a patch with the use of Typed Assembly Language (TAL), including type safety. The authors provide tools to aid the programmer applying the patches. Low overhead is achieved by the use of TAL because patches and programs are made of native code, giving performance benefits compared to systems like Java.

In his dissertation Hicks [25] details the concepts mentioned above. The whole approach of DSU is based on dynamic patches. An update consists of one or more dynamic patches. A dynamic patch is a new version of the module (a file) which contains the required data to support the dynamic update of that module. The advantage of using dynamic patches is that it provides robustness by respecting important safety properties, including type safety. Changes are classified into two categories: updates to data and code, and updates to type definitions.

Updates to code and data are done by *code relinking*. Figure 2.8 depicts an example of code relinking. In code relinking all references to old definitions are “fixed up” (the dynamic linker goes through the entirety of the program) and redirected to point to the new code. In other words, code relinking is a method in which the dynamic linker changes all occurrences in the program to point to the new code. When relinking is used the program is relinked after the patch is loaded. Code relinking is implemented by reusing the code in the dynamic linker.

When it comes to type definition updates the Hicks *et al.* use type renaming approach. In this approach, as depicted in Figure 2.9, there is a fixed notion of a

Figure 2.9: An example of type renaming [25].

type definition, and it relies on the compiler to determine a new type that logically replaces the old one by syntactically renaming occurrences of the old name with the new one. As a result when the patch is applied, existing instances of the old type are left as they are; converting of old instances is done by the stub function and/or the state transformer function, which reside in the patch, at update time or later if needed. Renaming provides more flexibility as to when and how values of changed type will be converted.

Similarly to interprocedures in PODUS, Hicks uses *stub functions* to handle change type. A stub function has the same type as the old version, and is used as a mediator between old callers and new definitions to get the types right. *State transformer functions* are equivalent to the combination of all mapper procedures in PODUS; these translate existing state, static data, to a form usable by the new version. Stub and state transformer functions aid the updates of data and code.

Hicks provides an automatic generation of the skeleton for the dynamic patches. The advantage of an automatic patch generator is that it makes the system easier to use, and it also minimizes human error. The patch generator is divided into two stages: identification and generation. The identification phase takes the old and new version of the file to patch these are parsed and type-checked. After this phase is

finished the patch generator reports: the functions that changed, the functions that changed but retained their old type, the data declarations that did not change, the data declarations that changed, the named types that did not change, and the named types that changed. After the identification phase has finished it generates state transformer and stub functions, when possible. Type conversion functions (files) are constructed to the extent possible for data conversion from old to new version of a type, and vice versa.

DSU's framework is implemented in Typed Assembly Language (TAL) for intel IA32 instruction set, called TALx86. DSU extends TALx86 by adding a type-safe dynamic linker and it provides dynamic updating for programs written in Popcorn². Dynamically loadable files are compiled so that their external references are resolved through a global offset table (GOT). At load-time, the entries of the table are resolved with the exported definitions of the running program; which in turn are tracked by the dynamic linker within a global dynamic symbol table.

Hicks' approach to hot swapping is composed of the following steps: a) construct a dynamic patch, b) compile the dynamic patch to a loadable TAL file, c) dynamically link the file into the running program, and d) transition the running program to use the patch. In order to perform dynamic updating by dynamic linking the methodology is the following: a) dynamically link patches, b) transform the state in the program (using the state transformer function), c) transition to using the new code. Figure 2.10 depicts the elements of hot swapping in DSU.

As mentioned previously *state transfer* is accomplished by transformer functions.

The following sections briefly introduces some of the research done in the field of virtual machines, programming languages, and applications with respect to hot swapping. We begin describing work done for C++ and Java; for the latter we present some java implementations that implement hot swapping.

²The Popcorn language is similar to C, and includes some standard enhancements such as more flexible variable declarations and a C++ -like namespace mechanism. Unsafe features, such as pointer arithmetic, the generic address operator, and pointer casts, are missing [25].

Figure 2.10: Dynamic Software Updating.

2.5.2 Dynamic Classes for C++

Hjálmtýsson and Gray present an implementation for dynamic update of an executing C++ program at the class level [27]. Their approach does not require special pre-processor or compiler support. However the programmer has to code using a generic template proxy class to provide hot swapping for those classes.

This framework allows objects of multiple versions of a dynamic class to coexist. All new objects are created with the new implementation, *i.e.*, new version, and existing objects continue to exist with their current versions. Once existing objects are destroyed only the new version will be used. If the programmer wants to directly migrate objects, the framework provides a method that an object can use to identify if its version is the most current version. The authors note that the programmer would need to explicitly write state-capture and restoration routines for each class version.

The implementation has an interface monitor that listens to messages that pass through the dynamic class interfaces. The monitor, a class proxy, maintains a map with relationships between class names and the current implementation version. The interface monitor maintains a pointer to the newly loaded version of a class directing

Figure 2.11: Two-level mapping in dynamic C++ classes [27].

constructor calls to that class.

Since multiple versions of a dynamic class exist within a program, there can be multiple versions of each method within the program's address space. The correct implementation has to be called when a method is invoked. To solve this problem the authors propose a two-level indirect method resolution at runtime. The first level is version-dependent mapping, which is implemented in the dynamic class proxy; and the second level is the method mapping within a version, which can be achieved with standard C++ virtual methods and the associated vtables. The version of each object remains unchanged for the lifetime of the object, in Apache this is called *child server generation*. The method mapping is achieved in C++ by defining all methods in the interface class as virtual. C++ adds a vtable to each derived implementation class, and resolves all method calls through the vtables. Figure 2.11 depicts the two-level indirection. The proxy contains a pointer to the object, which in turn contains a pointer to the vtable, and the vtable contains pointers to the methods of the object's implementation version.

This framework is simple and quite portable since it does not require changes to compiler or special preprocessor support. However there are some implementation trade-offs. Using a proxy introduces some overhead, it also limits the class to a single constructor, and all method access must take place through the dereference operator. Similar to Conic, §2.4.1, there is a clear separation between the class interface and the implementation. However, the proxy interferes with interface polymorphism. Even if one interface inherits from another, the proxy class instantiated on the derived

interface is not related to the proxy class instantiated on the base. Thus, contrary to interface inheritance, the proxy of the derived class cannot be used where the proxy of the base interface is expected. The solution the authors propose is the use of a template function that performs explicit dynamic casting.

The proxy approach is, as well, implemented in S-Modules and Gang's work (see §2.5.5, §2.5.6).

State Transfer. Since the authors use version coexistence there is not a state transfer *per se*. However, they include a method that an object can use to determine if its version is the most recent version, allowing the developer to migrate objects of a particular class. Nevertheless, there is no proposed guidelines on how a programmer would have to write state-capture and restoration routines for each class version. The authors do mention that these routines would produce and accept version-independent representations of an object's state.

2.5.3 Dynamic Java

Andersson *et al.* [2] specify and implement run-time support for dynamic architectural reconfiguration. See Appendix A for an overview of the Java Virtual Machine. They divide the class update into layers as follows:

- Reload the class file. Done when the compilation phase alters the class file.
- Rebuild the method and attribute table. Done when attribute and/or methods are added or removed from the class, including inherited ones.
- Rebuild the constant pool. The constant pool maps information on where to find attributes and methods in other classes. An update on a higher level cause lower level updates thus this information has to be updated.

Objects are updated incrementally rather than all at once. Objects are replaced when using a reference to an object for which there is a new version of its class available. Therefore, objects of different versions of the same class can coexist. In order to do this the authors make minor modifications to the virtual machine. They assume the existence of a mechanism to convert an object of the old version into

an object of the new version; given references to: the object, the old version of the class, and given the new version of the class. They suggest, as an example of this mechanism, to use a method in the new version class to convert an object of class k to an object of class k' . If there were references to the old object, these must be updated to point to the object of the new version. Once these are changed the object can be garbage-collected. This approach is similar to the one used in dynamic classes for C++ Section 2.5.2.

Referencing and dereferencing objects is accomplished by modifying some virtual machine instructions like: `aload`, `aaload`, `getfield` for referencing; and `getfield`, `putfield` and `invokevirtual`, `invokeinterface` for dereferencing. By modifying the first two one can assert that only references to updated objects are pushed to the run time stack. Dereferencing waits until the object is dereferenced, thus reducing the number of checks (compared to referencing), and as well the time between the check and the use of the object. This approach supports the use of multiple versions thus there is not much emphasis on state transfer.

Unless the proposed changes become part of the Java runtime this mechanism is limited in its portability. On the other hand, *state transfer* might not prove to be too difficult since changes are done to the virtual machine and low level access is gained by doing this.

2.5.4 Dynamic Virtual Machine

Malabarba *et al.* [35] present an approach for supporting dynamic evolution for Java programs. The unit of change used by this approach is a *class*. In order to provide for dynamic classes they modify the Java Virtual Machine (JVM), *dynamic classes-enabled virtual machine*, *DVM*, and add a dynamic class loader. In this work, the authors present a series of definitions and formalisms to comprise a formal model of objects, classes, and inheritance in Java.

The majority of the evolution functionality is integrated within the Java run-time. Type safety is provided when an upgrade is compiled. Moreover, the DVM checks for static type violations by examining the current class, the old class version and any dependents. Their design extends Java's dynamic linking mechanism, instead of replacing it. Two methods are added to the class loader: `reloadClass` and `replaceClass`.

The former is similar to `loadClass`, however it does not fail if the class is already defined in the system. `replaceClass` defines a class C' as a new definition of an existing class C and initiates instance update. `reloadClass` receives a class as a parameter and `replaceClass` receives the old and new classes.

The DVM requires that classes be replaced logically at load-time; their model uses the global update when it comes to updating instances. Global update of all objects defined by class C , is such that the DVM locates all instances of C and updates them, as well as its subclasses to reflect the new definition. This is what Hicks 2.5.1 calls code relinking. The replacement is initiated by the user by explicit calls to the classloader in the application program.

One disadvantage of having a modified virtual machine is that it becomes platform dependent and will not easily transfer to other JVM's. Changes introduced to the virtual machine introduce a 5% of penalty when the class replace lock mechanism is used according to their own performance analysis.

State Transfer. Their changes do not include an automatic state transfer, this is left to the programmer. He or she has to specify a transform function for the class being replaced, which assigns value to new data fields, and translates old data fields that have a new type, *i.e.*, a wrapper function. The approach for state transfer is similar to the proposed approach in this thesis.

2.5.5 Java S-Modules

Ning Feng [15] proposes a design infrastructure for software updating in the realm of object-oriented programming. Her approach to solve the hot swapping problem is by applying the Proxy Pattern. The intent of the proxy pattern is to control access by providing a surrogate or placeholder for it [18]. In Feng's implementation each S-Module has one proxy object (S-Proxy object) associated with it. This is a similar approach to the one taken in Dynamic Classes for C++ (see §2.5.2).

Each application, named S-Application, is composed of S-Components, which in turn are made up of an S-Module and its corresponding S-Proxy; non-S-Modules, and a swap manager. The swap manager controls the swapping of S-Components. Non-S-Modules have a direct interface to other modules through an S-Proxy. An S-Module follows a three level class hierarchy:

- an `S_ModuleSuper` class; an abstract class which defines all the attributes and common interfaces for all S-Modules
- an `S_ModuleTypeX` class; an abstract class for all S-Modules of the same type X
- an `ConcreteS_ModuleTypeX`; a concrete class and implements the behaviour interface of that type of S-Module

An S-Module is expected to be collected by the garbage collector after being swapped out. An S-Module A has a dependency list which maintains all the S-Modules that A depends on. When a deadlock condition is found in a dependency map the S-Modules involved cannot be swapped. There is a particular order in which modules are loaded based on the dependency list. An S-Module A has a descriptor which is used to identify A of a certain type, which later helps to build the dependency list and also for mapping attributes between versions. There is no mention on how to cope with the updating of existing instances when a class changes.

State Transfer. Feng's proposal for state transfer is based on attribute mappings from the old S-Module to the new S-Module. The mapping responsibility, in terms of old state information, is left to the new S-Module class. The S-Modules have an administration interface and the state transfer process relies on the existence of these fixed APIs.

2.5.6 Hot-Swapping Technique for Object-Oriented Paradigm

Gang's work [3] combines the object-oriented paradigm and mobile code technology as an approach to performing hot swapping. He embraces the same terms as in 2.5.5, as well as some of the concepts. His technique takes advantage of Java's support on features like network mobility, security, platform independence, dynamic linking, and dynamic extension, memory management and others.

The framework presented consists of the following elements: a specification for application configuration, a swap manager, and a system administrator. The swap manager handles the activities of the update and manages the S-modules. A swap manager provides the following services: *listening service*, listens for messages sent by the system administrator, *security service*, *life cycle service*-controls the state transfer

of S-proxies and S-modules in the swap transaction, *storage service*, stores messages and new S-modules that has instantiated, *naming service*, provides a naming service to locate every S-proxy according to its name, *state monitor*, maintains integrity of the application, *time control*, duration control of the hot swapping transaction, *logging service*, handles information for the system administrator, *transaction service*, it decides whether the hot swapping transaction succeeds or not.

Similar to Feng's (§2.5.5) approach the disadvantage of having a proxy is that this is made non-swappable. Interfaces can be made dynamic, but the proxy uses reflection in case of deleted or added methods. Gang's work pays close attention to state consistency of the objects. He employs *logging* and *shadowing* to guarantee state consistency of an object. *Logging*: persistent object values are updated in place and all changes are recorded in a log, useful for rollback actions. *Shadowing*: the update of any persistent object values is deferred. The new and stale values of each object part of a transaction are kept in a persistent storage and switch is made when the transaction is committed.

He proposes a two-phase commit transaction model combined with thread-pool technique to ensure robustness and to minimize the interruption of the software application services, based on the Jini [36] transaction design.

We note that in Java S-Modules §2.5.5 and Gang's approach §2.5.6 it would have been beneficial to have a performance analysis due to overhead introduced by the S-Proxy modules.

2.5.7 SwapBox: a Hot-Swapping Framework for Swappable Java Beans

Tan presents a framework [50, 49] for component-based applications, specifically the JavaBean model. His work builds on top of Sun's BeanBox, a test container for JavaBeans. The BeanBox is an standalone application.

The SwapBox deals with the common issues involved in hot swapping, like:

- Granularity. The basic unit of change the hot swapping supports; in this case a JavaBean component.
- Reference indirection. With the event-based communication mechanism an

adapter is interposed between beans. These provide a reference indirection to swappable JavaBeans. The hot swapping system requires the beans to be connected to event adapter. Which are generated automatically.

- Constructing a new state. Capture the old bean's object state and restore it on the new bean's object state. However it does not capture the running stack of the old bean. Mapping rules are used to map the state of the old bean to the state of the new bean, in case the interface changes.
- Levels of change. The system supports implementation, interface, and data structure change.
- Timing for upgrading. This is concerned with program correctness. The system provides solutions as when the swap can occur, these can be selected at run-time.

The mapping rules, mentioned in constructing new state, are a one-to-one relationships expressed in XML format and parsed at the time of hot swapping. Mapping rules are established at runtime by the programmer. The SwapBox uses Java reflection to extract the state information of both, old and new, beans. First, it retrieves methods that reveal state information, and it displays the list on a GUI. Then, it retrieves the methods in the new bean to perform initialisation, and it displays the list on the same GUI. The programmer then chooses the mapping relation between the old state name and the new state name. Even though this feature provides flexibility to the system it incurs in high timing overhead. This approach is infeasible for a full deployed operational system, since human intervention at state transfer is unacceptable.

The proposed SwapBox does not modify the BeanBox's code, rather it inherits the BeanBox's classes and overrides methods which have different behaviour. The author provides a *carleton.swapbox* package which contains all the classes belonging to the SwapBox. Tan uses the same concept of SwapManager used in 2.5.5 and 2.5.6. The SwapManager's tasks include: set a timeout for the swap transaction, parse the swap transaction file to retrieve parameters specific to the particular transaction, create event adapters for the new bean, create new states, clean up the old bean, rollback in case of exception or timeout.

The swap transaction is done as follows: a set of event adapters for the new bean is created. Then services are blocked for the old bean and it transfers the states from the old bean to the new bean. If no exceptions occur the old bean is swapped with the new bean and the old bean is removed from the SwapBox.

The commonality of the S-Modules (§2.5.5, §2.5.6), SwapBox (§2.5.7), and Dynamic C++ classes (§2.5.2) is the use of a proxy element to provide a separation between interface and implementation.

2.6 Operating Systems Layer

In this section we discuss some of the work that has been done in the realm of operating systems and hot swapping.

2.6.1 DAS

The Dynamically Alterable System (DAS) was conceived by Goullon *et al.* for research and education at the Technische Universität Berlin in the late 1970's. The system allows dynamic changes to all system components except in the kernel which contains part of the modification facility [20].

DAS is one of the early examples of dynamic program updating. In their work Goullon *et al.* use the term *replugging* to refer to dynamic modifications: a distinct implementation is plugged into the unchanged specification socket of a component (called in DAS module). They also introduce the term *dynamic restructuring*, which means dynamic modification that requires data conversion, either on demand or on request.

Replugging is done by a facility that is functionally independent of all other system components. Indirect access to segments as a built-in feature of the system's kernel, and the required synchronization is achieved by the busy/waiting technique (reader/writer exclusion problem).

When a process in a component calls a different component, DAS carries out an address space transition by a special CALL mechanism. The process moves to a new address space. This new virtual address space is made of a *stack segment*, a

Figure 2.12: DAS address space change caused by calling an object [20].

object segment, a *own segment*, and a *code segment*. The *stack segment* is a copy of the previous stack segment before the CALL. The *object segment* contains data of the object to be handled. The *own segment* contains the component's own data. The *code segment* it comprises the code for all the new component's procedures. The CALL mechanism places the descriptor information into the address mapping hardware. Keeping track of the segments is done by a linked list of segment descriptors (descriptor chain). Figure 2.12 illustrates the effect of an address space change caused by calling an object, provided that the calling space contains a free segment and no object segment.

Replugging is done with an indirection facility that allows the replacement of one segment by another one, this is done with the descriptor chain. That is, replugging without any data restructuring, stand-alone code segment, or replugging-on-request (including data restructuring) are all done by changing the links within the descriptor.

Data restructuring is done along the replugging action and it is always done upon request. Goullon *et al.* use an algorithm that copies a value from a source object to an initialized destination object. These operations [in/out] are part of the component. *In* operation takes care of the destination object and the *Out* operation is done on the source object. Thus one object is filled with information while the other is emptied and left in its initial state. The mechanism works independent of the type of source

and destination. If there are differences between the structures of the source and destination then these are overcome by the mode of the *out* operation of the old component and the *in* operation of the new component. The above described is the *state transfer* mechanism under DAS. These in/out operations are similar to the interfaces described in our abstract architecture depicted in Figure 2.1.

The design of DAS made it highly coupled to the hardware used, a PDP-11/40E³. Thus the implementation of this system required modification of microcode. By having a machine-dependent architecture we find a system complex to port to other platforms and obsolete nowadays.

2.6.2 K42

Here are some of the main features of K42:

- **Performance.** K42 is designed to scale up to run well on large multiprocessors and support large-scale applications. It also scales down to run on small multiprocessors.
- **Customizability.** K42 allows the application to decide how their resources are handled by the OS. This can be done by choosing from existing components or by writing new ones.
- **Applicability.** K42 is intended to support a wide variety of processors and systems architectures, from embedded processors to high-end servers.
- **Wide Availability.** K42 open source and open to the community for experimenting with policies and implementation strategies.

K42 provides full support to the Linux application programming interface, application binary interface, libraries, device drivers, file system and other code without changes. This is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects.

K42's proposal is structured with object-orientation in mind. Well-defined interfaces are necessary for interposition and replacement of components. The system is

³The PDP-11 -Digital Equipment Corporation (DEC)- computers were a popular minicomputer series that allowed user-written microcode to be implemented [1]

Figure 2.13: Stages in replacing an instance A of object j by instance B [4].

structured using modular, object-oriented code; it is designed to scale to very large shared-memory multiprocessors; takes advantage of 64-bit processors; it avoids centralized code paths, global data structures, and global locks; and it moves system functionality from kernel to server processes.

The dynamic replacement algorithm follows these steps: 1) instantiate the replacement component, 2) establish a quiescent state for the component to be replaced, 3) transfer state from the old component to the new component, 4) replace all references in the new component, and 5) remove the old component. This procedure is depicted in Figure 2.13.

The object-oriented structure facilitates hot swapping. Objects are self-contained with well defined interfaces. Each object is accessed through a single pointer indication, which is maintained in an Object Translation Table indexed by the object identifier. “K42 has a generation count mechanism to determine when all threads that were started before any given point in time have completed or reached a safe point” [4]. This approach is illustrated in our abstract architecture as a proxy for each module (*cf.*, 2.3).

State Transfer. Similar to other systems, K42 cannot determine automatically what state must be transferred. On the other hand, it provides means to negotiate a best common format that both objects, and in some cases pointers to complex data structures may be passed. An interesting feature is parallel state transfer. Parallel

state transfer can be accomplished on a multiprocessor architecture. No specifics on how to perform the state transfer are given.

2.6.3 Kea

Kea is an operating system kernel developed at the University of British Columbia [53]. The work on it started in 1994. The Kea operating system provides a means through which applications can dynamically reconfigure, replace, or extend existing services in an incremental and safer manner.

Kea provides lightweight abstractions of a machine's physical resources (CPU, memory, interrupts, etc), which can then be used to construct higher level services (network protocols). These higher level services communicate with applications using interprocess communication (IPC), and often with remote procedure calls (RPC) on the next layer. Kea's form of RPC is general and it allows the destination of an RPC to be changed independently of the caller, also enables services to be dynamically and transparently reconfigured or extended. The combination of the following Kea abstractions make reconfiguration possible under Kea:

- **Domain.** These are virtual address spaces, offering mappings from virtual to physical memory. Domains are the entities to which system resources are allocated.
- **Threads.** A thread is the context in which code execution takes place, and from the application point of view provides similar services and properties to those in other operating systems. Kea separates the kernel thread state (scheduler and security information) from the current execution state (domain, machine registers and stack). The execution state is called an activation, and each thread can have multiple activations associated with it.
- **Inter-domain calls (IDC).** IDC is the RPC-like abstraction. IDC is more efficient as it eliminates several unnecessary steps often made in a general RPC system.
- **Portals.** Portals are the entry points of IDCs. They provide a handle into both a domain and an entry point within that domain. Figure 2.14 depicts the effect of a portal invocation. An application invokes a portal, through a

Figure 2.14: Portal invocation and portal remapping in Kea [53].

portal identifier (box), and the system performs the appropriate IDC to the service implementing that IDC. The underlying implementation maps the local (application level) portal identifier into a system portal structure (oval), which determines the number and size of arguments actually copied to the destination service. Portals are similar to the proxy elements in our abstract architecture (see §2.1), however portals are not linked to each component.

In Kea, services are implemented as set of interfaces. An interface is a definition of allowable interactions between the implementor of a service and the clients of that service. Portals provide the property of making reconfiguration possible, they can be remapped to a different domain/entry-point at runtime. Figure 2.14 illustrates, as well, an example of remapping; an application using a File Service *A* (dashed line, IDC call), and then switch to a File Service *A'*, with same semantics but with improved performance. When a new service is introduced a new proxy (portal service) is introduced as well. The important features are that the applications portal identifiers do not change, and that remapping is done transparent to the application, and does not affect other applications using the old service.

This approach requires the application call to the portal to remain unchanged in order to communicate with the proxy (portal system). There is no mention on mechanisms to perform *state transfer* or on how to eliminate old services. On the other hand, old services continue to serve applications until a remapping is done. The coexistence of elements may provide reasons no to transfer state from the old service to the new service. There is no description on how to add new services, that is how these new services are available to the Kea kernel.

2.7 Summary

In this chapter we reviewed different approaches to solve the hot swapping software problem. Many of the solutions come from the realm of distributed applications and fault-tolerant systems.

From our survey we can point out that one of the main difficulties in hot swapping components is to determine the effect of the change on the running system. It is clear that arbitrary changes to software cannot be done if we want to preserve state. For example, if we replace a piece of code the program and the state is not preserved, the program may not be able to execute afterwards or it might not produce the expected results.

Our research has not uncovered any work that allows for hot swapping of arbitrary code (see §2.2). We believe that a system has to be designed and built to cope with hot swapping. In order to do hot swapping we need a clear understanding of what type of changes can be done, when can the change be performed, and how to accomplish this change. Based on the review done in this chapter we can say that completely general legacy code can not be made hot swappable unless it undergoes a re-design to support hot swapping. We want to note that only a few of the strategies reviewed this chapter present performance data, very little information is available in this regard.

State Transfer depends on various factors, among them syntactic and semantic dependencies. Some of the techniques reviewed provide means to cope with the problem of syntactic dependencies. However, syntactic dependencies alone are not always sufficient for a system to determine the calling relationships between modules. At times a group of modules may interact in ways that cannot be automatically detected. For example, a group of procedures that control different parts of a system may not call each other, but still depend on each other's actions. Therefore, making the programmer responsible for *when* to perform the state transfer is the best pragmatic solution.

Most of the research efforts reviewed in this chapter make assumptions about programming language, programming technique, or run-time environment. In some cases some systems included their own programming language, compilers and editor. In general, it is very unlikely the programmers will make significant changes to their

code to adapt for hot swapping. Using standard programming styles, languages, tools and run-time environments makes hot swapping more transparent for developers. It is clear that the more transparent it is for developers and designers the more likely it will be used.

A summary for the more relevant techniques is depicted in Table 2.3. This table shows some of the state transfer features these techniques implement. The columns are described as follows:

- *Component state.* Most of the techniques require the component to reach certain state before hot swapping. A component can be in quiescent state or active in order to do a swap. Quiescent, idle, non-active, and passive state are all used as synonyms across the literature. A component in quiescent state is not performing transactions and any transaction it has already initiated have terminated.
- *State Transfer.* This column indicate how the state is initialised in the new module. Some of the techniques use the compiler to transfer the state or it is done by the virtual machine, other techniques require the programmer to develop state-capture and restoration methods or routines.
- *State Storage.* Most of the techniques retain the state in memory, however Argus uses persistent storage.
- *Transferable Elements.* What elements are carried on to the next version. These may vary from static data, and user-allocated data (global variables) to register, and stack information.
- *Version Coexistence.* Some of the reviewed approaches allow the existence of the old version and the newly loaded version to run simultaneously.

System	Component State	State Transfer	State Storage	Transferable Elements	Version Coexistence
Podus DYMOS	quiescent idle	mprocedures compiler	memory memory	static data symbols table objec code data	Yes Yes
PolyLith	quiescent	encode/decode funcs.compiler	memory	static data stack	–
Argus	quiescent	functions	persis- tent	user-allo- cated data	No
Durra DSU	quiescent active	functions state trans- former funcs compiler	memory memory	– all state	– No
OSVC	quiescent	state trans- former func. compiler	memory	stack register	–
Dynamic C++	active	functions	memory	user-allo- cated data	Yes
Dynamic Java	active	Virtual M.	memory	stack	Yes
Dynamic VM	quiescent	Virtual M.	memory	stack	–
S- Modules	quiescent	functions	memory	user-allo- cated data	No
Gang	quiescent	functions	memory	user data	No
SwapBox	quiescent	functions	memory	user data	No
K42	quiescent	kernel	memory	all state	–

Table 2.3: State transfer features on relevant hot swapping techniques.

Chapter 3

A Framework for State Transfer (FST)

In this Chapter we present our framework for state transfer in hot-swapping applications. We start by a brief discussion on state transfer concepts and existing techniques. Section 3.2 presents a series of guidelines to follow in order to perform state transfer in hot-swapping applications. We then propose in Section 3.3 a design pattern for State Transfer. We finish the Chapter by presenting our design.

3.1 State Transfer

State transfer aims, among other goals, to preserve as much of the original state and semantics as possible. We can use state transfer to implement hot-swapping by restarting a new program or component with an old component's state. State transfer is used to implement process and/or object migration, checkpointing and general-purpose persistence.

Process migration is the movement of a currently executing process to a new processor [32]. Advantages of process migration are: load balancing, fault tolerance, easier system administration, and data access locality. Process migration systems have two approaches to save state: in the first one, the program saves its own state, and in the second one an automated checkpointing technique is used to save and restore the state.

The simplest checkpointing techniques are architecture-specific: these approaches copy the pages of the program's heap and stack, plus the state of the registers; and are then saved in a file. There are libraries (such as `libckpt`) to capture user-space information. However, these libraries cannot capture OS-resident data, such as mappings from file descriptors to open sockets, pipes, etc. With consistent checkpointing, the state of each process is saved separately on stable storage [14]. These techniques are often used for failure recovery and/or process rollback.

A general-purpose persistent state is the state that needs to be transferred between the existing version and the new version. It allows any user defined data-type to be made permanent [26, 19].

The authors of IBM's K42 [4] consider a state transfer an activity that may include porting: active connections, buffers and outstanding timers. An example of state transfer is when an object that caches file state is swapped, the authors invalidate the cached information and pass the control information. Converting an internal list of pages into a list of generic page descriptors.

The unit of replacement in our design is called *Module*. A component is comprised of one or more modules. The life of the module is tied to the life of the component. In order to replace a module the whole component has to be swapped out. However, only the module is what the state is captured from.

3.2 Proposed Guidelines for State Transfer

Based on our literature review and the common issues in hot swapping applications, discussed in Section 2.7, we suggest that the following matters should be considered as guidelines when designing a hot-swapping application:

- What is relevant to capture? That is, identify the state that needs to be transferred to the new version.
- How does one capture it? That is, provide the means to serialise and unserialise the persistent state.
- When is this capturing needed? Identify the point in which the component will save its state or how often saving is required.

- When is it safe to perform the transfer?
- What issues might prevent the state from being saved or loaded?
- What to do if state cannot be saved/loaded?

Moreover, these issues could be considered in the design of modular application that envisions a future incorporation of hot-swapping into the system.

The following are examples of **what is relevant to capture?** during a state transfer:

- Save data structures
- Identifiers generated by an object which are unique within the lifetime of the system, program or module
- The stack or heap
- Global variables
- Classes or objects
- File descriptor usage

Note that other kinds of data may be needed to be saved depending on the system.

To answer the question **how does one capture it?** we find that, in the case of component and object oriented frameworks a mechanism for object serialization is often provided. Serialization is the process of saving object state to a fixed storage [51]. The advantage is that the process is handled at the level of run-time objects.

For example, using checkpointing techniques helps determining **when is this capturing needed?** The program is coded to perform its own updating. The previous section 3.1 gives a brief overview regarding checkpointing technology.

The following are some of the different approaches to answer **when is it safe to perform the transfer** question:

- The object or module is in a quiet state¹ and it agrees to be swapped
- By forcing a crash or killing the object, that is, halting and then restarting object with the new state
- By not taking action. All new objects are created with the new state, and existing objects continue with their current state until these are killed or garbage collected

To exemplify **when is it safe to perform?** we present an approach to support dynamic Java architectures. Andersson *et al.* in [2] suggest that safer transferring is done by deferring the update. Objects are not replaced at once when a new class is updated or available. Instead, objects are replaced when using a reference to an object for which there is a new version of its class available. At the same time, the capturing is done when the new object is referenced, and not periodically. However, Gupta has shown that the problem of correct timing is, in general, undecidable ([11],§2.2). Thus, leaving this task to the programmers to decide and identify correct timing conditions to apply changes.

What to do if state cannot be saved/loaded?: A default behaviour must be envisioned in case the state is not saved or loaded. The following are issues, among others, which **might prevent the state from being saved or loaded**:

- Abnormal termination of the module or component
- A predefined time constraint or threshold for loading has been reached
- Corrupted state
- A race condition is met
- Incompatible serialisations

In multitasking architectures choosing a poor time for retrieving or saving the state could cause race conditions. We require the programmer to construct the module to “know” that it is hot-swappable, in fact enforcing timing constraints at software construction time, rather than at runtime.

¹Chapter 2 provides various definitions on quiescent or idle states

3.2.1 State Loading Policies

Modules can define a set of policies to manage the state. For example, defining what a valid state is can be based on policies. A state is valid if it complies with state transfer policies set by the designer, *e.g.*, expiry date and time of the state in persistent storage. Policies are defined as any syntactic rule the designer can provide to the system for restricting or allowing a state to be loaded from persistent storage. For example, a designer may decide not to load the state, and use a default state, if the system has a load larger than a predetermined value.

Policies can be implemented in the following components of a system:

- As part of the application's configuration file
- Saving and setting the policies at the time the XML state file is saved to persistent storage, or
- Coded as part of the module in the method to recover the state from persistent storage.

The last one may be the least flexible of them. However it may save time in certain cases, *e.g.*, if the policy is for preventing the loading of an expired state, no parsing is involved since the expiration date can be based on the time stamp of the file.

3.3 State Transfer Pattern

The following design pattern presents a description of a proposed solution for the general problem of state transfer in the context of hot-swapping applications. The pattern focuses on allowing components or modules to transfer their state to new module versions. This pattern offers an abstract structure which should be customized by developers to each specific application.

State Transfer Pattern

Context:

We are working in a system made of independent components, *i.e.*, modules. Components need to be upgraded when new versions are introduced. Modules are initialized every time the system is restarted. We want to keep the system running across re-initializations without losing the state of its modules. The application being developed is module or component-based and requires persistence through updates and changes to the modules.

Problem:

How can we make the state of a module persistent through out system upgrades, re-initializations, and restarts?

Forces:

- Maximise readability of state representation. We want to save the state in a format that is readable across platforms. Using known standards vs. proprietary formats. Implementing readability may result in a loss of performance; on the other hand, the use of known standards increases portability for the state representation
- Maximise performance by minimising the added overhead, CPU and memory usage. Note however that increased performance comes with a trade-off between flexibility and performance. The use of exclusive tools and programming languages may increase the performance achieved by the application, nevertheless the cost to pay for this performance can be the reduction of adaptability to other environments
- Maximise adequate use of existing design by minimising the additional logical implementation through the use of existing APIs. Cause the least amount of changes on the existing module and system design, new implementations should foresee state transfer in their designs. Use existing design vs. introducing proprietary APIs. The use of existing APIs may present limitations, on the other hand the introduction of proprietary APIs may cause a ripple effect for changes on the whole design and its application

Solution:

- Use data structures allocated in memory to keep track of the current state. For example, use a table in memory to trace the current state during the life time of the module or component. Add a function to update this table, for example a trace-state method
- Add a method or function to save the state of the module from memory to persistent storage
- Add a method or function to provide a means to recover the state of the module from persistent storage and load it
- Use a serializable standard data representation to save the state in persistent storage. The use of the W3C XML text format standard is an example of it
- Define a default initialisation state in case of state recover failure

Rationale:

The designer and/or developer has to respond to:

- What is relevant to capture? That is, identify the state that needs to be transferred to the new version
- How does one capture it? That is, provide the means to serialize and unserialise the persistent state
- When is this capturing needed? Identify the point in which the component will save its state or how often saving is required
- When is it safe to perform the transfer?

Once these questions have been identified we can use the following tools to handle the answers. For example, a table, such as a hash table, is a simple and widely implemented data structure that we can use to keep trace of the module's state. For capturing the state and when to perform the transfer, we expect that every module

includes a *finalizer* or *destroy* method. This method does the “clean-up” or “termination house-keeping”. When the *destroy* method is invoked the state tracked by the function, *e.g.*, `trace_state()`, is serialized into permanent storage. The same reasoning applies to the *constructor* method and the object’s initialization, except that at this point –if applicable– the state is retrieved from permanent storage. In OO implementations this could be implemented at the garbage collection level. However this implies modifying the VM and we want to avoid intrusive implementations such as this one. Finally, The XML format is chosen to preserve the state in order to provide portability across platforms since it is a standard data representation and it is widely used.

Applicability:

The “state transfer” pattern can be used under any of following circumstances:

- State transfer can be used to implement hot-swapping by restarting a new version of an application with an old application’s state
- State transfer can be applied to process migration; in which a process is moved across the network to different machines *e.g.*, for load balancing
- Checkpointing techniques can apply this pattern. During normal execution, checkpointing technology, saves the state of each process periodically to stable storage for later retrieval

Known Uses:

Polyolith (§2.4.4) provides two operations, *encode/decode*, that help capturing and restoring the stack data structure. Argus (§2.4.5) saves to disk the state stable state of its modules. The programmer decides what portion of the module state is made stable. Argus provides procedures to compute the new state from the old state, similarly to DYMOS (§2.4.3). Durra (§2.4.6) implements a functions to initialise a task with the state of the old process. It leaves the decision of what to transfer to the programmer. The dynamic classes for C++ (§2.5.2) technique for hot swapping also

provides state-capture and restoration routines for each class version.

Related Patterns:

Component Configurer. This pattern allows component creation, destruction and migration without disturbing other components [44].

Swappable Staging Environment. This pattern is concerned with long-term scalability and the ability to evolve the system over a period of time. [40].

Memento. Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later [18].

3.4 Proposed Design for State Transfer

This section describes our design for implementing state transfer in an existing hot-swapping application. We start by introducing the assumptions we build on to develop our solution. The following Sections then define the goals, approach, and persistent storage representation.

3.4.1 Design Assumptions and Boundaries

Our proposed solution works on the assumption that an existing hot-swapping mechanism, *i.e.*, the ability to load and unload modules at run-time, is part of the application. There are a number of applications that implement dynamic loading of modules and/or components. For example, Apache http web server, Linux operating system with its loadable kernel modules, or IBM's K42. In other words, we are not building a new hot-swapping architecture. In this way, we can focus on unresolved problems and make some contributions to the field. However, we propose that any modular system should include a discussion on state transfer.

The approach we follow allows transfer variables, tables, or internal computations that could be considered state. We do not transfer open handles, such as a handle to an open socket. However, the examples below propose solutions to transfer open

handles. That is, we handle what the object represents capturing the semantics of it. Examples of capturing the semantics of the object:

- Open socket. A socket can be converted to a URL to be accessed in the new module. State is saved using an object that encapsulates the semantics of the socket
- File handle. It can be converted to a filename reference and a byte offset into the file, plus whether the file is being read or written

3.4.2 Goals

The goals we have set to develop our approach based on the pattern and the proposed guidelines, described in Sections 3.3, 3.2 are the following:

- Non intrusive to the existing architecture
- Minimal added overhead
- Ease of use
- Maximum flexibility of state representation

Non intrusive to the existing architecture. One of the problems with many of the existing approaches to hot-swapping is their proprietary architecture and design. The framework we present builds up on modular architecture without adding or modifying the existing APIs.

Minimal added overhead. The penalty for adding state transfer to modules should be as low as possible. Our goal is to have an added overhead of less than ten percent. That is, a performance similar to the one achieved by PODUS (of 90%) by the non-hot swapping application.

Ease of use. In general, the less complex a new framework is the easier it is to be adopted and implemented.

Maximum flexibility of state representation. The state is saved in a format that is readable across platforms. Thus, we selected XML as standard data representation to be used when saving the state to persistent storage.

Figure 3.1: A component is formed of one or more modules.

3.4.3 Approach

As mentioned earlier, the unit of replacement in our design is called *Module*. A component is comprised of one or more modules. The life of the module is tied to the life of the component. Figure 3.1 illustrates the containment relationship between component and modules. In order to replace a module the whole component has to be swapped out. However modules in the core of the application cannot be hot-swapped. Each module provides various services to the component. Our solution is meant to be applied on a forking or threading architecture. Nevertheless, these principles can be applied to a design in which a component and a module are equivalent; that is, the component is the unit of replacement.

Figure 3.2 shows the research target and its environment. An application is composed of an application core and components. Each component can contain several hot-swappable modules which interact with resources, the operating system and the application core. The administrator interacts with the configuration and document files. For example, modules are loaded or unloaded by editing the configuration file and by restarting the server core forcing a re-parsing of the configuration file. However, this is not the only method for interacting with the application, there are alternatives to this method. For example, a telnet-type service can be implemented in the application core and control commands (*e.g.*, load, unload) can be sent to it.

Every module includes a *finalizer* or *destroy* method or function. The method does the “clean-up” or “termination house-keeping”. When the *destroy* method is invoked the state is saved to persistent storage before exiting. In a similar manner a

Figure 3.2: A Hot-swapping Application

method *constructor* is invoked automatically when the module is created (instantiated), *cf.*, Figs. 3.1 and 3.3. In OO implementations this could be implemented in the constructor and destructor methods or at the garbage collection level. However the latter implies modifying the VM and we want to avoid intrusive implementations such as this one.

However, this does not prevent the application from using checkpointing techniques by saving the state to persistent storage not only at exit time but at regular intervals during the processing of requests or services.

Figure 3.3 shows the state machine diagram for a module. To manage the state transfer we employ the following methods: `get_state`, `save_state`, `trace_state`. `Get_state` retrieves the stored state from persistent storage. `Trace_state` function keeps track of the current state of the module by using a table in memory. `Trace_state` are optional checkpoints, in which the developer can periodically save the state to the scoreboard. This table is later dumped to persistent storage by using the `save_state` function at a predetermined point and/or at exit time (clean up). When the component is in processing state it will remain there processing and

Figure 3.3: State machine for a module with state transfer functionality

servicing transactions until it receives a signal to terminate.

The hash table approach can be extended to the use of a scoreboard table with attributes as columns and variable names as rows. A scoreboard table is open to different implementations but it could be implemented as a linked list of arrays. The design of the scoreboard has to be carefully done in order to foresee the number of attributes to be kept. A trade off of using a scoreboard table is concurrent access and the need of locks, which might impact performance. An alternative to using a table in memory is the use of global variables or structures for each of the elements we want to keep track of. However this approach has the usual drawbacks the use of global variables carry. It is common practice of programmers to avoid and minimize the use of global variables.

Figure 3.4 depicts the scenario of saving and loading the state after the component is initialised. At the time of initialisation the module reads the stored state from persistent storage. The configuration manager coordinates the life time of the module. Then, after loading the state the module is ready to process requests from clients or other modules. In this particular scenario tracing of the state is implemented during

Figure 3.4: Sequence diagram for a module with state transfer functionality

the life time of the module, however this is not required for all implementations. After the terminate signal is received the state is saved to persistent storage and the module exits.

3.4.4 Storing the State

There are applications and languages that are capable of serializing their current states in binary format, an exact representation of the data stored in computer memory. This method provides for very fast storing and restoring operations²; however, users lose the ability to alter the states by means of simple editing tools and state portability is lost. A benefit of using a XML schema for state representation is that it assists and facilitates hot swapping approaches that are deployed in heterogeneous distributed systems. Another benefit of XML data representation produces a version-independent representations of an object's state.

In order to provide maximum flexibility of state representation, as mentioned in

²An interesting comparison between Java XML serialization and .NET serialization is done in [24]

the proposed guidelines §3.2 and pattern §3.3, we decided to use XML format as standard data representation. XML stands for eXtensible Markup Language. It is a text-based markup language. XML is similar to HTML in the sense that it identifies data using tags. It is called extensible because it is not a fixed format such as HTML. In XML tags signify what the data means, rather than how to display it. XML is a language to define and describe other languages, *i.e.*, a metalanguage. XML is not just for web pages, it describes a class of objects called XML objects, it can be used to store any kind of structured information, and to encapsulate information in order to migrate it between different environments otherwise which would be unable to communicate [43].

An XML document must be well formed in order to be parsed correctly. A well formed document is one that follows the norms laid by the recommendation of the W3C [43]; *e.g.*, all elements must have a closing tag, all elements must be properly nested, all XML documents must have a root element, attribute values must always be quoted among other syntax and structural rules.

XML provides a mechanism to define constraints on the logical structure, this mechanism is the document type definition (DTD). “An XML document is valid if it has an associated document type declaration and if the document complies with the constraints expressed in it” [43]. This leads us into XML Schemata. XML Schema was approved as a W3C Recommendation on 2 May 2001 [55]. An XML Schema achieves some of the same functionality as DTDs in rather different ways. The Schema describes what can and cannot be included in an XML document. Both DTDs and XML Schemata are used to validate XML documents and to provide meta data information about those XML documents. Schema extends the functionality provided by DTDs.

We propose the use of XML Schemata over a DTDs since the former provides some advantages over DTDs such as, expressing hierarchies of data types, schemata are more verbose, widely available parsing tools, and the schemata themselves are XML documents among other advantages. We do not want our schema to be limited in terms of the XML vocabulary employed; that is, it should allow for unforeseen data. In order to provide for growth of our schema we have chosen a design with extensible

variable content by using the eXtreme eXtensibility design pattern [10] and the guidelines set by the World Wide Web Consortium [55]. The eXtreme eXtensibility design pattern describes means for designing XML Schemata that allow data to be gathered in an independent, distributed fashion, and with no limitations on the XML vocabulary. The latter is the feature we are most interested in. That is, aggregation of data to a base XML schema. An alternative to our XML schema design is to partition the schema from general to the particular. That is, to have a general hot swapping schema and then have particular schemata for different application domains.

Appendix D describes our Schema and explains how it can be extended in needed cases and provides an example. However, the decision on whether to use Schemata, DTDs or none is ultimately left to the designer. We discourage the practice of not using a metadata language for the associated XML state document. The proposed XML schema can be divided onto the following sections:

- Root element. It defines the branching point to a list of elements that constitute the root element. The root element can be extended by following the eXtreme eXtensibility design pattern.
- Content type. It proposes a basic list of elements definition, this is the extension to the root element definition.
- Basic definitions. This section provides element's attributes, element type and basic definitions.
- Array type definition. This type describes the primitive array object.
- IP type definition. Primitive description for storing IPs based on a regular expression definition.

The proposed XML schema (*cf.*, Appendix D) can handle data structures such as arrays, lists, unions, or plain variables. For example, in the case of preserving the state of a variable we find that there are three main elements to consider when preserving the state of a variable: a) variable name, b) type, and c) value . In our schema variable name and type are attributes of the tag and the value is enclosed in the corresponding open and close tags. However, not all of these elements are

```
<?xml version="1.0"?>
<state>
  <ip>134.117.5.236</ip>
  <old_var name="random" type="unsigned long">739490</old_var>
</state>
```

Figure 3.5: Example of a saved state in XML data representation.

required to be present in all cases. For example, storing an IP value only requires the name of the variable and the IP address. An example of a state representation in the proposed solution is depicted in Figure 3.5. In this example a module wants to keep track of requests coming from certain IP and it generates a random number. When this module is updated these values are saved following the proposed schema.

By having the corresponding attributes in the XML data representation we are introducing flexibility to our FST since loading restrictions, *e.g.*, new types, can be decided at the design time of the new module generation and it is not linked to the old generation or to the XML schema. This provides the advantage that the next generation of module designer is completely free to change the semantics of the saved state.

Since the process of saving the state cannot predict what changes the new module may have, our XML schema does not propose any type of mapping to the new values. We take a similar approach as Hicks §2.5.1 and Feng §2.5.5 do when it comes to mapping rules. The mapping is done in the new module, and it is the new module's responsibility to recognize and map the old attributes to the new attributes according to its mapping rules.

There are mainly two types of XML parsers with respect to their processing of the XML document: DOM and SAX. DOM parsers construct a Document Object Model by reading the contents of the XML file into memory, this approach is not well suited for large documents due to its memory and CPU usage. SAX parsers report parsing events *e.g.*, start and end elements to the application through callbacks. These events can be handled and action can be taken on finding required elements. Memory usage does not grow with the size of the document. The decision on whether to use SAX or DOM depends on the application and/or the designer.

3.5 Summary

In Section 2.7 we presented a table (§2.3) summarising some of the state transfer features for the introduced hot swapping techniques. The following is the mapping of the dimensions from Table 2.3 to our approach:

- **Component State.** The proposed FST requires the component to reach a quiescent state before swapping. That is, a component is in quiescent state if it is not performing transactions and any transaction it has initiated have already been terminated
- **State Transfer.** In this case state transfer is achieved by including state capture and state restoring functions or methods at the time the object or component is destroy or created
- **State Storage.** The FST provides the option to trace the state using memory allocation during the life time of the component or storing the state
- **Transferable Elements.** Elements that can be carried on to the next version are: user allocated data (global variables), and tables. Open handles to sockets or files is suggested to be done by capturing the semantics of it (§3.4.1). For example, for a file handle save the filename and the byte offset into the file, and attributes of the file (open for read, write, append, etc).
- **Version Coexistence.** Since the approach is mainly for state transfer, version coexistence relies on the application on which FST is being applied. However if coexistence is provided then provisions when retrieving the state have to be taken at load time. That is, since multiple stored version states might exist then the retrieved state might not be the latest state.

Chapter 4

HTTP Servers

The test case presented in this thesis is based on the Apache HTTP Server. Thus, in order to understand how our framework fits in Apache we introduce in this chapter the concepts behind an HTTP Server and some of its most relevant parts to the framework presented in Chapter 3. Section 4.1 begins with an overview of HTTP servers. Section 4.3 then introduces the Apache HTTP server.

4.1 An HTTP Server

In general, a HyperText Transfer Protocol HTTP server (the term *HTTP server* is commonly used as a synonym to Web server) listens for user agent's requests and then replies to them according to the HTTP protocol. The user agent or client (in most cases a browser) requests a resource (*e.g.*, data objects like HTML files, images, binary files, services, etc.). The server receives the request and maps the requested resource to a file or to a program which, in turn, produces the requested data. Finally the server sends this response back to the client.

Most HTTP communication is initiated by a user agent. A simplified overview of the user-agent (browser) server interaction is shown in Figure 4.1. The user clicks on a hyperlink or requests a Uniform Resource Locator (URL) on their browser. Then the web browser translates the URL into an address or link and establishes a TCP/IP connection with the server. Using this connection, the browser sends a request method (for example a GET request) to the HTTP server extracting the resource name from

Figure 4.1: Browser-HTTP server interaction sequence diagram.

the URL or the link.

Then the HTTP server accepts the request and processes it. It sends the resource in its response using the connection, and finally closes the connection. The browser receives the data and examines it. For example, an HTML document may contain links to further resources and/or services. The browser has to request these resources or services from the HTTP server. After receiving the last resource needed to display the HTML document, the browser successfully renders the page and waits for the user.

An HTTP server can be implemented in a few hundred lines. However a production server is far more complex. Here are some of the most relevant features of large scale HTTP servers:

- handles concurrent requests (multiprocessing / multi-threading)
- provides portability to multiple platforms
- offers compliance with HTTP server RFC 2616, and backward compatible

- capable of performing access control (security and authentication methods)
- provides dynamic web content (CGI, server scripts, server-side includes)
- supports for virtual hosts
- enables administrators for comprehensive configuration and provides means for programmers to add more features and functionality (flexible)
- provides proxy functionality
- applies changes to configuration and features without stopping the server (robust)
- provides status information provided by the server (logging)
- offers documentation for developers, other than comments in source code

4.2 Apache and its Hot Swapping Features

Apache is the most popular HTTP server in the market (*cf.*, Appendix B). Part of its success relies on its robustness, flexibility and broad range of functionality. This section presents the supporting principles and comparison to existing techniques for Apache's hot swapping capabilities.

We know that the problem of correct timing for performing a hot swap is, in general, undecidable (*cf.*, §2.2). Thus the approach some techniques follow is to leave this decision to the programmer. In DYMOS, *cf.*, §2.4.3, the timing for loading a module is based on conditions from other modules. For example:

```
update M when A,C, idle
```

This command updates module *M* when *A* and *C* are idle. In a similar fashion, Apache provides a *graceful restart* (described in detail in §4.3.2). A graceful restart allows components (processes) and transactions currently in execution to finish their tasks to completion. That is, the component is idle. For example,

```
httpd -k graceful
```

replacing a component	Argus	Apache
executing transactions	wait	http -k graceful
abort transactions	crash	http -k restart

Table 4.1: Similitude between Argus and Apache when replacing a component.

Argus (*cf.*, §2.4.5) is a fault tolerant system which is resilient to crashes. In order to replace components (guardians) the system can either abort currently executing transactions or wait for them to finish.

In Argus, to replace a component, the old component is crashed and the new is started. Even though crashing a process might seem radical, Apache provides a facility for “killing” all executing process (components) using a *restart* (described in detail in §4.3.2). Table 4.1 compares Argus and Apache on their approach on replacing components when these are executing. A key feature of Apache’s hot swapping implementation is the fact that it is implemented in a preforking architecture to support multitasking (*cf.*, §4.3.2). In a very similar approach Gupta implements its On-line Software Version Change (OSVC) technique (*cf.*, §2.4.7). That is, by running the user programs as child processes of the modification shell. OSVC uses the system call **ptrace** to inspect or modify the address space and machine registers of its child. OSVC provides a modification shell with a series of commands to handle running programs. Apache provides this functionality by reading commands from the command line and passing the control to the parent process, which in turn takes action with the child processes.

4.3 The Anatomy of Apache HTTP Server

Apache is a robust and W3C compliant implementation of an HTTP Server. In this Section we discuss the Apache HTTP Server. This will give us an understanding of Apache’s modular architecture and its hot swapping mechanisms.

As we describe in the following sections, and in the previous section, hot swapping capabilities are provided in Apache through its modular design, the Apache portable runtime, the preforking architecture and mainly the ability to compile the modules as dynamic shared objects (§4.3.3), these exist separately from the main httpd binary

file.

Apache's APIs and multi-processing modules facilitate and comply with the requirements set in Section 3.4.2. The APIs provide sufficient flexibility to implement the FST without external support or changes to APIs. Which in turn respects the exiting architecture. The capability to restart gracefully the server and its implementation of the leader/follower pattern allows the implementation of the FST for state transfer while there is no activity on the component.

4.3.1 Apache's APR

Apache has provided support for non-Unix like systems since version 1.3b3. Version 2.0 introduced the Apache Portable Runtime (APR). APRs and Multi-Processing Modules (*cf.*, §4.3.2) provide a faster and more stable support for non-Unix platforms such as BeOS, OS/2 and Windows. These platforms are now implemented in their native Application Programming Interface (API), avoiding the often buggy and poorly performing POSIX-emulation layers.

The APR enhances and adds new functionality to Apache's API. The APR abstracts all the system-dependent parts of Apache. This includes the following platform dependent features:

- File handling (Input/Output)
- Sockets
- Enhanced memory management
- Pipes
- Multitasking (threads and processes)
- Locking mechanisms (including file locking, interprocess locking, and interthread locking, asynchronous signals)
- Network Input/Output
- Dynamic linking, dynamic shared objects

- Time
- Authentication
- and any platform-dependent feature

A side benefit of this APR, is that Apache could be considered as a universal network server that can be enhanced to provide almost any functionality.

In the section that follows, some relevant features of the Apache HTTP server are explained, detailing the features that provide hot swapping mentioned in §4.2. We focus on the most recent Apache version.

4.3.2 Apache 2.0

Apache 2.0 is a major new version and introduces a good number of core and module enhancements, including better operating system portability, multi-threaded processing (where applicable), layered I/O, multi-protocol support and a new Apache API for modules.

Apache 2.0 has, among other new features, an easier portability to different operating systems (APRs §4.3.1), is multi-threaded, layered I/O (known as input/output filters), and it has IPv6 support. Apache's success is probably due, besides the fact that it is free, to its broad range of functionality and flexibility that provides to developers and administrators.

Multi-Processing Modules

Part of the flexibility that the Apache HTTP server provides is by its architecture design to work on a very wide variety of platforms in a range of different environments. Different operating systems and different environments require different features, or they may vary on the way they implement the same feature. Apache has proposed a solution to this problem by its modular design. The modular design allows the webmaster to select which features will be included in the server by choosing which modules to load either at compile time or at run-time [16].

The modular design of Apache 2.0 is extended to the most basic functions of the web server. Multi-Processing Modules (MPMs) are in charge of binding to network ports on the machine, listening for incoming requests, and dispatching children to

handle the requests. Extending the modular design to this level brings a number of benefits, [16, 21]:

- more clean and efficient support of wide variety of operating systems
- the server can be customized to the needs of the site
- modules are exchangeable
- modules encapsulate the multiprocessing architecture
- each module's structure is alike the standard module architecture

MPMs cannot be dynamically included and have to be selected at compile time. One and only one MPM must be loaded into the server at any time. In our test case the Apache HTTP server was compiled with the default MPM for the Unix/Linux platform. Apache 2.0 includes, at the moment, the following MPMs:

- BEOS - optimized for BeOS (Be Operating System). Single process model with threading
- Preforking and Netware – mpm_netware, an exclusively threaded web server optimized for Novell NetWare, default for Netware. Prefork a non-threaded, pre-forking web server, default for Unix
- OS/2 - mpmt_os2. Constant number of processes, variable number of threads.
- Windows - mpm_winnt, MPM optimized for Windows NT. Single process with threading
- Worker - implements a hybrid multi-process multi-threaded server, uses threads to serve requests, performs better than preforking
- Leader, PerChild and Threadpool - experimental MPMs, alternatives to pre-forking and worker for Linux OSs

Preforking Architecture

At start-up, Apache creates a pool of processes or threads (tasks), leaving most of

them idle. When a request comes in, it will be processed by an idle process/thread. The preforking architecture, used in our test case, is based on the Leader/Followers design pattern to coordinate work among threads and/or processes. The leader/follower pattern can be applied to event-driven applications, where multiple service requests arrive from a set of event sources and must be processed efficiently by various [multiple] threads that share the event sources [45].

On-line transaction processing applications often process high volume of multiple type of events which arrive simultaneously. To address this problem, the leader/follower pattern resolves three forces: 1) Service requests can arrive from multiple event sources. An important design force is determining efficient demultiplexing between threads (processes) and event sources. However, it may not be practical to assign or associate each event source to a thread. 2) Maximise performance, key sources of concurrency, such as context switching, synchronization, and cache coherency management, must be minimised. 3) Prevention of race conditions. These may occur if multiple threads try to modify or access certain types of events simultaneously.

The solution the leader/follower pattern proposes is as described below. Design a process pool mechanism that allows multiple processes to take turns dispatching (demultiplexing) and processing events. In this mechanism, one thread at the time, the leader, waits from an event to occur from a pool of event sources. Simultaneously, other processes, the followers, are queued up waiting their turn to become leader. Once the current leader detects an event from the pool of events, it first promotes a follower process to become the new leader. It then changes its role to processing thread, which demultiplexes and dispatches the event to a designated event handler that performs application-specific event handling in the processing thread. Multiple processing threads can handle events concurrently while the current leader waits for new events. After handling an event, a processing thread goes back to a follower role and waits in the queue for its turn to become leader again.

Figure 4.2 shows how Apache implements this pattern. In this mechanism one task (process or thread) –*the leader*– waits (*i.e.*, listens) for an event to occur (*i.e.*, request). Meanwhile, other tasks –*the followers*– are queued up waiting for a request to happen, then eventually becoming the new leader. Once a request arrives the leader changes its state to *processing*, it dispatches the request using the connection made as

Figure 4.2: Apache implementation of the leader/follower pattern.

listener. Multiprocessing threads or processes can handle requests concurrently while the new current leader listens for new requests. After handling the request, a child server checks the scoreboard and if applicable (*i.e.*, there is no change in generation), the child server reverts to a follower state; otherwise the child server dies. Each time the configuration is read, due to a graceful restart, a restart, or a fresh start of the web server, a generation of child servers (processes) is created. Apache uses a scoreboard to communicate between its parent and child processes. The scoreboard can be an anonymous shared memory area or a file-based shared memory.

The Life Cycle

One of the main aspects that differentiates among MPMs is the use of different means to create and organise child servers, also called workers, and the way these are created during the restart cycle. Aside from the underlying model chosen Apache has to go through the following stages [21, 16]:

- First-time initialisation. Allocate resources, read configuration, open logs, become a daemon (httpd process)

- Process the restart loop. Re-read configuration and launch several child processes
- Control the number of child processes launched, based on configuration directives
- Enter in the request-response cycle
- Process HTTP requests
- Clean-up before deactivation

If the port selected for server-listening is below 1024 -the default port is 80- then it is necessary to have root privileges to start Apache. Once started the main `httpd` process continues to run as root user, but the child server runs as a less privileged user. In order to stop or restart Apache, one has to send a signal to the running daemon. There are three signals that can be sent to the server: `TERM`, `HUP`, and `USR1`.

`TERM`

Sending a `TERM` or `stop` signal to the parent causes the server to immediately try to kill all of its children. Then the parent itself quits. `USR1` or `graceful restart` The `USR1` or `graceful` signal causes the parent to recommend the child servers to exit after their current request or to exit immediately if they are in *listening* state. The parent re-reads its configuration files and re-opens the log files. As each child dies the parent replaces it with a child from the *new generation*. It is important to note that this process respects the number of child servers and/or threads, maintaining the appropriate values throughout the restart process.

`HUP` or `Restart`

Sending a `HUP` or `restart` signal to the parent, like in `TERM`, causes the parent to kill all its children. However in this case the parent does not exit. It re-reads the configuration files and re-opens the log files. Then it spawns a *new generation of child servers*. Appendix C depicts the restart cycle.

Child initialisation

Child initialisation consists, in general, of the following steps:

- Get a sub-context for global allocations for the child, gets access to common resources like the scoreboard, creates memory pool and accepts mutex
- Reinitialize modules, these have been initialised by the server earlier. However it done a second time in case it allocates system resources or it depends on process ID (PID).
- Setup time-out handling. Apache uses alarms in order to avoid infinite blocking of the child server
- Initialisation of the alarm timer and clearing the transient pool used in the request-response pool
- Set status to ready in the scoreboard

4.3.3 The Module's Anatomy

In Section 4.3.2 we described the the life cycle of Apache. In this Section we explain how a module fits in the Apache's architecture and we describe the module's structure.

Building blocks

The Apache HTTP server is a modular program where the administrator can select the functionality in the server by selecting a set of modules. Modules are independent pieces of code which can provide or enhance functionality of the Apache HTTP server. Modules can either be statically or dynamically included with the core. For static inclusion, the module's source code has to be compiled into the httpd binary when the server is built. Alternatively, modules can be compiled as Dynamic Shared Objects (DSOs), these exist separately from the main httpd binary file. DSOs can be compiled at the time the server is built or they can be compiled at a later time with a tool such as Apache Extension Tool (apxs). Modules can be loaded at startup or restart of the server.

The DSO support to load individual modules is provided by a module named `mod_so` which must be statically compiled into the Apache core. There are only two modules that cannot be put into a DSO: `core` and `mod_so`. Practically all other modules can be placed into a DSO. Once the module is compiled into a DSO, lets

say `mod_test.so`, one can use `mod_so`'s `LoadModule` directives in the configuration file (`httpd.conf`) to load the module at server restart or startup.

On modern Unix-like operating systems there exists a mechanism analogous to the dynamic link library (DLL) on Windows. This mechanism is often called dynamic linking/loading of DSOs. There are two ways to do this loading. In the first way, the *share libraries* or *DSO libraries* are hard-coded references into the executable program so that at start-time the Unix loader (`ld.so`) is able to locate the library in path hard-coded or via linker option or in paths configured by the environment variable `LD_LIBRARY_PATH`. Then it resolves the unresolved symbols in the executable program which are accessible in the DSO. In the second way, usually called *shared objects* or *DSO files*, the files usually stay inside a program-specific directory and there is no automatically established link to the executable program were they are used. Rather the executable program manually loads the DSO at run-time into its address space via `dlopen()`.

The natural choice for implementations is the shared library approach, since it is what the DSO mechanism was designed for, and it is used for almost all types of libraries the operating system provides. On the other hand, only few software packages use shared objects to extend their functionality at run-time. However, since version 1.3 Apache uses the module concept to extend its functionality and internally uses a dispatch-list-based approach to link external modules into the core functionality of the server [16].

DSO advantages over Apache server:

- The server can extend its functionality at run-time because the actual server process can be assembled at run-time via `LoadModule` directive in the `httpd.conf` configuration
- Third party modules can be added even after server is built and installed
- `apxs` provides great flexibility in building and installation DSO modules into the server. The developer can work outside the Apache source tree. Thus running `apxs -ica` command followed by an `httpd -k restart` adds the module to the running Apache server

DSO disadvantages over Apache server:

- Not all operating systems support the DSO mechanism
- There is an approximately 20% decrement on performance at startup because of symbol resolution
- Execution time is an approximately 5% slower in some platforms
- There are certain platforms in which DSO modules are restricted to only use symbols from the Apache core, from the C library (libc), and all other dynamic or static libraries used by the Apache core, or from static libraries (*e.g.*, `libYYYY.a`) containing position independent code containing position independent code. There are two options to fix this: A) manual loading using `dlopen()`, or B) ensuring the Apache core itself already contains a reference to it.

Modules interact with the Apache core through a common programming interface. Apache provides an Application Programming Interface (API) that helps isolating the modules from the HTTP protocol mechanics and from each other. Each module can access the server's data structures through the API. The Apache API provides a wide set of functions that a module can call. The functions receive as parameters complex data structures and return complex data structures.

Each module contains a module-info about the handlers provided by the module itself and the configuration directives the module can process. Module info holds references to four handlers for predefined hooks used for configuration purposes only. All other handlers for hooks are registered by using the function commonly named `register_hooks`. Inside this function the module can add its functions to the list of functions called for each hook.

All Apache processes contain the same executable code. That is, the master server and child server are identical. The executable code consists of the core, the statically loaded and the dynamically loaded modules. Figure 4.3 illustrates the module's structure. The hooks (operations) depicted in `register_hooks` class are not required to be implemented as part of this class.

We can see in Figure 4.4 how modules and the Apache core interact. A module is composed in general of the following parts: a module info, a configuration handler,

Figure 4.3: Class diagram of a module's structure.

Figure 4.4: Overview of a module's anatomy.

register hooks handler, function handlers, optional functions and filters. Each part is called at some point during the life of the component. A component is formed of multiple modules, the life of the module is tie to the life of the component. Modules interact with the server core through Apache's multiple APIs.

Modules register handlers for hooks in the core or in other modules. The Apache core calls all registered hooks when appropriate. A hook is a transition in the execution sequence where all registered handlers will be called, thus triggering an event that executes event handlers. A module can implement four types of handlers:

- Handlers for hooks. A hook function is one that Apache will call at some point during the processing of a request
- Handlers for processing configuration directives. A module can provide its own set of configuration directives which can be used in the configuration file. For example, our approach uses these directives to implement timeout for state retrieval, and path and file name for the persistent storage.
- Filters. A filter is a process that is applied to data sent or received by the server. Data sent by the clients is handled by input filters and data sent by the server to the client is handled by output filters. For example, deflate filter compresses data before sending it to the client.
- Optional functions. This feature lets a module register functions with the core as optional. This allows other modules to use those functions, the modules query the core server to check if those functions have been registered. The core uses the name of the function as the key to find the pointer that was stored when the function was registered.

Hooks have some advantages. The list of hooks can be extended arbitrarily. Optional hooks can export their own hooks. Although hooks are dynamic, these are type safe; that is, the compiler will complain if the type of the function registered for a hook does not match the hook. In general, a hook is a point at which a module can request to be called. Each hook specifies a function prototype, and each module can specify one or more functions that get called at the applicable moment.

Figure 4.5: Hooks available to modules at different stages of the child life cycle.

Figure 4.5 shows the hooks available for modules, these are grouped in two general categories: *Server Setup* and *Processing a Request*.

4.3.4 Request Processing

The request-response cycle is the most important task of the HTTP server. Every Apache child server handles the request-response cycle as long as it lives (*cf.*, §4.3.2). The following section describes the life cycle of the child servers. Figure 4.6 shows the request-response loop.

Depending on the MPM model chosen the child server will behave differently under each MPM model. The child server is suspended until the mutex or the job queue signals either that it will become the next listener or that a new job is in the queue. The transitions in the “Wait for TCP request” depict the leader-follower model for

Figure 4.6: Request-response cycle in Apache [21]

Preforking MPM. The child server acquires a mutex and it is suspended until the TCP connection is established and then it releases the mutex. Then it follows the hooks shown in the rectangle “Pre-processing a request” in Figure 4.5. The HTTP/1.1 protocol allows persistent connections, allowing a client to make multiple requests without waiting for each response, this translates in a more efficient use of a single TCP connection. This is depicted in the keep-alive loop. The TCP connection is negotiated in the headers of both the request and the response. Once this connection is established the child server reads the header and processes the request. Modules that implement the handler hook are called and then finalise the request cycle, afterwards the response is sent.

Chapter 5

Implementing State Transfer in Apache

Apache provides an underlying model for hot-swapping. As mentioned in Chapter 4 the server's functionality can be enhanced and changed to fit almost any environment. In this chapter we apply our FST to Apache modules. Apache's preforking architecture is used for the implementation of the test case. Note that Preforking is the default Multi-Processing Module (MPM) for Unix platforms. We profit from Apache's modular architecture and dynamic implementation to implement our test case.

5.1 Overview

We focus our attention on transferring the module's state from one child server generation to another after the administrator sends a graceful restart or a restart to the Apache web server. Recall that a child server is constituted of several modules (see § 4.3.3). In order to implement our test case we replicate an existing Apache module: `mod_status`, referred to as `mod-A`. The replicated version is referred to as `mod_server_s` or `mod-A'`. This module keeps track of a significant number of Apache's internal data; like performance (*e.g.*, server uptime, requests per sec, Kb/sec, Mb/sec, total accesses, etc.) and the status of all child processes.

`mod_status` module is used as a case study because the internal data that it

gathers during execution is reset each time the administrator does a restart or a graceful restart. Our implementation transfers variables (type and value), and internal computations considered as state. We do not transfer open handles, like open sockets or connections to clients, since these are closed after the connection is terminated, *i.e.*, files and sockets are transient. In order to activate `mod_status`, requests are sent to `http://localhost/server-status` URI (*i.e.*, load the page in the browser is equivalent to requesting the module); correspondingly, `mod_server_s` module is activated by requesting the `http://localhost/server-s` URI.

In the context of the HTTP server and following the criteria set in Section 3.2 we respond to the guideline questions proposed there:

- What is relevant to capture? In our case we want to keep track of access counter since this gets reset every time a restart is done. The access counter is the total number of objects requested from the web server.
- How does one capture the state? Before the child server dies the counter is copied to a hash table in memory.
- When is state capture needed? Since the internal computation we want to save is volatile only when the child server dies, tracing the state is done on child's exit.
- When is it safe to perform the transfer? The state is loaded when the main server spawns a new generation of child servers and when each child server is initialised.
- What issues might prevent the state from being saved or loaded? Policies that might indicate state expiry time outs, a corrupt state, race conditions encountered if multiple restarts are done, among others.
- What to do if state cannot be saved/loaded? If the state is not loaded the module is initialised to a default behaviour similar to the un-modified version of the module.

As described in Section 3.4.2 the features of our implementation are:

- *Non-intrusive to the existing architecture.* The framework we present builds up on Apache's modular architecture without adding or modifying the existing APIs.
- *Minimal added overhead.* The penalty for adding state transfer to modules is quite low, < 5% (see §5.5).
- *Ease of use.* In general, the less complex a new framework is the easier it is to be adopted and implemented.
- *Maximum flexibility of state representation.* We save the state in a format that is readable across platforms. Thus we developed an XML schema to be used when saving the state.

As mentioned earlier, a child server is composed of a series of modules. The module is the unit of replacement in Apache's design. The life of the module is tied to the life of the child server. In order to replace a module the whole child server has to be swapped out. However, modules in the core Apache server cannot be hot-swapped. Each module provides and implements different services to serve requests from clients.

Although, our implementation was done under Linux this implementation can be extended to Apache's Win32 architecture. The initialisation phase of the Win32 architecture is akin to the preforking architecture (see §4.3.2). Under Win32 architecture the master process supervises that the worker process works correctly. When started the worker process only contains a single master thread which in turn it spawns a fix number of threads, these are equivalent to child servers under the preforking architecture. The Windows multitasking architecture only supports graceful restart or shutdown.

5.2 FST Implementation

As mentioned previously, the state is loaded when the child server is initialised, *i.e.*, spawning of child servers, process creation, and saved when the child server dies, *i.e.*, process death. Table 5.1 shows the conditions under which child servers are spawned or

Spawning	Killing
First-time initiali- sation	
Restart (normal or graceful)	
Server-Pool Size Regulation (MPM specific)	
	Stop

Table 5.1: Child servers initialisation and death under different conditions.

killed.

The administrator controls the master server by sending signals directly to it. The three main signals the server responds to are:

- SIGTERM – stop server, *i.e.*, `httpd -k stop`
- SIGHUP – restart server, *i.e.*, `httpd -k restart`
- SIGUSR1 – perform a graceful restart, *i.e.*, `httpd -k graceful`

The first signal causes Apache web server to stop, *i.e.*, kills the child servers, and thus saving the child servers' state. The latter two spawn new child servers, causing the child servers to load the state.

The following subsections depict sequence diagrams and state diagrams that are based on the state machine and sequence diagrams presented in Section 3.4.3; in particular Figures 3.3 and 3.4 respectively.

5.2.1 Loading and Saving a Module's State

Figure 5.1 depicts the scenario of initialisation of the web server. The child servers are composed of multiple modules and these are depicted in Figure 3.4 as modules. The first time the Apache web server is started it follows the same initialisation process to that of any Apache web server not implementing FST. In case there is a state in persistent storage and it is valid then the state is loaded. A state is valid if it complies with state transfer policies set by the designer, *e.g.*, expiry date and time of the state in persistent storage, and it is not corrupt. In Apache web server we can implement policies, (see §3.2.1), in three different areas:

Figure 5.1: Loading a state after an `httpd -k start`.

1. In Apache's configuration file, *i.e.*, `httpd.conf`
2. In the XML-state file, by coding the state transfer policies when the child server is killed
3. Hard coded as part of the module

Apache allows the administrator to introduce new directives into the configuration file (*cf.*, §4.3.3). These are read the first time the web server is initialised. An alternative is to use the `child_exit()` function to code the policies into the XML-state file. Then, when the state is parsed the directives are applied.

Figure 5.2 shows the scenario of saving the state when the Apache web server is restarted. This scenario is equivalent, in terms of saving the state, to a stop (shutdown) followed by a start. Similarly to Figure 5.1, child servers map onto modules in Figure 3.4 and Admin and Server entities are grouped as Configuration Manager. For a shutdown or a non-graceful restart the master server sends a `SIGHUP` to the pool of child servers. The operating system takes care of the distribution of the signal. After

Figure 5.2: Saving a state after an httpd -k restart.

receiving the signal the child servers process the `trace_state()`, `save_state()`, and `child_exit()` functions. The master server then gets a notification regarding that the child processes are terminated. The master server restarts the initialisation and spawns a new generation of child servers.

Figure 5.3 depicts the scenario of saving and loading the state after the administrator sends a graceful restart to the server. A graceful restart kills idle child servers and sends a “Char or Death” (CoD) to the “Pipe of Death” (PoD)¹. Busy child servers check the PoD after these finish serving client requests. These child servers check, as well, the scoreboard in order to compare their generation against the current one. The child server waiting for connections (the listener) receives a dummy_connection that is terminated immediately forcing the listener to check the PoD and the scoreboard, and eventually dying.

Before exiting each child server gets the state saved in memory by using `state_trace()` function and saving it to persistent storage before child termination.

¹The master server uses either signals or a pipe to control the child servers

Figure 5.3: Loading and saving a state after an `httpd -k graceful` restart.

Figure 5.4: Child Server State Machine.

5.2.2 The State Machine of a Child Server

Figure 5.4 depicts the state machine of a child server after it has been spawned by the master server. This state machine is a particular application of the state machine depicted in Figure 3.3. However, two states are added for this implementation: *child_server_pool* and *leader*. The *child_server_init* state reinitialises the child server, the child server was initialised previously by the master server, a second reinitialisation is done in case there are system resource dependencies or allocations. During this process the state is loaded. *child_init* is a predefined hook which Apache provides. However not every module implements or uses this hook. If the module wants to provide hot swapping this has to be added to the module. As mentioned earlier, in the graceful restart scenario, the state is loaded if it complies with predefined policies. If the state is not loaded, and it is required in order to perform properly, then the default state takes over.

After the *child_server_init* state is processed the child server inspects *check_leader* to

define its role. If there is an existing leader waiting for incoming connections the child server goes to the *child_server_pool* state. If there is no leader, the child server goes into *leader* state and waits for a connection request. In order to accept connections the preforking architecture uses an accept mutex which distributes input connections among multiple child servers. The accept mutex is a semaphore used to enforce mutual exclusion for access to a resource. It controls the access to the TCP/IP service and guarantees that, at any time, exactly one process waits for TCP/IP connection requests. If a restart signal comes the child server transitions into *child_server_cleanup*.

Once a connection arrives the child server releases the mutex and starts *processing* requests. If required, as part of the hot swapping modifications, checkpoints can be added to the request processing hook, shown as *trace_state* on the *processing* state. If checkpoints are added, the state is traced by using a hash table in memory which later, at *child_server_cleanup* state, this trace will be saved to persistent storage. Graceful restarts and requests on the same connection keep the child server in the *processing* state.

After the child server finishes processing the requests for a connection, it validates if it belongs to the same generation. If the child server belongs to the same generation it returns to the *child_server_pool* state. If there was a restart or a graceful restart it moves to *child_server_cleanup* state before it dies. During this *child_server_cleanup* state a supplementary action is added in order to save the state to persistent storage.

As most of the techniques reviewed in §2 Apache performs a quiescent type of hot swapping.

5.3 Parsers

Two open source XML C parsers were considered: libxml2 [52] and gdome [9]. The implementation for our application was done employing both libraries in order to provide a better judgment of choice for the reader. Libxml2 is known to be very portable. We discerned that for small XML state transfer files the difference in performance can be neglected (see Table 5.5). On the other hand we found libxml2 to be more supported and more flexible than gdome. However the ultimate decision on whether to use libxml2, gdome or any other parser is left to the designer and/or programmer.

```
module AP_MODULE_DECLARE_DATA s_server_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                /* dir config creator */
    NULL,                /* dir merger -- default is to override */
    NULL,                /* server config */
    NULL,                /* merge server config */
    status_module_cmds,  /* command table */
    register_hooks       /* register_hooks */
};
```

Figure 5.5: Standard Apache Module Structure.

5.4 Code Samples

As mentioned previously the `mod_status`, or `mod-A`, module is used to implement the FST. In general, code-wise, each module ends with its module structure (see Figure 5.5). This is the module's info, illustrated as an attribute in Figure 4.3. References to handlers are included in a handler commonly named `register_hooks`, the name may vary. Except for the configuration management hooks, all handlers are dynamically registered using this function. As described in 4.3.3, hooks are functions that will be called at some point during the processing of a request. The command table, `status_module_cmds` is the hook used for reading directives from the configuration file, `httpd.conf`, our implementation adds directives (*cf.*, 5.8) explained below.

`register_hooks` provides handlers for each module. In Section 4.3.3 we presented graphically (see Figure 4.5) the predefined hooks available to modules. In Figure 5.6 we show the use of `ap_hook_child_init` as part of our solution to load and save the state. The three hooks (`handler`, `post_config`, `child_init`) are, in this module, part of the original implementation. The latter two initialise the module. However `child_init` was enhanced with our function `get_state()`, which is described below. If a module wants to implement hot swapping, and it is not part of its implementation, then it has to include `ap_hook_child_init` as part of its hooks.

There are more hooks that can be included in `register_hooks`, whether these are custom or predefined hooks (see Figure 4.3). As mentioned earlier, we use the predefined hook

```
static void register_hooks(apr_pool_t *p){
    ap_hook_handler(status_handler, NULL, NULL, APR_HOOK_MIDDLE);
    ap_hook_post_config(status_init, NULL, NULL, APR_HOOK_MIDDLE);
    ap_hook_child_init(x_child_init, NULL, NULL, APR_HOOK_MIDDLE);
} /*end register_hooks() */
```

Figure 5.6: Available hooks for modules.

```
/*
 * Our process initialiser reads state from file
 */
static void x_child_init(apr_pool_t *p, server_rec *s) {
    xml_parse = get_state(pathstate,s);
    apr_pool_cleanup_register(p, s, x_child_exit, x_child_exit);
} /*end x_child_init() */
```

Figure 5.7: Child server initialiser including `get_state()` and `x_child_exit`.

`ap_hook_child_init` to include our functions for state transfer.

Figure 5.7 shows the additional functions in `child_init` hook. The first function, `get_state()`, retrieves and loads the state from persistent storage. The second function, `apr_pool_cleanup_register`, saves the state to persistent storage by calling `x_child_exit`. Note that the original `mod_status` module does not implement method calls in `child_init` function, it only gets the process id of the child server.

The variable `pathstate` in Figure 5.7 is configured through the configuration file *httpd.conf* as a directive. State-related policies and restrictions can be included in the global configuration file, *httpd.conf*, within the section corresponding to the module's directives, - e.g., inside `<IfModule mod_server_s.c>` and `</IfModule>`. Figure 5.8 depicts module's directives relevant to a module implementing FST.

The function `get_state()`, can be implemented using a DOM or SAX parser library. This function implements restrictions and policies for the intended objects to be retrieved.

```
# StateTimeout life of state file in secs. i
# StateTransferFile path to temp state files
#
<IfModule mod_server_s.c>
StateTimeout 5
StateTransferPath /tmp/httpd
</IfModule>
#end hot_swap
```

Figure 5.8: Including directives as policies for state transfer in the httpd.conf file.

The default behaviour can be defined in the XML file or inside the `get_state()`. Figure 5.9 is a partial example of a parsing using *gdome* parser. In this figure functions starting with `ap_` are part of Apache's API, such as `ap_log_error`. Functions starting with `gdome` are part of *gdome*'s library. The code represents the forming of the tree in memory from persistent storage. After reading the node for the tag `old_car_count` its attributes are retrieved in the for loop following the `get attributes` comment.

As mentioned previously, the routine `trace_state` is used to add a trace for the functions that wish to keep track of their state. This is the checkpoint activity, `trace_state` in *processing* state, for the module (see Figure 5.4).

In `trace_state()`, *cf.*, Figure 5.10, we are passed the server record (if available), the request record (if available), a pointer to our private configuration record (if available) for the environment to which the callback is supposed to apply, and the state to be saved. We turn this into a textual representation and add it to the hash table. We profit of Apache's memory allocation mechanisms and we use `apr_table_set` as the hash table, in a similar fashion we use `apr_pstrcat` to manipulate strings.

Finally, we have the routine `x_child_exit()`, *cf.*, 5.11, which is invoked when a restart or graceful restart is done. This function calls `trace_state()` (if needed) and then calls `save_state()` to save the state to persistent storage conforming the XML schema defined in Appendix D. Note that `trace_state()` is not needed in every implementation since some internal computations are global to the module and can be retrieved directly when calling `save_state()`. Remember, `x_child_exit` is derived from `apr_pool_cleanup_register` in `x_child_init`, *cf.*, Figure 5.7. A module implementing hot swapping with the proposed FST would have to include this function as well as `save_state`. `x_child_exit` retrieves

```

/*****HS xml parsing *****/
static int get_state(char *filename, server_rec *s) {
: (part of the code not listed)
    /*----- constructor of th DOMImplem. struct -----*/
    domimpl = gdome_di_mkref();

    /*----- load state from xml file -----*/
    doc = gdome_di_createDocFromURI(domimpl,filename,
                                   GDOM_LOAD_PARSING,&exc);
    if (doc == NULL) {
        ap_log_error(APLOG_MARK,APLOG_ERR, 0, s,
            "mod_server.s.c get_state failtcreate error: %d",exc);
        return(1);
    }

    /*----- get root elem of doc ----- */
    root = gdome_doc_documentElement (doc, &exc);
    if (root == NULL) {
        ap_log_error(APLOG_MARK, APLOG_ERR, 0, s,
            "mod_server.s.c get_state failtogetroot: %d",exc);
        return(1);
    }

    /*----- get a nodelist with all children of this node (root)*/
    childs = gdome_el_childNodes (root, &exc);
    if (childs == NULL) {
        ap_log_error(APLOG_MARK, APLOG_ERR, 0, s,
            "mod_server.s.c get_state failtogetnodes: %d",exc);
        return(1);
    }
: (part of the code not listed)
    /*----- get attributes -----*/
    attrs = gdome_n_attributes(nd,&exc);
    num_elms = gdome_nnm_length(attrs,&exc);
    for (i=0; i < num_elms; i++) {
        nd = gdome_nnm_item(attrs,i,&exc);
        name = gdome_n_nodeName(nd,&exc);
        value = gdome_n_nodeValue(nd,&exc);
    } //end for
: (part of the code not listed)
    return(OK);
} /*end get_state() -----*/

```

Figure 5.9: Partial listing of `get_state()` function.

```
static void trace_state(server_rec *s, request_rec *r,
                        state_cfg *mconfig, const char *note,
                        const char *value) {
    char *where;
    apr_pool_t *p;
    char *key;

    apr_pool_create(&p, NULL);

    where = (mconfig != NULL) ? mconfig->loc : "nowhere";
    where = (where != NULL) ? where : "";

    key = apr_pstrcat(p, note, ":", where, NULL);

    /*----- add key/value pair to the table -----*/
    apr_table_set(static_calls_made, key, value);
} //end trace_state()
```

Figure 5.10: Partial listing of `trace_state()` function.

counters from Apache's scoreboard, because it is part of the data we want to preserve, depicted in the for loop after the comment `get info from scoreboard`. The state then is saved using our `save_state` function which is a normal implementation of an XML data representation. That is, `gdome` library functions are used to generate the tree and later on save it to disk.

The state is saved in persistent storage using, either SAX, DOM, or any other XML library. Thus the routine `save_state()` is implemented akin `get_state()`. We have two implementations of `save_state()` and `get_state()`. One using SAX and one using DOM. This was done in order to measure performance.

In some cases it is impossible to foresee the new type or constraints of the variable, structure or object receiving the old state. By having the corresponding attributes in the XML data representation we are introducing flexibility to our FST since loading restrictions, *e.g.*, new types, can be decided at the design time of the new module generation and it is not linked to the old generation or to the XML schema. Figure 5.12 depicts a state saved from `mod_server_s` module in our implementation.

5.5 Performance Evaluation

In order to evaluate the overhead introduced by the FST in a module, *httpperf* [37] and *autobench* [5] were used to perform benchmarking measurements.

httpperf is an application to measure web server's performance. It generates a variety of HTTP workloads. The program is written in C and compiled under Linux. Among its features we can point out its ability to generate and sustain server overload and its support for the HTTP/1.1 protocol. The latter is an important feature since it provides persistent connections to the web server.

Autobench is a Perl front-end to *httpperf* which automates the process of testing, and provides tools to generating a file with results. It generates HTTP loads by increasing the number of requested connections per second on each iteration.

Before describing the performed tests we point out some notes. The tests performed on the server are not stress or performance-tuning tests. The testing is mainly done to provide a measurement on the impact of the FST in a module. Appendix E describes the environment used for the tests performed to evaluate the FST implementation.

Two types of tests were done. The first one consisting of a simple static benchmarking. The second one an automatic static benchmarking. Due to the number of connections and

```

static apr_status_t x_child_exit(void *data) {

: (part of the code not listed)

/*==== get info from scoreboard -----*/
    for (i = 0; i < server_limit; ++i) {
        ps_record = ap_get_scoreboard_process(i);
        for (j = 0; j < thread_limit; ++j) {
            ws_record = ap_get_scoreboard_worker(i, j);
            res = ws_record->status;
            if (ap_extended_status) {
                lres = ws_record->access_count;
                if (lres != 0 ||
                    (res != SERVER_READY && res != SERVER_DEAD)){
                    count += lres;
                }
            }
        }
    } //end for thread_limit
} //end for server_limit
trace_state(s, NULL, hcfg, "count", apr_ltoa(s->process->pool, count));

/*==== write state to xml =====*/
if (!apr_is_empty_table(static_calls_made)) {
    save_s = save_state(s);
    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, s,
        "mod_server_s.c child_exit save state");
} else {
    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, s,
        "mod_server_s.c child_exit empty table");
}
return APR_SUCCESS;
} //end x_child_exit()

```

Figure 5.11: Partial listing of `x_child_exit()` function.

```
<?xml version="1.0"?>
<state>
  <ip>134.117.5.236</ip>
  <old_var name="count" type="unsigned long">739490</old_var>
</state>
```

Figure 5.12: Example of a saved state in XML data representation.

requests sent to the web server, child servers are spawned and killed on a self regulating behaviour based on web server configuration. Thus there was no need to artificially force restarts to load and save the state. Because our approach does not modify Apache's hot swapping mechanism no overhead is added to it. Modules are loaded by sending a graceful or a restart signal to the server. However our approach does add overhead for loading and saving the state. This is what we are measuring with the following benchmarks.

Simple Static Benchmarking

Table 5.2 shows the resulting comparison between `server-status` (`mod_status`) and `server-s` (`mod_server_s`) modules after running the simple static test. The test opens a total of 1,000 TCP connections at a rate of 120 per second. On each connection 10 HTTP calls are performed (a call consists of sending a request and receiving a reply). The timeout is set to 10sec. The timeout is the number of seconds that the client waits for a response from the server. If this timeout expires, then *httperf* considers the corresponding call to have failed.

	server-status original mod.	server-s state transfer	change	percentage
Connection rate [conn/s]	36.5	37.3	+0.8	+2%
Connection time [ms] avg.	16,182.7	15,798.2	-384.5	-2%
Request rate [req/s]	118.6	119.7	+1.1	+1%
Reply rate [replies/s] avg.	109.0	109.9	+0.9	+0.9%
Reply time [ms]	730.0	798.9	+68.9	+9.4%
Errors total	696.0	701.0	+5.0	+0.7%
Errors client-timeout	639.0	670.0	+31.0	+4.8%

Table 5.2: Simple static test for `mod_server` and `mod_server_s`.

The readings for the FST module, `server-s`, in the simple static test (*cf.*, 5.2) show marginal overheads and latencies of the `server-s` module with respect to the original module,

server-status.

Automatic Static Benchmarking

The automatic static testing consists of running *httperf* nine times against the web server. The test increases the requested request rate each time it runs *httperf*. In this test the connection rate is increased by a step of 20 connections per second, starting at a rate of 40 connections per second and finishing at a rate of 200/s. Note that on each connection 10 requests are made, thus the starting requested rate is 400 req/s. The number of TCP connections remained the same as in the first test, at 1,000 TCP connections. The timeout is set to 10sec. At the end of the ninth run the number of requests per second demanded is 2,000.

In Figure 5.13 the x axis represents number of requests per second and the y axis represents the maximum reply rate. On average, comparing mod-A (server-status) to mod-A' (server-s), the latter shows a better reply rate in the automatic static testing than in the single static testing. We believe that the increment in mod-A' (server-s) is due to the gradual increment of requests to the server. In this figure both modules show a curve with small variations between each other after 800 requests per second are demanded. mod-A' reaches a larger reply rate than mod-A, this is most likely due to the spawning of child servers the server having a better response when running the test again mod-A'.

Table 5.3 shows the numeric values corresponding to Figure 5.13. We believe that the differences are due to the ramp-up process of increasing demanded requests to the server. In this table positive percentage numbers reflect an overhead when testing against mod-A'. This overhead is caused by the state transfer when server controls the number of minimum and maximum spare servers to serve clients (*cf.*, §E). Nevertheless the overhead introduced is not very large and certain well under the 10

Table 5.4 shows the connection rate, request rate, and response time comparison between mod-A (server-status) and mod-A' (server-s). This second table is an aggregation to Table 5.3. It is important to note that average reply time is 5.26% in mod-A' as opposed to the 9.4% increment mod-A' had during the simple static test. Note in the table that not in every run the reply time for mod-A' was grater than mod-A; for example, when 1,400 requests per second were made mod-A' replied at 574.8ms as opposed to 839.9ms with mod-A.

Simple tests were performed to benchmark parsing times for SAX and DOM implementations. The XML data representation file used was the same as in the benchmarks for

Figure 5.13: Automatic static testing comparison for **server-s** and **server-status**.

`mod_status_s` module (`mod-A'`). Table 5.5 shows the parsing time for the XML state representation shown in Figure 5.12. As mentioned in Section 3.4.4 the choice for using SAX or DOM, or any other parser, is left to the designer or programmer. Similar performance tests could be done before deciding which parser to use. Two C programs were tested performing the same operations as the `get_state()` and `save_state()` in `mod_servers_s` module. We can note that retrieving and forming the tree, in both parsers, takes longer than saving the file to disk.

It is important to remark the spawning rate of child servers. Apache has a process for performing idle server maintenance. This process is called whenever a timeout occurs and the graceful restart has finished. A timeout happens when there are remaining children to start after all terminated child servers have been replaced. For operating system-performance reasons the master server does not immediately create the needed number of child servers. This process is done exponentially, the idle spawn rate is the number of child servers that will be spawned on the next maintenance cycle if there are not enough idle child servers. It is doubled up to a maximum spawn rate -set to a default of 32-, and reset only when a cycle goes by without the need to spawn [16].

D demanded request rate/s	mod-A				mod-A'				change avg.
	reply rate [rep/s]			std dev	reply rate [rep/s]			std dev	
	min.	avg.	max.		min.	avg.	max.		
400	100.0	107.3	114.1	4.4	97.2	108.0	118.4	5.4	+0.652%
600	93.4	105.7	116.6	7.4	89.2	105.5	111.2	8.6	-0.189%
800	90.9	105.1	115.1	8.9	89.9	106.7	121.2	10.3	+1.522%
1000	89.1	104.4	115.2	8.8	95.2	108.0	120.2	8.8	+3.448%
1200	92.0	105.2	117.2	8.6	96.0	106.1	112.6	6.8	+0.855%
1400	91.8	102.9	109.5	7.0	94.6	104.8	111.0	6.3	+1.184%
1600	94.0	104.5	109.6	6.3	95.4	104.8	109.4	5.5	+0.287%
1800	92.4	104.3	110.2	7.2	95.8	105.5	113.0	8.5	+1.150%
2000	92.0	103.4	107.8	6.6	93.0	106.0	112.6	7.9	+2.514%

Table 5.3: Automatic static test for **mod-A** and **mod-A'**. Reply rate.

D demanded request rate/s	mod-A			mod-A'		
	conn rate [conn/s]	req rate [req/s]	reply time [ms]	conn rate [conn/s]	req rate [req/s]	reply time [ms]
400	21.0	112.7	795.8	20.7	113.2	817.2
600	26.9	113.0	757.8	29.3	113.7	741.9
800	30.4	113.7	808.5	31.5	115.1	787.7
1000	32.1	113.5	806.4	32.7	118.0	773.4
1200	32.9	114.6	579.7	34.1	118.2	768.5
1400	34.8	114.5	839.9	34.4	116.8	574.8
1600	35.3	116.9	429.2	35.5	116.6	742.4
1800	35.6	116.7	544.9	35.7	116.4	643.0
2000	35.6	114.4	393.8	36.0	116.7	420.7
Averages	31.62	114.44	661.77	32.21	116.07	696.62

Table 5.4: Automatic static test for **mod-A** and **mod-A'**.

[ms]	get_state()	save_state()
Gdome	1,795.49	967.62
SAX	989.45	723.82

Table 5.5: Average time comparison for SAX and Gdome implementations.

Chapter 6

Conclusions

6.1 Summary

The main goal of this thesis is to present a framework for state transfer in hot swapping applications. We summarised the essential characteristics of hot swapping applications.

In Chapter 2, a comprehensive review of past and relevant research, mechanisms, frameworks, languages, and operating systems to approach hot swapping was presented. Drawing from the analysis done on these systems we presented an abstract representation of hot swapping systems.

Chapter 2 lead us into a discussion of state transfer in Chapter 3. As a result of the review of past techniques we presented guidelines to approach state transfer in hot swapping applications. Drawing from these guidelines we proceeded to propose an individual pattern for state transfer. We introduced a state representation based on a proposed XML schema.

In order to demonstrate the applicability of the state transfer in hot swapping applications, an implementation was developed (Chapter 5). The implementation was adapted to the most popular HTTP server: The Apache HTTP Server. Thus, in Chapter 4 we began by introducing the concepts behind an HTTP Server. We then, discussed the fundamentals of Apache's hot swapping features.

We enhanced Apache modules by applying our FST framework to a module which reports server status. Performance tests were conducted for SAX and DOM libraries to parse the XML state representation. Measuring performance in hot swapping applications is a non-trivial task, on the other hand not much data is found on this regard (§2). However, we decided to perform comparison tests between the modified version of the module and the

original version in order to obtain the added overhead after implementing the FST (§5.5).

6.2 Contributions of the Thesis

The research presented in this thesis makes a number of specific contributions that orbit around the design and implementation of our FST framework:

- We have compiled an essential set of characteristics of hot swapping applications.
- We proposed an abstract representation of hot swapping architectures.
- We have developed guidelines for state transfer in hot swapping applications are proposed.
- We proposed a XML schema class for data representation for state transfer. This is flexible enough to be used under existing hot swapping applications.
- We have defined a pattern for state transfer in the context of independent components that lose their state after reinitialization.
- We show measurement of overhead when performing state transfer. Ours is one of the few that document performance data, at least during state transfer. We have demonstrated that the our approach provides a minimal overhead, which meets the $< 10\%$ requirement originally proposed.
- We have presented a comprehensive survey on hot swapping techniques.

6.3 Future Work

Hot swapping techniques have been studied for some time now. It is only recently that commercial or open source applications are starting to implement these mechanisms into their systems. Nevertheless there are many open research questions. The following as possible directions to enhance either the present research or some of the reviewed techniques (see §2):

- The present work provides enough material in order to extend the State Transfer design pattern into a pattern language.
- Rollback operations can be considered and matured under the proposed framework.

- Our FST framework could be extended to implement process migration or utilized for checkpointing techniques (see §3.1).
- In this thesis we have implemented our FST for Apache HTTP modules. Some investigation can be done in order to extend FST to the Linux operating system and its loadable kernel modules (LKMs). Under the kernel environment we face restrictions and new challenges. As opposed to applications, kernel modules are not user space programs and as such they run with no restrictions on the system and can easily crash it.
- Performance data collection on hot swapping techniques is much needed in order to be able to compare among systems and techniques.
- Extend the formal work, done by Gupta [11], for studying hot swapping to other languages than procedural, such as object oriented.

Our approach is the result of drawing from past efforts and overcoming some of their limitations. Mainly, the FST has similitudes to past techniques such as requiring the component to be in a quiet state to perform the state transfer, state transformer functions, timing of state transfer is left for the designer to decide, save/get state functions, or initialisation routines. Among the differences under our approach OS-level data structures are not captured, pointer or open handles. However we propose a semantic capturing of open handles (§3.4.1). Moreover our FST can be applied in particular, when OS-resident data structures need not be captured. The FST enhances other techniques by providing a portable and standardised format (XML) for storing onto persistent storage the state. The FST provides provisions for tracing the state during the life time of the module in case needed. In terms of implementation our approach is one of the least intrusive techniques reviewed. The FST does not require to extend or modify existing data structures and primitives.

It can be considered as an alternative to our XML schema design to partition the schema from general to the particular application. That is, to have a general hot swapping schema and then have particular schemata for different application domains.

Appendix A

The Java Virtual Machine

Figure A.1 illustrates the java virtual machine [54]. Relevant to this figure is the class loader and the execution engine. The class loader loads class files from both the program and the Java API. The infrastructure of the class loader contributes to many features of Java such as dynamic linking and dynamic extension, code mobility, and security. The bytecodes that the class loader loaded are executed in an execution engine. The execution engine can vary in different implementations so that it communicate with the operating system through native method invocations.

Figure A.1: The Java Virtual Machine.

Appendix B

A Bit of History on Apache

Most of the documentation we found about Apache web server refers to customizing, installing, configuring or optimizing the server; in general it is about usage and administration. There is very little documentation about the implementation of Apache. A good source of information is “Writing Apache Modules with Perl and C” [48]. The book explains the Apache API and its design. An excellent source of information is “The Apache Modelling Project” [21]. A natural source of information is the source code distribution of Apache itself.

The underlying ideas of the World Wide Web evolved from the work done at the European Laboratory for Particle Physics (CERN) in Geneva, Switzerland. The Apache web server was developed as an evolution of the National Center of Supercomputing Application (NCSA) HTTP daemon (NCSA HTTPd 1.3). Development of the NCSA HTTPd server stopped at version 1.3. However at that stage a lot of people had adopted and used the server. Many webmasters started developing patches to suit their own environments, as well as fixing exiting bugs.

In 1995 a group of webmasters volunteered to consolidate all the information related to the server and placed this information in a publicly accessible domain. The *Apache Group* was formed from people that had made substantial contribution to the Apache server. The first public release of Apache was done in April 1995 with version 0.6.2. The name Apache is derived of “A PAtCHy server”, since it was the evolution of a series of patches to the NCSA HTTPd 1.3 web server. The redesign of the architecture and the introduction of new features brought releases of versions 0.7.x and 0.8.8. The modular structure and API features came with version 1.0 on December 1995.

It took only a year for Apache to become the most popular HTTP server (see table B.1). During the next years the community began to grow and Apache received many new features and was ported to different operating systems. Ever since version 1.2, Apache has been fully compliant with the HTTP/1.1 specification.

In June 1999 the Apache Software Foundation (ASF), a non-profit corporation, is founded and incorporated. The first Apache conference (ApacheCon) is held in March 2000 and Apache version 2.0 Alpha 1 is introduced to the public. At the time of the writing of this thesis, the latest version of Apache is 2.0. However, due to its popularity, version 1.3 is still being further developed and supported [16].

Developer	June 2004	Percent	July 2004	Percent
Apache	34'710,235	67.22	35'122,146	67.37
Microsoft	11'021,807	21.35	11'115,660	21.32
Sun	1'664,927	3.22	1'656,671	3.18
Zeus	763,152	1.48	754,721	1.45

Table B.1: Netcraft survey comparing the top four most popular HTTP servers [39].

Appendix C

Apache's web server restart cycle

The following is a brief description of the main operations depicted in Figure C.1. These operations correspond to the Apache's preforking architecture.

- create static pools. Creates and initialises memory areas using Apache's own APIs.
- register information about prelinked modules. Apache is made out of modules. Dynamic or static modules have to be registered. This task register modules.
- read command line and set parameters. The administrator can override some of the defaults set in the configuration file (`httpd.conf`) by passing parameters to the server on the command line.
- read per server configuration. There is a *main server* which is made of definitions outside of `<VirtualHost>` sections in `httpd.conf` and there are *virtual hosts*, defined by `<VirtualHost>` sections. The configuration information read at this point is static for the life of a server's generation. This configuration information is read twice due to restarts.
- scoreboard. Used to store internal server process information, like each child's process id and its generation.
- prefork pre-configuration. Each process is typically a child process of the process that it created. For example, there are child process of the shell, once the shell terminates all of its child process are terminated as well. After reading the configuration a detach process is done. This process consists of: a) create a clone of the current process, b)

stop the parent process, c) the clone detaches I/O from the shell, d) the clone sets its process group ID to the group ID of process number 1 (init).

- `one_process_mode`. This is used for debugging purposes. There is no preforking is done when this option is selected at command line. Only one main process is created.
- `idle_server_maintenance`. This cycle is in charge of maintaining the required number of child servers, either by spawning off more or kill them. The values to configure the number threshold are set in `httpd.conf` file.

Figure C.1: Apache's restart cycle [21].

Appendix D

XML Schema

As mentioned in § 3.4.4 we select XML Schemata over DTDs. One of the mandates of hot-swapping is to introduce flexibility in an application. Keeping truth to this mandate we want to provide flexibility even in the design of our XML Schema. Thus, our proposed Schema for State Transfer is based on the proposed design from W3C [55] and on the eXtreme eXtensibility design pattern [10]. This allows the XML class to grow by allowing unforeseen data to be added without restricting the XML vocabulary employed. The XML language specification below is meant to be standalone. That is, it is not dependent on other schemata but it is extensible to other schemata. A schema definition gathers a class of XML documents, often called instance documents, the purpose of this definition is to verify that the instance document conforms with a particular schema definition.

The State Transfer schema, state.xsd

```
<?xml version="1.0"?>
```

```
<xsd:schema targetNamespace="StateTransfer"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:state="FST">
```

```
<xsd:annotation>
```

```
<xsd:documentation xml:lang="en">
```

```
Copyright 2004 Gerardo Reynaga. All rights reserved.
```

```
</xsd:documentation>
```

```
</xsd:annotation>

<!--
The root element.
-->
<xsd:element name="state">
  <xsd:complexType>
    <!--
      An unordered list of elements. See "state:ContentType"
      for more details about what is allowed in this list.
    -->
    <xsd:group ref="state:ContentType" minOccurs="0"
      maxOccurs="unbounded" />
  </xsd:complexType>
  <xsd:attribute name="id" type="xsd:ID" use="optional" maxOccurs="1"/>
</xsd:element>

<!--
This type defines the unordered list of elements that
constitute the root element.
-->
<xsd:group name="state:ContentType">
  <xsd:choice>
    <xsd:element name="old_var" type="state:ElementType"/>
    <xsd:element name="array" type="state:ArrayType"/>
    <xsd:element name="ip" type="state:IPType" />
    <xsd:element name="comment" type="xsd:string"/>
  </choice>
</group>
```

```
<!--
Element's attributes
-->
<attributeGroup name="ElementAttributes">
  <!-- Name of the variable the element represents and its type.-->
  <attribute name="name" type="string" />
  <attribute name="type" type="string" />
</attributeGroup>

<!--
Element type definitions
-->
<complexType name="ElementType">
  <group ref="state:BasicType" minOccurs="0" maxOccurs="1" />
  <attributeGroup ref="state:ElementAttributes" />
</complexType>

<!--
Basic Type definitions
-->
<group name="BasicType">
  <choice>
<element name="string" type="xsd:string">
  <element name="integer" type="xsd:integer" />
  <element name="posint" type="xsd:positiveInteger" />
  <element name="negint" type="xsd:negativeInteger" />
  <element name="nonnegint" type="xsd:nonNegativeInteger" />
  <element name="nonposint" type="xsd:nonPositiveInteger" />
  <element name="long" type="xsd:long" />
  <element name="unsignedlong" type="xsd:unsignedLong" />
  <element name="int" type="xsd:int" />
  <element name="unsignedint" type="xsd:unsignedint" />
  <element name="short" type="xsd:short" />
  <element name="unsignedshort" type="xsd:unsignedShort" />
```

```

    <element name="byte" type="xsd:byte" />
    <element name="unsignedbyte" type="xsd:unsignedByte" />
    <element name="decimal" type="xsd:decimal" />
    <element name="float" type="xsd:float" />
    <element name="double" type="xsd:double" />
    <element name="boolean" type="xsd:boolean" />
    <element name="duration" type="xsd:duration" />
    <element name="dateTime" type="xsd:dateTime" />
    <element name="date" type="xsd:date" />
    <element name="time" type="xsd:time" />
    <element name=" gYear" type="xsd: gYear" />
    <element name="gYearMonth" type="xsd:gYearMonth" />
    <element name="gMonth" type="xsd:gMonth" />
    <element name="gMonthDay" type="xsd:gMonthDay" />
    <element name="gDay" type="xsd:gDay" />
    <element name="anyURI" type="xsd:anyURI" />
  </choice>
</group>

<!--
Array type definition
-->
<!--
This type describes the primitive array type; for example,
int[] char[] [] Primitive types in an array are simply
represented as text separated by a space (0x0020). If the
character 0x0020 is present as part of an array of characters,
it is escaped as \u0020.
Examples:
<array id="174" length="2" type="int[]">3 4 </array>
<array id="200" length="6" type="char[]">a b \ \u0020 c d </array>
-->
<complexType name="ArrayType" mixed="true">
  <sequence>

```

```

    <group ref="state:ArrayComponentType" minOccurs="0" maxOccurs="1"/>
  </sequence>

  <!-- Length of the array. -->
  <attribute name="length" type="int" />
  <attribute name="type" type="string" />
</complexType>

<!--
If the component type of an array object is an object type, each
component is represented as a "string", "object" or "array" element.
-->
<group name="ArrayComponentType">
  <choice>
    <element name="string" type="xsd:string" minOccurs="0"
      maxOccurs="unbounded" />
    <element name="array" type="state:ArrayType" minOccurs="0"
      maxOccurs="unbounded" />
  </choice>
</group>

<!--
IP type definition
-->
<simpleType name="IPType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="([01]?[d\d]?|2[0-4]\d|25[0-5])\.
      [01]?[d\d]?|2[0-4]\d|25[0-5])\.
      [01]?[d\d]?|2[0-4]\d|25[0-5])\.
      [01]?[d\d]?|2[0-4]\d|25[0-5])\" />
  </xsd:restriction>
</simpleType>

</schema>

```

As schemata become larger, it is often wanted to divide their content into multiple files in order to provide easiness of maintenance, access control, and readability. In order to provide room for growth in the schema presented above we depict an example on how to extend the base schema.

Policies schema, policy.xsd

```
<?xml version="1.0"?>
<schema targetNamespace="StateTransfer"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:state="FST">

  <annotation>
    <documentation xml:lang="en">
      Policies for state transfer
      Copyright 2000 Gerardo Reynaga. All rights reserved.
    </documentation>
  </annotation>

  <!--
  Time to live for a state
  -->
  <xsd:simpleType name="StateTimeout">
    <xsd:restriction base="xsd:unsignedLong">
  </xsd:simpleType>

</schema>
```

The following example extends state.xsd schema by adding a policy to the state transfer by using policy.xsd document. In state.xsd document the following element is included:

```
<!-- include policies constructs -->
<include schemaLocation="policy.xsd"/>
```

The line should be included before the root element. It is important to note that the target namespace of the included components must be the same as the target space of the including schema, that is **StateTransfer**.

An alternative design to our XML schema design is to partition the schema from general to particular applications. That is, to have a general hot swapping schema and then have particular schemata for different application domains.

Appendix E

Testing Environment

The following is the environment used for the deployment and testing of our test case. The benchmarking process consists of a mechanism to generate a controlled stream of web requests.

Web server The server consists of Apache 2.0.48 running on a generic computer with one Intel 650Mhz P3 processor, and 128Mb of RAM. The server operating system was Red Hat Linux release 8.0 kernel version 2.4.18-14.

Network The tests were conducted at the School of Computer Science labs. The network utilized is a 100 Mbps Ethernet. All the machines used in the test run in the same subnet.

Client Machine The client machine used to generate the load for the benchmarking runs on a 2Ghz P4 Intel processor, with 1Gb of RAM. The operating system is Red Hat Linux release 8.0 kernel version 2.4.18-27.8.0.

Software *httperf* is an application to measure web server's performance. The version used was 0.8. It generates a variety of HTTP workloads. Among its features one finds the ability to generate and sustain server overload and its support for the HTTP/1.1 protocol. *Autobench* is a Perl front-end to *httperf* which automates the process of testing, and provides tools to generate a file with results. The version used was 2.1.2

The parameters: demanded request rate, high rate, number of connections, number of requests per connection, and the timeout are taken as suggested in [37]. The web server configuration, corresponding to the child spawning, is as follows E.1:


```
# prefork MPM
# StartServers: number of server processes to start
# MinSpareServers: minimum number of server processes
#                   which are kept spare
# MaxSpareServers: maximum number of server processes
#                   which are kept spare
# MaxClients: maximum number of server processes
#               allowed to start
# MaxRequestsPerChild: maximum number of requests
#                       a server process serves
#                       if 0, then the process will never expire
<IfModule prefork.c>
StartServers      100
MinSpareServers   100
MaxSpareServers   150
MaxClients        150
MaxRequestsPerChild 0
</IfModule>
```

Figure E.1: Child server configuration parameters.

Bibliography

- [1] Pdp11, 2000. URL: <http://www.village.org/pdp11/faq.html> [Accessed: July 2003].
- [2] J. Andersson, M. Comstedt, and T. Ritzau. Run-time support for dynamic java architectures. In *ECOOP'98 Workshop on Object-Oriented Software Architectures (WOOSA '98)*, July 1998.
- [3] Gang Ao. Software hot-swapping techniques for upgrading mission critical applications on the fly. Master's thesis, Department of Systems and Computer Engineering, Carleton University, February 2000.
- [4] J. Appavoo and et al. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60 –76, 2003.
- [5] Autobench. A perl front end to httpperf web server benchmarking tool, 2004. URL: <http://www.veritest.com/benchmarks/webbench/default.asp> [Accessed: June 2004].
- [6] M.R. Barbacci and R.W. Lichota. Durra: an integrated approach to software specification, modeling and rapid prototyping. pages 67 –81, 11-13 June 1991 1991. Rapid System Prototyping, 1991. 'Shortening the Path from Specification to Prototype'. Second International Workshop on.
- [7] T. Bloom. DYNAMIC MODULE REPLACEMENT IN A DISTRIBUTED PROGRAMMING SYSTEM. Technical Report MIT/LCS/TR-303, 1983.
- [8] T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102 –108, March 1993.
- [9] Paolo Casarini. Gnome dom engine, libgdome, 2004. URL: <http://gdome2.cs.unibo.it/index.html> [Accessed: June 2004].

- [10] Roger L. Costello. eXtreme eXtensibility (XML Schema Design for the Semantic Web), January 2003. URL: <http://www.xfront.com/eXtreme-eXtensibility.html> [Accessed: June 18 2004].
- [11] Gupta D., Jalote P., and Barua G. A formal framework for on-line software version change. *Software Engineering, IEEE Transactions on*, 22(2):120 –131, Feb 1996.
- [12] Gupta Deepak. *On-line Software Version Change*. PhD thesis, Department of Computer Science & Engineering. Indian Institute of Technology, Kanpur, November 1994.
- [13] Gupta Deepak and Jalote Pankaj. On-line software version change using state transfer between processes. *Software - Practice and Experience*, 23(9):949–964, 1993.
- [14] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.
- [15] Ning Feng. S-module design for software hot-swapping. Master's thesis, Department of Systems and Computer Engineering, Carleton University, November 1999.
- [16] The Apache Software Foundation. Apache HTTP Server, 2004. URL: <http://www.apache.org> [Accessed: June 2004].
- [17] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, Vol 14 No 2, February, pages 111–128, 1991.
- [18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [19] Alex Garthwaite and Scott Nettles. Transactions for java. In *First International Workshop on Persistence and Java, Drymen, Scotland, Sep 16-18, 1996*, September 1996.
- [20] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. pages Page(s): 298 –307, 1978.
- [21] Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, and Oliver Schmidt. The apache modelling project, 2004. URL: <http://apache.hpi.uni-potsdam.de/> [Accessed: Jul 2004].

- [22] Object Management Group. UML 2.0 Superstructure Specification, August 2003. URL: http://http://www.omg.org/technology/documents/modeling_spec_catalog.htm [Accessed: June 2004].
- [23] D. Gupta and P. Jalote. Increasing system availability through on-line software version change. In *Proc. 23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23)*, pages 30–35, Toulouse, France, 1993. IEEE Computer Society Press.
- [24] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic. Object serialization analysis and comparison in java and .net. *SIGPLAN Not.*, 38(8):44–54, 2003.
- [25] Michael Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.
- [26] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.
- [27] Gíslí Hjálmtýsson and Robert Gray. Dynamic C++ classes - A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conf.*, pages 65–76, jun 1998.
- [28] Christine R. Hofmeister. Dynamic reconfiguration of distributed applications. Technical Report CS-TR-3210, 1994.
- [29] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *Software Engineering, IEEE Transactions on*, Vol:11 No:4, Apr 1985, pages 424 –435, 1985.
- [30] J. Kramer and J. Magee. A model for change management. In *Distributed Computing Systems in the 1990s, 1988. Proceedings., Workshop on the Future Trends of*, pages 286–295, Hong Kong, 1988.
- [31] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
- [32] Ramon Lawrence. A survey of process migration mechanisms, 1998.
- [33] Insup Lee. *DYMOS: A DYNAMIC MODIFICATION SYSTEM*. PhD thesis, Computer Sciences, UNIVERSITY OF WISCONSIN - MADISON, 1983.

- [34] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, March 1988. ISSN: 0001-0782.
- [35] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *Proceedings of the European Conference on Object-Oriented Programming*, June 2000.
- [36] Sun Microsystems. Jini architecture specification, 2003. URL: <http://www.sun.com/software/jini/specs/> [Accessed: June 2004].
- [37] David Mosberger and Tai Jin. `httperf` –a tool for measuring web server performance, 2003. URL: http://www.hpl.hp.com/personal/David_Mosberger/httperf.html [Accessed: June 2004].
- [38] Barbacci M.R., Doubleday D.L., and Weinstock C.B. Application-level programming. In *Distributed Computing Systems, 10th International Conference on*, pages 458–465, 28 May-1 June 1990.
- [39] Netcraft. Web server survey. July 2004 Web Server Survey Finds 50 Million Sites, 2004. URL: <http://news.netcraft.com> [Accessed: July 2004].
- [40] Andy Longshaw Paul Dyson. *Architecting Enterprise Solutions. Patterns for High-Capability Internet-Based Systems*. Wiley, 2004.
- [41] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.
- [42] J.M. Purtilo and C.R. Hofmeister. Dynamic reconfiguration of distributed programs. *Distributed Computing Systems, 1991., 11th International Conference on*, pages 560–571, 20-24 May 1991.
- [43] Extensible Markup Language (XML) W3C Recommendation, 2004. URL: <http://www.w3.org/TR/2004/REC-xml-20040204/> [Accessed: May 2004].
- [44] Francisco Assis Rosa and António Rito Silva. Component configurer: A design pattern for component-based configuration. In *Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97)*. Siemens Technical Report 120/SW1/FB. Munich, Germany: Siemens, 1997. URL: <http://citeseer.ist.psu.edu/rosa97component.html> [Accessed: September 2004].

- [45] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley and Sons, New York, NY, 2000.
- [46] M.E. Segal and O. Frieder. On-the-fly program modification: systems for dynamic updating. *Software, IEEE , Volume: 10 Issue: 2 , March 1993*, pages 53 –65, 1993.
- [47] Margo I. Seltzer and Christopher Small. Self-monitoring and self-adapting operating systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 124–129, 1997.
- [48] Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, Inc., 1999.
- [49] L. Tan and B. Esfandiari. The swapbox: A test container and a framework for hot-swappable javabeans. MICON. Mitel Networks Corporation, 2001.
- [50] Lei Tan. Swapbox: a hot-swapping framework for swappable java beans. Master's thesis, Department of Systems and Computer Engineering, Carleton University, September 2001.
- [51] Peter Tröger and Andreas Polzee. Object and process migration in .net. In *Workshop on Object-oriented Dependable Real-time Systems (WORDS 2003)*, Guadalajara, México, Januray 2003. IEEE Computer Society Press.
- [52] Daniel Veillard. The xml c parser and toolkit of gnome, 2004. URL: <http://www.xmlsoft.org/> [Accessed: June 18 2004].
- [53] Alistair C. Veitch and Norman C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. In *In Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (ICCDs)*, pages 236 –242, May 1996.
- [54] Bill Venners. *Inside the Java Virtual Machine, Second Edition*. McGraw-Hill, 1999.
- [55] World Wide Web Consortium (W3C). XML Schema Part 0: Primer - W3C Recommendation, 2 May 2001, 2001. URL: <http://www.w3.org/TR/xmlschema-0/> [Accessed: May 2004].