Dynamic Adaptation for Distributed Systems

Jingtao Sun

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences
SOKENDAI (The Graduate University for Advanced Studies)

Dynamic Adaptation for Distributed Systems

国立大学法人
総合研究大学院大学
SOKENDAI（THE GRADUATE UNIVERSITY FOR ADVANCED STUDIES）

Jingtao Sun

A dissertation submitted to the Department of Informatics
School of Multidisciplinary Sciences
(SOKENDAI)The Graduate University for Advanced Studies
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

September 2016

Acknowledgements

First, I would like to express my deep-felt gratitude to my supervisor, Professor Ichiro Satoh of the Information Systems Architecture Research Division at the National Institute of Informatics (also the Department of Informatics at the Graduate University for Advanced Studies) for his constant support and mentoring. His serious and responsible work style in academia had a very deep impact on me. He guided me not only in my work and research but also in the course of my life. I have also been extremely fortunate to have been able to work so closely with him.

I also wish to thank my advisers professor Shin Nakajima, professor Kento Aida, associate professor Asanobu Kitamoto of the National Institute of Informatics, and professor Tatsuo Nakajima of Waseda University, who provided many insightful comments and suggestions on my work and dissertation. It has been a pleasure to work with them.

Further, I am lucky to have worked closely with Dr.Sisi Duan of the computational data analysis group at Oak Ridge National Laboratory. She gave me a lot of astute help in our collaborative researches. I greatly appreciate her help, especially when we discussed our researches during what was my day and her night.

Above all, I want to thank my dear friend, Mingqing Wu. Especially when I lost myself in despair, thank you for your concern about me, and thank you for the dinner you cooked for me. Thanks also go to my parents, Xiangfeng Sun and Yanqiu Li, and my collaborators Kai Xu of Wayne State University, USA, Yuan Yuan of the University of Alberta, Canada, secretary Hiroko Maruyama of National Institute of Informatics, professor Kazuya Tago, assistant professor Chihiro Shibata, and messrs. Zhan Jin and Ruhui Ye of the Tokyo University of Technology. Without their supports, I could not imagine how I could have written my dissertation throughout the whole process.

Abstract

Applications are typically executed on fixed distributed system architectures, and they merely interconnect a huge number of software components through networks. However, the complexity and dynamism of distributed systems are beyond our ability to build and manage such systems through conventional approaches, such as centralized and top-down architectures. On the other hand, the requirements of applications or the structure of systems change from time to time. For instance, software components that applications consist of may be dynamically added to or removed from distributed systems, and networks between computers may be rapidly disconnected and reconnected. Therefore, modern distributed systems demand availability, dependability, and reliability to adapt themselves to various changes, even dynamically, to self-adapt their architectures.

This dissertation first presents the requirements and proposed approaches. The key idea behind the proposed approaches are to introduce the relocation of software components, which define functions, between computers as a basic mechanism for adaptation. Second, it introduces the design of *Mimosa*, which is an adaptive and reliable middleware that adapts to various changes through relocation of software components in distributed systems. Third, a policy-based language is described for specifying, and analyzing user-defined adaptations. Since the language is defined on a theoretical function, it enables the results of conflicts and divergences from adaptions to be to analyzed sequentially. Although the proposed approaches are based on adaptive deployments of software components but not on those of adaptive functions inside any software components. Therefore, my proposal can more effectively adapt to types of change between distributed systems and applications, even change the fixed system architectures. This is because user-definitions of policies were separated from software components, and then the software components could be invoked from different computers.

This dissertation describes the design and implementation of the approaches with five distributed applications, and experiments were run on my system with the proposed applications over a distributed system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems are currently undergoing a revolution. Two advanced technologies began to change the situation. The first involved the invention of powerful microprocessors and memory chips and the second involved the invention of high-speed computer networks. These technologies were not only feasible, but also made it easy to connect a large numbers of computers into systems through high-speed networks, which are called distributed systems.

Distributed systems have been widely used in various fields of computing, such as artificial intelligence, cloud computing, ubiquitous computing, mobile computing, disaggregated computing, big data, machine learning, the Internet of Things (IoTs), and machine to machine (M2M) [10] [3] [57] [28] [75] [13] [58] [94] [25]. Distributed systems are complicated and dynamic by nature because their structures, and the applications running on them tend to change dynamically. For instance, applications running on computers or software components may be added to or removed from distributed systems, and networks between computers may be frequently disconnected and reconnected. Existing distributed systems should have the capabilities to adapt themselves to such changes, and even dynamically change their architectures to support their business targets and behaviors.

How to build adaptive distributed systems has become one of the foremost challenges for researchers. This chapter first introduces the motivation for my research. It then describes the requirements and challenges to dynamically achieve adaptive distributed systems. Finally, the main contributions of this dissertation are presented.

1.1 Motivation

Distributed systems need to support availability, adaptability, and dependability, because they are often used for mission-critical purposes. However, the configurations of particular systems have currently been built and operated with specific architectures. Specific architectures have lost their effectiveness with the expanded scale and use of distributed systems. Distributed applications, on the other hand, have been implemented for multiple purposes and multiple users with various requirements that have changed dynamically. Therefore, software running on distributed systems should be resilient so that the systems can adapt software to various changes at runtime, and even change the architecture of existing distributed systems to support their business targets and behaviors. Software running on distributed systems should also be reuseable so that it can adapt to various changes in different distributed systems.

Adaptation is an effective way of resolving dynamic changes between distributed systems and applications. Many researchers have focused on dynamic adaptation for distributed systems through a variety of approaches [89] [8] [17] [21] [90] [29] [20]. However, adaptation is not only needed to avoid harmful effects, but also to reap the benefits of potentially favorable effects. Therefore, relocation of software components is focused on in this dissertation between computers for adaptation on distributed systems. Although it seems very simple, it can provide very strong adaptability in both general-purpose applications and existing distributed systems. Conversely, although the conditions of adaptation written inside software programs is a solution, such programs are difficult to maintain, and it is not easy for junior developers to develop efficient adaptation programs. Therefore, it is believed that adaptation programs should be separated from business logic. Adaptation from business logic is distinguished in this dissertation by using the principle of separation of concerns, so that developers can concentrate on business logic rather than adaptation programs as much as possible.

Based on these motivations, this dissertation addresses a policy-based middleware to adapt to various changes in existing distributed systems. It was assumed that distributed applications would consist of one or more software components, which might have been running on different computers through networks. The proposed approach incorporates two key ideas:

- The proposed adaptation does not cause unnecessary failures, and supports a wide variety of changes on existing distributed systems. It focuses on the relocation of software components between computers as a basic adaptation mechanism (The details are presented in Chapter 4).

- A policy-based language needs to be designed for developers to define nature-inspired relocation policies for describing application-specific adaptations to distinguish adaptation concerns from software components on distributed systems (The details are presented in Chapter 5).

Based on these two ideas, when changes occur and their defined conditions of policies are satisfied, software components are automatically *relocated*, *duplicated then relocated*, between computers or *removed from computers* to adapt themselves to changes on distributed systems.

1.2 Requirement

Distributed systems are used for multiple purposes and need capabilities to adapt themselves to various changes in results from their dynamic properties [93]. Therefore, seven requirements were first defined for distributed systems, and my approach should meet all of these requirements.

- *Self-adaptability*: Software components may be running on different computers. Therefore, software components should coordinate themselves to support their applications with partial knowledge about other computers.

- *On-demand deployment of software*: Computers may have limited resources so that they cannot support software for various applications beforehand. Software for the applications needs to be dynamically deployed at appropriate computers to coordinate multiple computers for individual applications.

- *Separation of concerns*: All software components should be defined independently of the proposed adaptation mechanism as much as possible. As a result, developers should be able to concentrate on their own application-specific processing.

- *Availability*: System failures are inevitable since numerous software components run on distributed systems. The proposed adaptation should not only avoid causing more system failures, but also needs to adapt to them as well as provide a non-stop distributed system when these failures occur. Therefore, an effective solution is to relocate software components between computers. This is because it just changes the location of software components instead of changing their functions. Furthermore, it also enable applications to remain available after their adaptations in distributed systems.

- *Reusability*: There have been many attempts to provide adaptive distributed systems. However, as the approaches and parameters in most of them have strictly and statically depended on their target systems, they have needed to be re-defined overall for reuse in other distributed systems. The proposed adaptation should be abstracted away from the underlying systems for reasons of reusability.

- *Non-centralized management*: There is no central entity to control and coordinate computers. The proposed adaptation should be managed without any centralized management for reasons of avoiding any single points of failures and performance bottlenecks to attain availability and scalability.

- *General-purpose and adaptation independence*: There are various applications running on distributed systems. Therefore, the approach should be implemented as a practical middleware to support general-purpose applications. All software components should be defined independently of the proposed adaptation mechanism as much as possible. As a result, developers should be able to concentrate on their own application-specific processing.

Computers on distributed systems may have limited resources, for instance, processing, and storage resources. On the other hand, the bandwidths of networks on several distributed systems tend to be narrow and their latency cannot be neglected. Therefore, the proposed approach should be available with such limited resources and such networks, whereas many existing adaptation approaches explicitly or implicitly assume that their target distributed systems have enriched resources.

1.3 Challenge

There are a few challenges on be confronted in designing adaptive distributed systems. Depending on the structure of distributed systems and the requirements of applications, software components should be flexible and adaptable. These challenges can be tackled in four steps:

- A middleware system needs to be designed and implemented to support adaptation in both systems and applications so that it does not rely on different execution environments. This middleware system needs the capability to manage and relocate software components between computers and determine which software component should be relocated to where. The methods of relocating software components should be dynamically invocated at destination computers through the middleware system.

- A policy-based language needs to be designed and implemented for users so that they can easily define their requirements as polices, such as the requirements of applications or the structures of system changes or network states. These policies need to include a pair of *condition* and *action* parts. If the condition part can be satisfied, software components should be relocated to the destination computer and then their operations need to be restarted.

- A policy interpreter needs to be designed and implemented in our middleware system for executing user-defined policies. Because the interpreter does not need to generate intermediate object code and memory requirements are much less, it is very consistent with the designed middleware system.

- Several applications also need to be developed to verify the performance and effectivity of the proposed middleware systems and policy language, and even import the necessary mechanism to analyze the conflicts and divergences between user-defined policies.

Many researchers have proposed various approaches for adaptation on distributed systems [89] [44] [43] [23] [68] [14] [21]. They have focused on changing parameters, coordination of functions or modify software for adaptation, instead of focusing on the relocation of software components between computers for adaptation. Their approaches have been effective, but they may create unnecessary failures in existing distributed systems, thereby causing distributed systems to stop and reduce availability and reusability.

1.4 Contribution

This dissertation describes how I hoped to tackle these challenges to provide a general-proposed approach to adapt components themselves to changes by relocating software components for distributed systems and applications. Several applications were used to overcome the challenges and test the proposed approach to validate its feasibility. The six key contributions of the research described in this dissertation are below:

- A middleware called *Mimosa* was designed and implemented (explained in Chapter 4), which allows software components to self-adaptively relocate or duplicate/relocate or remove themselves from destination computers when the structure of distributed systems or the requirements of applications are changed.
- The Mimosa system not only relocates the software components in distributed computers, but it can also relocate their states of software components to destination computers. It can make the relocated software components continue their operations at the destination computer by dynamically invoking their life-cycle methods.
- A dynamic method invocation (DMI) mechanism was also developed, in which the common object request broker architecture (CORBA) [79] was studied, so that our mechanism could easily hide differences between the interface of objects at the original computer and others.
- A policy-based language was designed for users who need to define their requirements as policies throuth process calculus [61]. It is basically defined as a pair of information items on where and when the software components are deployed.
- Each application-specific component can have one or more policies, therefore user-defined policies may cause various conflict or divergence, so that the adaptation mechanism need to be able to analyze them through the properties of software component descriptions with the proposed policy-based language, and provide the necessary modified proposal to policy-makers.
- Distributed applications are formed by different software components. Therefore, through the relocation of software components between computers, the architectures of existing distributed systems can be dynamically changed. For instance, the system architectures can also be dynamically changed in both the directions of the Client/Server and Peer-to-Peer.

Application developers no longer need to write complex and adaptive code inside their programs with the proposed developer-friendly adaptation middleware and policy-based language, then developers just need to focus on their own work. In addition, general-purposed distributed applications can run on the middleware system and they do not need to do anything for adaptation. When changes occur, the middleware system can determine where software components are relocated so that they can dynamically adapt themselves to various changes, according to user-defined policies.

1.5 Organization

This dissertation is organized into eight chapters, which include the introduction, rrevious studies are described in Chapter 2, which are separated into three types, and two applicated approaches are then compared with my proposed approach. The six remaining chapters can be divided into three parts.

The first part (Chapter 3) introduces the underlying concept behind the research, and why I chose to relocate software components for adaptation on distributed systems. Some scenarios are presented in this chapter. I will describe the validity of the proposed approach in contrast with existing studies, and why it can solve problems that existing research cannot.

The second part (Chapters 4 and 5) introduces a reliable middleware, called *Mimosa* (Chapter 4), including its specific structure, and how it can adapt to various changes in distributed systems through relocating software components between computers. As the *Mimosa* middleware system does not depend on the execution environment, it can run on all operating systems. Next, a policy-based language for users is presented to define functions as policies for relocating software components. As it was designed based on theoretical foundations, it enables the effects of adaptations to be analyzed. Both the software and adaptation can be reused in different distributed systems because adaptation is separate from application-specific software concerns.

Five generally proposed distributed applications are introduced (Chapter 6) based on the approaches to demonstrate the relocation of software components for adaptation on a distributed system, according to user-defined policies. Implementations, evaluations, and conclusions are covered in the third and last part (Chapters 7 and 8). Chapter 7 explains the implementations and evaluations of the *Mimosa* middleware system and our policy-based language with the five distributed applications. A summary of this dissertation and an outline of future work are given in Chapter 8.

Chapter 2

Related Work

This chapter provides an introduction to research areas with which this dissertation is concerned. The following sections describe the backgrounds and explain the most influential researches in the area of adaptation on distributed systems.

Previous studies on adaptation areas can be classified into three types. The first has focused on parameter-level adaptation for distributed systems. It modifies program variables that determine behavior. The second has focused on coordination-level adaptation for distributed systems. It provides adaptation through dynamic changes in the coordination of different software components within a computer. The third has involved software-level approaches to adaptation, e.g., the genetic algorithm (GA), genetic programming, and swarm intelligence. It provides adaptation through a dynamic redefinition of software.

Based on the three technologies, there are two applicated approaches for adaptation. Architecture approaches enables existing system structures from one type to change to another type to provide adaptation. Location approaches have used *Mobile agents* to adapt agents to changes in their system environment, and it can change locations by migrating agents to dynamically adapt to their behaviors on distributed systems.

The next section surveys existing studies on how to provide adaptation on distributed systems.

2.1 Parameter-level Adaptation

The first type is parameter-level adaptation for distributed systems. An often-cited example, the Internet's Transmission Control Protocol (TCP) [69] allows its behavior by changing values which could control window management. It also adjusts retransmissions in response to apparent network congestion.

In dynamic environments, the value of parameters may change over time. Epifani et al. presented a framework, called KAMI [24]. They provided a solution by changing the parameters on a runtime system. The updated model that provided for software engineers have advantages and disadvantage. Before the software engineers build their systems, they should give full consideration to all requirements. Once their original systems are constructed, parameter adaptation does not allow new algorithms and software components to be added to applications. It can tune parameters or direct an application to use a different existing strategy, however it cannot adopt new strategies.

Herrera and Lozano proposed an approach to adaptive parameters, in which their approach was based on the use of fuzzy logic controllers [33] through which the parameters dynamically changed to provide adaptation for distributed systems. Programs could be rewritten in their research; however, they could not adapt themselves to changes inside systems, and the software components could not be reused.

Chang et al., [14] present an adaptation approach for distributed applications to adapt to changing resource characteristics. Their approach modify program variables that determine behavior. However, It does not allow new algorithms and components to be added to an application after the original design and construction.

Blair et al. explored the role of the *Aster* framework in supporting dynamic adaptation within the context of the Open-ORB middleware platform [8]. Following a detailed examination of adaptation, I concluded that *Aster* [37] could usefully be extended to meet my requirements. The key extensions to Aster included the incorporation of weakest properties and environmental parameters in architectural descriptions to accommodate re-configurations due to changing non-functional parameters in the former and environmental conditions in the latter. However, when changes occurred in their research project, the defined configurations or re-configurations were difficult to reuse.

A model-driven middleware was presented in [45]. They focused on component-based parameter adaptation to adapt changes for applications at a runtime system. They designed their user interface for implementing components with a number of different variants, e.g., OneWayUI, TwoWayUI, PlayBackUI. However, their approach has an inherent weakness. Changes in the environment can not only from the applications, but also may come from their system itself or the networks. Therefore, adaptive behaviors are limited in parameter-level adaptation.

Ting Liu and Margaret Martonosi presented a non-VM-based middleware system for managing distribuetd sensor systems, called Impala [53]. It allows software updates to be received through the transceiver of nodes, and to be applied to the running system dynamically. However, they defined a set of application parameters and system parameters to represent at runtime. However, adaptive behaviors are limited in their researches, it does not allow new algorithms to be added to applications, and does not allow software components to be relocated between computers.

2.2 Coordination-level Adaptation

The second type is coordination-level adaptation for distributed systems. These approaches can be divided into two categories.

The first changes the coordination of programs for adaptation, such as [89] [44] [43] [23].

Uribarren et al. presented a context-based coordination-level middleware platform, which was a configurable, adaptable, heterogeneous, and interoperable middleware that was abbreviated to CAHIM. CAHIM provides interoperable mechanisms of communication between applications and devices [89]. Like mine, their middleware platform provided good portability, and was independent of the underlying hardware, operating system, and of the application itself. CAHIM can be simply summed up as the collection, distribution, transformation and inference of generic information. For instance, one of the main scopes of CAHIM is to communicate context information, and it focuses on how to make software components run on different devices, and then to become aware of their network and related resources. However, CAHIM can adapt and provide pervasive services to distributed applications. Although CAHIM based on contexts and generic information can change the coordination of software components, CAHIM middleware does not support the relocation of software components for adaptation and it needs more computing resources for generic information than my approach. In addition, CAHIM cannot change fixed system architectures to provide various adaptations to distributed systems.

Keeney and Cahill presented a context-aware policy-based dynamic adaptation framework, called Chisel [44] [43]. Chisel is based on decomposing the particular aspects of service objects and using meta-types, but it does not provide core functionality to the multiple behaviors of components. When the context of users or applications changes, service objects are driven by a human-readable adaptation policy to adapt themselves to different behaviors as the execution environment. If an application needs to adapt in Chisel [43], it is usually the non-functional requirements or behaviors of some objects contained in this application that need to be changed, rather than the domain of the application that needs to be changed. Therefore, applications running on Chisel can be adapted by changing these behaviors without changing the applications themselves. Like the one proposed here, Chisel separates policy and service objects. However, unlike the one proposed here, they did not rise the approach to the level of language, they used the concept of meta-types and reflection to implement the adaptation mechanism, and the one proposed here uses policies to define the relocation of software components for adaptation. Policies in Chisel make it difficult to describe complex changes to users, and it is difficult to solve conflicts in user-defined policies. Unlike the system proposed here, their system is built on a fixed architecture, and therefore, unlike that proposed here, their approach can not dynamically change the system architecture to provide adaptability to applications to adapt themselves to changes on distributed systems.

Rouvoy et al. presented a component-based middleware, i.e., for mobile users in ubiquitous computing environments that was abbreviated as *MUSIC*, for adapting to changes in ubiquitous and service-oriented environments [71] [72]. The MUSIC project focused on various changes from service providers. Services functionalities could be dynamically changed to adapt to changes through coordination-level adaptation of software components. Their research was like mine in that they separated adaptation concerns from business logic concerns, and designed their middleware for complex distributed environments and various generic applications. They offered adaptivity through given changes in execution contexts. Software components that defined service functionalities could be dynamically configured with conforming components by coordinating adaptation processes. Their approach essentially had limitations in adaptation. For instance, their approach could not dynamically increase or decrease or duplicate functions of components for adaptation. In addition, although their proposed approach could change the coordination of components, it required system resources to optimize the context for adaptation and it could not dynamically correspond to changes from distributed systems themselves.

Mirkoet Morandini et al., presents a goal-oriented approach to specify variability in system requirements which is called Tropos4AS (Tropos for self-adaptive systems) [63]. Tropos4AS tries to capture already at design time the information needed for autonomous decision making in self-adaptive systems, allowing designers to model features such as the ability to select among different alternatives depending on the environmental context, user's preferences, and system failures to be prevented. However, this research focus on the coordination-level, but they don't separate the adaptation concern from the business logic concern like mine.

Brian et al., presented a simple high-level directives and a sophisticated runtime algorithm for coordinating adaptation approach [23]. Application developers can describe when an adaptation must happen, as a functions of the relative computational progress of the affected processes and then scheduled automatically by runtime system. However, they just can change the cooperation of processes, therefore the adaptive behaviors is limited, unlike mine. For instance, this approach can not dynamically add or delete processes on various computers.

The second uses *Policy-based language* [20] [97] [42] [16] [17] to separate adaptive behaviors from bussiness logic for adaptation, and it is an ideal mechanism to drive general-purpose dynamic adaptation framework/middleware [59], since the adaptation mechanism can be completely decoupled from adaptation management. Several research groups have theoretically designed and implemented Policy-based languages for adaptation on distributed systems.

Lymberopoulos et al. [56] proposed an object-oriented declarative framework for adaptations based on their policy specification language that was called *Ponder*[20]. However, the Ponder language focused on specifying management and security policies.

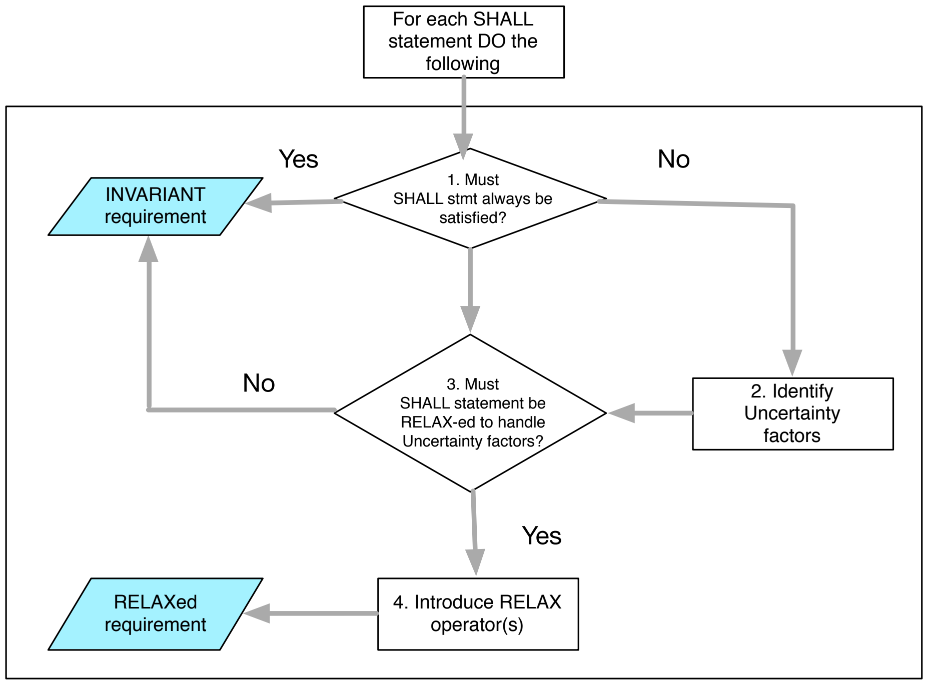
```
inst oblig policyName "{"
    subject [<type>] domain-Scope-Expression ;
     [ target [<type>] domain-Scope-Expression ;]
    on event-specification ;
    do obligation-action-list ;
    [ catch exception-specification ; ]
    [ when constraint-Expression ; ] "}"
```

Figure 2.1: Obligation policies

Unlike my research, they used an *obligation policies* format to specify the actions of components in coordination-level adaptation. When changes occurred, the Ponder language manager was run to provide capabilities for action for themselves to respond to the changing circumstances. The definition of obligation policies is given in Figure 2.1. I can see from the figure that Obligation is an event-triggered condition-action rule, which explicitly identifies the subjects that are responsible for running the management actions on target objects. Both subject and target objects are specified in terms of domain scopes, which are a method of grouping objects which the policies define, such as timer events and external events. These Internal Events are collected and distributed by Ponder's monitoring services. In addition, composite events can be specified by using event composition operators that *Ponder* language supports [20]. When the defined policies are sent to Differentiated Services (DiffServ) elements, the policy actions dynamically change the parameters and reconfigure the policy objects, and then the behaviors of objects are modified.

Remark The user-defined policies that were defined by Ponder language just change the coordination of the components; they cannot dynamically add/remove functions to/from their systems. Moreover, as Ponder language is dependent on the domain scope, the biggest difference is that it cannot provide functions that automatically select the destination by themselves, and it does not support dynamic changes to system architectures to adapt to various changes on distributed systems.

Whittle et al.fs research was different to that of Lymberopoulos et al. and they presented *RELAX* [97] [15], i.e., a new specification language to develop the engineering language requirements of dynamically adaptive systems (DASs), while demonstratively addressing factoring uncertainty in requirements and processes. *RELAX* uses a variation of threat modeling to determine where the requirements need to be updated for adaptation. Typically, RELAX use a modal verb, such as *SHALL* (or WILL) that defines actions or functions inside software components to provide adaptation on the prescribed behaviors of textual requirements.



For each SHALL statement DO the following
Yes
No
1. Must SHALL stmt always be satisfied?
INVARIANT requirement
No
3. Must SHALL statement be RELAX-ed to handle Uncertainty factors?
2. Identify Uncertainty factors
Yes
RELAXed requirement
4. Introduce RELAX operator(s)

Figure 2.2: RELAX process

The core of the *RELAX* language [97] [98] is the operators, which are designed to enable system administrators to identify requirements that could temporarily change under user-defined certain conditions. Figure2.2 shows how the steps of the process translate traditional requirements into RELAX-ing requirements. The set of *RELAX* operators can be organized into modal, temporal, and ordinal operators, and uncertainty factors (The details are provided in Subsection 2.1 of [97]). Each relaxation operator defines constraints on how a requirement is relaxed at system runtime by using these user-defined conditions to adapt components to changes. In addition, uncertainty factors, such as, MON (monitor), ENV (environment), REL (relationship), and DEP (dependency), ensure requirements are relaxed, which require adaptive behaviors.

Remark Although RELAX can adapt to changes through transferring text to requirements, and then these SHALL statements can be relaxed, however the decision of whether a requirement is invariant is an issue for the system stakeholders, aided by the requirements engineer. As same as Ponder language, RELAX language also cannot provide the adaptive functions to automatically select the destination for components, conflicts in RELAX-defined rules may occur, and existing systems may stop working: RELAX does not provide specific solutions. Moreover, RELAX does not support dynamic changes to system architectures to adapt to various changes on existing systems.

Kagal et al. [42] proposed a policy language called *Rei* [42]. Based on deonic concepts and grounded, Rei was designed for pervasive computing applications. It supports unanticipated dynamic adaptation, but must be used in conjunction with a separate adaptation mechanism. Refrains in Ponder, Rei policy language is mostly focused towards *security* policies. There were four basic policy types in Rei, such as rights, prohibitions, obligations and dispensations that was correspond to positive and negative authorizations, and obligations. These constructs denoted by PolicyObject are represented as

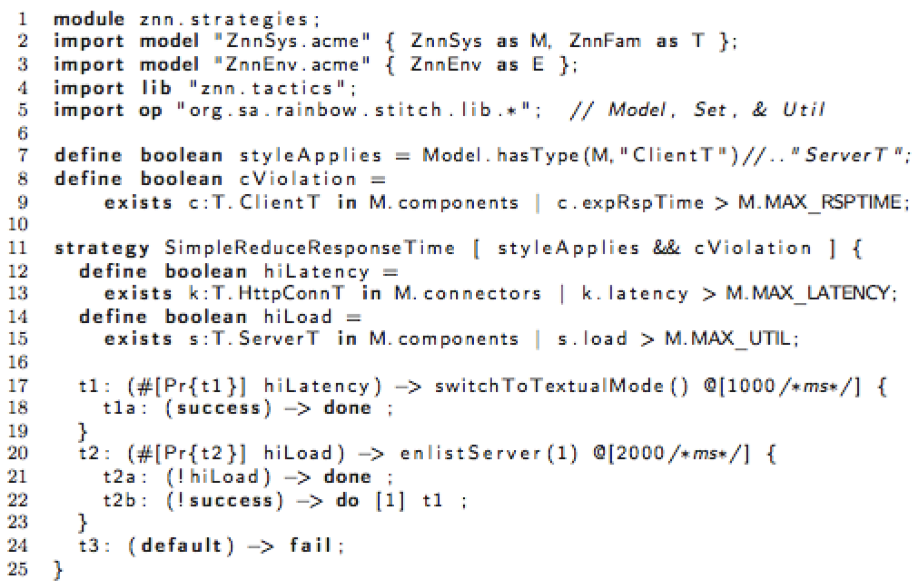
PolicyObject(Action, Conditions)

where, *Action* is a domain dependent action and *Conditions* are constraints on the actor, action and environment. A set of rules can be associated with a managed domain entity, and any time of an action was to be performed on that entity. For instance, a rule that states that all employees of 'UMBC' can perform printAction1 is represented as the follows:

has(Variable, right(printAction1, (employee(Variable, 'UMBC'))))

Remark Rei language was requested to verify that the action, and was provided a mechanism to define actions that can be used in obligation rules. In addtion, Rei provided the ability to reason about rules and respond to queries but does not provide a mechanism to enforce policy rules or perform actions. Therefore, the adaptability of Rei supported is limited than mine, and it does not support duplication of software components for adaptive changes for their systems. More over, it cannot allows dynamic changes to be adapted to various changes through system architectures changed in distributed systems.

Cheng et al. [16] proposed a language called *Stitch*. It is responsible for repairing strategies within the context of an architecture-based self-adaptation framework *Rainbow* [17]. The *Stitch* language supports definition of the adaptation strategies through a control-theoretic point of view in which systems and dynamic models. In addtion, *Stitch* also represents uncertainties in adaptation outcome and timing delays. The follwing example shows how to use the Stitch language to adapt to response time 2.3.



```
module znn.strategies;
import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
import model "ZnnEnv.acme" { ZnnEnv as E };
import lib "znn.tactics";
import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util

define boolean styleApplies = Model.hasType(M,"ClientT")//.."ServerT";
define boolean cViolation =
    exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;

strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
  define boolean hiLatency =
    exists k:T.HttpConnT in M.connectors | k.latency > M.MAX_LATENCY;
  define boolean hiLoad =
    exists s:T.ServerT in M.components | s.load > M.MAX_UTIL;

  t1: (#[Pr{t1}] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
    t1a: (success) -> done ;
  }
  t2: (#[Pr{t2}] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
    t2a: (!hiLoad) -> done ;
    t2b: (!success) -> do [1] t1 ;
  }
  t3: (default) -> fail;
}
```

Figure 2.3: Example of stitch language

Remark Unlike mine, *Stitch* used an architecture evaluator to detect when a target-system is in a state suitable for repair when the conditions of applicability are satisfied. It also cannot provide functions that automatically relocate to the destination by themselves or make duplications of functions to be relocated to where the computers need. In addtion, the policies which defined by *Stitch* have to complies and executes repairs, but my research implemented as interpreter, and my research supports relocation of software components for adaptation, unlike *Stitch*. Furthermore, my research supports conflict and divergence of user-defined policies. It is not available in *Stitch* language.

Joonseon et al.,[1] proposed a high-level policy description language for ubiquitous environment. The programming environment contains a high-level ubiquitous programming framework, a run-time system, and programming support tools for program analysis and monitoring. By using the policy description language, programmers can describe a high level specification on context space, context-based security, and context-based adaptation for ubiquitous applications. The syntax of adaptation rules of the proposed policy language is described as follows:

$$
\begin{array}{lll}
d \in \text{Adaptation-Rule} & ::= & r \Rightarrow a \text{ if } b \mid d_1 \ ; \ d_2 \\
a \in \text{Action} & ::= & p_1.id(p_2) \mid p_1!id(p_2) \mid a_1 \ ; \ a_2
\end{array}
$$

Remark Unlike mine, the proposed approach does not focus on components instead of context for adapt to changes, although it can describe the adaptive conditions and actions as policies, but it does not support the relocation or duplication of software components for adaptation and allow new software components to be dynamically added to their systems. In addtion, There is no way to provide a solution for conflict and divergence of user-defined policies.

Based on the requirements of my policy language (in chapter 5.2.2), the Table 2.1 shows the differences of the existing adaptation languages. To compared with mine, Ponder language can define the adaptive conditions, but it do not support relocation of software components for adaptation. However, Ponder language provides a policy compiler to resolve the different types of constraints at compile time, not like mine. Although RELAX language supports monotonicity as other languages, but it does not separate the concerns for adaptation developers, and the developed programs have to be compiled. However, RELAX language does not support reusability and scalability([97]). Rei language and Joonseon's proposed policy language separated conditions and actions for adaptations like mine. However, both of them does not support relocation of software components as actions, they focus on variables, and the adaptive programs does not support reusability and scalability. Not like mine, Stitch language is a compiled language instead of interpreted languages, therefore the adaptive programs need to be compiled and create a middle code. Stitch also does not separate adaptive conditions and actions from their programs.

Table 2.1: Comparison with existing language.

Requriements/Languages	Ponder	RELAX	Rei	Stitch	Joonseon's	Mine
Monotonicity	O	O	O	O	O	O
Independence	X	X	O	X	O	O
No-compiling	X	X	O	X	O	O
Reusability	O	X	X	O	X	O
Scalbility	O	X	X	O	X	O

However, I defined the policy language is an interpreted language, and separation of concerns of adaptation. In addition, the condition part and action part are separated in mine for describing adaptations. The former is written in a first-order predicate logic-like notation, where predicates reflect information about the system and applications. The latter is a specified action which responsible for relocation of software components for adaptation on distributed systems. Furthermore, I proposed the policy language support reusability and scalability for developers.

2.3 Software-level Adaptation

The third type is software-level adaptation. Common approaches to software-level adaptation enables software to be dynamically modified in accordance with environment changes.

Genetic programming (GP) [47] [48][33] is an automated method of creating a working computer program from a high-level problem statement of a problem. Genetic programming starts from a high-level statement of what needs to be done and it automatically creates a computer program for adaptation. Therefore, applications which implemented by GP, it can adopt new algorithms for addressing concerns when unforeseen during development. However, it cannot predict adaptability on a distributed system. It also needs a large number of computing resources for adaptation. In addition, genetic programs cannot support reuse, and they do not support the relocation of software components. Therefore, developers find it difficult to define the destination of the software components.

Computational reflection aspect oriented programming (AOP) [46] enables software to be open to dynamically defining itself without compromising portability or exposing parts unnecessarily, where the software implementing a crosscutting concern, called an aspect, is developed separately from other parts of the system and woven with the business logic at compile- or run-time. Many researchers have introduced AOP into adaptive distributed systems. For instance, McKinley et al., [68] proposed a middleware system with compositional adaptation by using AOP. They can modify parts of programs running on single computers but do not support distributed systems themselves. Satoh et al., [75] proposed a bio-inspired adaptation by introducing the notion of cellular-differentiation into distributed systems to change available functions in accordance with the frequency of invoking the functions from the external system; however the adaptation cannot migrate any functions between computers.

Bonabeau et al. provided a detailed look at models of social insect behavior and how to apply these models to the design of complex systems [9]. They demonstrated how these models replaced emphasis on control, preprogramming, and centralization with designs featuring autonomy, emergence, and distributed functioning. These designs proved to be immensely flexible and robust, able to adapt quickly to changing environments, and to continue functioning even when individual elements failed. However, most swarm intelligence approaches have only focused on their target problems or applications but they are not general purpose, whereas distributed systems are. Software adaptation approaches should be independent of applications. In addition, computers in real distributed systems have no such room to execute such large numbers of computations and analyses.

Ji Zhang et al., proposed an approach to create formal models for the behaviors of adaptive programs [101]. Their approach separates the adaptation behaviors and non-adaptive behaviors from the specifications of adaptive programs. In addtion, their approach also presented a process to construct adaptation models that automatically generate adaptive programs from the models, and verify and validate them. The proposed approach focused on the behavior of adaptive programs not on relocation like mine. The quiescent states (e.g., states in which adaptations may be safely performed) of an adaptive program can be defined in the specific adaptation context which includs the program behavior before, during, and after adaptation, the requirements for the adaptive program, and the adaptation mechanism. Behaviors of the programs can be changed when adaptation, however, it is difficult for users to define where and how adapt to such changes for developers. Moveover, the proposed approach just require invocating the methods of the adaptive programs, but they don't support relocation of software components for adaptation and the fixed architecture can not be dynamically adapt to the changes from existing distributed systems.

Christos et al., present a middleware platform, and a policy language that has been designed and implemented for adaptive changes in mobile applications [22]. Based on Event Calculus [78], the policy language described that handles adaptation allows the dynamic modification of the adaptive behaviors in order to overcome potential conflicts and satisfy the user requirements. In addtion, their approach allows sharing of application status information among all applications running on the system. Like mine, the proposed approach rise adaptation to the language-level, however, they need to modify the behaviors for adaptation, therefore the proposed approach is difficult to dynamically add/remove components to/on existing systems, unlike mine.

2.4 Other Approaches

Based on the above three adaptation technologies, there are two applicated approaches for adaptation.

2.4.1 Architecture Approach

Architecture approach was presented in previous work [80] [26] [30] [17] [65][18] [87] [8] to adapt to various changes by changing their system architecture styles for adaptation on distributed systems.

Danny et al., presented a comprehensive reference model, named Formal Reference Model for Self-adaptation (FORMS) [96]. FORMS supports a small number of formally specified modeling elements which is correspond to the key concerns for self-adaptive software systems, and a set of relationships which is guided to the composition of the proposed approach. On the other hand, based on documenting, FORMS gives us a potential reusable architectural solution to adapt to changes through change the *parameter's* invocation by they defined interface. Like mine, FORMS not only adapt to environment changes, but also change the architectures. e.g., the model of self-adaptive system is desributed as follows:

However, when the execution of systems changes to complex and dynamic, the proposed approach is not sample for adaptive changes between different models, and change back the model is also a different task. Compared with their model, my approch support the relocation of software component, therefore, the fixed and complex architectures of distributed systems can be dynamically adapted in both directions.

Jacqueline Floch's group proposed a middleware system, called MADAM (mobility and adaptation-enabling middleware), the aims of their approach dynamically adapt to various changes of applications for mobile computing [26]. MADAM supports adaptability of applications, and it allows services to be composed in a flexible manner through *parameter's* changes. In the proposed approach, they use a UML profile to self-adapt changes on systems, which the architect will use to model the architecture. I also have a try to use UML format to define requirements of applications for adaptive distributed systems [40]. However, to execute UML need cost

$$
\begin{array}{l}
\textit{SelfAdaptiveSystem} \\
\hline
\textit{baseLevelSubsystems} : \mathbb{P}\, \textit{BaseLevelSubsystem} \\
\textit{metaLevelSubsystems} : \mathbb{P}\, \textit{MetaLevelSubsystem} \\
\textit{metaMetaLevelSubsystems} : \mathbb{P}\, \textit{MetaMetaLevelSubsystem} \\
\hline
\#\textit{baseLevelSubsystems} \geq 1 \\
\#\textit{metaLevelSubsystems} \geq 1 \\
\#\textit{metaMetaLevelSubsystems} \geq 1 \\
\forall\, \textit{mls} : \textit{metaLevelSubsystems};\ \textit{cm}, \textit{ce} : \textit{ReflectiveComputation} \bullet \\
\qquad \textit{cm} \in \textit{mls.computations} \land \textit{ce} \in \textit{mls.computations} \land \\
\qquad \mathrm{dom}\, \textit{cm.sense} = \{\textit{bls} : \mathbb{P}\, \textit{BaseLevelSubsystem} \mid \\
\qquad\qquad \textit{bls} \subseteq \textit{baseLevelSubsystems} \bullet (\textit{bls}, \textit{cm.state})\} \land \\
\qquad \mathrm{dom}\, \textit{ce.adapt} = \{\textit{bls} : \mathbb{P}\, \textit{BaseLevelSubsystem} \mid \\
\qquad\qquad \textit{bls} \subseteq \textit{baseLevelSubsystems} \bullet (\textit{bls}, \textit{cm.state})\} \\
\forall\, \textit{mmls} : \textit{metaMetaLevelSubsystems}; \\
\textit{cm}, \textit{ce} : \textit{ReflectiveComputation} \bullet \\
\qquad \textit{cm} \in \textit{mmls.computations} \land \textit{ce} \in \textit{mmls.computations} \land \\
\qquad \mathrm{dom}\, \textit{cm.sense} = \{\textit{mls} : \mathbb{P}\, \textit{MetaLevelSubsystem} \mid \\
\qquad\qquad \textit{mls} \subseteq \textit{metaLevelSubsystems} \bullet (\textit{mls}, \textit{cm.state})\} \land \\
\qquad \mathrm{dom}\, \textit{ce.adapt} = \{\textit{mls} : \mathbb{P}\, \textit{MetaLevelSubsystem} \mid \\
\qquad\qquad \textit{mls} \subseteq \textit{metaLevelSubsystems} \bullet (\textit{mls}, \textit{ce.state})\} \\
\hline
\end{array}
$$

computing resource more than a interpreter type language. In their research, they need centralized management, and do not support relocation of software components.

Moreira et al., proposed an architecture, called FORMAware. Based on reflection [1], the proposed approach blends run-time architectural representation with a reflective programming model to address *coordination-level* adaptation [64]. It opens up composition architecture through a replaceable default style manager that permits to execute architecture reconfigurations. This manager enforces the structural integrity of the architecture through a set of style rules that developers may change to meet other architectural strategies. Each reconfiguration runs in the scope of a transaction that I may commit or rollback. In addition, FORMAware prescribes a method to formally carry out architecture constrain verifications whenever architectural adaptation (e.g. add, plug, unplug, remove, replace components) is required, since the architecture structure is opened up and maintained by the reflective component model approach.

Remark FORMAware is similar with mine, the difference is that they do not upgrade to the definition of adaptations into language-level, they through reconfigurations to change the architecture style, and I use relocating software components for adaptation on architecture-level.

[1]Reflection technology is be used for invoke methods from objects for adaptation.

Garlan et al. presented a framework, called *Rainbow* [29] for specifying architecture-based self-adaptation. The aims of the proposed approach focused on reduce the cost and improve the reliability of making changes to complex systems through change the *coordination* of the adaptive programs. Although Rainbow supports automated, dynamic system adaptation via architectural models, however their approach was not solely aimed at distributed systems, it supports adaptive connections between operators of components, which might be running on different computers.

Yang's [100] group presents an architecture-based software adaptation through *coordination* of agents on their systems. On the basis of explicating and reasoning about architectural knowledge, they are mainly to automate the software adaptation in running system. The proposed architectures themselves can also be introspected and altered at runtime, to control the adaptation. They use the architectural reflection to observe and control the system architecture, while use the architectural style to ensure the consistency and correctness of the architecture reconfiguration. In addition, the proposed approach not only forms an adaptation feedback loop onto the running system, but also it separates the concerns among the architectural model, the target system and the facilities use for adaptation. However, they did not specify how to define the conditions of adaptation to change their structure style, therefore, it is difficult to dynamically adapt to frequent changes in existing systems.

Shang-Wen Cheng et al., described an approach for dynamic adaptation, which is supported by the use of *software* architectural models to monitor an application, and guide dynamic changes to it [18]. The use of externalized models permits one to make reconfiguration decisions based on a global perspective of the running system, apply analytic models to determine correct repair strategies, and gauge the effectiveness of repair through continuous system monitoring. Their approach is based on the 3-layer view illustrated in Figure 2.4.

The Runtime Layer is responsible for observing a system's runtime properties and performing low-level operations to adapt the system. It consists of the system itself, together with its operating environment (e.g., networks, processors, I/O devices, communications links, etc.) The Model Layer is responsible for interpreting observed system behavior in terms of higher-level, and more easily analyzed, properties. The Task Layer is responsible for determining the quality of service requirements for the tasks. In their system each architecture is identified with a particular architectural

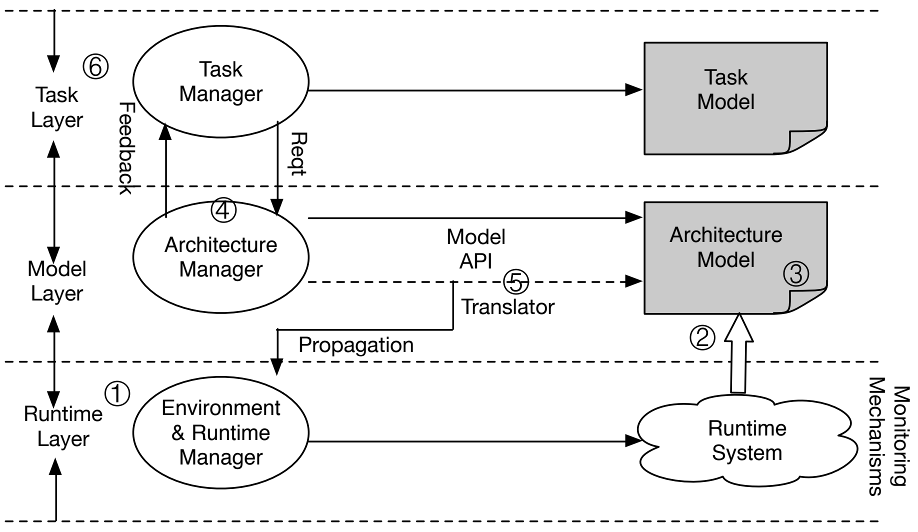


Figure 2.4: Architectural model

style. An architectural style defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed. However, in their proposed approach, their middleware doesn't support the relocation of software components, and the rules is difficult for defining by developers. Once the change in system configuration, how to change it back to the original structure it is not easy thing.

Jeff Kramer and Jeff Magee proposed an architectural approach for self-managed systems [49]. The architectural provided the required level of abstraction, and generality to deal with the challenges. In the proposed approach, the vision of self-management at the architectural level are described, where a self-managed *software* architecture is one in which components automatically configure their interaction for adaptation as *software-level adaptation.*

Figure 2.5 described a three layer reference model. It is that consist of component control, change management, and goal management. The proposed approach provided a context for discussing the main research challenges. At the component layer, the main challenge is responsible for providing change management which reconfigures the software components, ensures application consistency, and avoids undesirable transient behaviors. At the change management layer, decentralized configuration management is required which can tolerate consistent views of the system state, but still converge to a satisfactory stable state. At the goal management layer, some form of on-line planning is required. However, the adaptation concerns is written inside of

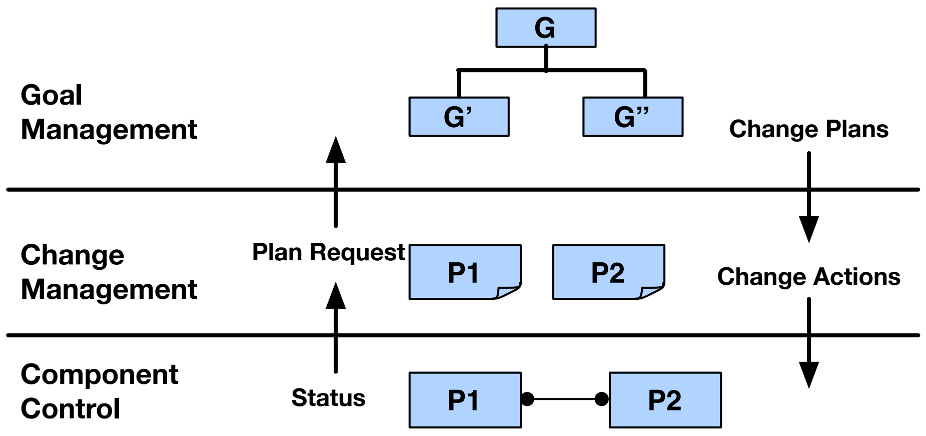


Figure 2.5: Architectural framework

components for adaptive changes in distributed systems. Unlike mine, the adaptation in their approach can't be reused. When environment changes frequently occurred in such system, their research can't be suited well.

2.4.2 Location Approach

There have been a few attempts to introduce to the location approach for adaptation [6] [35] [38] [88] [101] [66] [12] [92] [70] [32] [54] [76] [97] [63] [73] [49] [96] [95].

Israsel Ben-Shaul's group introduced two middleware systems in location approach [6][5][34][7] [35][36]. One of the middleware system is named Hadas, it focused on intra-component self adaptability. Another middleware system is named FarGo, it focused on inter-component self adaptability. Both of their systems were fully implemented, and by migrating the code of software components between computers for self-adaptation.

The FarGo system [35][36] focus on dynamic inter-component structure. It proposed a dynamic mechanism which laid out in a no-centralized manner between distributed systems and applications. The mechanism of FarGo system provides mobility of software components to adapt themselves to changes for self-adaptation. It is attachment the remote components into the same address, and then detachment the co-located components into different addresses. FarGo system provides an interface named *reference*, it can make duplications of the target components which follows the source code to destination computers.

FarGo system proposed two levels of adaptation for distributed applications.

- The dynamic relocation of software components may be viewed as an adaptation of distributed system architecture.
- Evolved the semantics of inter-component reference dynamically for adaptation.

Unlike the FarGo system, Ben-Shaul et al. proposed another system called Hadas [5][34][7]. It was designed to adapt to changes in distributed applications. Adaptation is a major element of Hadas and it dynamically supports the deployment of existing software components and autonomously addresses sites.

In particular, 1) Each software component in Hadas can be defined by users. 2) As each component is mutable, it extends a special class of adaptive objects termed Ambassadors, which can dynamically be deployed to the remote side of computers. 3) Each software component has an interface consisting of *methods* and *data members*, which can be evolved at runtime. 4) Each component contains two sections, the first is a fixed section, and the second is an extensible section. The data items and methods of the components are defined in the former as class-based items that may not be changed during their lifetimes. The latter comprises the mutable portion of the components through their structures and behaviors to dynamically adapt to changes. For instance, new/old items (data, objects, or methods) can be dynamically added/removed or changed for components on-the-fly. 5) Through Hadas's built-in meta-methods, which are responsible for structural and behavioral changes, individual components adapt themselves to changes in distributed applications. Readers can find detailed samples in their previous work [5][34][7].

Both of their approaches provided migration of software components for adaptation from the adaptability perspective. Their major distinction was the FarGo systemfs focus on reference as the subject of adaptability; whereas, Hadasf focus was on the definition of software components.

These studies represented two location research projects for adaptation, and they were very similar to my research. I also used the relocation of software components to adapt themselves to changes on distributed systems. However, their research and mine had two major distinctions. First, I not only migrated the source code, but also migrated the state of software components to destination computers for adaptation. However, their research just migrated source code for adaptation. When changes occurred, the software components could not be restarted to immediately deal with their tasks, whereas those in my research could. Second, Hadas components could define changes by users inside of components; however, I distinguished adaptation concerns from business logic concerns for adaptation. As Hadas provided meta-methods embedded into components, neither of the components nor adaptive functions could be reused, whereas my research not only provided software components, but also user-defined adaptive programs that could be reused. I promoted these adaptive definitions to language level.

Suda and Suzuki [83] proposed a bio-inspired middleware system, called Bio-Networking, for disseminating network services in dynamic and large-scale networks that have large numbers of decentralized data and services. The system provides reusable software components for developing, deploying and executing cyber-entities (CEs). Low-level operating and networking details can be embedded in CEs, and high-level runtime services provided to adaptability to their services and behaviors. In addtion, the system enables CEs to be replicated, moved, and deleted, like mine does. However, [83]'s policies for migration and duplication of agents has already been decided beforehand, and the users cannot define that in Bio-Networking middleware system. Conversely, mine proposal provides policy language to define adaptive conditions and behaviors for their systems to adapt to various changes. Furthermore, unlike mine system, Bio-Networking's target is a large number of homologous computers, and the destinations of agent migrations depended on the number of service requests in addition to the locations of clients.

Wu et. al, and Fok et. al [99] [27] are focus on sensor networks. Wu's group present a simplified analytical model for a distributed sensor network by using mobile agents. They formulate the route computation problem in terms of maximizing an objective function, which is directly proportional to the received signal strength and inversely proportional to the path loss and energy consumption [99], and Fok's group presents a mobile agent middleware for self-adaptive applications in wireless sensor networks [27], called *Agilla*. It provides a programming model in which applications consist of evolving communities of agents that share a wireless sensor network. The agents can dynamically enter and exit a network and can autonomously clone and migrate themselves in response to environmental changes like mine. However, these approaches do not separate the adaptation from agents, therefore, their adaptation can't be reused, and their proposals can not dynamically change the fixed system architectures.

Paolo et. al [4], and Gray et. al [31] proposed agent-based middleware for mobile computing. Such middleware facilitates service-specific optimization and allows users to adapt to local resources. Mobile users can change locations and dynamically adaptive mobility-enabled applications to the properties and characteristics of their network connections and hardware devices. In addition, mobile agents simplifies dynamic personalization by following user movements and tailoring service depending on personal preferences. Gray's group described a mobile agent system *Agent Tcl* [31] that is under development at Dartmouth College. They present a system to support agents that provide network sensing and routing services. It support agents allow an agent to transparently migrate between a mobile computer and a permanently connected machine or between one mobile computer and another regardless of, when the mobile computers connect to the network. However, both of them did not separate the adaptation from agents, therefore, their adaptation can't be reused.

Radu et al., presents a framework, called DACIA, for building adaptive distributed applications [52]. In DACIA, distributed applications are viewed as consisting of connected components that typically implement data streaming, processing, and filtering functions. It also provides mechanisms for runtime reconfiguration of applications to allow them to adapt to the changing of operating environments. The key contribution of DACIA supports migration of components from original host to different host during execution, while maintaining communication connectivity with other components. The proposed approach is similar with mine, however the runtime reconfiguration of applications are difficult to be reused. Once the software components is relocated, the new reconfiguration could cause conflicts, in their paper, they did not consider this issue.

Ito et. al., [39] presents a communication infrastructure of agent-based middleware for ubiquitous computing environments. The proposed approach provide an adaptive communication mechanism that can select communication schemes flexibly, according to the properties of inter-agent communication, and resource status. In addtion, this mechanism enables the agent platform to adapt to various inter-agent communication requirements in a ubiquitous computing environment with limited resources. However, the proposed middleware doesn't separate the adaptation from agents.

2.5 Discussion

The existing studies have been proposed various approaches through different cases for adaptation. However, to satisfy I defined requirements (in Chapter 1), they all have some issues remain unresolved. The Table 2.2 shows the differents between existing research and mine. Parameter-level adaptation has inherent weakness, such researches cannot dynamically allow new components to be added into their systems, but mine reserach can solve this problem throuth relocation of software components between computers. To compared with coordination-level adaptation, there are two important differents with mine. Such researches can coordinate functions between computers to provide adaptability, but the function can not be relocated to destination computer to execute, network environment restricts the effect of those approaches. In addition, some policy language was presented in those approaches, but almost all languages only can describe the adaptive conditions. It was not specifically designed for relocation of software components, therefore, almost all languages do not support migration for adaptation in distributed systems. It is why I designed a policy language to this dissertation. In addition, software-level approaches were support modification for adaptation in single computer. For distributed systems which was composed of a large number of computers, the adaptive behaviors ware limited than mine.

Architecture and location approaches also were presented, and mine approach have distinguished between the two proposals. Compared to the former, users do not need to describe the styles of their architecture. Only through the relocation of software components, the system architectures can be dynamically changed in distributed systems. Compared to the latter, users can define their requirements as policies, and then mine proposal can automatically relocate the specified software component to the destination computers for adaptation.

Table 2.2: Comparison with existing adaptation researches.

Research/Requriements	Parameter-level	Coordination-level	Software-level	Architecture	Location	My Research
Self-adaptability	X	O	X	X	O	O
On-demand	O	O	X	O	X	O
Separation of concerns	X	O	X	X	X	O
Availability	O	O	O	O	O	O
Resuability	X	O	X	X	X	O
No-centralized	X	X	O	X	O	O
General-purpose	X	X	X	X	O	O

2.6 Summary

This chapter introduced four types of approaches to adaptation. A number of systems and research influenced this dissertation and made similar contributions to those presented here. However, they did not solve the requirements of modern distributed systems or the challenges faced like those in the present research.

In dynamic environments, various changes may change over time and time. The first common type involved parameter-level adaptation for distributed systems. Such researches are often adapted changes to variables in distributed systems, and then supported various services to users. However, those approaches can not change the fixed system architecture to provide adaptability in their systems, and then they do not allow the new algorithms or software components to be added to their systems when the requirements of users or system environment are changed. Therefore, the presented approaches have a common weakness that is the adaptive behaviors are limited in distributed systems.

The second type involved coordination-level adaptation for distributed systems. Most research has been proposed by using policy-based or rule-based language to define the conditions of changes outside components. However, as far as is known, existing research has not had any language focus on dynamically defining the destination to relocated software components for adaptation on existing distributed systems. Therefore, I developed a policy-based specifying language to define the conditions and events for each change, assigned when and where the software components should to be relocated, and when changes occurred between the system itself or applications. Moreover, the present research could dynamically change the system architecture to adapt components to various changes.

The third type involved software-level approaches to adaptation. Software components in these approaches could be reconfigured/rewritten through genetic programming or swarm intelligence. These approaches were initially common in non-distributed systems. Therefore, they only focused on their target problems or applications but they were not general purpose. Software-level approaches should be independent of applications. Furthermore, distributed computers have no such room to execute such large numbers of computation resources for analysis.

Following the above three tecnologies, there are two applicated approaches for adaptation. *Architecture approaches*, which enabled the system architecture to be dynamically changed for adaptation. However, it needed certain formats to define the changes, thereby changing the style of the system. The main problem was who defined the changes, and which architecture was best. It was very difficult to use, and developers were required to be familiar with well-known properties of their systems, whereas existing systems may be down/stop to execute tasks. As was previously explained, I previous approach was of this type, but I focused on the relocation of software components for adaptation, according to user-defined policies. The proposed approach does not reduce the functions of components, it just moves the location of execution; therefore, it is more reliable on existing distributed systems. *Location approaches* often used migration of agents to adapt to changes on distributed systems, whereas the proposed approach migrates source code between computers; code is not only relocated, but also the state of components to the destination. Therefore, tasks running on distributed systems can restart immediately. In addition, unlike existing approaches, the present research discussed here raises the description of requirements to language level.

Next, the concepts and solutions to certain key ideas will be presented with several scenarios.

Chapter 3

Proposed Approach

This chapter focuses on introducing the main concept underlying the proposed approach. First, why existing researches cannot solve these problems are explained through several scenarios, which include the dynamic changes to system architecture between Client/Server and Peer-to-Peer systems, the dynamic addition/removal of computers, the disconnection/reconnection of networks, the dynamic distribution of resources, and the dynamic reductions in the number of message deliveries in publish/subscribe systems. The proposed approach can resolve all these issues.

Second, key ideas behind the proposed approach on how to solve these problems will be introduced by using policy-based relocation of software components to define functions between computers. Although the approach is very simple, it can adapt to various changes between general-purpose applications and distributed systems, which are the most important in this research; the architecture of distributed systems can be dynamically changed to adapt to various changes. For instance, the architecture between Client/Server and Peer-to-Peer systems can be changed. Moreover, the proposed approach can not only relocate software components, but can also relocate their states to the destination for adaptation. This means temporarily halted processing can be immediately restarted on another computer.

Finally, since the proposed approach does not have centralized management, and adaptation concerns are separated from business logic concerns, both software components and policies can be reused on-demand in different distributed systems.

3.1 Overview

Many researchers have proposed various approaches to adapt to changes on distributed systems [20] [38] [88] [89] [54] [86] [56] [29] [7] [52] [63] [74] [101] [85] [66]. These approaches have outlined parameter-level, coordination-level, software-level adaptation, and other approaches are applicated through them which are introduced in Chapter 2. However, these approaches cannot support the policy-based relocation of software component to define functions for adaptive changes between the architectures of distributed systems and requirements of distributed applications. This chapter introduces the proposed approach on how to resolve problems that existing approaches have not been able to solve.

The proposed approach can fully meet the demands presented in Chapter 1. Next, five scenarios will be introduced to illustrate its effectiveness through our policy-based relocation of software components.

3.2 Scenarios

A widely distributed system can generally be anticipated that is comprised of numerous computers, which form a collection with immense aggregate computing, communication, and storage capabilities. For described various changes on distributed systems, five scenarios are outlined as follows.

Adaptive Peer-to-Peer and Client/Server Architecture

Client/Server and Peer-to-Peer are the most classic architectures for distributed systems. There are many applications based on these architectures for constructing file-sharing, chatting, and multimedia applications. However, both of them have advantages and disadvantages. For instance, if specific Peers are frequently requested to provided data, the Peer-to-Peer architecture cannot meet these needs, but the Client/Server can (Figure 3.1). On the other hand, if high delay between Server and Client, the Client/Server architecture cannot meet these needs, but the Peer-to-Peer can. Such architectures should be dynamically changed as the environment changed.

The proposed approach can solve this problem. For instance, Peer-to-Peer systems generally have client and server functions in Peers (Figure: 3.1). It was assumed that server functions could be relocated to be concentrated on a computer in such systems, which then dispersed client components within the remaining computers to change their architecture. In other words, a Peer-to-Peer system architecture could be dynamically changed to a Client/Server architecture. In addition, the proposed

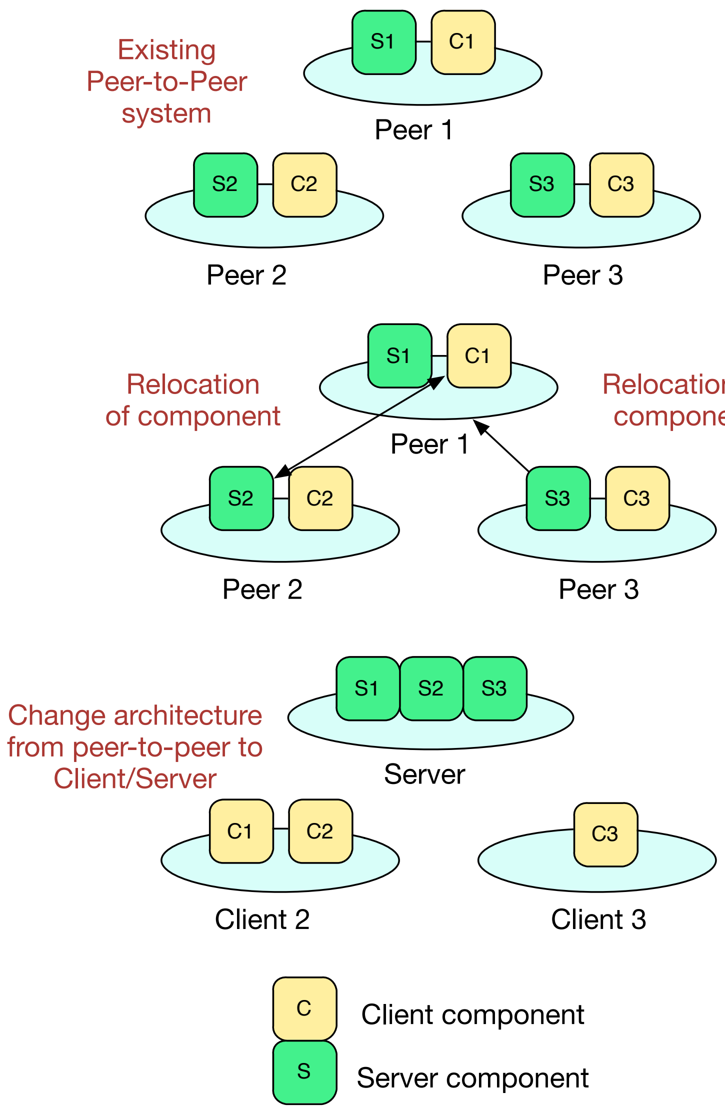


Figure 3.1: Adaptive system architecture

approach can be used to easily change system architectures to adapt to changes. However, while other research techniques cannot do this, the proposed approach can change the system structure in both of them, and gain all of the benefits from Peer-to-Peer and Client/Server networks.

The opposite of existing research, such as that on software-level and coordination-level approaches try to adapt to changes of the system architecture for adaptation. However, they need to rewrite the software components themselves, or change the coordination of components to adapt to changes instead of relocation of software components. Both of them can only adapt to software components themselves when changes occur, and both of them are difficult to be reused. Parameter-level approaches can change parameters of functions for adaptations, but it does not support to adapt to changes as architecures for adaptations. Architecture approaches can change the style of the system architecture through user-defined policies/rules, but their policy-based language can only describe the conditions of the changes, and these types of research cannot easily define their architectures to adapt to various changes, and they cannot immediately return the state before changes occur. Location research can migrate software components to the destination. However, they do not support adaptive changes to the system architecture, and they do not separate adaptation concerns from business concerns. Therefore, the software components cannot be reused for different distributed systems. Moreover, as far as is known, such approaches have no language that can define when conditions are changed, where the software components should be relocated for adaptation.

Adaptive Addition and Deletion Computers

In distributed systems, computers and software components of which an application consists may be added to or removed from them (Figure: 3.2). When the environment or system requirements changed, dynamically increase or decrease computing resources and software components are required. The proposed approach can solve the problems through dynamically relocating software components according to user-defined policies. System developers only need to define what computers or software components and when they need to adapt themselves to changes for adaptation. These software components will then freely be relocated/duplicated then relocated/removed from the whole distributed system for adaptation.

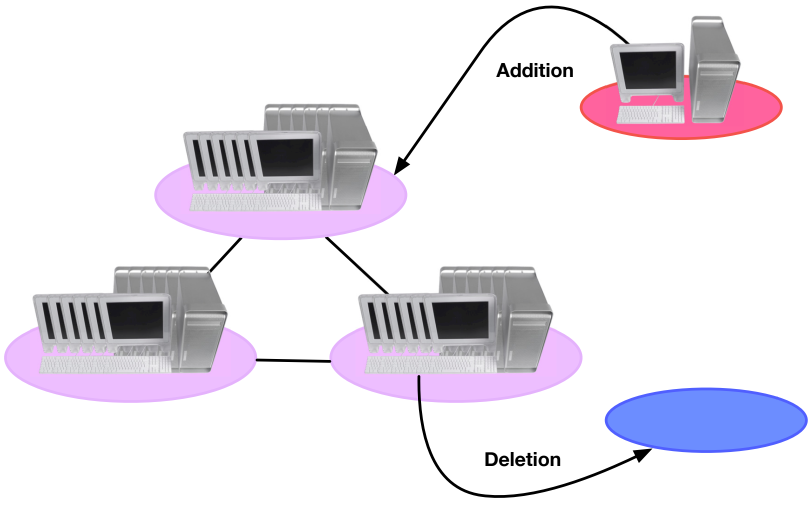


Figure 3.2: Adaptive addition & deletion of computers

The opposite of existing research, through parameters changed, parameter-level approaches can adapt to changes in distirbuted systems, software-level approaches can provide reconfigured/rewritten adaptation programs inside their components, coordination-level approaches can change how the methods of components are invoked, which are running on computers, architecture approaches need policies/rules to define adaptive functions outside components to adapt to changes on distributed systems or change the architecture for adaptation. However, these types of approaches can adapt themselves to static changes instead of dynamic changes, for instance, to add computers/software components to or remove computers/software components from the external environments of their systems. Although location approaches can migrate components as agents to destinations, this technology does not support user-defined actions at the language level for specific adaptation. Our approach provides several policy formats for users to easily define policies. Policy-makers just require under what conditions they are defined and when they need to be relocated to destination then computers/software components will automatically be added to/remove from distributed systems.

Adaptive Network Environment

Although penetration of networks determines the level of a country's scientific and technological advances, network delays still occur in the same place even if networks are very common in developed countries due to changes in the increasing number of users. Adaptation technology is required like that in these unstable networks. For instance, frequent network connections, disconnections, and then reconnections within a certain time (Figure: 3.3) will cause high levels of delay and packet losses in their networks. The proposed approach can solve the problems through dynamically relocating software components between computers. The proposed approach is an effective way in this scenario to adapt to network failures between changes in frequently connected and disconnected networks. One or more policies can be predefined to describe when networks are connected and when software is relocated to its destinations. In contrast, these components can package their codes/objects to wait for network connectivity. When a network is reconnected, our system can dynamically relocate the software components to the destination side. Because our system not only relocates the software components, but also relocates their state, these components can go on working at destination computers for users. When components have finished processing, they just need to take the results back to the start location. The proposed approach can greatly reduce the communication delay time between computers to enable adaptive network changes.

The opposite of existing research, such as that on parameter-level, software-level and coordination-level approaches focus on adaptation on local computer instead of computers. However, they cannot be applied to frequent changes and cannot be reused. Architecture approaches still cannot solve the problem. Although this research can change the architecture style to provide adaptation through user-defined policies/rules, there is essentially no way to reduce the number of communications between components through networks. Location approaches can migrate software components to destinations for adaptation. However, they have not separated adaptation concerns from business concerns, and as they have written adaptation inside software components, these approaches cannot be reused for different distributed systems. As far as is known, these have no existing languages that can define where the software components should be relocated to adapt to changes on existing distributed systems. However, my research can define a pair of condition and action for relocating the software components for specificing adaptation.

Adaptive Resource Management

There have been many approaches in distributed systems to adaptive resource management. If system resources can realistically be used, our systems can greatly improve the usage of computers, and reduce the cost of adding new servers. For instance, Amazon's cloud involves global scale distributed systems, from which users can borrow resources that they need. Although users most cases often use system resources for a fixed period of time, they still cost money when they are not used (Figure: 3.4). The proposed approach can solve the problems through dynamically relocating software components between computers. In addition, the proposed approach is more effective in these cases. For instance, when users do not use system resources, software components can be relocated to destination computer and to sleep the computers. In contrast, when more computing resources are needed that are beyond the upper limit of resources that can be borrowed, programs and the state of objects can be migrated to local computers through the relocation of software components, or resources can be dynamically allocated to who needs them during specific time, both of users and providers of resources can gain benefits through my approach.

Although existing researches try to slove this problem, however, software-level approaches need a numbers of resources to predict, and it is difficult to self-adjust system resources based on user requirements. Parameter-level and coordination-level approaches can adapt to such changes on existing distributed systems, but both of them should to define adaptation programs inside of software components, however, they can not dynamically add new resource, and cannot be reused in frequent changes.

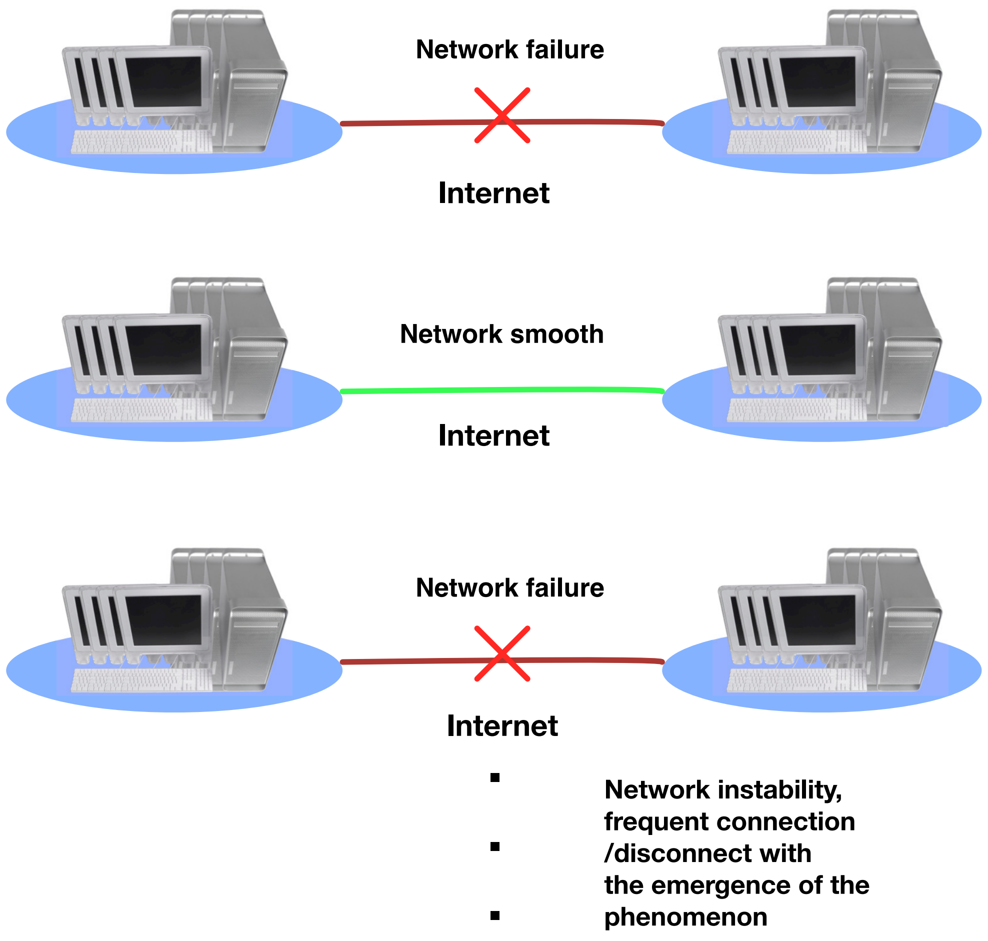


Figure 3.3: Adaptive network environment

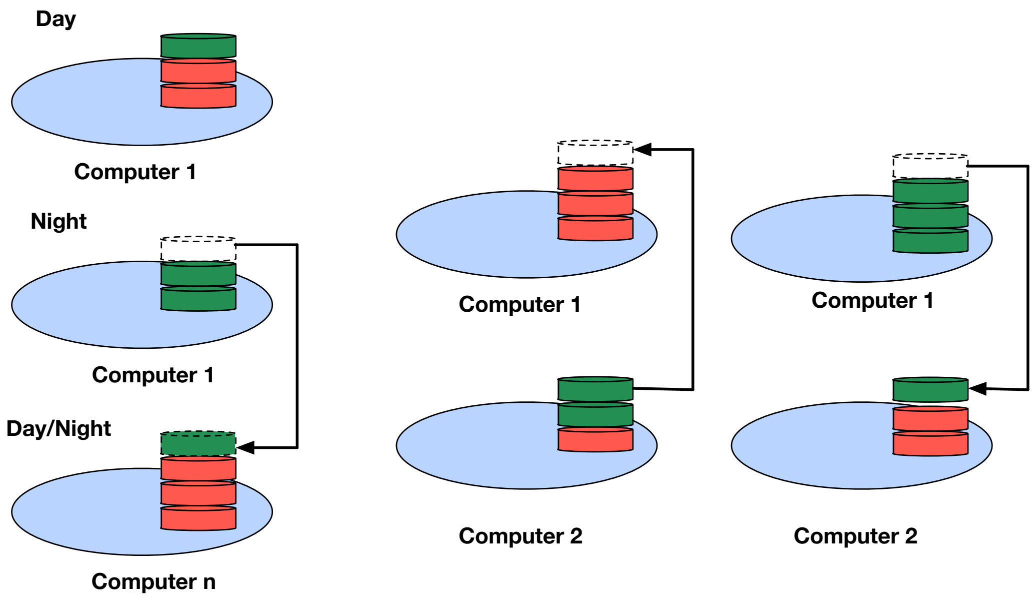


Figure 3.4: Adaptive resource management

Architecture approaches and location approaches do not separate adaptation concerns from those of business. Location approaches focus on migrate agets between computers instead of migration of software components for adaptation. In addition, these approaches are difficult to be reused in diffident types of distributed systems.

Adaptive Publish/Subscribe Events

The last scenario introduces how to adapt events in publish/subscribe (pub/-sub) systems. Pub/sub is a messaging pattern where senders of messages are called publishers. The messages are then to be directly sent to specific receivers, who are called subscribers. However, the published messages are divided into classes without knowledge about which subscribers they were sent to. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge about which publishers sent them.

Messages in pub/sub need to be transferred through various brokers, who choose the shortest routes for application receive events (Figure: 3.5). However, when needs are changed by user-defined applications, messages will be lost or system latency will be increased. The proposed approach can solve this problem in this case. When requirements change, software components can be relocated from brokers to subscribers, according to user-defined policies. Some copies of the software components can also be made, which are then relocated to a number of subscribers. Message loss through transmission and delay time can be reduced by relocating software components.

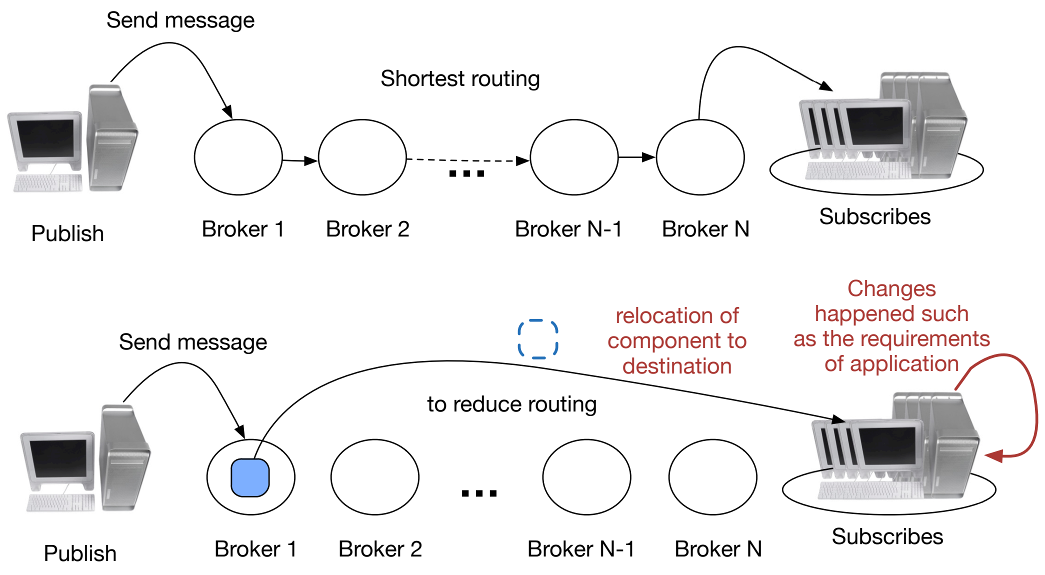


Figure 3.5: Adaptive events of publish/subscribe system

Existing researches focus on the shortest routing for exploring instead of adaptation. Only a few of researches describe adaptation for pub/sub system. However, unlike I described this scenario, parameter-level, software-level and coordination-level approaches, supports adaptation, but in their researches, they need numerous resources to adapt to changes, and just changing the coordination of components cannot solve adaptation problem. Architecture approaches also cannot solve the problem because they do not need to change the style of system architecture to provide adaptation through user-defined policies/rules. Near with mine, location research can migrate software components to adapt to changes. However, they cannot adapt to messaging control modes by relocating software components on pub/sub systems with language.

3.3 Approach

Although reconfiguration software components can feasibly be endorsed through these scenarios or software component coordination can be changed to provide adaptation, such approaches make it difficult for developers to define functions, and such software components cannot be reused. However, the proposed adaptation is very simple, and it can adapt to more cases than those with the existing researches. A policy-based middleware was attempted to materialize this idea for these reasons, which could relocate software components to adapt to changes on existing distributed systems.

- I focus was on the relocation of software components for adaptation. This is because this approach does not need to increase or decrease the number of functions for systems; only the location where software components are executed is changed. Although it seems simple, the proposed approach does not reduce the reliability of distributed systems, but improves adaptability between systems themselves and applications, and even their system architectures can be dynamically changed for adaptation. Two items should be considered.
- The adaptation programs should be separate from software components to define changes between distributed systems and applications for reuse on different distributed systems.
- There may be many applications in distributed systems, which contain various components running on them. Therefore, many changes may simultaneously occur. Individual components in the new approach can have one or multiple policies to control software components.

Figure 3.6 presents the steps in the approach on how to provide adaptation on distributed systems.

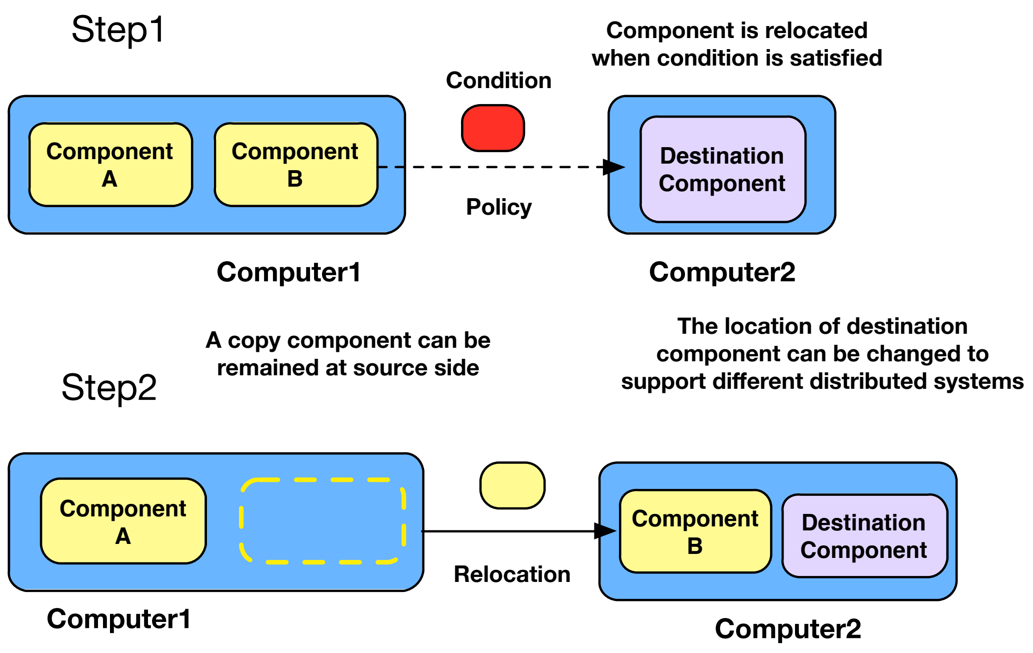

Step1
Component is relocated when condition is satisfied
Condition
Component A
Component B
Destination Component
Policy
Computer1
Computer2
Step2
A copy component can be remained at source side
The location of destination component can be changed to support different distributed systems
Component A
Relocation
Component B
Destination Component
Computer1
Computer2

Figure 3.6: Approach solutions

- First, environmental changes should be defined as policies by using the policy-based language (in Chapter 5), which are saved at a policy database.

- Second, when the defined conditions of policies are satisfied, the actions of policies will be invocated, then specific software components can automatically be relocated to destination computers through the system and network monitor, which can determine the relocated destination of software components.

- Third, these software components can relocate to or duplicate themselves and then relocate to destination computers. When the software components relocated to destination computers, the methods of such software components can be dynamically invocated through the original designed dynamic method invocation mechanism.

Middleware and policy language are introduced on the basis of this idea in the chapter 4 and 5, which were developed for adaptation on distributed systems with several applications.

3.4 Discussion

In this section, I discuss how my proposal can satisfy the requirements of my definition in 1.2 and explain the differences between related works and the proposed approach.

- *Self-adaptability.* The proposed Mimosa middleware can satisfy *self-adaptability* well. This is because the Mimosa middleware system can dynamically monitor the external environment changes, and if the changes are matched with the conditions of policies, software components can automatically be relocated to destination computers through it.

- *On-demand deployment of software.* Users can define their own policies to control the software components to adapt to changed behaviors, therefore, the proposed approach can deploy the software on-demand.

- *Separation of concerns.* Because the proposal separates adaptations from software components, the proposed approach can significantly reduce the burden for different users. This is because developing adaptation programs requires a lot of experience and high programmability. *Separation of concerns* can make different levels of developers dedicated to their work.

- *Availablity.* The proposed middleware system supports duplication of software components through user definition of *copyTo(x)* function (details in Chapter 5). Therefore, *multiplexing* can effectively improve the *availablity* so as not to cause failures. This is because the software components can be duplicated, and executed at different computers beforehand.

- *Reusability.* Because the user-defined adaptation policies are not required to be built in software components, both of the policies and software components can be *reused* to adapt to changes on Mimosa middleware.

- *Non-centralized management.* The proposed approach can provide dynamically adaptation through the relocation of software components in distributed systems, therefore, the approach donot bound any system architectures. On the contrary, dynamic adaptation does not require centralized management in my proposal.

- *General-purpose and adaptation independence.* The proposed approach is designed and implemented for general distributed applications, and my proposal separate adaptations from software components, therefore, the proposal can be better used in general-purpose application and independence adaptations for distributed systems.

Remark Compared with the proposed proposals, such as parameter-level approaches, and coordination-level approaches can change the parameters of functions and cooperation of processes. Software-level approaches need to modify program variables for adaptation. However, those approaches will reduce the reliability for distributed systems. Furthermore, such adaptation approaches are not separated concerns of developers, and adaptations are built inside of software components, therefore, such components are difficult to be reused. On the other hand, architecture approaches and location approaches are near with mine, but the main difference is their migration policies have been decided beforehand. However, the proposed proposal can freely define various policies for adaptive environment changes in both general-purpose applications and existing distributed systems.

Relocation of software components for adaptation seems simple, but it can provide strong adaptability. The *advantages* of the proposed approach are listed as follows:

- Software doesnot need to be rewrited in my proposed approach, and not only the changes of distributed systems, but also the changes of applications can be dynamically adapted to through relocation of software components.

- The execution position of software can be changed in my approach, however, the functions of distributed systems doesnot need to be increased or decreased through the whole system.

- The state of software components can be relocated from original computer@to destination computer.

- Through changing the execution position of software, the architectures of distributed systems can be changed dynamically.

3.5 Summary

This chapter presented five scenarios to explain why the existing research could not solve these problems, such as dynamically changing system architectures, dynamically adding/removing computers, connecting/disconnecting networks, dynamically distributing resources, and dynamically reducing the number of message deliveries in pub/sub systems. It then mainly introduced the proposed approach on how to solve these problems through the policy-based relocation of software components, and explained why existing research could not resolve these problems. The last section provided detailed descriptions of why the focus was on the relocation of software components for adaptation on distributed systems, explained why the policies were separated from software components for adaptation. How software components were relocated in two steps on user-defined policies were introduced which based on the key ideas of mine (The details are given in Chapters 3 and 4).

The next chapter mainly introduces the design of the middleware system I proposed and a policy-based language on how to define functions outside software components based on the key ideas of mine. In addition, how the interpreter is located inside the adaptation manager is also introduced to enable polices to be easily executed on the new middleware, which does not consume large amounts of system resources.

Chapter 4

Mimosa: a Dynamic Adaptation Middleware through Relocation of Software Components for Distributed Systems

The work presented in this chapter was first described in an earlier paper by Sun et al. [82] [40]. Unlike that in existing approaches, Mimosa is presented here, which is a dynamic adaptation middleware for distributed systems. Software components automatically relocate or duplicate/relocate or remove themselves between computers with the Mimosa system, according to user-definitional policies. Application developers only need to focus on their development programs, and Mimosa provides self-adaptations on their distributed systems.

The Mimosa middleware system that is proposed here has four distinct benefits: I. Not only the source code of software components, but also the states of the software components can be relocated to destination computers. Ⅱ. The software components can not only relocate themselves to destination computers, they can make copies of themselves and spread themselves via networks. Ⅲ. Developers can easily hide the differences between the interface of their objects at original and other computers by using the developed dynamic method invocation mechanism. Ⅳ. Because Mimosa was developed with Java language, general-purpose applications can easily be constructed on any operating systems.

4.1 Strong and Weak Relocation

The state of execution is relocated with the source code so that computation can be resumed at the destination. According to the amount of detail captured in the state, component relocation can be classified into as two types: **strong and weak relocation**.

- **Strong relocation**: is the ability of components to relocate over a network, carrying their code and execution state, where the state includes the program counter, saved processor registers, and local variables, which correspond to variables allocated in the stack frame of the components' memory space, i.e., global variables. These correspond to variables allocated in the heap frame. A component is suspended, marshaled, transmitted, unmarshaled, and then restarted at the exact position where it was previously suspended on the destination computer without loss of data or execution states.

- **Weak relocation**: is the ability of components to relocate over a network, carrying their code and partial execution state, where the state is variables in the heap frame, e.g., instance variables in object oriented programs, instead of its program counter and local variables declared in methods or functions. A component is relocated to and restarted on the destination with its global variables. The runtime system may explicitly invoke specified component methods.

Although strong migration can cover weak migration, this is a minority. This is because the execution state of software component tends to be large and the marshaling and transmitting of the state over a network need heavy processing. Therefore, the latter was chosen for the study presented in this dissertation to design the Mimosa middleware system. Compared to existing studies, the proposed approach can provide adaptation between general-proposed applications and distributed systems without losing availability, dependability, or reliability. In fact, the proposed approach can also adapt to changes of structures of distributed system dynamically as architecture-level approach.

4.2 System Model

The requirements of applications in distributed systems and their structures may often change, however software components can easily adapt to these changes. Existing adaptation technology needs large amounts of computational resources, but the range of adaptation is limited, or adaptive content cannot be predicted. Therefore, an attempt was made to relocate the software components to adapt to changes in distributed systems which was proposed by me. This section introduces my system model that can make software components adapt to various changes in their systems, networks, and applications.

I define an adaptive model (4.1) is defined as:

```
Data : User-defined Policies
Result: Determine whether to adapt to changes through
 relocate the software components
initialization;
read policy context;
while conditions is satisfied do
    if action.duplicatation is true then
        duplicate software components;
        relocate component to destination;
    else if action.relocation is ture then
        relocate component to destination;
    else
        remove component;
    end
end
```

Figure 4.1: Adaptive model

The above figure indicates that Mimosa system developers need to define the policies first with the proposed policy-based language (the details are given in Chapter 4). Mimosa has three options according to user-definitional policies: I. Make copies of the software components and relocate themselves to multiple destination computers. Ⅱ. Relocate the running software components to designated computers in their network. Two options are provided for developers in this case. For instance, automatically select the destination, or manually specify it. Ⅲ. The software components can finish their tasks at the departure point or at destination computers, which is determined by developers.

4.3 Approach: Dynamic Adaptation for Distributed Systems

The proposed approach dynamically deploys components to define application-specific functions at computers according to the policies of the components to adapt distributed applications to changes on distributed systems.

4.3.1 Software Components

It has been assumed that an application consists of one or more software components, which may be running on different computers. Each component is general-purpose and is a programmable entity. It can be deployed at another computer according to its deployment policy, when it has started to run. It is defined as a collection of Java objects like JavaBeans components in the current implementation. It also has an interface, called a *reference*, to communicate with other components through dynamic method invocation developed in the common object request broker architecture (CORBA) [79]. The interface supports the notion of being mobility-transparent in addition to that of inter-component communication, in the sense that it can forward messages to co-partner components after it has migrated to another computer through a network. Consider the four items below.

- The approach supports relocation of software components but no adaptations inside software components. When more than one dimension must be considered for adaptation, representing the policies and choices between policies tends to be too complicated to define and select policies. Therefore, support for at most one dimension is intended, e.g., the dynamic deployment of components.

- Each component has one or more policies, where a policy specifies the relocation of its components and instructs them to migrate to the destination according to conditions specified in the policy. The validation of every policy can be explicitly configured to be one-time, within specified computers, or permanent within its component.

- Each policy is specified as a pair of a condition part and at the most one destination part. The former is written in a first-order predicate logic-like notation, where predicates reflect information about the system and applications. The destination part refers to another components instead of the computer itself. This is because such policies should be abstracted away from the underlying systems, e.g., network addresses, so that they can be reused on other distributed systems. The policy deploys its target component (or a copy of the component) at the current computer of the component specified as the destination, if the condition is satisfied.

- The approach also provides built-in policies for adaptation as extensions of this primitive relocation policy. In fact, it is not easy to define relocation policies, because such policies tend to depend on the underlying systems.

Since components for which other components have policies can be statically or dynamically deployed at computers, the destinations of policies can easily be changed for reuse by other distributed systems. Next, I will describe my built-in policies for adaptation in distributed systems.

4.3.2 Adaptation Policy

My approach provides a policy-based language to help users to define their adaptation outside of software components, it is developed by using a ***Condition-Action*** format, therefore, users can easily describe the changes of environment as policy's condition, when their defined conditions are satisfied, the appropriate action will be invoked through methods of the software components. I also defined a set of essential and useful adaptation policies for distributed systems. Each policy is only activated when the condition specified in it is satisfied, where the condition is written in a first-order predicate logic-like notation [67], where predicates reflect information about the system and applications (The mathematical definition is presented in Chapter 5). The five policies format is presented as follows:

- Attraction: Frequent communication between two components yields stronger force. Both or one of the components are dynamically deployed at computers that other components are located at.

- Repulsion: Components are deployed at computers in a decentralized manner to avoid collisions between them. This policy relocates software components from regions with high concentrations of components to those with low concentrations.

- Spreading: Copies of software components are dynamically deployed at destination computers and propagated from one computer to another over a distributed system. This policy progressively spreads components to define functions over the system and reduces the lack of functions.

- Evaporation: Excess of components results in overloads. The same or compatible functions must be distributively processed to reduce the amount of load and information. This policy consists in locally applying functions to synthesize multiple components or periodically reduces the relevance of functions.

- Time-to-Live: After a certain time is reached, software components or their clones can be migrated to destination computers. This policy can periodically send software component to destination computers that allow members of their group to synchronize message changes.

Through the above formats, the user can define the necessary procedures to adapt to changes according to their requirements. Once the adaptive program started, the following middleware will help them to achieve their dynamic requirements.

4.4 Mimosa: System Design

This section introduces the design of the proposed Mimosa system, and presents the core technologies for adaptive distributed systems.

4.4.1 Adaptive System Architecture

The proposed middleware architecture consists of three parts: a component runtime system, an adaptation manager, and software components of applications (Figure. 4.2). From this architecture, it can be noted that:

- The first part is a component runtime system. It consists of three parts where the ***component deployment manager*** is responsible for executing software components. The ***component discovery service*** is responsible for relocating software components. However, it can control the behavior of components, fetch and determine themselves where the software components should be moved. ***Relocation transparent method invocation*** is responsible for enabling them to invoke methods on the destination side's software components through my defined ***Activity*** interface. However, the software components need to be marshaled in the first part by using the method invocation mechanism, and then migrate themselves from one computer to the destination. When these software components arrive at destination computers, data items are unmarshaled, and they then communicate with the components of destination computers, according to naming inspections.

- The second part is the adaptation manager. It also consists of three parts where the policy language interpreter is responsible for executing user-defined policies. The policies are written in a ***Condition-Action*** format policy language and a database system to maintain the policies. The ***Destination Address Service*** is responsible for managing, determining the software components who should be addressed to which destination computer. The ***System&Resource&Network Monitor*** is responsible for monitoring dynamic changes from the external systems.

- The third part consists of distributed applications, which can be designed by using any java-based general purposes.

4.4.2 Component Runtime System

Each runtime system allows each component to have at most one activity through the Java thread library. When the life-cycle state of a component changes (The details are provided as follows). When it is creates, terminates, duplicates, or relocates to another computer, the runtime system issues specific events to the component. Each component can have more than one listener object to capture such events that implements a specific listener interface to hook certain events issued before or after changes have been made in its life-cycle state. The current implementation uses the notion of dynamic method invocation studied in CORBA [79] so that it can easily hide differences between the interfaces of objects at the original and other computers.

```
public abstract class Activity extends JFrame implements
                                           ActionListener {

        private static final long serialVersionUID = 1L;

        public Activity() {
             System.out.println("Definition Software Component");
        }
          public abstract void create(CompURL url);

          public abstract void duplicate(boolean flag);

          public abstract void relocate(CompAddress address,
                                   InvocationMethod invoMethod);

          public abstract void terminate();
}
```

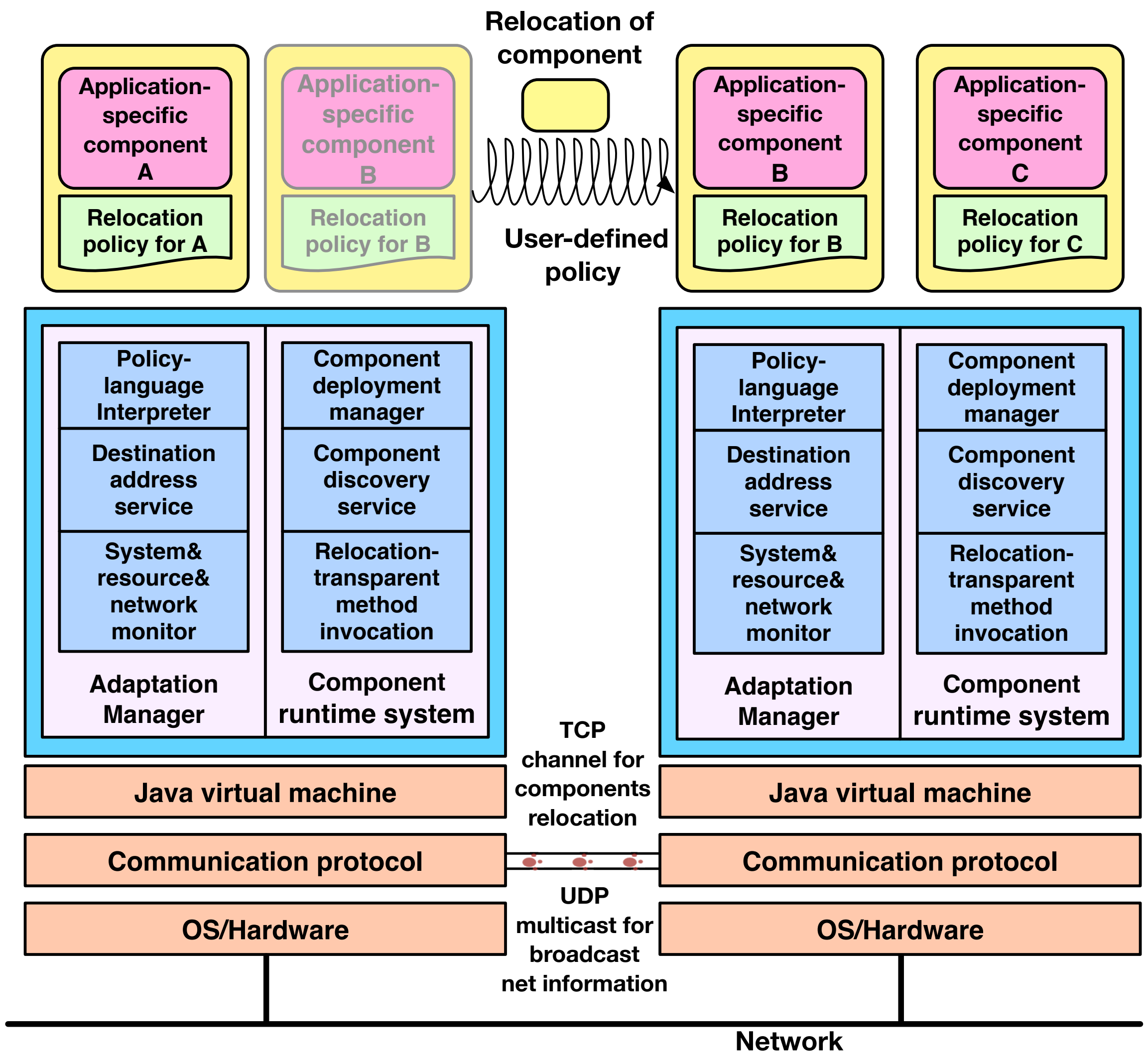

Relocation of component
Application-specific component A
Relocation policy for A
Application-specific component B
Relocation policy for B
User-defined policy
Application-specific component B
Relocation policy for B
Application-specific component C
Relocation policy for C
Policy-language Interpreter
Destination address service
System& resource& network monitor
Adaptation Manager
Component deployment manager
Component discovery service
Relocation-transparent method invocation
Component runtime system
Policy-language Interpreter
Destination address service
System& resource& network monitor
Adaptation Manager
Component deployment manager
Component discovery service
Relocation-transparent method invocation
Component runtime system
TCP channel for components relocation
Java virtual machine
Java virtual machine
Communication protocol
Communication protocol
UDP multicast for broadcast net information
OS/Hardware
OS/Hardware
Network

Figure 4.2: Mimosa system architecture

Each runtime system can exchange components with other runtime systems through a TCP channel by using distributed objects technology. When a component is transferred over the network, not only the software component but also its state is transformed into a bitstream by using Java's object serialization package and then the bitstream is transferred to the destination. The component runtime system on the receiving side receives and unmarshals the bitstream.

Even after components have been deployed at destinations, their methods should still be able to be invoked from other components, which are running at local or remote computers. The runtime systems exchange information about components that visit them with one another in a peer-to-peer manner to trace the locations of components.

4.4.2.1 Component Marshaling

A running program of data items, such as objects and values, cannot be directly transmitted over a network. Before migrating, both of them should be transformed into external data representations, i.e., binary or text forms. (Figure 4.3). Marshaling is the process of collecting data items and assembling them into a form suitable for transmission in a message. Unmarshaling is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

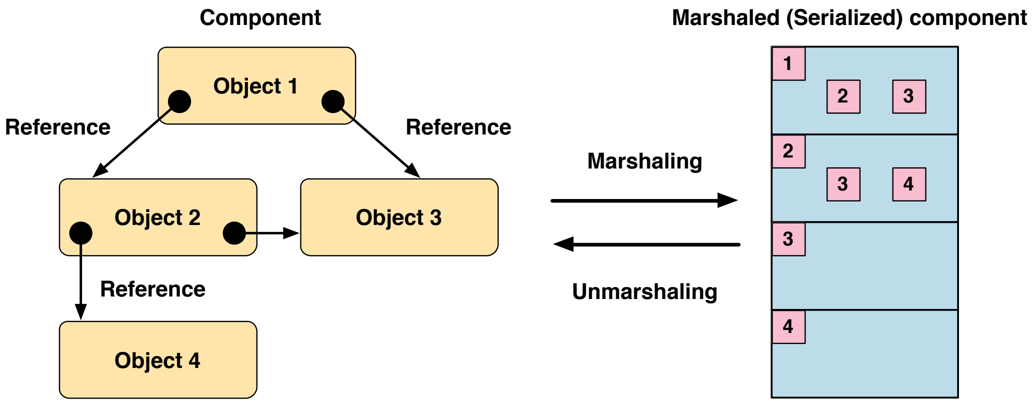


Figure 4.3: Marshaling of component

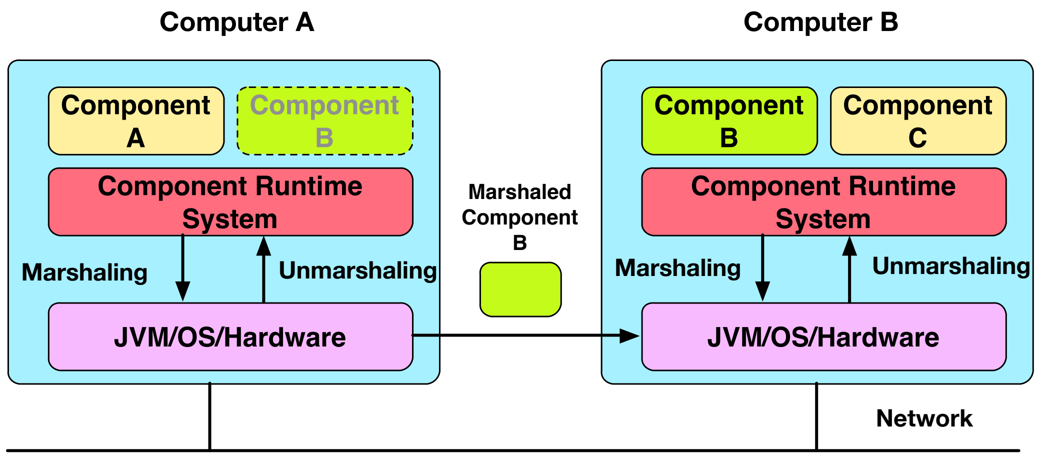


Figure 4.4: Relocation of component between computers

[2] The marshaling and unmarshaling processes are carried out by runtime systems

[2]Note that marshaling and serialization are often used without any distinction between them.

in the Mimosa systems. The runtime system at the left (on the sender-side computer) of Figure 4.4 marshals a software component to transmit it to a destination through a communication channel or message and then the runtime system at the right (on the receiver-side computer) of Figure 4.4 receives the data and unmarshals the component.

4.4.2.2 Dynamic methods Invocation

Each component has to be an instance of a subclass of abstract class Activity. The component class consists of certain methods invoked in the life-cycle of a software component. I also developed a new "ClassLoader" for invoking transmitted software components through their objects and class files. It is because the default class loader does not support invoking their paths of the user implemented software components. When the software components are relocated to destination computers, I use the package/class/method names of the software components to allow the components to invoke themselves by using Java's reflect package.

4.4.3 Adaptation Manager

Figure 4.5 explains the policy-based relocation of components. Each adaptation manager periodically advertises its address to the others through UDP multicasting, and these computers then return their addresses and capabilities to the computer through a TCP channel. It evaluates the conditions for its storing policies, when the external system detects changes in environmental conditions, e.g., user requirements and resource availability.

- When a component has an attraction policy for another component, if communications between the former and latter reach more than a specified number, the policy instructs the former to migrate to the current computer of the latter.
- When a component has a spreading policy for another component, if the current computer of the latter does not have the same or compatible components, the policy makes a copy of the former copy of it and instructs the copy to migrate to the current computer of the latter.
- When a component has a repulsion policy for another component, if there are a specified number or more of the same or compatible components of the former at

The latter is a process of flattening and converting an object, including its referring objects, into a sequence of bytes to be sent across networks or saved on a disk.

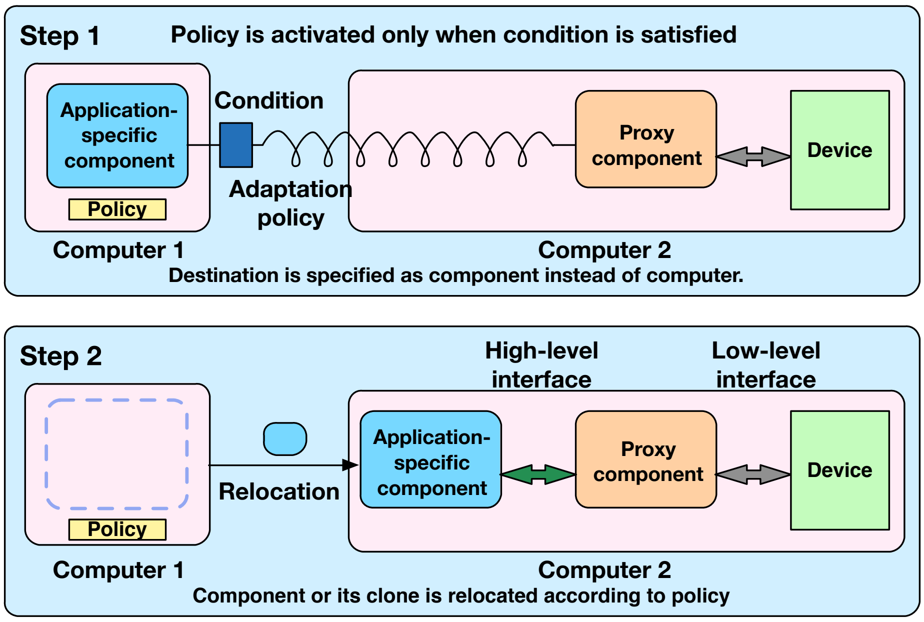

Step 1
Policy is activated only when condition is satisfied
Application-specific component
Policy
Condition
Adaptation policy
Proxy component
Device
Computer 1
Computer 2
Destination is specified as component instead of computer.
Step 2
High-level interface
Low-level interface
Policy
Relocation
Application-specific component
Proxy component
Device
Computer 1
Computer 2
Component or its clone is relocated according to policy

Figure 4.5: Policy for relocation

the current computer, the policy instructs the former to migrate to the current computer of the latter.

- When a component has an evaporation policy, if there are a specified number or more of the same or compatible components of the former at the current computer it terminates.
- Time-to-live policy often requires combination with the above four policy to be effective. When the user-specified time is up, it will complete its next mission.

Each policy has a condition part, which is written in a first-order predicate logic-like notation. The part specifies when its policy should be activated. For example, when the condition of the attraction policy is the movement of the co-partner component, the target component follows the movement of the co-partner. This is useful when the two components need to frequently interact and/or require heavy data-transfer on each interaction yet they cannot be programmed inside a single component. The number of components are reduced when the condition of the evaporation policy is that of the target component and specified components are at the same computer.

4.5 Benefits

Our middleware can provide a variety of benefits for different users. Because my middleware system can be self-adaptive for various changes between existing distributed systems and applications. Therefore,

- Application developers do not need to develop or modify their complex adaptation programs inside of software components. I can effectively reduce the burden and improve development the speed for application developers.
- System developers also do not need to develop adaptation programs. Therefore, my middleware can effectively coordinate the adaptive behaviors between distributed systems and applications.
- Policy definers can easily and simply define what is expected from adaptive conditions and adaptive behaviors through our language.

On the other hand, my language is an interpreted language, it does not generate intermediate code to help users to execute their adaptation programs efficiently.

4.6 Discussion

An adaptive middleware system called Mimosa is proposed. It can relocate software components between computers as a basic mechanism. As software components are relocated through networks, **Security** was one of the most important issues in my research.

Many researchers have explored mechanisms to enhance security for adaptive systems. However, there are two main security problems: the protection of hosts from malicious relocation and the protection of relocation of components from malicious hosts. It is difficult to verify whether incoming components are malicious or not. Nevertheless, there are two solutions to protecting hosts from malicious component relocation. The first is to provide access-control mechanisms, e.g., Java's security manager. They explicitly specify the permission of components and restrict any component behaviors that are beyond their permissions. The second is to provide authentication mechanisms by using digital signatures or authentication systems. They explicitly permit runtime systems to only receive components that have been authenticated, have been sent from authenticated computers, or that have originated from authenticated computers.

Although these technologies have not been cited in the current design, I intend to improve the Mimosa middleware in follow-up work.

4.7 Summary

This chapter introduced the system model by using user-defined policies. It also described the requirements and the design of the Mimosa middleware system. The new approach has three main advantages:

- Because parts of software components are relocated from one computer to others, the same state components can still be executed in different computers to provide services to users.

- Because the size of components[3] is smaller than that of large data, the cost of network translation can be reduced, and more computational resources can be saved for adaptations.

- System architectures can be dynamically changed because distributed applications are formed by different software components that contain server and client functions through the relocation of software components.

It is hoped that the proposed Mimosa system can meet the needs of more developers in the future, and become a universal handling approach for adaptive distributed systems.

[3]In my current system, the size of components is 10 KB.

Chapter 5

A Policy-based Language for Specifying Adaptation

Distributed systems often need to support availability, adaptability, and scalability because they are often used for mission-critical purposes. However, adaptations change distributed systems themselves so that the adapted systems may not satisfy the requirements of the systems. Adaptations may result in uncertainty in the sense that their effects may not be able to be predicted beforehand. They need to have the degree of uncertainty resulting from their adaptations reduced. Distributed systems execute multiple applications whose adaptations are simultaneously different. Although individual adaptations may be appropriate, they may cause serious problems, e.g., *conflict* and *divergence*, in distributed systems. For instance, when two or more adaptations are activated according to their conditions, one adaptation denies the effects of one or more other activated adaptations. Adaptations may be activated infinitely, if their conditions are satisfied after they have been performed. This chapter introduces a language to define requirements as policies for specifying adaptations.

5.1 Overview

Many researchers have proposed several language [20] [90] [19] [16] [98] [1] to define requirements for adaptation on distributed systems. This section outlines the proposed language based on the general purpose middleware *Mimosa* for adaptive distributed systems [82]. The proposed language consists of *conditions* and *actions* for each policy. Both of them are defined based on a theoretical foundation to verify the validation of adaptations. The former is written in a *first-order predicate logic-like notation* , where predicates reflect information about the system and applications. The latter represents the deployment and duplication of components in the proposed adaptation instead of any application-specific behaviors, including communications and state transition, of the components. The foundation presented in this approach is constructed as a process calculus to specify adaptations [62][81]. This enables the effects of adaptations to be analyzed and also specified, e.g., where and what functions are provided from after adaptation. Mimosa middleware system can specify and interpret policies for adaptations on the basis of the foundation outside software components.

5.2 Language Requirement

Many researchers presented a numbers of policy languages for distributed systems. For instance, Seamons proposed a clearly requirements [77] for their language. However, unlike such researches, I proposed the policy language is designed for specified software components to adapt to changes in distributed systems through relocation software components between computers.

5.2.1 Problem statements

Before describing the requirements of the proposed policy language, problems in adaptive distributed systems are stated independently of the proposed adaptations.

- Since the special nature of distributed systems, only within a single computer-implemented adaptation is limited, therefore I think that the policy language should considered more adaptive behaviors between computers.
- Exisiting adaptations technology often result in uncertainty in the sense that their effects may not be able to be predicted beforehand. This is serious problem

in mission-critical distributed systems, therefore, adaptation should be defined by the users customize to satisfy their assigned requirements.

- There may be no guarantees as to whether activated adaptations can be stopped. An adaptation often creates the conditions to satisfy other adaptations. Chain reaction adaptations may occur or certain adaptations may be activated infinitely.

- Distributed systems are used for multiple applications, which may require adaptations to the systems. Therefore, more than one adaption may have conflicts, although individual adaptations would be unaffected.

5.2.2 Requirements for language

Adaptations in distributed systems need to be managed with partial knowledge about other computers, so that these four problems above in adaptations between different computers cannot be accurately detected at runtime. Therefore, it is believed that adaptations in distributed systems should be verified beforehand rather than checked at runtime. Furthermore, distributed applications should often be defined independently of adaptations, and vice versa, as much as possible. This is because most adaptations in distributed systems depend on the underlying systems or user's requirements rather than the applications themselves.

The proposed policy language should satisfy the requirements as follows:

- Monotonicity. For trust negotiation of proposed language, it should be monotonic. This is because if two parties successfully negotiate trust, then that same negotiation should also succeed.

- Independence. The proposed language should abstracted away from the underlying systems based on the concept of *separation of concerns*, the language enables adaptations to be defined independently of applications.

- No-compiling. The proposed language need to control the software components which running on the proposed Mimosa middleware system, therefore the user-defined policies in needs of immediate execution without having to generate intermediate code.

- Reusability. User-defined policies should be reused to satisfy the dynamic changes between distributed systems themselves and applications.

- Scalability. The proposed language should support the scalability to users so that satisfy their requirements as much as possible.

5.2.3 Advantages

This approach focuses on the specifications of the proposed adaptations rather than other existing adaptations for distributed systems. Policies to relocate software components as a basic adaptation mechanism are introduced. It provide users to define their own functions outside of software components. If functions inside software components are adapted, other components, which communicate with the adapted ones, may have serious problems, e.g., downed systems and security leaks. In addition, through the software components between computers, the architecture of distributed system can be dyanimcally changed (in chapter 3). Furthermore, relocation of software components does not lose the potential functions of components. This is because the proposed approach only migrate the execution of software components. It may seem to be simple but it makes their applications resilient without losing availability, adaptability, or reliability.

5.3 Approach: Policy-based Specification Language

This section defines the proposed language for specifying adaptations in a process-calculus-style [62]. I used several notations to better describe the language, as summarized in Table 5.1. The proposed middleware enabled users to specify user-defined policies for adaptations by means of the expressions. Each expression contained conditions for triggering adaptation and the destinations of relocation. Software components were relocated to the specified destinations when the conditions were satisfied.

Table 5.1: Notations.

Notation	Meaning
current	Current node
$\mathcal{L} = \{\ell_1, \ell_2, \ell_3, \cdots\}$	Location names
$\mathcal{X} = \{x_1, x_2, x_3, \cdots\}$	Location variable names
$\mathcal{C} = \{A, B, \cdots\}$	The identifiers of components with conditions

5.3.1 Policy Expression

$\mathcal{D} = \{D, D_1, D_2\}$ is defined as a set of located process expressions, which is *the smallest set containing the expressions in* Table 5.2. In the expressions, C represents a condition expression, and *E ; 0* is often abbreviated as E. The τ is an action invoked as a callback function, and ϵ indicates that there are no components.

Table 5.2: Policy expressions.

Expression		Meaning
D, D_1, D_2 ::=	ℓ[E| P]	Located component
|	$D_1 \parallel D_2$	Distributed component
E, E_1, E_2 ::=	C **then** G	Conditional action
|	E_1 ; E_2	Sequential composition
|	$E_1 + E_2$	Alternative selection
|	**0**	Termination
G ::=	moveTo(x)	Migration
|	copyTo(x)	Duplication & migration
|	remove	Elimination
|	τ	Internal execution
P, P_1, P_2 ::=	P_1,P_2	Composition
|	A	Component
|	ϵ	No component

The proposed language describes the intuitive meanings of several constructors in the language. For instance,

- ℓ_1**[E | A]** means that component A located at ℓ_1 is executed as an expression specified as E.
- $D_1 \parallel D_2$ represents distributed components D_1 and D_2 executed in parallel.
- E_1 **;** E_2 is executed as E_2 after E_1, and $E_1 + E_2$ behaves as E_1 or E_2.
- ℓ_1**[C then moveTo(**ℓ_2**) | A]** means that if condition C is true, component A located at ℓ_1 is relocated to ℓ_2, where moveTo(ℓ_2) represents the relocation to ℓ_2.
- ℓ_1**[C then remove | A]** means that if condition C is true, component A terminates.
- ℓ[C_1 **then copyTo(**ℓ_2**)** + C_2 **then callback ; remove | A, B**] means that if condition C_1 is true, two components A and B are copied, and then the copies

are deployed at ℓ_2. Otherwise, if condition C_2 is true, the policy executes a callback function in A and B and then terminates A and B.

5.3.2 Policy Conditional Functions

The policy conditional functions of the proposed language is defined as first-order logic predicates. The set C of conditions is the smallest set containing the following expressions:

$$C, C_1, C_2 := \phi \mid \neg \mathrm{C} \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid true \mid false,$$

where ϕ is a logical predicate symbol and returns either true or false with zero or more parameters.

The ϕ is a user-defined function in components or a system's built-in function. The former is provided as an application-specific method from the component that its policy is assigned to. The current implementation provides several built-in functions as:

- exist(A,ℓ) (exist : $\mathcal{P} \times \mathcal{L} \rightarrow$ true or false) returns *true* if the same or compatible component(s) of A exist at location ℓ; otherwise, it returns *false*.
- delay(time) (delay : $\mathcal{T} \rightarrow$ true or false) blocks the subsequent executions for the *time* interval and then returns *true*, where $\mathcal{T}$ is an infinite set of relative time values.
- received(m, ℓ, A) (requested : $\mathcal{M} \times \mathcal{N} \times \mathcal{L} \rightarrow$ true or false) returns *true* if the component that the policy is assigned to receives a message labelled as m from component A.
- receivedFrom(n, t) (receivedFrom : $\mathcal{N} \times \mathcal{T} \rightarrow$ true or false) returns *true* if it received messages from more than a number of of computers, specified as n, within a certain duration specified as t, otherwise, it return *false*.
- largerSize(d, s) (largerSize : $\mathcal{D} \times \mathcal{S} \rightarrow$ true or false) return ture if the received data size is bigger than program size, otherwise, it return *false*.
- highThroughput(d, b) (throughput : $\mathcal{D} \times \mathcal{B} \rightarrow$ true or false) return ture if the received data size is divided by a bandwidth greater than 1, otherwise, it return *false*.

- detect(target) (detect : $\mathcal{T} \rightarrow$ true or false) returns *true* when the movement of a target is discovered, otherwise, it return *false*.

User-defined functions are implemented inside components or the runtime system. User-defined functions defined in components can be accessed inside the components.

5.3.3 Control Structures

There are two types of control structures, branches and loops, can be used in the proposed language, it can be repeated a sequence of statements over and over or to choose among two or more possible courses of action.

- An *if* statement notifies the computer to take one of two alternative courses of action, depending on whether the value of a given boolean-valued expression is true or false.
- An *if-else* statement notifies the computer when they executes an if statement, it evaluates the boolean expression. If the value is true, the computer executes the first statement and skips the statement that follows the "else".
- A *while* loop will repeat a statement over and over, but only so long as a specified condition remains true.
- An *for-each* statement is also a loop control statements, it is used to repeat a given statement over and over when the condition is true.

5.4 Design of Policy Formats

Figure 5.1 presents typical adaptation policies. It can help users easily defining their adaptations through the proposed policy language.

5.4.1 Attraction Policy

The following policy (Figure 5.1) is assigned to component A in the computer. If a computer, called *another*, has the same or compatible component of component A, the policy instructs A to migrate to *another*, and then it follows E.

$$\ell[\text{exist(A, another) then moveTo(another) ; E} \mid \text{A}]$$

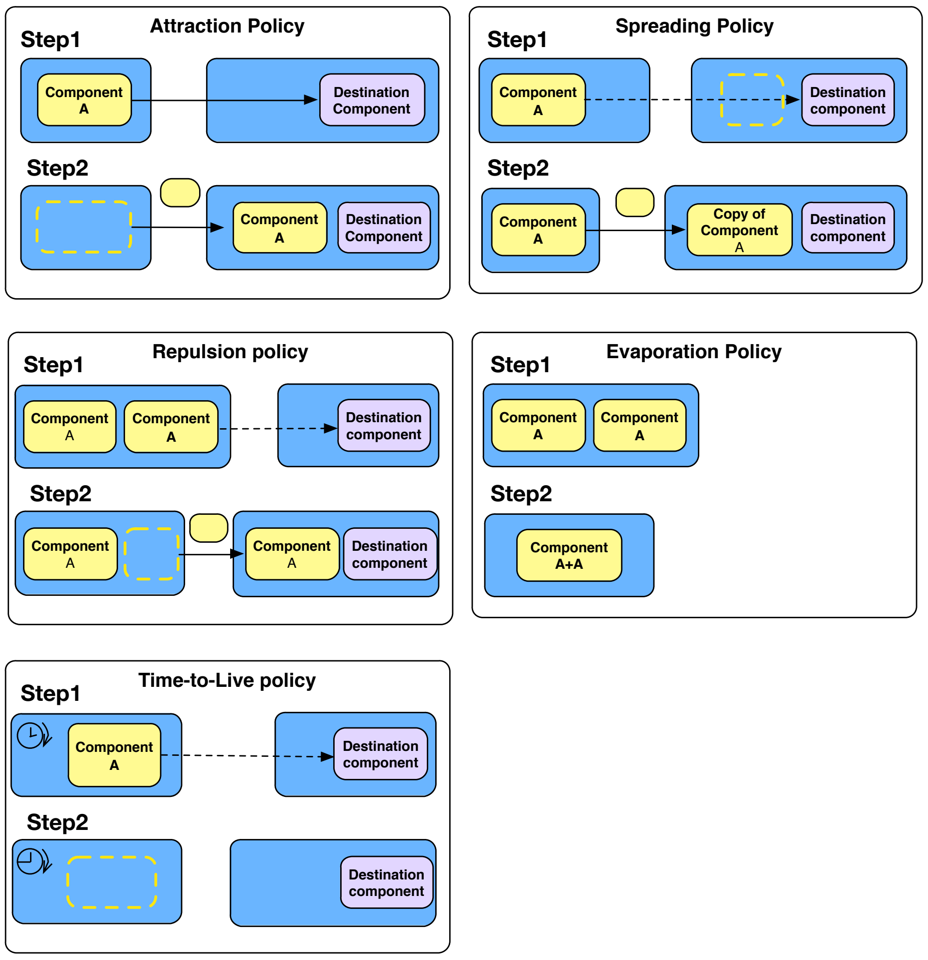


Figure 5.1: Policy formats

5.4.2 Spreading Policy

The following policy (Figure 5.1) is assigned to component A. If a computer, called *another*, does not have the same component A, the policy makes a copy of component A and deploys the copy at *another* computer, and then it follows E.

$$\text{current}[\neg\text{exist}(\text{A, another}) \text{ then copyTo(another) ; E} \mid \text{A}]$$

5.4.3 Repulsion Policy

The following policy (Figure 5.1) is assigned to component A. If the same or compatible components of A exist, the policy instructs component A to migrate to another computer, called *another*, and then it follows E.

$$\text{current}[\text{exist}(\text{A, another}) \text{ then moveTo(another) ; E} \mid \text{A}]$$

5.4.4 Evaporation Policy

The following policy (Figure 5.1) is assigned to component A. If the current computer has one or more components A, the policy eliminates component A there.

$$\text{current}[\text{exist}(\text{A}, \text{another}) \text{ then remove } ; 0 \mid \text{A}]$$

5.4.5 Time-To-Live Policy

After a certain time has passed, the following policy (Figure 5.1) terminates component A.

$$\ell[\text{delay}(t) \text{ then remove } ; 0 \mid \text{A}]$$

Remark Users can individually, or with a combination of these policies, define the changes for adaptation on distributed systems according to these five kinds of policy formats.

5.5 Policy Conflict and Divergence

According to the characteristics of distributed systems described in Subsections 5.1 and 5.2, multiple adaptations may be in conflict with others. In addition, inappropriate definition within a policy, adaptive conditions may lead to the emergence of divergences. McEvoy et al. [60], and Lupu and Sloman [55] have explored these problems [60] [55] in distributed systems. Unlike their research, I designed language can formally analyze several properties of adaptation through components that are described.

5.5.1 Conflict

Before describing the policy conflict, I need to analyze what a conflict is. Policy conflict means that there are inconsistent and even mutually exclusive phenomena during the interiorization of policies. Conflict was divided into two patterns in this research. The first was in a location computer, the adaptation of a component is described as:

$$\ell[C_1 \text{ then } E_1 + ... + C_n \text{ then } E_n \mid \text{A}]$$

this is a *local conflict*, if more than two of C_1,...C_n are positive at a location. The second was a conflict between computers, if and only if more than two of C_1,...C_n can be positive at every location, this was a *global conflict*.

Suppose that the same software component A has two conditions as the figure 5.2 shows. For instance, the condition 1 is a function that determine the network status, e.g., 4G,3G,Wi-Fi, and condition 2 is a function that determine delay time. When both of the two conditions are positive at location computer, local conflict may occur. Because the proposed policy language focus on static analysis, therefore, the solution of local conflicts can be detected by evaluating the overlaps of conditions of two or more policies at certain locations. According to the first-order predicate logic, the proposal language can notify users that the user-defined conditions may conflict and need to be modified.

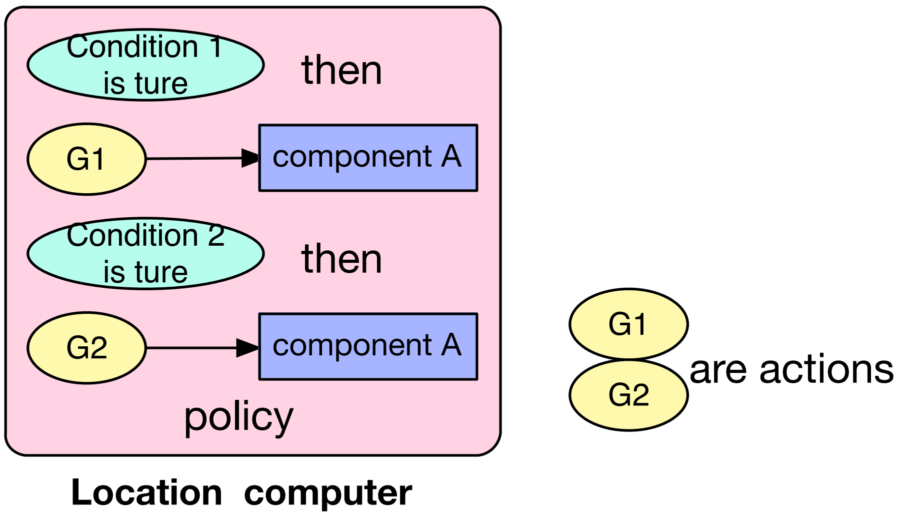


Figure 5.2: Local conflict

Global conflicts may be occured in two cases (Figure 5.3).

- Case 1: For the same software component which is executing at a location computer, two or more user-defined policies may lead to global conflict. For instance, condition 1 is the delay (time) and it return ture, and the condition 2 is net status and it return ture, in this case, two conditions are true for one software component, therefore, global conflicts may occurred. Global conflicts can be detected by the overlaps of conditions of two or more policies at location computer.

- Case 2: According to the grammar of my policy language, policies need to be executed at each computer. Therefore, even if the conditions of policies are same and positive, the software component are different, so that this case's global conflitc never be occurred.

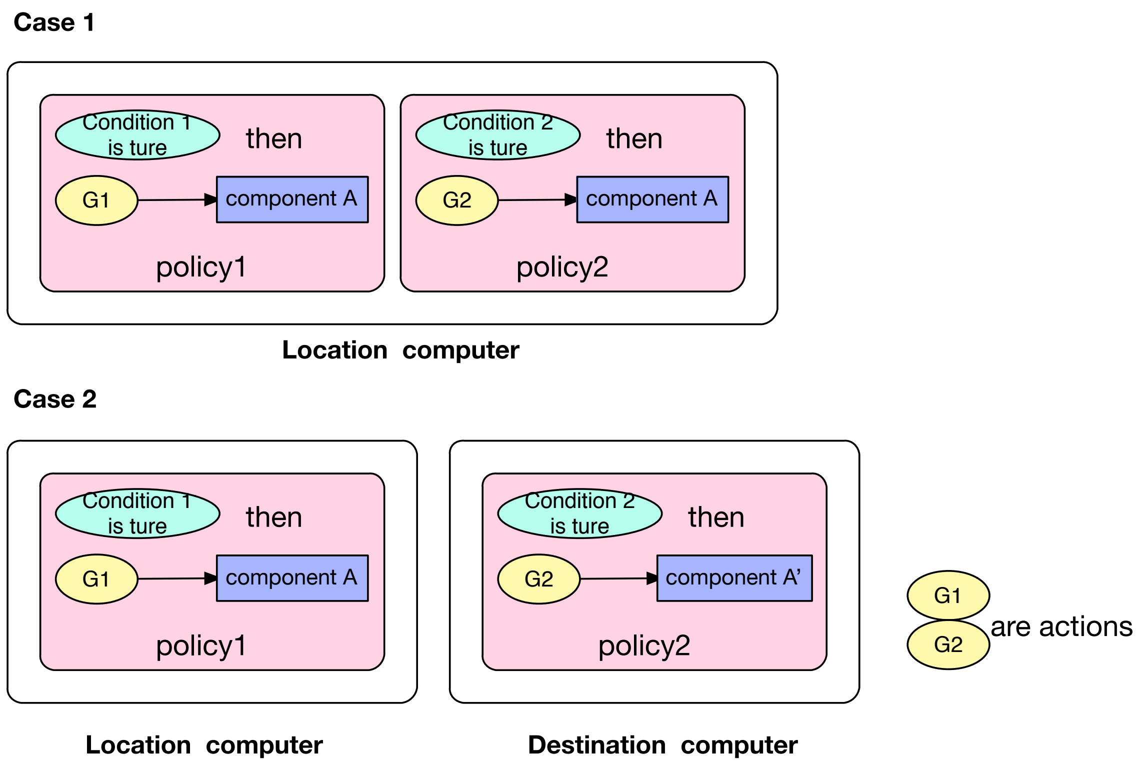


Figure 5.3: Global conflict

Currently, the detection method of such global conflict is still restricted at conditions of arbitrary locations in static-level, once such conflict was found, the user needs to modify or redefine their adaptation policies.

Remark There may no guarantee that activated adaptations can be stopped. Suppose an adaptation is invoked and executed at a change because the change satisfies the condition of the adaptation. After the adaptation is executed, the condition is still satisfied so that the adaptation is invoked again. When the condition of the effectiveness of an adaptation is inappropriate, the adaptation may go into an infinite loop; it becomes divergent. I will disscuss it in next section.

5.5.2 Divergence

Before describing the policy divergence, I need to analyze what divergence is. Policy divergence means that when the conditions of a policy are still satisfied, the same actions of the software components are invoked for non-stop adaptation. The adaptation of divergence that occurs in a component can be described as:

$$\ell[\text{C then E} \mid \text{A}]$$

Here, there is local divergence, if C is still positive after executing E at a location. Figure 5.4 shows this situation. It is very difficult to find divergent at static conditions, however, at present, the proposal can

- Confirm whether an adaptation is divergent or not by evaluating C condition at every step of E, because adaptations are written in the language can be interpreted as their possible itineraries.
- Although this practice restricted to the description of C, for instance, the C condition is just expressed as ture with number of times. Currently implementation provide a discovery method by comparing the user-defined number of times with the actual number of times. Once the actual number of times exceeds a user-defined number of times, it will be treated a divergence.

If a component is divergent, its adaptation may be endless. Such adaptation should be modified so that it is not divergent[4].

Remark Currently, the proposed language to discovery divergence is restricted, this is because the proposal approach focus on static condition, and the solution of divergence confined to detect the adaptation conditions before executing policies. Although I added to a comparing method by counting the number of times for loop, however this proposal still has the following inabilities:

- It can not be applied the conditions are positive forever.
- The proposed language do not support detecting divergence in destination in current implementation.

[4]Please note that the proposed language do not support detecting divergence in destination by current implementation

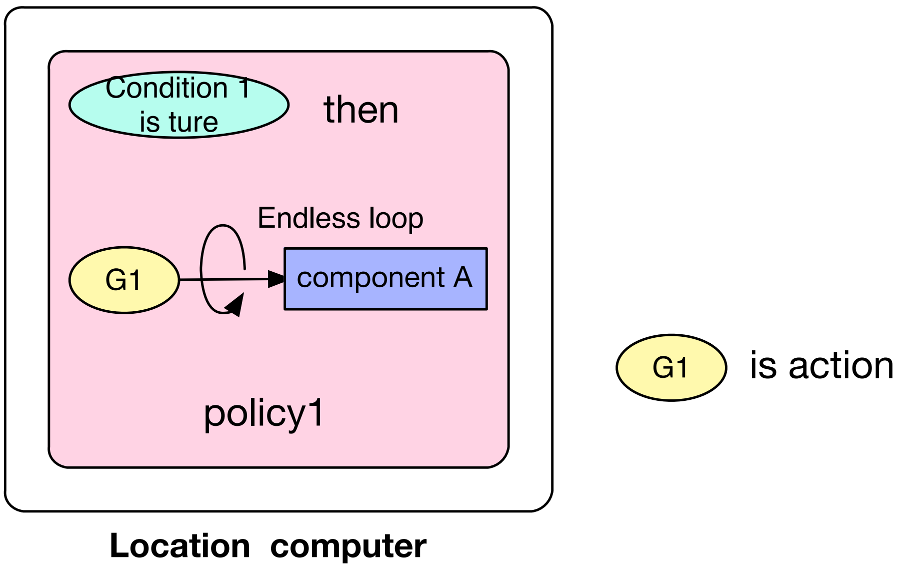


Figure 5.4: Divergence

5.6 Discussion

I designed a policy-based language for adaptation on distributed systems. Users can easily define their policies by using the five policy formats that are provided. All policies can be immediately defined, but can also be called from their policy database. The proposed language procive an interpreter for executing user-defined policies on Mimosa middleware system, it also can satisfy the defined requirements, such as monitinicity, independence, no-compiling, reusability, and scalability in section 5.2.

Currently I design language only provide some basic variables, and functions, however, in the future, with the further development of new applications, I need to expand the proposed policy language. In addition, although the language can analyze parts of the policy conflicts and divergence, but it is still limited by analysis the adaptive conditions individually. Therefore, more adaptation policies need to be defined and evaluated, and new algorithm for founding conflicts and divergences need to be imported in future.

5.7 Summary

This chapter outlined the theoretical foundation of policy-based language for specifying adaptive distributed systems. The current foundation could only specify adaptations in the proposed middleware system, where it introduced the relocation of software components whose distributed applications occurred between computers as a basic mechanism for adaptation on distributed systems. It enabled adaptations to be analyzed and rationalized based on a process-calculus-based formal system.

Five policy formats were explored in Subsection 5.4: Attraction policy, Spreading policy, Repulsion policy, Evaporation policy and Time-to-live policy. If users defined these once, these policies could be reused when the same conditions were satisfied. I provided a policy database to conserve policies. Therefore, when the software components were relocated to destination computers, the policies that were saved in the destination policiesf database could be recalled through the names of component's methods. When conflict and divergence occurred, the proposed policy language analyzed the reasons, notified users, and then let them make decisions for adaptation.

The proposed policy-based language especially strengthened the relocation of software components, unlike that in existing research. I just need developers to define the name of the destination for relocation, and the language will then automatically explore its IP address through the Mimosa middleware system. Then, when the conditions of policies are satisfied, the software components will automatically relocate themselves to the destination for adaptation. When the software components migrate to destination computers, the proposed dynamic method invocation mechanism will recall their methods to restart the tasks for execting. In addition, some condition expressions were provided. Such as branch statements, if statement, if-else statement, and loop statements of while, for-each for users to define the conditions and actions of their policies. Especially, by using the for-each statement, and loop the action to easily spread components to various computers.

Chapter 6

Adaptive Applications on Distributed Systems

Various changes between distributed systems and applications could be dynamically adapted based on two technologies, i.e., the *Mimosa* middleware system and policy-based language.

This chapter introduces five typical applications for adaptation on a distributed system. I. *Remote Information Retrieval* is a typical example to relocate software components among computers for adaptation. Ⅱ. *Primary-back replication and Chain replication* are very common approaches to distributed systems. However, both of them have suitable and unsuitable cases. When changes occur on distributed systems, they should be adapted. Their architectures can be dynamically changed from one type to another one through the software componentsf relocation. Ⅲ. Because the resources of sensors are limited, *sensor networks* also need adaptation against changes. By using the proposed approaches, the software components can make copies, and spread/relocate the copies to other sensors for dynamic adaptation. Ⅳ. *Model-View-Control (MVC) applications* are one of the best examples of disaggregated computing. Various changes can be dynamically adapted by relocating their components between servers and disaggregated computers. V. Approaches to a *publish/subscribe system* were used in the last application. Brokers can be self-adaptively connected to cope with failures and guarantee delivery of messages through the relocation of software components on brokers.

6.1 Remote Information Retrieval

Data have recently become huge and have been increasing year by year. However, when the processing is heavier on servers or there is network instability, remotely fetching data through networks will become bottlenecks. When this occurs, the search function of clients should be relocated to the destination computer that is closest to the data side to improve efficiency. Data dispatch can substantially be reduced between local and remote computers by using the proposed approaches. I then just need to return the results to local computers.

Suppose users are trying to search certain text patterns from data located at remote computers like Unix's *grep* command. A typical approach is to fetch files from remote computers and locally find the patterns from all the lines of the files. However, the approach is not efficient if the sum of the volume of its result and the size of a component for searching patterns from data is smaller than the volume of target data. The component should be executed at remote computers that maintain the target data rather than at local computers. However, it is difficult to select where the component is to be executed because the volume of the result may not be known in advance.

The proposed approach can solve this problem by relocating such components from remote computers to local computers and vice versa while they are running. Figure 6.1 presents our system for adaptive remote information retrieval, which consists of client, search, and data access manager components. The first is located on the computer *client side* and the third is on the computer *server side*. The second supports the finding of text lines that match certain patterns provided by the first in text files that it accesses via the third. The *search* component has the following policy with a function called *largerSize()* [5] that compares the sizes of two data or programs.

client_side [largerSize (receivedData, programSize)
then moveTo(server_side) | search_component]

The above policy relocates a search component from the computer *client_side* to the *server_side* when the volume of its middle result is larger than the size of the component; otherwise, it relocates components from remote to local computers. The *largerSize()* function was implemented as a public method of the search component.

[5]Current policy language provides only the most basic functions, but in practical applications, depending on the environment, custom functions should be defined to satify the requirements of users

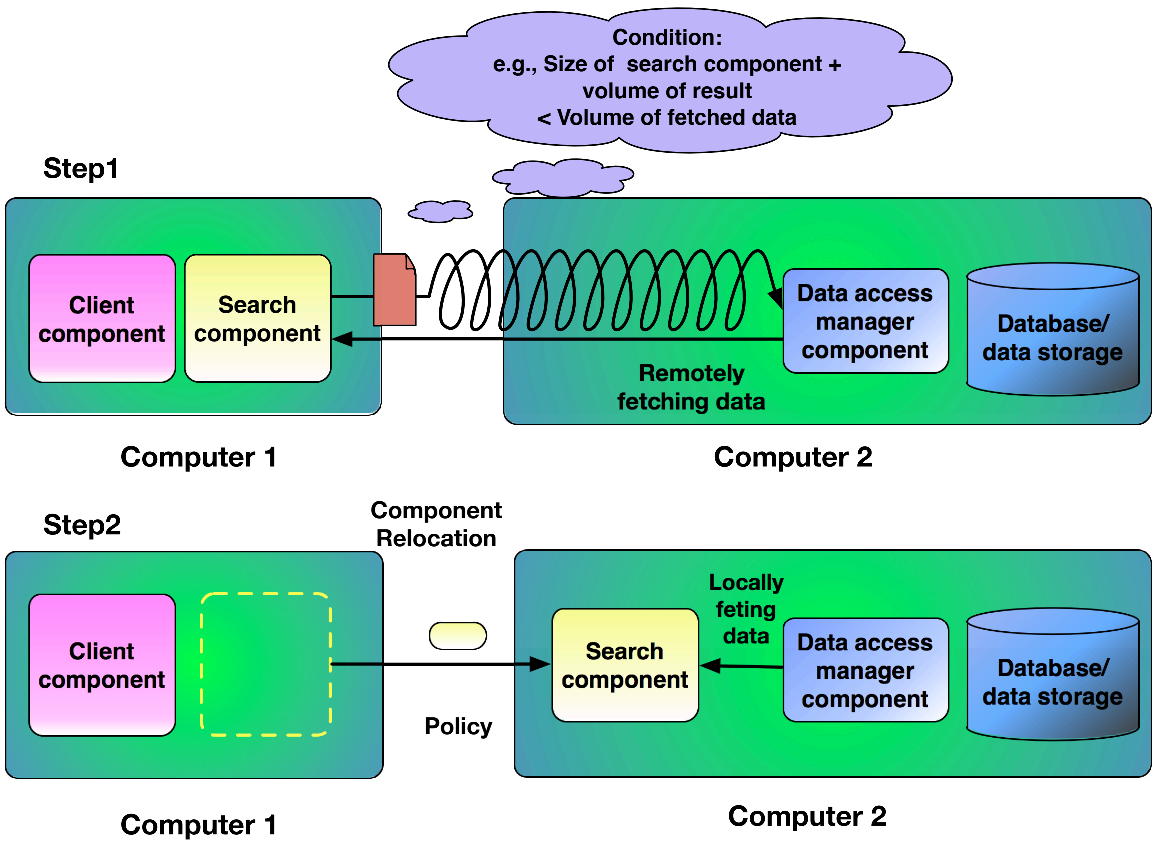


Figure 6.1: Adaptive remote information retrieval

The proposed approach enabled the search component to have its own adaptation policy and manage itself according to the policy independently of these components themselves. It was independent of its underlying systems because the destinations of the component relocations were specified as components instead of computers themselves. The proposed adaptation also can be reused by changing the locations of such destination components.

6.2 Primary-back up and Chain Replication

The second application enabled adaptive management in data replication on multiple computers. Although there have been many data replication approaches on distributed systems, the primary-backup approach is one of the most typical [2]. A client only sends an update request to one designated primary server. The server updates its replica and then forwards the request to one or more backup servers to update the replica and waits for responses from the backup servers before responding to the client. The chain replication approach is a replication protocol to support large-scale storage services, e.g., key-value stores, to achieve high throughput and availability while providing strong consistency guarantees [91]. A client sends update requests to the backup server with the maximum number (head) to update its replica, while forwarding the request to update the replica of the server with the next lowest number until it reaches the server with the minimum number (tail). The tail server responds to the client (As shown in Figure 6.2).

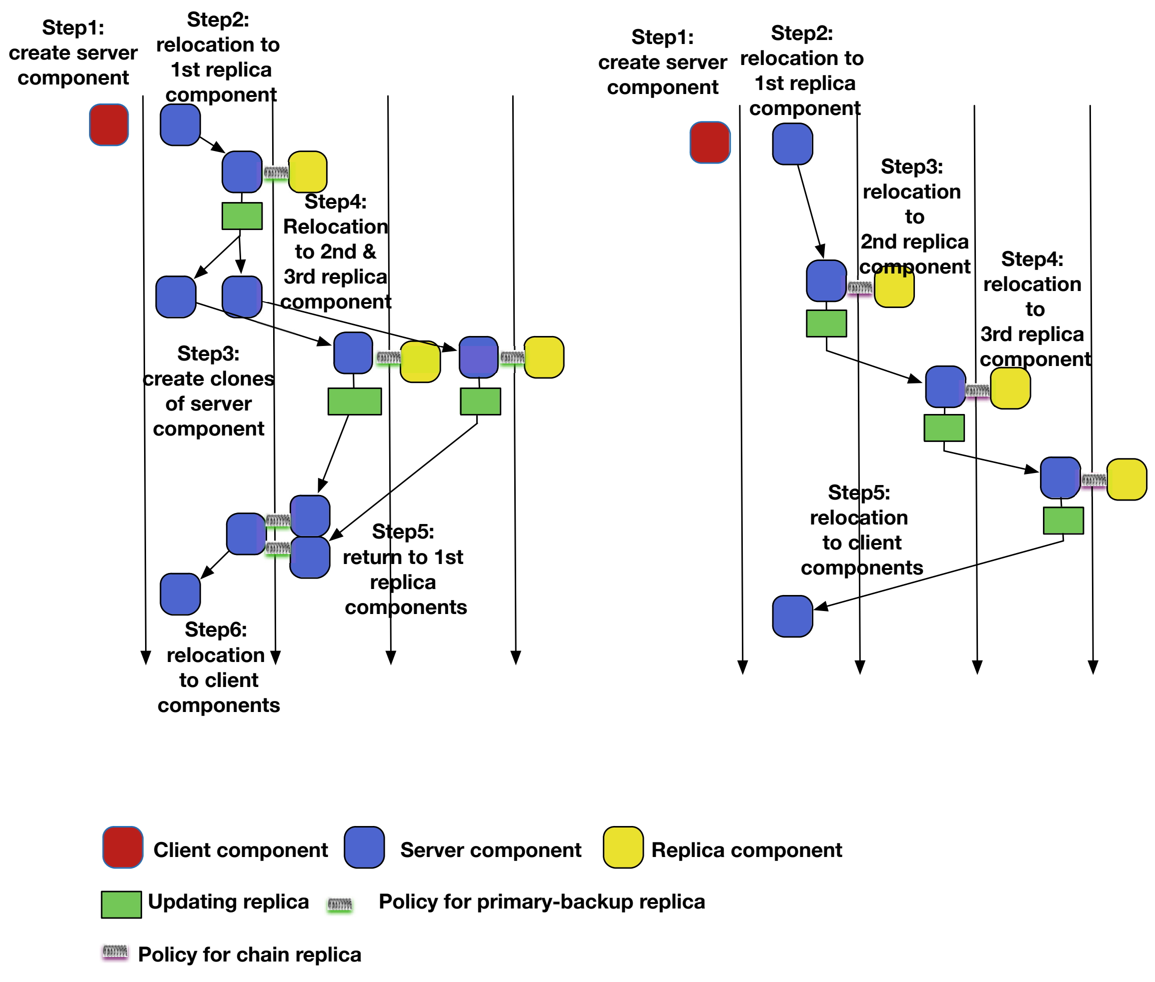

Step1: create server component
Step2: relocation to 1st replica component
Step3: create clones of server component
Step4: Relocation to 2nd & 3rd replica component
Step5: return to 1st replica components
Step6: relocation to client components
Step1: create server component
Step2: relocation to 1st replica component
Step3: relocation to 2nd replica component
Step4: relocation to 3rd replica component
Step5: relocation to client components
Client component
Server component
Replica component
Updating replica
Policy for primary-backup replica
Policy for chain replica

Figure 6.2: Adaptive consistency for data replication

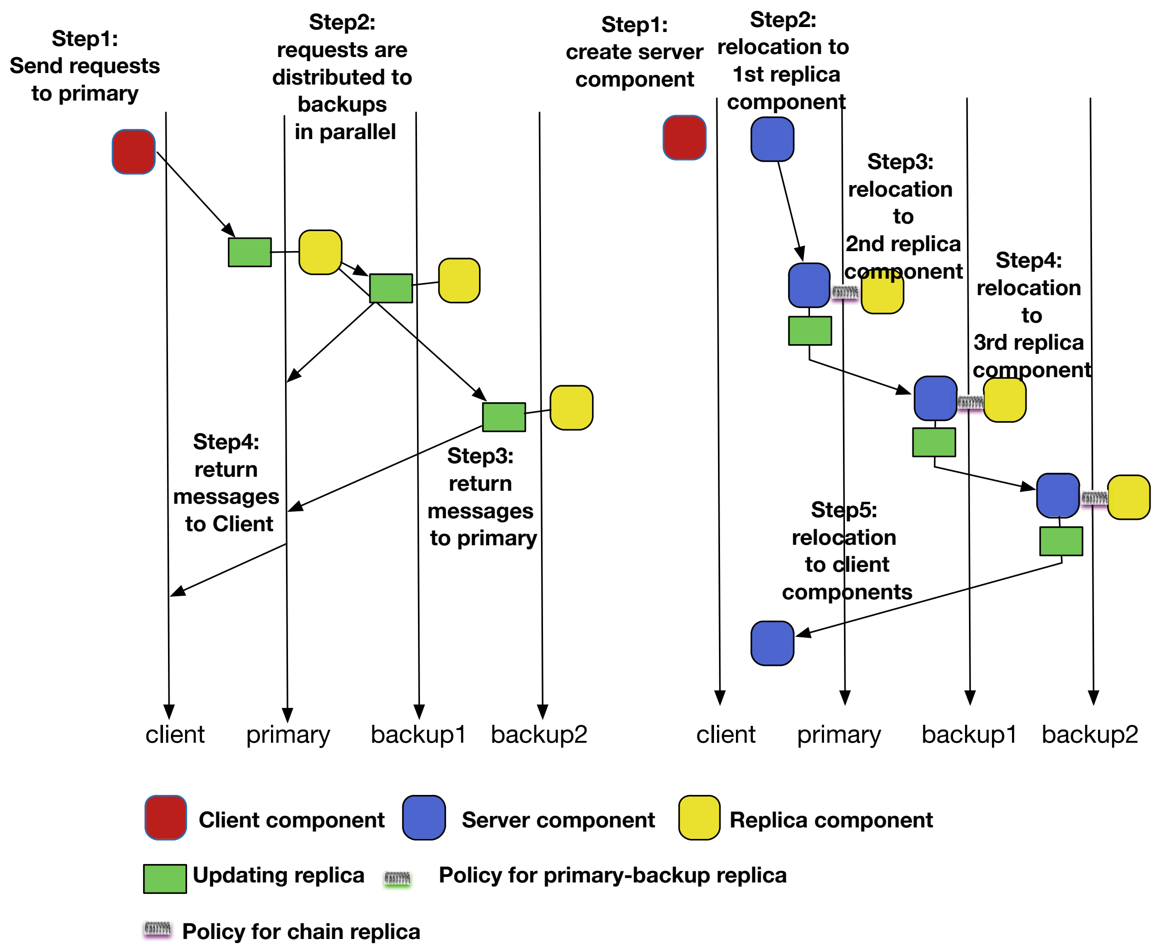
Both these approaches have advantages and disadvantages. For instance, the primary backup approach must wait for acknowledgements from the backups for prior updates, whereas the chain replication approach can execute sequencing requests before prior updates have not been completed. Chain replication is at a disadvantage for reply latency to update requests since it disseminates updates serially, compared to primary-backup, which disseminates updates in parallel. Therefore, the approaches should be selected according to the requirements of applications.

The proposed approach enables the two approaches to be easily transformed into each other. As can be seen from Figure 6.2, the application consists of three kinds of components: a *client*, a *server*, and a *replica manager*. The first receives update requests from the external system. The second has polices for one of either the primary backup or chain replication approaches. The third component is statically deployed at a computer that keeps the replica, is assigned with its own number, and is responsible for updating the replica on behalf of server components.

In the primary-backup approach, the client component first creates a server component after receiving an update request and then deploys at computers with the minimum number and blocks itself until the server returns to it. The server component creates its clones and deploys them at computers that have other replica components. Each server component asks the replica components to update the replica at its destination, and then it returns to the computer that has the parent server component. The parent waits for all its clone components to arrive and then migrates to the computer that has the client component.

In the chain replication, the client component first creates a server component after receiving a update request and waits for the next update request. Next, the server component migrates to the computer that has the replica component with the maximum number. After asking the replica component at the destination, it migrates to the computer that has the replica component with the next lowest number until it reaches the computer that has the replica with the minimum number. The server component migrates to the computer that has the client component.

Figure 6.3 presents the architecture can be changed from Primary-backup to Chain replication, which consists of client, server, replica components, and updataing replica. The following policy is located on the computer, called *primary_side*, whereas the *highThroughput(d, b)* is a function returns *true* if the received data_size is divided by a bandwidth greater than 1. When the *server_component* finished the updata with *replica_component* through a function, called updata_replica(), then the *server_component* migrate from *primary_side* to *backup 1* server. By using this method, the replication approach's architecure can be changed from parallel to chain. In addition, because only a single server (the tail means the backup 2) is involved in processing a query and that processing is never delayed by activity elsewhere in the chain. Therefore, when high throughput in a distributed system, the chain is better than primary-backup replication.



Step1: Send requests to primary
Step2: requests are distributed to backups in parallel
Step3: return messages to primary
Step4: return messages to Client
client
primary
backup1
backup2
Step1: create server component
Step2: relocation to 1st replica component
Step3: relocation to 2nd replica component
Step4: relocation to 3rd replica component
Step5: relocation to client components
client
primary
backup1
backup2
Client component
Server component
Replica component
Updating replica
Policy for primary-backup replica
Policy for chain replica

Figure 6.3: Primary-backup change to Chain

primary_side [**received(** *message, primary_side, server_component* **)** **then**
callback(updata_replica());
highThroughput(*data_size* , *bandwidth*) **then**
moveTo(*backup_server* **)** | **server_component**]

On the other hand, figure 6.4 presents the architecture can be changed from Chain to Primary-backup replication, which also consists of client, server, replica components, and updataing replica. The following policy is located on the computer, called *head_side*, whereas the delay() function is detect the latency. When the server_component finished the updata with *replica_component* through a function, called updata_replica(), then when latency is become high, the *server_component* make copies, then migrate the copies from *head_side* to two computers, one is called *next_side* server and another is called *tail_side*. In this case, primary-backup is better than chain replication, this is because the requests were distributed to backups in parallel. By using this method, the replication approach's architecure can be changed from chain to a parallel.

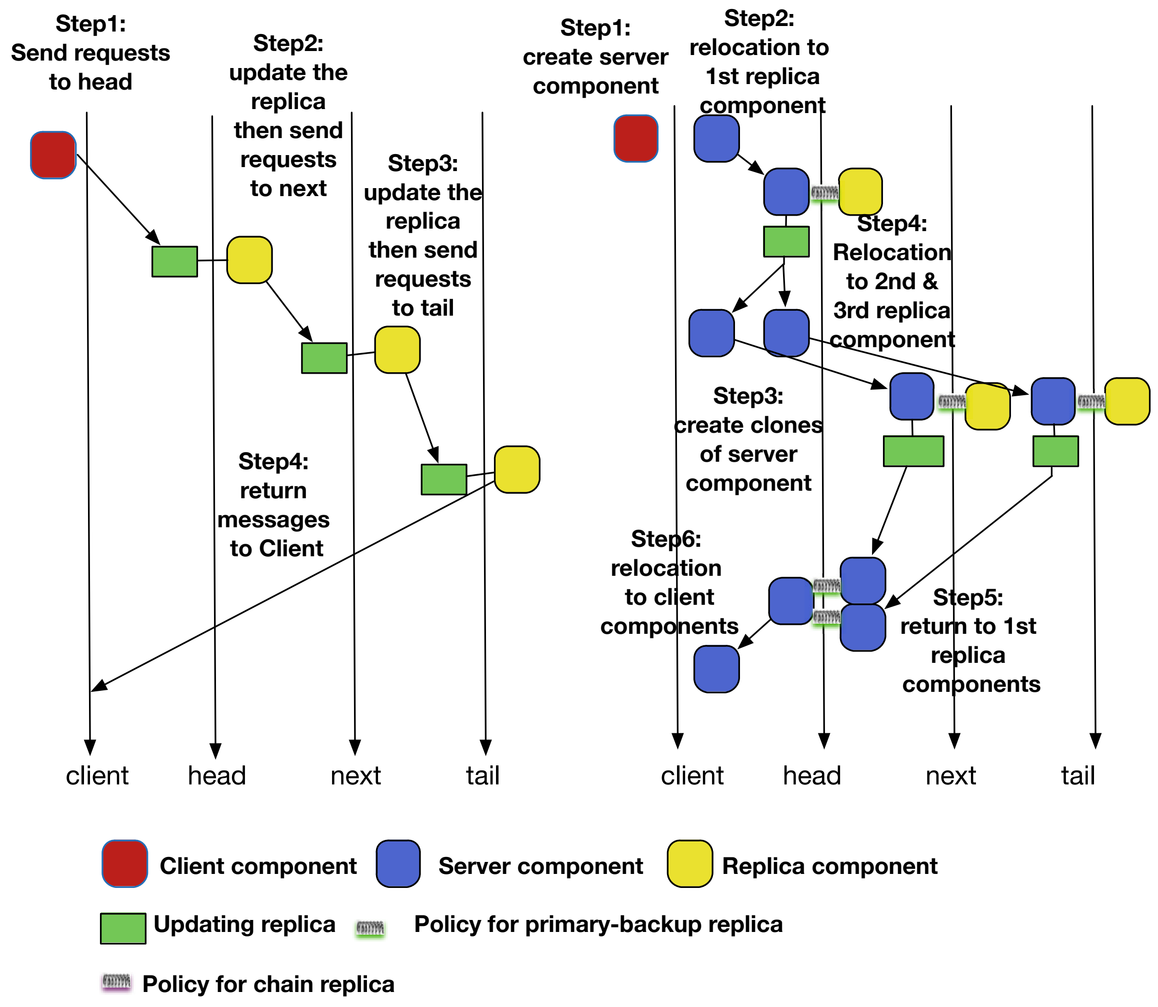

Step1: Send requests to head
Step2: update the replica then send requests to next
Step3: update the replica then send requests to tail
Step4: return messages to Client
client
head
next
tail
Step1: create server component
Step2: relocation to 1st replica component
Step4: Relocation to 2nd & 3rd replica component
Step3: create clones of server component
Step6: relocation to client components
Step5: return to 1st replica components
client
head
next
tail
Client component
Server component
Replica component
Updating replica
Policy for primary-backup replica
Policy for chain replica

Figure 6.4: Chain change to Primary-backup

```
head_side [ received( message, head_side, server_component ) then
callback(updata_replica());
delay( time ) then copyTo( other_servers )
| server_component]
```

The whole system can adapt itself to one of either of the primary-backup or chain replication approaches by changing the policies of the server component. This application means that the proposed approach could enable a distributed system to be adapted in its architecture level between primary backup and chain replication. As it had no centralized management system, it was useful in providing scalable and dependable distributed systems, as was discussed in Chapters 4 and 5. The server components themselves in primary backup and chain replication were the same. This means that our approach could separately define adaptations for components from the application-logic of the components.

6.3 Spreading Components for Sensor Nodes

The third application is the dynamic deployment of software components over a sensor network. It is a well known that after a sensor node detects environmental changes, e.g., the presence or movement of people, in its area of coverage, some of its geographically neighboring nodes tend to detect similar changes after a period of time. Software components should be deployed at nodes where and when environmental changes can be measured. The basic idea behind this example is to only deploy software components at nodes around such changes.

I assumed that the sensor field was a two-dimensional surface composed of sensor nodes and it monitored environmental changes, such as motion in objects and variations in temperature. Each software component had the *spreading* and *time-to-live* policies described in Subsection 5.4 in addition to its application-specific logic, e.g., monitoring environmental changes around its current node, where the destination components of the former were neighboring sensor nodes and the condition of the latter was the detection of changes within a specified time. It was assumed that such a component was located at nodes close to the changes. When an adaptation manager receives an event from sensors to notify of changes, it evaluates the policies of the component; if there are no components at neighboring nodes, it creates clones of the component and deploys them at the neighboring nodes (As shown in Figure 6.5). When the changes move to another location, e.g., when people are walking, the components located at the nodes near the change can detect the change in the same way because the clones of the components are deployed at the nodes. The policy for each of the Sensing components is described as:

$$\begin{gathered}\textbf{current}\ [\ +\ \textbf{delay(}\ \textit{time}\ \textbf{)}\ \textbf{then remove}\\ +\ \textbf{(detect(}\ \textit{target}\ \textbf{)}\ \textbf{then moveTo(}\textit{target}\textbf{)}\ ;\\ \neg\ \textbf{exist}\ (\textit{Sensing, neighbor\_of\_target})\ \textbf{then}\\ \textbf{copyTo(}\ \textit{neighbor\_of\_target}\ )\ )\ |\ \textbf{Sensing}]\end{gathered}$$

where *current* is an element of $\mathcal{X}$, and specifies the current computer of the component and *detect*(target) is a user-defined function that returns *true* when the movement of a target is discovered. I can select the destinations by using the *estination(Spec)* according to the specification of neighboring nodes. Each clone is associated with a time-to-live (TTL) limit by using a *detect* function, and *neighbor_of_target* is an element of X and specifies the spatially neighboring nodes around the current node. Although a node can monitor changes in interesting environments, it sets the TTLs of its components to their own initial values. It otherwise decrements TTLs over the passage of time. When the TTL of a component becomes zero, the component automatically removes itself according to the policy to conserve computational resources and batteries at the node. The TTL limit of each Sensing component is reset when the component detects the target again.

6.4 Distributed Model-View-Control application

The fourth application is distributed Model-View-Control (MVC) for adaptation. It is one of the most typical applications of disaggregated computing, where the original MVC is a software architectural pattern for implementing user interfaces [50]. The *concept* of disaggregated computing was initially advocated in Microsoft's EasyLiving project [11]. There have been several similar projects, e.g., PointRight [41] and BEACH [85]. Existing work on disaggregated computing has basically aimed at sharing input/output devices between computers to construct virtual computers on distributed computers and has assumed centralized management systems have coordinated different computers. As was previously explained, their federations themselves could not be reused because they were defined inside applications and strongly depended on the target systems in addition to the applications. They also assumed that computers were initially provided with software to control devices, but it was difficult to retain such software beforehand. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted by the user. Whereas the original MVC assumes that MVC parts are located at the same computers, distributed MVC permits these parts to be provided by different computers.

The proposed distributed MVC consists of two parts: *proxy* and *application-specific components*. The first was implemented as proxy components to control and monitor devices, screens, keyboards, and mice, through low-level interfaces provided

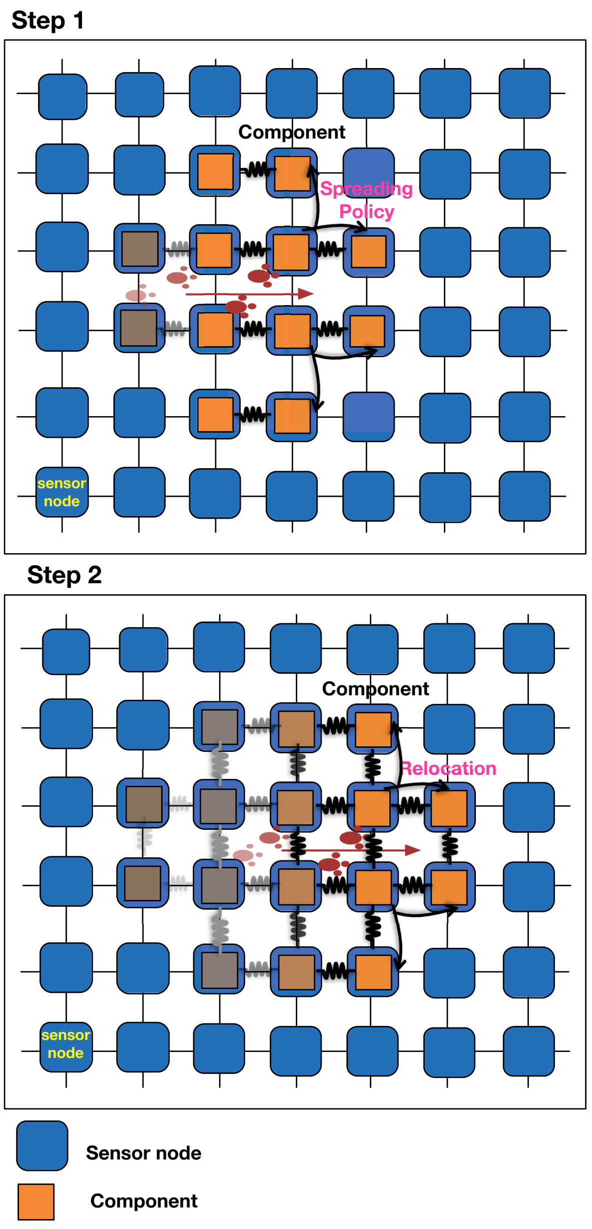


Figure 6.5: Adaptive sensor networks

by the underlying systems, e.g., hardware and operating systems that were coordinated with application-specific components through high-level interfaces. The second was implemented as several components, Model, View, and three control components, called Keyboard-based control, Virtual keyboard-based control, and Pointer-based control components with their adaptation policies (As shown in Figure 6.6).

- The model component maintains content data inside it. When receiving content input messages from control components, it updates the data and sends output messages to the View component. It has two attraction and repulsion polices. The former enables it to be deployed at computers whose computational resources, e.g., processors, memory, and storage satisfy its requirements. The latter avoids having to work two or more Model components within a specified area for reasons of consistency. An example of the attraction and repulsion polices is described as:

  home_server [**exist(***keyboard-based_control, NotebookPC* **) then moveTo(***NotebookPC***)**
  | **keyboard-based_control**]

  home_server [**exist(** *pointer-based_control, TabletPC* **) then moveTo(***TabletPC***)**
  | **pointer-based_control**]

- The View component receives output messages from Model components. It has an attraction policy whose destinations are proxy components for screen devices that satisfy its requirements on screen.

- Keyboard-based control, Virtual keyboard-based control, and Pointer-based control components forward events input from their target devices to application-specific components. They have a spreading policy whose destinations are proxy components for devices that satisfy its requirements, e.g., keyboards, screens with software-based keyboards, touch-screens, or mice.

A smart television with Virtual keyboard-based control and Pointer-based control components, in addition to the View component, were placed in a room, where these components were connected to the Model component. When a person entered the room with a tablet-PC with the component runtime system, the system detected the

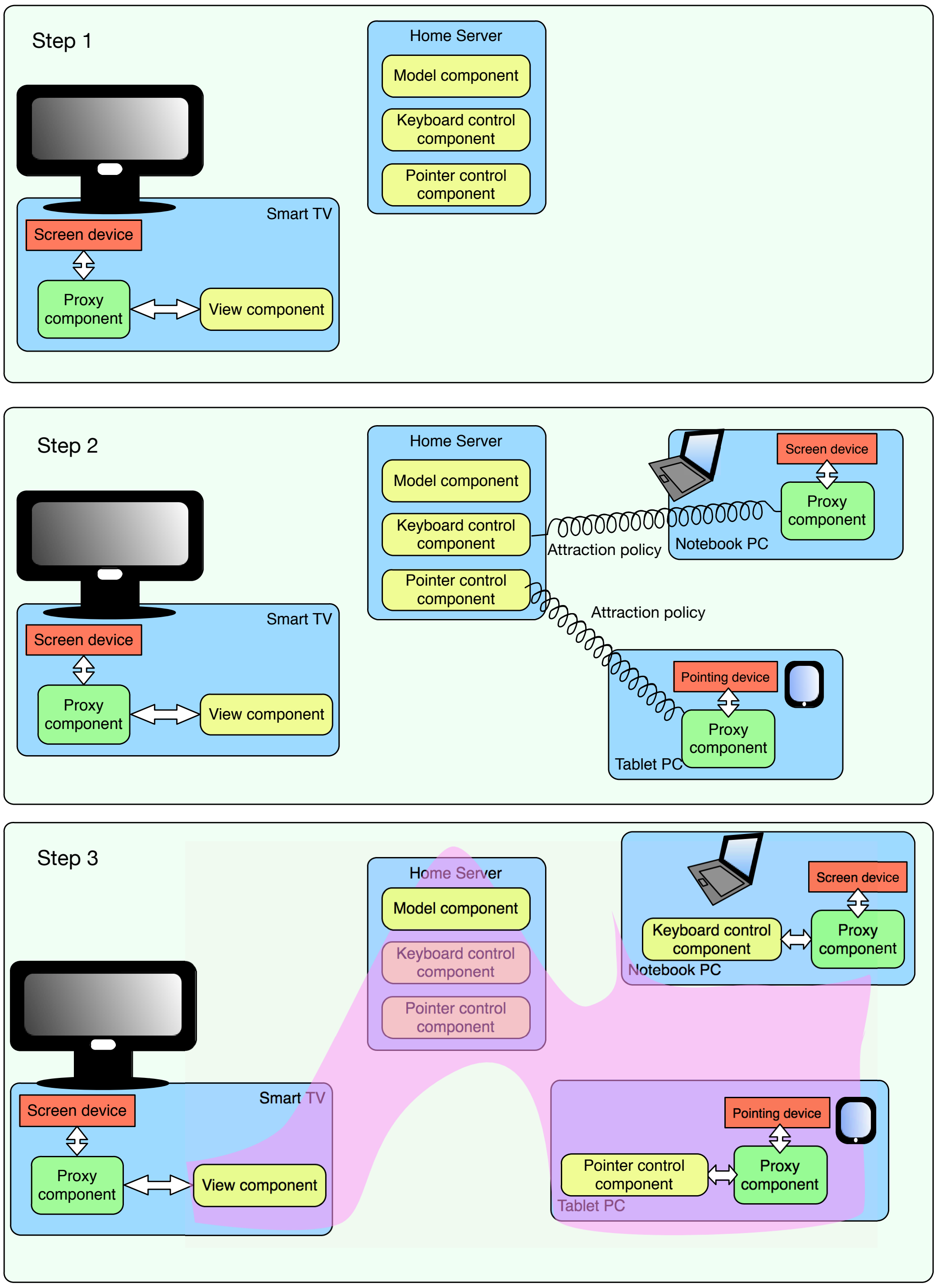


Figure 6.6: Adaptive model-view-control application

presence of the tablet-PC by using the external location sensing system. It deployed Virtual keyboard-based control, and Pointer-based control components at the tablet PC. These components forwarded their events to the Model component by using my message forwarding mechanism. The system deployed the Keyboard-based control component at the PC, if the room contained a notebook or desktop PC, when a number of inputs was required by the application.

6.5 Reliable Publish/Subscribe System

The fifth application is adaptive changes of publish/subscribe system. There have been numerous research efforts to developing reliable pub/sub systems. Most of them have required dynamic applicability to be adaptive to various changes in systems and applications. For instance, message loss and broker/link failures need to be handled in the presence of environmental changes. In addition, the software components of an application consist may need to be migrated from one node to another, so as to be adaptive to limited computing resources and high loading at a node. However, most existing approaches propose solutions in the software layer while the pub/sub system structure itself is not able to be adaptive to routing changes.

Some past research efforts have been devoted to developing reliable pub/sub systems [86]. Most of them have guaranteed that messages will eventually be delivered. Previous efforts have relied heavily on the topology to guarantee message order in the presence of failures, either through redundant nodes or links. However, redundant nodes incur high costs in replication, and redundant links usually require brokers to store large amounts of redundant information, which limits the scalability of systems and may even render brokers unusable.

I assumed that an asynchronous model was presented, where messages could be delayed, duplicated, dropped, or delivered out of order and brokers could crash and subsequently recover. Up to $\lfloor \frac{n-1}{2} \rfloor$ crash failures are tolerated for any n brokers between any pair of publishers and subscribers. In other words, there are at least 2f + 1 brokers on the path to handle f brokers in two failures.

In this example, a pub/sub system has at least three brokers (Figure 6.7). The broker 2 is the only way for other brokers to go through. When broker 2 has too many same functions for software components, this will lead to failure. Therefore, I can regularly terminate some of the software components. In addition, the software components can be relocated to new computers or some computers can be deleted for adaptive environmental changes. A Spreading policy can be used with a function

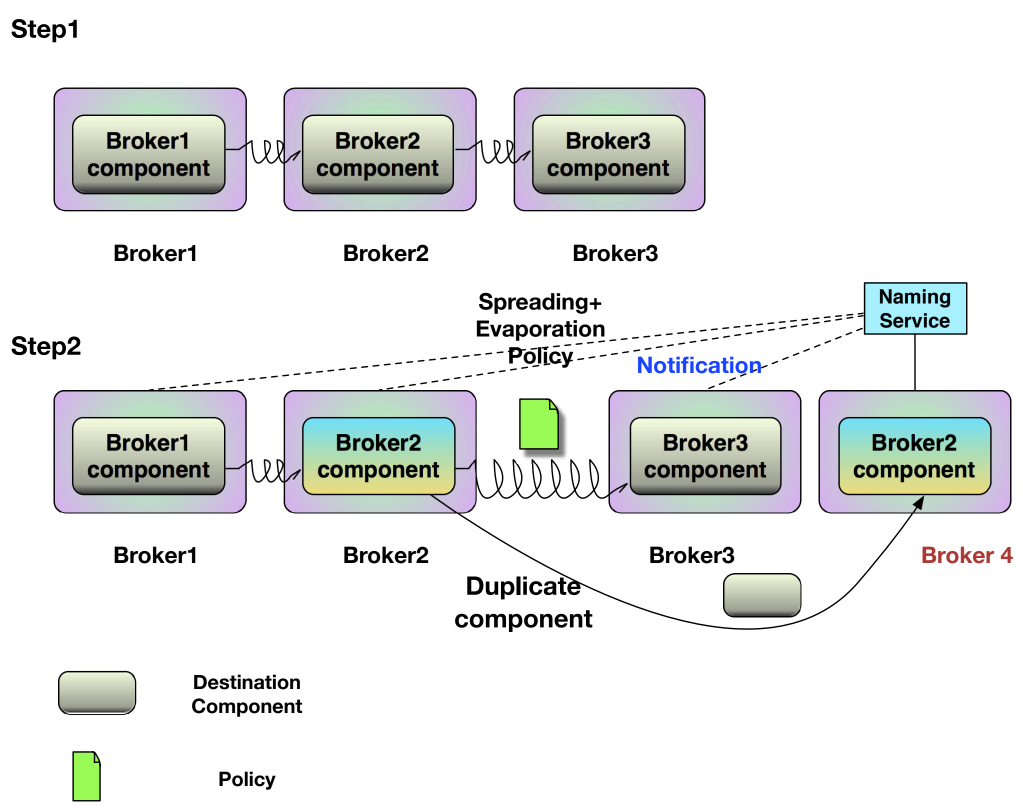


Figure 6.7: Adaptive pub/sub system

called *determination_Component()* and a combination of other policies. The combination of Spreading with an Evaporation policy as an example follows.

$$\begin{gathered}\textbf{current}\ [\ +\ \textbf{receive}(\ \textit{message, broker2,}\\ \textit{application-specific\_component}\ )\ \textbf{then remove}\\ +\ (\neg\textbf{exist}(\ \textit{determination\_Component(), broker2})\\ \textbf{then copyTo}(\textit{broker4})\ )\ |\ \textbf{application-specific\_component}]\end{gathered}$$

When a new broker is added to the pub/sub system, the new brokers will send a notify message to other broker members of their group. Then, when communication has finished, the *determination_Component* will automatically remove broker 4, go on to another computer, or go back to broker 2. This needs to be defined by users according to a real situation.

6.6 Summary

This chapter introduced five applications to adapt to changes on distributed systems. By using the proposed Mimosa system and policy-based language, the software components could be relocated between computers/nodes to adapt to changes. Both of them ascertained that the proposed approaches were effective, and I believe that the five general-purpose applications demonstrated the performance and capabilities of the proposed adaptation technologies. While individual examples only demonstrated one part of the way to define policies, it is hoped that developers can use the policy-based language flexibly, so as to create more distributed applications. I also hope they can share their applications with more users. In future, I also plan to expand this research and develop more applications. Both of the five applications can be found in previous papers in the *References* section.

The next chapter introduces the implementation of two core technologies, which were evaluated and verified with these applications.

Chapter 7

Implementation and Evaluations

This chapter can be divided into two parts that introduce the implementation and evaluations of my proposed approaches.

The first part introduces the implementation of the Mimosa middleware system, and policy-based language for specifying adaptation on distributed systems. Both of them were implemented in Java language; the former was responsible for relocating the software components between computers, and the latter was responsible for defining the changes between distributed systems and applications as policies. When the requirements were changed, these policies automatically directed the software components to relocate to destination computers to adapt to changes.

The second part explains how the strength and performance of the proposed middleware and the policy language were tested and verified through several evaluations by using the distributed applications that were introduced in Chapter 6. A detailed discussion is then presented on both of them.

7.1 Implementation of Mimosa Middleware

As was described in Chapter 4, the *Mimosa* middleware system is composed of a software component runtime system and an adaptation manager (Figure 4.2). I implemented it by using Java language.

The *component runtime system* was implemented as a mobile agent platform, but it was constructed independently of any existing middleware systems because existing middleware systems, including mobile agents and distributed objects, have not supported the policy-based relocation of application-specific components. The current implementation basically uses the Java object serialization package to marshal or duplicate components. The package does not support the capture of stack frames of threads. Instead, when a component is duplicated, the runtime system issues events for it to invoke its specified methods, which should be executed before the component is duplicated or migrated, and it then suspends its active threads.

It can encrypt components before migrating them over the network and it can then decrypt them after they arrive at their destinations. Moreover, since each component is simply a programmable entity, it can explicitly encrypt its individual fields and migrate itself with these and its own cryptographic procedure. The Java virtual machine could explicitly restrict components so that they could only access specified resources to protect computers from malicious components. Although the current implementation cannot protect components from malicious computers, the runtime system supports authentication mechanisms to migrate components so that all runtime systems can only send components to, and only receive from, trusted runtime systems. Each component runtime system periodically advertises its address to the others through UDP multicasting, and these computers then return their addresses and capabilities to the computer through a TCP channel.

The *adaptation manager* runs on individual computers which exchanges information and evaluates the conditions of its storing policies and when the external system detects changes in environmental conditions, e.g., user requirements and resource availability.

A process for the relocation of a component according to one of its policies is described.

- When a component creates or arrives at a computer, it automatically registers its deployment policies with the database of the current adaptation manager, where the database maintains the policies of components running on its runtime system.

- The manager periodically evaluates the conditions of the policies maintained in its database.
- When it detects policies whose conditions are satisfied, it deploys components according to the selected policies at the computer that the destination component is running on.

Two or more policies may specify different destinations under the same condition that drives them. The current implementation analyzes whether there are conflicts in the policies of its visiting components by using a technique to find conflicts between multiple predicates studied in existing propositional or first-class logic systems. The destination of the component may enter divergence modes, as in Chapter 5.5. The manager analyzes whether or not the effects of adaptations are repetitious by using a runtime system like that in existing runtime checking techniques [84] based on Chapter 5.5. The current implementation does not exclude any conflicts or divergences but can predict the presence of typical conflicts or divergences. As mentioned in the previous section, the language permits us to specify the requirements of computers at which components are deployed instead of the addresses of the computers, where the requirements are defined as a set of the constraints or limitations that destination computers must satisfy. In the current implementation constraints are evaluated as a constraint satisfaction problem (CSP) by using an existing tool for symbolic CSP, named JaCop [51].

Each *component* is a general-purpose programmable entity defined as a collection of Java objects like JavaBeans and packaged in the standard JAR file format. It has no specifications for adaptation inside it. The original remote method invocation between computers was introduced instead of Java remote method invocation (RMI) because Java RMI does not support object migration. Each runtime system can maintain a database that stores pairs of identifiers of its connected components and the network addresses of their current runtime systems. It also provides components with references to the other components of the application federation to which it belongs, as was discussed in Chapter 4. Each reference enables the component to interact with the component that it specifies, even if the components are on different hosts or move to other hosts. These references are managed by using the original protocol for locating components by using UDP multicasting.

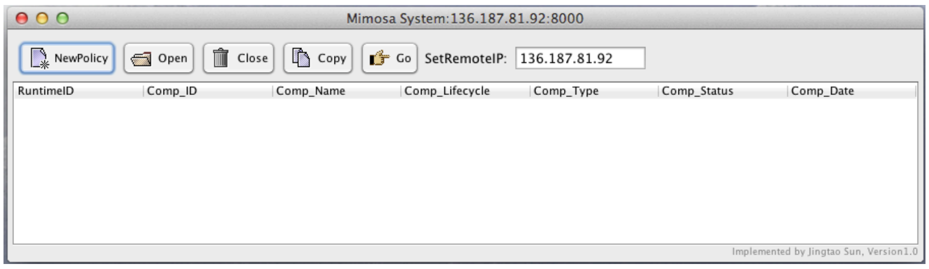
Suppose a plan is required to start two computers to run the Mimosa system. Table 7.1 describes how to start Mimosa system. There are two modes involved in running the Mimosa system.

Table 7.1: Execution of mimosa system.

Command line arguments
Java MimosaSystem 8000 8001 false
Java MimosaSystem 8001 8000 true

- First, users need to enter a path that the executable files of the Mimosa system has.
- Second, users need to input three arguments: the first is the local host number, the second is the remote host number, and the third argument is for users to choose types of Mimosa systems. When the argument is false, the Mimosa system will start a receive model to wait for messages arriving from the local network. When the argument becomes true, the Mimosa system will start a send model to initiate sending its IP_Address to the whole local network by using the UDP protocol.

Figure 7.1 has the interface of the Mimosa middleware. There are two parts: the sets of buttons, and the main body of the current status. The New Policy button is responsible for editing user-defined policies. The Open button is responsible for invoking components that have been developed as JAR files, and saved in individual computers. The Close button is responsible for closing currently executed components. The Copy button is responsible for manually copying the selected execution of components. The Go button is responsible for migrating the components to specified destination computers[6]. An automatic way of duplicating/relocating/removing components is also offered, according to user-defined policies.


Mimosa System:136.187.81.92:8000
NewPolicy
Open
Close
Copy
Go
Implemented by Jingtao Sun, Version1.0

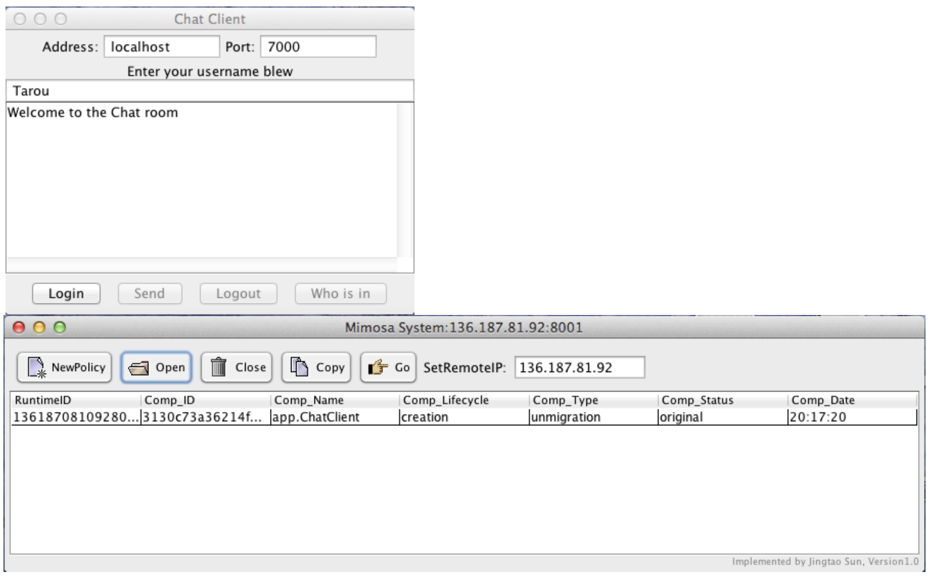
SetRemoteIP: 136.187.81.92

RuntimeID	Comp_ID	Comp_Name	Comp_Lifecycle	Comp_Type	Comp_Status	Comp_Date

Figure 7.1: Mimosa middleware graphical user interface

[6]We have left out functions, such as remove, copy, and relocation to explain and facilitate testing.

When components are running on the Mimosa system (Figure 7.2), each runtime system has a unique ID number. This has 32 significant digits, which consist of the current IP address, and port network. Each component also has a unique ID number. It is generated by Java's Universally Unique Identifier (UUID)[7]. Component_Name, Component_lifecycle, Component_Type, Component_Status, and Component_Date indicate the properties that components have. When the system state is changed, both properties will be changed. The component_Name needs to be pointed out because it contains the package name. This is because when the destination computers have the same component, it will create conflicts when it calls its own methods. If this occurs, system users will have to define its path and name with policies.


Chat Client

Address: localhost Port: 7000

Enter your username blew

Tarou

Welcome to the Chat room

Mimosa System:136.187.81.92:8001

SetRemoteIP: 136.187.81.92

RuntimeID	Comp_ID	Comp_Name	Comp_Lifecycle	Comp_Type	Comp_Status	Comp_Date
13618708109280...	3130c73a36214f...	app.ChatClient	creation	unmigration	original	20:17:20

Implemented by Jingtao Sun, Version1.0

Figure 7.2: Applications running on mimosa system

7.1.1 Experiments and Evaluations

This subsection focuses on testing the performance of the Mimosa system when applications are running on it. Table Table 7.2 summarizes the information obtained from computers that were used in the experiments.

Two terminals were started to test the performance of the Mimosa system. A Java Monitor[8] was used in this experiment to evaluate four parts, such as Used Heap

[7]UUID is an identifier standard used in software construction. A UUID is simply a 128-bit value. The meaning of each bit is defined by any of several variants.

[8]The JVM Monitor is a Java profiler integrated with Eclipse to monitor CPU, threads, and memory usage of Java applications.

Table 7.2: Experiment environment.

System State	System Information
CUP	2 Ghz Intel Core i7
OS	OS X Mavericks v10.9.2
Memory	8GB
Internet Speed	28.98Mb/s
Language	Java 1.7.0_45

Memory, Load Class Count, Thread Count, and CPU Usage. This was done according to three steps to test the Mimosa system.

- To Test the situation with the four parts when the Mimosa system was started.
- To test the situation with applications that were running on the Mimosa system.
- To test the situation with software components relocated to destination computers when changes occurred.

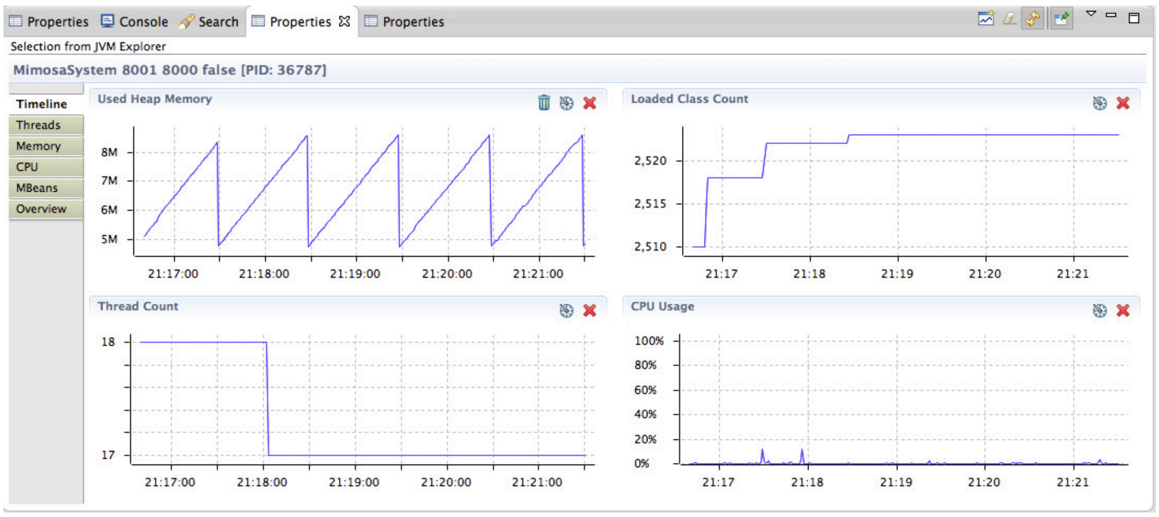


Figure 7.3: Experiment of receive model

Figure 7.3 presents the results obtained from starting the Mimosa system's receive model. Figure 7.4 presents the results obtained from starting the Mimosa system's send model. A comparison of the two figures reveals that used heap memory has relatively large changes e.g., enter a peak, and then a decline every second. This is

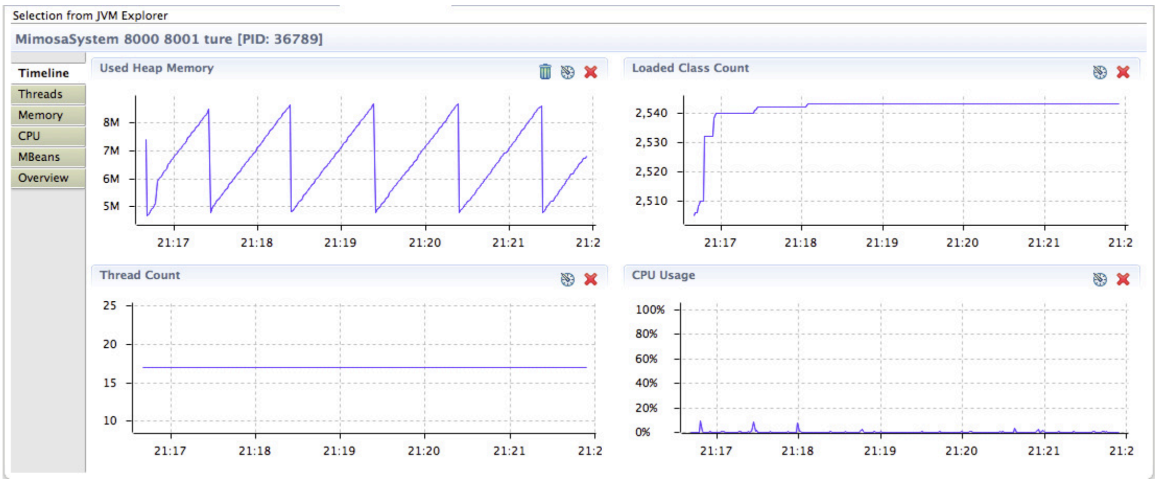


Figure 7.4: Experiment of send model

because I set the Mimosa system to interact with their own IP addresses every second through the UDP protocol.

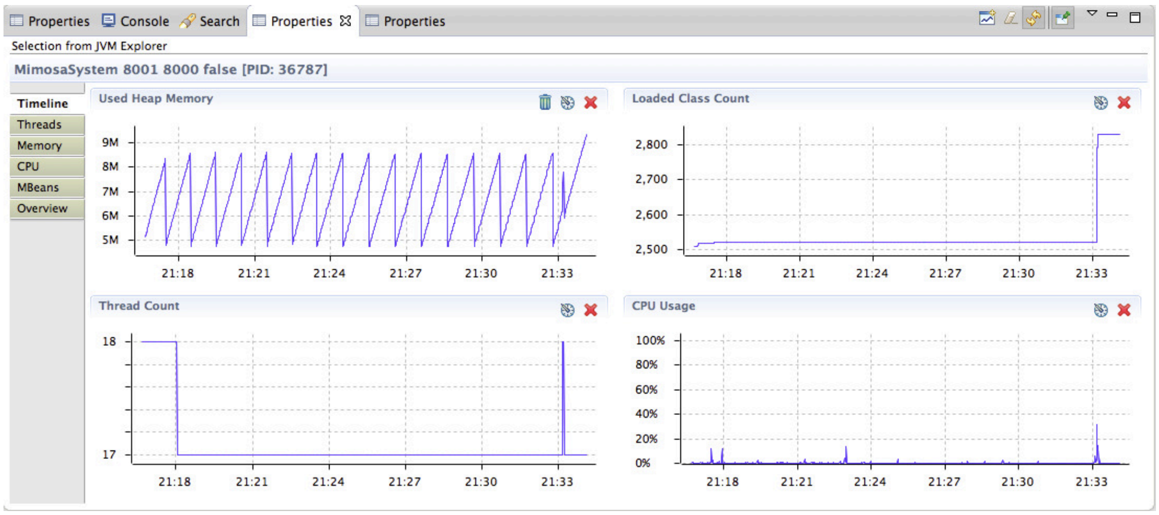


Figure 7.5: Experiment of start component

Figure 7.5 shows the execution of an application on the receive model's Mimosa system. This figure indicates several changes at 21:33:05 min from the four parts. This is because the start of an application needs computation resources; when it immediately returns to its original state means that this application has its setup completed.

When the requirements change, software components that are included in testing applications should relocate themselves to destination computers to adapt themselves to changes as well.

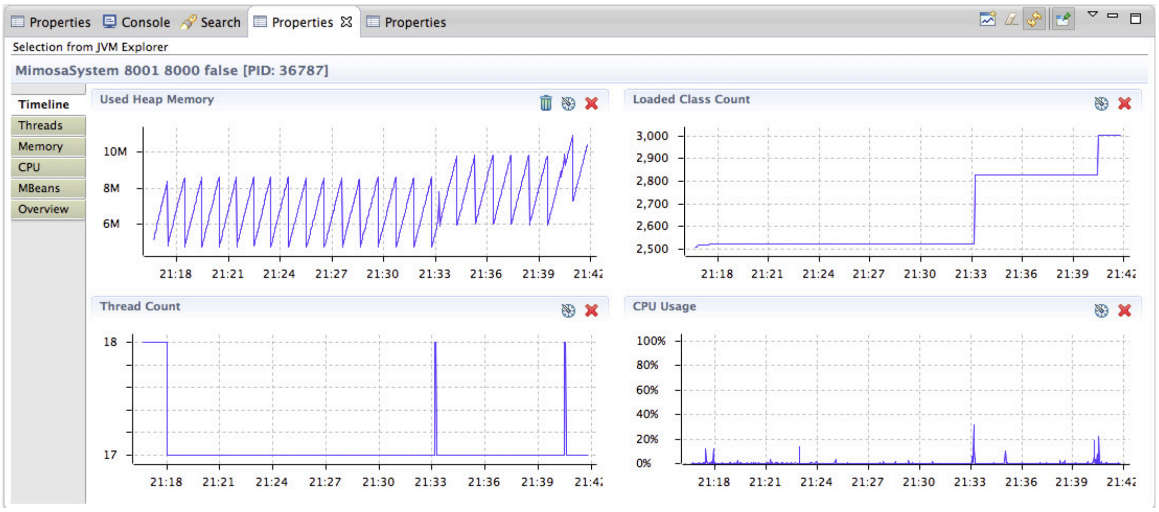


Figure 7.6: Experiment of relocated component

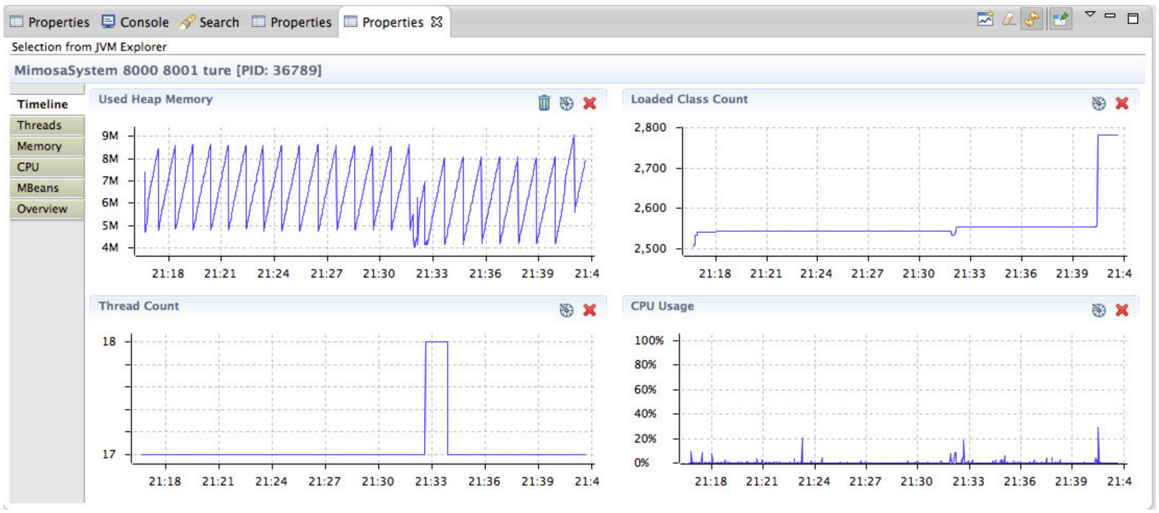


Figure 7.7: Experiment of received component

Figures 7.6 and 7.7 show the relocated components from the receive model's computer to the send model's computer. Both the properties of the two systems have a weak float, and they then return to their original state from the four parts at about 21:39:30 min. This is because the relocation of components between computers needs resources to execute migration.

These figures on testing the performance of the Mimosa middleware indicate the maximum CUP Usage and Used Heap Memory was close to 20% and 0.97%, and their minimum was close to 3% and 0.72%. Therefore, it was concluded that the proposed Mimosa middleware system was robust and performed well.

7.1.2 Discussion

As the Mimosa middleware system is based on mobile agents, it contains all their advantages. It can not only span different operating system platforms, but can also dynamically adapt to changes on distributed systems. Unknown computers delivering their addresses to known computers with the UDP protocol can clearly be identified, as can be seen through the description of the implementation and evaluation of Mimosa. When they are connected, they use the *TCP protocol to define sockets*, and then transfer messages between computers. The five figures for the experimental results indicate that the *Mimosa* middleware system consumes very few system resources, and performs well in relocating software components.

Next, the implementation of policy-based language will be introduced, and the utilization of user-definitional policies to evaluate network latency that was proposed with the five applications.

7.2 Implementation of Policy-based Language

As was discussed in Chapter 2, there are no mechanisms in existing research that completely fulfill the objectives and requirements of the research discussed here. Therefore, a policy-based language was developed to define policies to automatically select the destinations for relocated software components between computers.

The proposed policy-based language was implemented in Java language. Unlike other adaptation languages, the relocation of software components was specialized. For instance, relocation, duplication/relocation, and removal were provided for each component. Users could also choose automatic or manual means to define the destinations where software components needed to be relocated for adaptation. The following describes how this language was achieved.

The proposed policy-based language consists of two parts, e.g., the condition and action parts. The condition part is responsible for notifying user's defined conditions to naming the server, and getting back to the destination computer's IP address. Once the proposed monitor system identifies changes from applications or distributed systems themselves, and the conditions for user-defined policies are satisfied, the user-defined policies will automatically invoke the actions of each policy where their software components need to be relocated for adaptation. Users can choose "moveto()", "copyto()", or "remove()" functions for adaptive software components.

The proposed policy-based language is based on s-expressions of Scheme [8]. Therefore, parenthesized lists contains two parts, the formeris a prefix operator, and the latter is followed by its arguments. These policy programs can easily define, and dynamically evaluate pieces of user-defined code. Since the policy language is relatively small, and makes it easy to implement custom functions, users can freely expand their adaptations for conditions and actions. Moreover, the language can also manually set user's destinations to facilitate management and testing. Users only need to define a variable to save the specified IP address, and the software components can relocate to their destination computers, instead of using naming server retrieval.

Expression notation The expression of the policy-based language involves using a special notation where the expression is enclosed in parentheses. The first to appear inside parentheses is a function name or operator, and the second is the parameters. Table 7.3 lists some examples on how to use the language to define variables and functions.

Table 7.3: Expression notation.

Expression Example
(+ 2 3)
(<= a b)
(+ (* 2 3))
(let ((x 2) (y 3)) (+ x y))
(define a 322)
(define word "hello")
(define (function x) (if (<= x 0) "yes" "no"))

Key words The proposed policy-based language mainly provides a number of keywords (Table 7.4) as:

Operators and expressions The policy language mainly supports the following operator, condition expression, and loop expression, which are summarized in Table 7.5.

Users can easily define their conditions and actions as policy for relocating software components by using these designed operators and expressions. In addition, users can

[8]Scheme is primarily a functional programming language. It shares many characteristics with other members of the Lisp programming language family

Table 7.4: Mainly key words.

Key words	Grammar	Description
let	(let ((var val) ...) exp1 exp2 ...)	Valid only within the expression
define	(define var exp)	Globally valid (variable and assignment)
set!	(set! var exp)	Globally valid (assignment)
lambda	(lambda (var ...) exp1 exp2 ...)	Used to define function
begin	(begin exp1 exp2 ...)	Order execution

Table 7.5: Operators.

Relational Operators	Description
=	equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
null?	not null
Logic Operators	Description
not	Negation
or	Disjunction
and	Conjunction
Condition expression	Description
(if test consequent alternative)	If test is true, return consequent, otherwise alternative
(cond (test exp) ... (else exp))	Multiple branch judgment expression
Loop expression	Description
(for-each procedure list1 list2 ...)	Don't return to results list

define policies best suited to their applications and distributed systems as the policy language has strong distensibility.

7.2.1 Experiments and Evaluations

These experiment were used to evaluate the basic performance of adaptation using policy-based language as was discussed in Chapter 5. The cost of transmitting an object between two computers was 35 ms through the TCP protocol in my testing network environment (Table 7.2).

Next, the policy language was used to define different policies to test the five applications that were proposed. Each application contained several software components. All components were general-purpose and programmable entities, which were defined as a collection of Java objects and packaged in the standard JAR file format. They could migrate and duplicate themselves between computers according to user-defined policies. The current implementation in both the class files and the state of the software components could be relocated to the destination side.

We used a set of policies to test the delay time of the proposed applications [9]. The delay time in different tasks may largely vary when taking the internal processing of software components into account. Therefore, the processing time was ignored when assessing adaptation delay in this experiment. Instead, the latency due to relocation of software components was focused on. In addition, since the transmission time between nodes with different distances may also largely vary, the round-trip relocation of software components between computers was simply evaluated.

Table 7.6: Performance.

Adaptation	Time
Component duplication	15ms
Component relocation	35ms
Attraction policy	85ms
Spreading policy	140
Repulsion policy	115ms
Evaporation policy	14ms
Time-to-Live policy	90ms

[9] Each component was about 10 KB.

Table 7.6 shows the costs of executing five policies: attraction, spreading, repulsion, and evaporation, time-to-live described in Chapter 5.4. Each of the costs was the sum of interpreting the policies, marshaling, authentication, transmission, security verification, decompression, and unmarshaling after the target component or its clone after the adaptation manager detected the changes that should have activate the polices. Distributed systems potentially have several semantics of failures. Nevertheless, the current implementation supports the detection of crash failures at other runtime systems and disconnections in networks, where crash failures means that when a computer has troubles, it stops functioning properly. Each runtime system and adaptation manager consume smaller than 80 MB memory. It can transfer components to other runtime systems through TCP/IP. so that it is independent of any physical networks that support TCP/IP. Our approach is available in limited resources.

7.2.2 Discussion

Observations indicated that the most basic attraction policy caused the shortest delay time in both components. This is because it did not need to copy components like the other policies. Since the duplication of a software component took 30 to 40 ms, the execution time for the spreading policy was longer than that for the attraction policy. However, the evaporation policy generated the longest delay in the policies, other than that for the spreading policy, due to assessments of compatibility with other software components. Finally, since the repulsion policy and time-to-live policy did not need to be copied, both of them only have discrepancy about 25ms. In addtion, the time-to-live policy added a timestamp to each software component for execution.

7.3 Summary

This chapter outlined the implementation of an adaptive middleware system, and a theoretical foundation of policy-based language for users to define a pair of conditions and actions for adaptive changes on distributed systems. Both of them were implemented with Java language; the current implementation could only specify adaptations in the Mimosa middleware system, where the system automatically selected the destination for relocating software components, according to user-defined policies. We used various applications to test and validate the utility of the proposed middleware system and language.

Five policy formats could be written with the policy language, such as Attraction, Spreading, Repulsion, Evaporation, and Time-to-Live policies. If users defined these policies once, they could be reused when the same conditions were satisfied. A policy database was provided to save policies, and a naming server was used to manage all the information and to determine the destination of which computers were suited for relocation. When the software components were relocated to destination computers, the actions of user-defined policies could recall the methods of relocated software components to restart their tasks. The policy language could statically analyze the reasons for conflict and divergence and notify users; it then let users modify their decisions for adaptive distributed systems and applications.

The evaluations also revealed the strengthened performance of the current research. The average delay time for relocating software components between computers just needed 85 ms.

In addition, an IF expression, a cond expression, and a FOR-EACH statement were provided for defining policies. Users could especially define condition statements and loop actions by using For-Each statements to spread components to various computers.

Chapter 8

Conclusion and Future Work

This dissertation provided basic ideas for dynamic adaptation on distributed systems through the relocation of software components. This is because the relocation of software components just changes the execution of their location. Therefore, the approaches explained here could avoid system failures, and adapt to them as well as provide non-stop distributed systems. However, these approaches also separated adaptation concerns from software components. If the conditions of policies are satisfied according to user-defined policies, the proposed middleware system will automatically choose when and which software components need to relocate themselves to suitable destination computers for adaptation.

The final section of this dissertation overviews contributions that have been presented, and there is a brief description of several research topics that have not been tackled. Finally, several future studies will be discussed.

It is hoped that this research will help more people to develop adaptive applications on the new middleware system. I also plan to continue with these research achievements in the future.

8.1 Overview of Dissertation

Chapter 1 began with the motivation for this dissertation's aims. As different distributed systems have different requirements, several of these were defined for expected distributed systems. After that, basic ideas were introduced for adaptation on distributed systems. The key idea was to continue with policy-based language and adaptive middleware systems. A discussion on the challenges, contributions, and organization of this dissertation are presented in the last section.

Chapter 2 provided an overview and discussion of the most relevant current researches that were related to and influenced the design and operation of the new dynamic adaptation middleware called *Mimosa* (in Chapter 4). Related work was discussed in terms of its support for dynamic adaptation, such as parameter, software, and coordination levels and other approaches, e.g., architecture and location approaches. This chapter also compared differences in related work with the approaches discussed here, and introduce the advantages of my proposed approach.

Chapter 3 introduced the concepts underlying this research. Several scenarios were used to focus on how software components were relocated for adaptation on distributed systems. Unlike existing researches, the proposed approaches could not only relocate software components through user-defined policies, but could also transmit their states to destination computers for adaptation on distributed systems.

Chapter 4 presented the proposed Mimosa adaptive middleware system. It began with a system model, and then presented a detailed discussion on software components and policies. After that, the new Mimosa middleware system that included a component runtime system and an adaptation manager were introduced. The basic mechanism for this involved software components running on distributed systems that could be relocated to destination computers for adaptation when the conditions of policies were satisfied.

Chapter 5 presented a policy-based language, which was constructed as a process calculus to specify adaptations. This language was developed by using a theoretical foundation, and it enabled the effects of adaptations, such as conflict and divergence problems, to be analyzed. In addition, I proposed five policy formats for users to easily define the changes as policies.

Chapter 6 presented five applications to demonstrate the usefulness and limitations of the approaches. The first application called *Remote Information Retrieval* was a typical example to relocate software components between computers according to the largerSize function. The second was called *Primary-back replication and Chain* *replication.* Both of these approachesf architectures could be dynamically changed through the relocation of software components. The third application was *sensor networks.* The software components could make copies by using user-defined policies, and spread the copies to other sensors to notify them that similar changes would occur. The fourth application involved adaptive *Model-View-Control (MVC) applications.* It is one typical example of disaggregated computing, and it can relocate software components to control various devices. The fifth application was an adaptive *publish/subscribe system*; through the relocation of software components among brokers, the brokers can be connected self-adaptively to cope with failures and guarantee messages are delivered.

Chapter 7 introduced the implementation and evaluation of the proposed approaches.

In this chapter concluded the dissertation with an overview of its main contributions, and further works were presented.

8.2 Conclusion

This dissertation proposed several approaches to adapting distributed applications for distributed systems. I first introduced the Mimosa middleware system as a basic mechanism that could dynamically relocate the software components between computers. Second, a policy-based language to define policies for users was introduced, which contained five relocation policy formats, called *attraction, repulsion, spreading, evaporation, and time-to-live* to easily enable practical adaptations to be implemented. Software components were separated from their adaptations in addition to underlying systems, and the policies were specified outside the components. This was simple but provided various adaptations to support distributed systems without any centralized management.

The relocation of components between computers was useful to avoid network latency on distributed systems. It was constructed as a general-purpose middleware system instead of any simulation-based systems. Software components could be composed from Java objects. These components could be relocated to destination computers through serialization/deserialization and reflection and dynamically invocated by themselves. In addition, these approaches could be used with limited resources because they involved no speculative approaches, which tended to consume computational resources. Five policy formats were evaluated in a distributed system.

The results from tests and verifications indicated that the delay time of software components was 85 ms or less. These data not only corroborated the performance of the new systems, but also indicated that the impact of the proposals would be very limited for existing distributed systems.

8.3 Future Work

Several issues still remain for future work. One of the most important issues is that this study only provided adaptive software components by relocation and was not coordinated through software components themselves. Coordination-level adaptation can be completely added to future research work. Here, further directions this work can take are pointed out.

Security Problems As the software components can be freely configured through networks, security is a huge problem. There are two security problems in the proposed approaches. The first is protection of hosts from malicious relocation. The second is protection of the relocation of components from malicious hosts. It is difficult to verify whether incoming components are malicious or not. However, there are two solutions to protecting hosts from malicious component relocation. The first is to provide access-control mechanisms, such as Java's security manager. This method can explicitly specify the permission of components, and restrict any component behaviors that are beyond their permissions. The second is to provide authentication mechanisms by using digital signatures or authentication systems. These methods can explicitly permit runtime systems to only receive components that have been authenticated or have originated from authenticated computers.

Synchronize and Consistency Problems As software components can be relocated to destination computers, how to synchronize the processing of the original component and relocated components is worth considering, which is the same as the security problem. For instance, the system time of unrelocated computers and destination computers may be different. In addition, when data processing is in order, the components have the same named functions, and their software components plan to call one of them, which will lead to consistency problems. I need to combine user-defined policies to broker one of them, and re-allocate the order for the executing components. I intend to resolve these issues in the future.

Conflict and Divergence Since each software component can have multiple policies, conflict and divergence still occur. Although a method of analyzing them was proposed in this dissertation, however the focus was on the static level. The purpose of this dissertation is for adaptation, therefore, when conflicts occur, modifications are seek to policy developers, in future, I hope to find out a method to help developers automatically modifing the conflicts. As for the divergence, there still have two restrictions in this dissertation.

- The proposed language can not be applied the conditions are positive forever.
- The proposed language do not support detecting divergence in destination in current implementation.

In future, a well-semantic and a dynamic level approach needs to be tackled to solve these problems. I hope to develop a self-decided algorithm to reduce conflict and divergence as far as possible. I also need to write more policies to test and verify the approaches to continue the research carried out thus far, and develop more applications by using the approaches to evaluate their utility.

References

[1] Joonseon Ahn, Byeong-Mo Chang, and Kyung-Goo Doh. A policy description language for context-based access control and adaptation in ubiquitous environment. pages 650–659, 2006.

[2] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. pages 562–570, 1976.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli. Mobile agent middleware for mobile computing. *Computer*, 34(3):73–81, 2001.

[5] Israel Ben-Shaul, Avron Cohen, Ophir Holder, and Boris Lavva. Hadas: a network centric framework for interoperability programming. pages 120–129, 1997.

[6] Israel Ben-Shaul, Hovav Gazit, Ophir Holder, and Boris Lavva. Dynamic self adaptation in distributed systems. pages 134–142, 2001.

[7] Israel Ben-Shaul, Ophir Holder, and Boris Lavva. Dynamic adaptation and deployment of distributed components in hadas. *Software Engineering, IEEE Transactions on*, 27(9):769–787, 2001.

[8] Gordon S Blair, Lynne Blair, Valérie Issarny, Petr Tuma, and Apostolos Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. pages 164–184, 2000.

[9] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. From natural to artificial swarm intelligence. 1999.

[10] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1):139–159, 1991.

[11] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. Easyliving: Technologies for intelligent environments. pages 12–29, 2000.

[12] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. A conceptual framework for adaptation. pages 240–254, 2012.

[13] A. Doucet M. I. Jordan C. Andrieu, N. D. Freitas. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):pp.5–23.

[14] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *in the Proceedings of The Ninth International Symposium on High-Performance Distributed Computing*, pages 11–20. IEEE, 2000.

[15] Betty Hc Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Model Driven Engineering Languages and Systems*, pages 468–483. Springer, 2009.

[16] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, 2012.

[17] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. pages 2–8, 2006.

[18] Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for pervasive systems. pages 67–82, 2002.

[19] Alexandra I Cristea, David Smits, Jon Bevan, and Maurice Hendrix. Lag 2.0: Refining a reusable adaptation language and improving on its authoring. pages 7–21, 2009.

[20] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. pages 18–38, 2001.

[21] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE transactions on software engineering*, 5:pp.662–675, 1986.

[22] Christos Efstratiou, Adrian Friday, Nigel Davies, and Keith Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. pages 13–24, 2002.

[23] Brian Ensink and Vikram Adve. Coordinating adaptations in distributed systems. *in Proceedings of 24th IEEE International Conference on Distributed Computing Systems*, pages 446 – 455, 2004.

[24] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering*, pages 111–121. IEEE Computer Society, 2009.

[25] Zubair Md Fadlullah, Mostafa M Fouda, Nei Kato, Akira Takeuchi, Noboru Iwasaki, and Yousuke Nozaki. Toward intelligent machine-to-machine communications in smart grid. *Communications Magazine, IEEE*, 49(4):60–65, 2011.

[26] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62–70, 2006.

[27] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(3):16, 2009.

[28] George H. Forman and John Zahorjan. The challenges of mobile computing. *IEEE journal of Computer*, 27(4):38–47, 1994.

[29] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[30] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. pages 33–38, 2002.

[31] Robert S Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. 1996.

[32] George T Heineman and William T Councill. Component-based software engineering. *Putting the Pieces Together, Addison-Westley*, 2001.

[33] Francisco Herrera and Manuel Lozano. Adaptation of genetic algorithm parameters based on fuzzy logic controllers. *Genetic Algorithms and Soft Computing*, 8:95–125, 1996.

[34] Ophir Holder and Israel Ben-Shaul. A reflective model for mobile software objects. pages 339–346, 1997.

[35] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in fargo. pages 163–173, 1999.

[36] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. System support for dynamic layout of distributed applications. pages 403–411, 1999.

[37] Valerie Issarny and Chirstophe Bidan. Aster: A framework for sound customization of distributed runtime systems. pages 586–593, 1996.

[38] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A perspective on the future of middleware-based software engineering. In *2007 Future of Software Engineering*, pages 244–258. IEEE Computer Society, 2007.

[39] Taishi Ito, Hideyuki Takahashi, Takuo Suganuma, Tetsuo Kinoshita, and Norio Shiratori. Design of adaptive communication mechanism for ubiquitous multi-agent systems. *Information and Media Technologies*, pages 1028–1042, 2010.

[40] Ichiro Satoh Jingtao Sun. A policy-based middleware for self-adaptive distributed systems. pages 25–31, 2014.

[41] Brad Johanson, Greg Hutchins, Terry Winograd, and Maureen Stone. Pointright: Experience with flexible input redirection in interactive workspaces. pages 227–234, 2002.

[42] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. pages 63–74, 2003.

[43] John Keeney. Completely unanticipated dynamic adaptation of software. *Department of Computer Science, Trinity College Dublin*, pages 1–202, 2005.

[44] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. pages 3–14, 2003.

[45] Mohammad Ullah Khan, Roland Reichle, and Kurt Geihs. Architectural constraints in the model-driven development of self-adaptive applications. *IEEE Distributed Systems Online*, 9(7):1–10, 2008.

[46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[47] John R Koza. Genetic programming: on the programming of computers by means of natural selection. 1, 1992.

[48] John R Koza. Genetic programming iii: Darwinian invention and problem solving. 3, 1999.

[49] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. pages 259–268, 2007.

[50] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

[51] K. Kuchcinski and R. Szymanek. http://jacopguide.osolpro.com/guidejacop.html. 2008.

[52] Radu Litiu and Atul Prakash. Dacia: A mobile component framework for building adaptive distributed applications. *ACM SIGOPS Operating Systems Review*, 35(2):31–42, 2001.

[53] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. *ACM SIGPLAN Notices*, 38(10):107–118, 2003.

[54] Markus Luckey and Gregor Engels. High-quality specification of self-adaptive software systems. pages 143–152, 2013.

[55] Emil C Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6):852–869, 1999.

[56] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy based management framework for differentiated services networks. pages 147–158, 2002.

[57] K. Lyytinen and Y-J. Yoo. Ubiquitous computing. *Communications of the ACM*, 45(12):pp.63–96, 2002.

[58] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[59] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. pages 20–58, 2002.

[60] Darryn McEvoy, Sarah Lindley, and John Handley. Adaptation and mitigation in urban areas: synergies and conflicts. *Proceedings of the ICE-Municipal Engineer*, 159(4):185–191, 2006.

[61] Robin Milner. Functions as processes. *Mathematical structures in computer*, 2(02):pp.119–141, 1992.

[62] Robin Milner. Functions as processes. *Mathematical structures in computer science*, 2(02):119–141, 1992.

[63] Mirko Morandini, Loris Penserini, and Anna Perini. Modelling self-adaptivity: a goal-oriented approach. pages 469–470, 2008.

[64] Rui Moreira, Gordon Blair, and Eurico Carrapatoso. Formaware: Framework of reflective components for managing architecture adaptation. pages 115–129, 2003.

[65] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, (3):54–62, 1999.

[66] Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. Runtime software adaptation: framework, approaches, and styles. pages 899–910, 2008.

[67] Lindström Per. First order predicate logic with generalized quantifiers. *Theoria*, 32(3):pp.186–195.

[68] K Philip, P Eric, and HC Betty. Composing adaptive software. *IEEE Computer*, 37(7):56 – 64, 2004.

[69] Jon Postel. Transmission control protocol. 1981.

[70] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. pages 205–230, 2002.

[71] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. pages 164–182, 2009.

[72] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav. Composing components and services using a planning-based adaptation middleware. pages 52–67, 2008.

[73] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[74] Ichiro Satoh. Self-organizing software components in distributed systems. pages 185–198, 2007.

[75] Ichiro Satoh. Evolutionary mechanism for disaggregated computing. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on*, pages 343–350, 2012.

[76] Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. pages 95–103, 2010.

[77] K. E. Seamons, M. Winslett, Ting Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *Policies for Distributed Systems and Networks, 2002. Proceedings. Third International Workshop on*, pages 68–79, 2002.

[78] Murray Shanahan. The event calculus explained. pages 409–430, 1999.

[79] Jon Siegel. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA, 2000.

[80] P David Stotts and Richard Furuta. Dynamic adaptation of hypertext structure. pages 219–231, 1991.

[81] Jingtao Sun and Satoh Ichiro. Specifying distributed adaptation through software component relocation. pages 337–342, July 2015.

[82] Jingtao Sun and Ichiro Satoh. Dynamic deployment of software components for self-adaptive distributed systems. pages 194–203, 2014.

[83] Junichi Suzuki and Tatsuya Suda. A middleware platform for a biologically inspired network architecture supporting autonomous and adaptive applications. *Selected Areas in Communications, IEEE Journal on*, 23(2):249–260, 2005.

[84] Gabriel Tamura, Norha M Villegas, Hausi A Müller, João Pedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, et al. Towards practical runtime verification and validation of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 108–132. Springer, 2013.

[85] Peter Tandler. The beach application model and software framework for synchronous collaboration in ubiquitous computing environments. *Journal of Systems and Software*, 69(3):267–296, 2004.

[86] Muhammad Adnan Tariq, Gerald G Koch, Boris Koldehofe, Imran Khan, and Kurt Rothermel. Dynamic publish/subscribe to meet subscriber-defined delay and bandwidth constraints. pages 458–470, 2010.

[87] Richard N Taylor, Nenad Medvidovic, and Peyman Oreizy. Architectural styles for runtime software adaptation. pages 171–180, 2009.

[88] Anand Tripathi. Challenges designing next-generation middleware systems. *Communications of the ACM*, 45(6):39–42, 2002.

[89] Aitor Uribarren, Jorge Parra, Rosa Iglesias, Juan Pedro Uribe, and Diego Lopez-de Ipina. A middleware platform for application configuration, adaptation and interoperability. In *Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on*, pages 162–167. IEEE, 2008.

[90] Kees Van Der Sluijs, Jan Hidders, Erwin Leonardi, and Geer-Jan Houben. Gal: A generic adaptation language for describing adaptive hypermedia. pages 13–24, 2009.

[91] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. 4:91–104, 2004.

[92] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced runtime adaptation for java. *ACM Sigplan Notices*, 45(2):85–94, 2010.

[93] Thomas Vogel and Holger Giese. Language and framework requirements for adaptation models. *Models @ run. time*, pages 1–12, 2011.

[94] Rolf H Weber and Romana Weber. Internet of things. 2010.

[95] Danny Weyns and Michael Georgeff. Self-adaptation using multiagent systems. *Software, IEEE*, 27(1):86–91, 2010.

[96] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 7(1):8:1–8:61, May 2012.

[97] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. pages 79–88, 2009.

[98] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty Hc Cheng, and Jean-Michel Bruel. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2):177–196, 2010.

[99] Qishi Wu, Nageswara SV Rao, Jacob Barhen, SS Iyenger, Vijay K Vaishnavi, Hairong Qi, and Krishnendu Chakrabarty. On computing mobile agent routes for data fusion in distributed sensor networks. *Knowledge and Data Engineering, IEEE Transactions on*, 16(6):740–753, 2004.

[100] Qun Yang, Xianchun Yang, and Manwu Xu. A mobile agent approach to dynamic architecture-based software adaptation. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–7, 2006.

[101] Ji Zhang and Betty HC Cheng. Model-based development of dynamically adaptive software. pages 371–380, 2006.

Published Paper List

International Journals

[1]Jingtao Sun, Ichiro Satoh, "A Middleware-level Approach to Adaptive Distributed Systems," in International Journal of Computer Systems, (ISSN: 2394-1065), Vol.02, No.11, pp.481-489, 2015.

[2]Jingtao Sun, Ichiro Satoh,"Theory and Implementation of an Adaptive Middleware for Ubiquitous Computing Systems," in Journal of Information Processing, Vol.24 No.6, pp.1-9, 2016.

International Conferences

[1]Jingtao Sun, Ichiro Satoh, "Dynamic Deployment of Software Components for Self-Adaptive Distributed Systems," in the 7th International Conference on Internet and Distributed Computing Systems (IDCS 2014), 2014, LNCS 8729, pp.194-203.

[2]Jingtao Sun, Ichiro Satoh, "An Approach to Dynamically Adapting Distributed System Architecture,"in the Workshop on Future Technologies for Smart Information Systems in conjunction with The 33rd IEEE Symposium on Reliable Distributed Systems (SRDS 2014), 2014, pp.138-143.

[3]Jingtao Sun, Ichiro Satoh, "A Policy-based Middleware for Self-Adaptive Distributed Systems," in the Seventh International Conference on Dependability (DEPEND 2014), 2014, ISBN: 978-1-61208-378-0, pp.25-31. (Bset Paper Award)

[4]Jingtao Sun, Ichiro Satoh, "Specifying Distributed Adaptation through Software Component Relocation," in Workshop on Distributed Adaptive Systems in conjunction with The 12th IEEE International Conference on Autonomic Computing(ICAC2015), pp.337-342.

[5]Jingtao Sun, Sisi Duan, "A Self-Adaptive Middleware for Efficient Routing in Distributed Sensor Networks,"in the IEEE International Conference on System, Man, and Cybernetics (SMC 2015), 978-1-4799-8697-2/15, pp.322-327.

[6]Sisi Duan, Jingtao Sun, Sean Peisert, ”Towards a Self-Adaptive Middleware for Building Reliable Publish/Subscribe Systems,” The 8th International Conference on Internet and Distributed Computing Systems (IDCS 2015), 2015, ISBN: 978-3-319-23237-9, pp.157?168.

National Conferences

[1]Jingtao Sun, Ichiro Satoh, ”Dynamic adaptation in distributed file systems,” in The 126th Summer United Workshops on Parallel, Distributed and Cooperative Processing(SWOPP2013),IPSJ, pp.1-6.

[2]Jingtao Sun, Ichiro Satoh, ”Dynamic adaptation for Distributed System Architecture,” in Multimedia, Distributed, Cooperative, and Mobile Symposium(DICOMO2014),2014, pp.666-673.

[3]Jingtao Sun, Ichiro Satoh,”The Design and Implementation of Dynamic Adaptive Middleware for the environmental changes of distributed systems,” in the 162th Media Communication and Distributed Processing(DPS2015),IPSJ, 2015, Vol.2015-DPS-162 No.2, pp.1-8.

[4]Jingtao Sun, Ichiro Satoh, ”Reconfigurable Architecture with Dynamic Adaptability for IoT Environment,” in Multimedia, Distributed, Cooperative, and Mobile Sysposium(DICOMO2016),2016, pp.1116-1121.

Awards

[1]Jingtao Sun, Ichiro Satoh, ”A Policy-based Middleware for Self-Adaptive Distributed Systems,” The Seventh International Conference on Dependability (DEPEND 2014), 2014, ISBN: 978-1-61208-378-0, pp.25-31. *Best Paper Award*