Building Self-configuring Services Using Service-specific Knowledge

An-Cheng Huang

December 2004
CMU-CS-04-186

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Peter Steenkiste, Chair
David Garlan
Srinivasan Seshan
Ian Foster, Argonne National Laboratory and The University of Chicago

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2004 An-Cheng Huang

This research was sponsored by the US Air Force Research Laboratory (AFRL) under grant no. F306029910518, the National Science Foundation (NSF) under grant no. CCR-0205266, and the US Navy under contract no. N6600196C8528. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Distributed systems, networking, self-configuration, network sensitivity, run-time adaptation

Abstract

Network applications such as Web browsing, video conferencing, instant messaging, file sharing, and online gaming are becoming a necessity for more and more people. From a user's perspective, these network applications are used to access *services* offered by *service providers* through the Internet. Most of the services today are *statically integrated*, i.e., at design time, a service provider puts together a *service configuration* consisting of appropriate resources and components. One major problem with such services is that they cannot cope well with variations in user requirements and environment characteristics.

Self-configuration is an emerging approach for addressing this limitation. A *self-configuring service* is able to find an "optimal" service configuration automatically according to the user requirements and environment characteristics. There have been many previous research efforts in building such services. However, previous approaches either require a provider to build a custom self-configuration solution, resulting in high development cost, or they cannot take advantage of a provider's service-specific self-configuration knowledge, resulting in low effectiveness.

In this dissertation, we show that providers' service-specific knowledge can be abstracted from the lower-level self-configuration mechanisms such that service providers can build effective self-configuring services using a general, shared self-configuration framework. The use of a shared framework reduces the development cost, and being able to take advantage of a provider's service-specific knowledge increases the effectiveness of self-configuration.

This dissertation describes how a provider can express its service-specific knowledge in a *recipe* and how the *synthesizer*, the core element of our *recipe-based self-configuration* architecture, can perform global configuration and local adaptation accordingly. We also present a network-sensitive service discovery infrastructure that provides efficient support for component selection based on service-specific optimization criteria. We validate the thesis by developing a prototype self-configuring video conferencing service using our recipe-based approach. Our experimental results show that the abstraction and interpretation of the knowledge incurs negligible overhead, and our heuristic for complex component selection problems is effective. A different set of experimental results demonstrates the flexibility of the network-sensitive service discovery approach. Finally, simulation results show that our adaptation mechanisms work as expected and do not introduce unreasonable overhead.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Previous approaches . . . 6
- 1.3 Challenges . . . 7
- 1.4 Proposed approach . . . 8
- 1.5 Related work . . . 9
- 1.6 Contributions . . . 10
- 1.7 Evaluation methodology . . . 11
- 1.8 Road map . . . 12

2 Recipe-based Self-configuration Architecture **15**
- 2.1 A general architecture for self-configuring services . . . 15
- 2.2 Self-configuring services: previous approaches . . . 18
- 2.3 Proposed solution . . . 20
- 2.4 Recipe-based self-configuration architecture . . . 24
- 2.5 Scope . . . 28
- 2.6 Related work . . . 28
- 2.7 Chapter summary . . . 30

3 Network-Sensitive Service Discovery **31**
- 3.1 The network-sensitive service selection problem . . . 31
  - 3.1.1 Application examples . . . 32
  - 3.1.2 Problem formulation . . . 34
- 3.2 Current solutions . . . 34
- 3.3 Network-Sensitive Service Discovery . . . 36
  - 3.3.1 NSSD API . . . 36
  - 3.3.2 Supporting coordinated selection . . . 39
  - 3.3.3 A Simple NSSD Query Processor . . . 42
  - 3.3.4 Alternative Implementations . . . 44
- 3.4 Implementation . . . 45
  - 3.4.1 Extending SLP . . . 45
  - 3.4.2 Network Measurement . . . 46
  - 3.4.3 Selection Techniques . . . 46

3.4.4 Best-n-Solutions . . . 47
3.5 Evaluation . . . 48
3.5.1 Importance of Network Sensitivity . . . 48
3.5.2 Different Selection Techniques . . . 49
3.5.3 Effect of Load Update Frequency . . . 51
3.5.4 Local Optimization vs. Global Optimization . . . 53
3.5.5 Algorithms for Best-n-Solutions . . . 58
3.5.6 NSSD Overhead . . . 62
3.6 Related Work . . . 63
3.7 Chapter summary . . . 65

4 Synthesizer And Recipe Representation 67
4.1 Overview of the synthesizer architecture . . . 67
4.2 Recipe representation . . . 69
4.2.1 Self-configuring service examples . . . 69
4.2.2 Abstract mapping knowledge . . . 73
4.2.3 Physical mapping knowledge . . . 76
4.3 Synthesizer . . . 78
4.3.1 Complexity and algorithms . . . 80
4.3.2 Algorithm selection . . . 81
4.4 Implementation . . . 84
4.5 Evaluation . . . 85
4.5.1 Expressiveness of recipe representation . . . 86
4.5.2 Effectiveness and cost of recipe-based self-configuration . . . 89
4.5.3 Effectiveness of hybrid heuristic . . . 95
4.5.4 Algorithm selection using offline data . . . 98
4.6 Related work . . . 101
4.7 Chapter summary . . . 104

5 Run-time Adaptation 105
5.1 Problem statement . . . 106
5.2 Architecture overview . . . 109
5.3 Adaptation strategies . . . 110
5.3.1 Strategy format . . . 111
5.3.2 Strategy specification . . . 112
5.3.3 Example . . . 113
5.4 Customization . . . 115
5.4.1 Strategy selection and dynamic constraint . . . 115
5.4.2 Dynamic tactic binding . . . 116
5.4.3 Discussion . . . 118
5.5 Coordination . . . 119
5.5.1 Conflict detection . . . 120
5.5.2 Conflict resolution . . . 122

5.5.3 Identifying incompatibility 124
5.6 Implementation 125
5.7 Evaluation 127
5.7.1 Coordination 128
5.7.2 Customization 140
5.8 Related work 143
5.9 Chapter summary 147

6 Conclusions and Future Work **149**
6.1 Contributions 149
6.2 Discussion 150
6.3 Future work 152

List of Figures

1.1 A self-configuring multiplayer online gaming service. 3
1.2 A self-configuring video streaming service. 3
1.3 A self-configuring video conferencing service. 4
1.4 Service-specific approach. 6
1.5 Generic approach. 7
1.6 A high-level view of the recipe-based self-configuration architecture. 9

2.1 Required architectural elements for self-configuring services. 16
2.2 A self-configuring multiplayer online gaming service. 22
2.3 A video conferencing service example. 22
2.4 Trade-off between effectiveness and development cost. 24
2.5 Recipe-based self-configuration architecture. 25
2.6 Operations of a recipe-based self-configuring service. 26

3.1 A multiplayer online gaming service example. 32
3.2 A proxy-based End System Multicast (ESM) example. 32
3.3 A video streaming service example. 33
3.4 The NSSD API. 38
3.5 Using the API in the ESM example. 38
3.6 Coordinated selection of multiple components for a video conferencing service. 39
3.7 The extended NSSD API. 41
3.8 An example of “redundant” candidates. 42
3.9 Handling an NSSD query. 43
3.10 A sample filter for the game example. 45
3.11 Central cluster vs. distributed servers in a multiplayer game application. . . 49
3.12 Effect of per-packet processing: average maximum latency. 50
3.13 Effect of per-packet processing: average rank. 50
3.14 Cumulative distribution under 15 units per-packet processing. 51
3.15 Effect of load update frequency: average maximum latency. 52
3.16 Effect of load update frequency: average rank. 52
3.17 Cumulative distribution of maximum latency with update interval of 8. . . . 53
3.18 Relative global optimality for sessions in a typical weighted-5 configuration. 55
3.19 Relative global optimality for weighted-5. 56

3.20 Relative global optimality for weighted-25. 57
3.21 Relative global optimality for unweighted-25. 58
3.22 Fraction of time each VGW/HHP combination is globally optimal (unweighted-25). 59
3.23 Fraction of time each VGW/HHP combination is globally optimal (weighted-25). 60
3.24 Relative global optimality of pure-local in weighted-200. 61
3.25 Difference in global optimality between pure-local and clustering-3. 62
3.26 Difference in global optimality between pure-local and clustering-10. . . . 63
3.27 Distributions of relative global optimality. 64

4.1 Synthesizer architecture. 68
4.2 A video streaming service. 69
4.3 An interactive search service. 70
4.4 A video conferencing service. 71
4.5 Abstract mapping and physical mapping. 72
4.6 Abstract configuration API. 74
4.7 Abstract mapping knowledge for the video conferencing service. 74
4.8 A video conferencing recipe: abstract mapping. 75
4.9 Objective function API. 77
4.10 Physical mapping knowledge for the video conferencing service. 78
4.11 A video conferencing recipe: abstract and physical mappings. 79
4.12 Service-specific knowledge for the video streaming service. 86
4.13 A video streaming recipe. 86
4.14 Service-specific knowledge for the interactive search service. 87
4.15 An interactive search service recipe. 88
4.16 A simpler video streaming recipe. 88
4.17 Relative optimality: generic simulated annealing. 90
4.18 Configuration time per request: generic simulated annealing. 91
4.19 Breakdown of configuration time per request: generic simulated annealing. 92
4.20 Relative optimality: comparison for small-scale problems. 96
4.21 Configuration time per request: comparison for small-scale problems. . . . 96
4.22 Relative optimality: comparison for larger-scale problems. 97
4.23 Configuration time per request: comparison for larger-scale problems. . . . 97
4.24 Breakdown of configuration time per request: comparison for larger-scale problems. 98
4.25 Optimality of resulting configurations: small-scale problems. 99
4.26 Optimization cost: small-scale problems. 99
4.27 Optimality of resulting configurations: larger-scale problems. 100
4.28 Optimization cost: larger-scale problems. 100

5.1 Two service configurations for a conferencing session: IPM and ESM. . . . 106
5.2 A massively multiplayer online gaming service. 107

5.3 Extended synthesizer architecture for run-time local adaptation support. . . 109
5.4 Extensions to the recipe APIs for local adaptation strategies. 114
5.5 Two conflict resolution approaches. 122
5.6 Extensions to the recipe APIs for adaptation support. 126
5.7 Number of users/servers during simulation: hi-load-ng. 129
5.8 Number of users/servers during simulation: low-load-ng. 130
5.9 Adaptation time for join during simulation: hi-load-ng. 130
5.10 Adaptation time for join during simulation: low-load-ng. 131
5.11 Adaptation time for split during simulation: hi-load-ng. 131
5.12 Adaptation time for split during simulation: low-load-ng. 132
5.13 Number and percentage of delayed adaptations: hi-load-ng and low-load-ng. 132
5.14 Adaptation time distribution for delayed joins. 133
5.15 Adaptation time distribution for delayed splits. 133
5.16 Quality and cost: no-global vs global. 134
5.17 Adaptation time for join during simulation: hi-load-g. 135
5.18 Adaptation time for join during simulation: low-load-g. 136
5.19 Number and percentage of delayed adaptations: hi-load-g and low-load-g. . 136
5.20 Adaptation time for delayed joins during simulation: hi-load-g and hi-load-ng. 137
5.21 Cumulative number of splits during simulation: hi-load-g and hi-load-ng. . 137
5.22 Number and percentage of delayed adaptations: low-load-ng-noLC and low-load-ng. 139
5.23 Cumulative number of splits during simulation: low-load-ng-noLC and low-load-ng. 139
5.24 Number of users during simulations. 140
5.25 Overall average quality: no customization. 141
5.26 Overall average cost: no customization. 141
5.27 Overall average quality: with customization. 143
5.28 Overall average cost: with customization. 143

Chapter 1

Introduction

The use of network applications has evolved from academic/research activities to part of everyone's everyday life. Applications such as Web browsing, video conferencing, instant messaging, file sharing, and online gaming are becoming a necessity for more and more people. For example, the number of worldwide Web users is estimated at 454 million as of September 2004 [98], and another estimate shows that the number of Internet hosts has grown 290% since the year 2000 [73]. According to statistics from Valve [127], the developer of the popular multiplayer online game Half-Life [64], players worldwide spend a total of 3.4 billion minutes per month in Half-Life, surpassing the most popular TV show in the U.S. at the time, Friends [47], which generates 2.9 billion viewer minutes per month. As a result, improving the perceived performance for users of such network applications has become an increasingly important research direction.

From a user's perspective, these network applications are used to access *services* offered by *service providers* through the Internet. For example, when a user uses a web browser to access a web site, she is accessing a service provided by the entity that creates the contents and sets up the web server. A Napster [94] client can be used to access the service provided by Napster, who implements the applications and sets up the Napster servers. In a departmental network, the network administrator can provide a video conferencing service by setting up an H.323 Multipoint Control Unit (MCU) [74] in the network so that three or more users can hold a video conference using H.323-compliant video conferencing applications such as Microsoft NetMeeting [96]. A user who wants to play an online game with others can use the game client to connect to the Battle.net gaming service [9] that allows a player to find other players and provides game servers to host gaming sessions.

As can be seen from the examples, most of the existing services are *statically integrated*, i.e., at *design time*, a service provider puts together a *service configuration* consisting of appropriate resources and components according to estimates of the user requirements and environment characteristics. For example, the Battle.net gaming service provider needs to estimate the number of simultaneous users and provision the bandwidth and server capacity accordingly. In the video conferencing example above, the network administrator determines that only NetMeeting will be supported and deploys the MCU in the network.

With the proliferation of Internet usage, the network environment and user hardware/-

software capabilities are rapidly becoming more and more heterogeneous. Therefore, services are increasingly expected to be able to cope with variations of user requirements, resource availabilities, and other environment characteristics resulting from the heterogeneity both at *invocation time*, when a particular user request is received, and *run time*, when the service is being used by the user(s). However, statically integrated services cannot handle such variations well. For example, if, at design time, the video conferencing service provider did not anticipate users with handheld devices that have limited capacities, such users will not be able to participate in conferencing sessions hosted by the service. Similarly, if the Battle.net provider only placed servers in the U.S., European users may experience unacceptable performance. On the other hand, if a service provider configures the service for the worst-case scenario, for example, deploying components that can handle all possible requests, most components/resources may be idle most of the time, resulting in high and unnecessary deployment cost.

To summarize, the main limitation of statically integrated services is that the service configuration cannot be optimized according to a particular user request and the actual environment characteristics at invocation time, and similarly, the service configuration cannot be changed to adapt to run-time variations of user requirements and environment characteristics.

To address this limitation, one emerging approach is to add *self-configuration* capabilities to a service. The key feature of a *self-configuring service* is that it is able to find an "optimal" service configuration automatically without the service provider's intervention. Note that "finding the optimal configuration" means that the self-configuring service looks for the best configuration according to some service-specific criteria, i.e., it does not mean the service is always able to find the absolute best configuration. At invocation time, i.e., when a user request is received, the self-configuring service is able to automatically compose an optimal service configuration using the appropriate components and resources according to the particular user requirements and environment characteristics at that time. Similarly, at run time, i.e., when the service configuration is serving the user(s), the service is able to automatically modify the configuration to adapt to constantly changing user requirements and environment characteristics. To illustrate the concept of self-configuring services, let us look at several motivating examples.

1.1 Motivation

Figure 1.1 shows a self-configuring multiplayer online gaming service. Several players want to play an online game together, and they invoke the service by sending a request to the service, specifying the IP addresses of the participating players, the game they want to play, and so on. To satisfy this particular request, the service can compose a service configuration that consists of a single gaming server that supports the game requested by the players. Since there may be many gaming servers that are eligible, the service will want to select the server that can provide the best performance for the players in this particular gaming session. Specifically, the service will select the server that minimizes the maximum

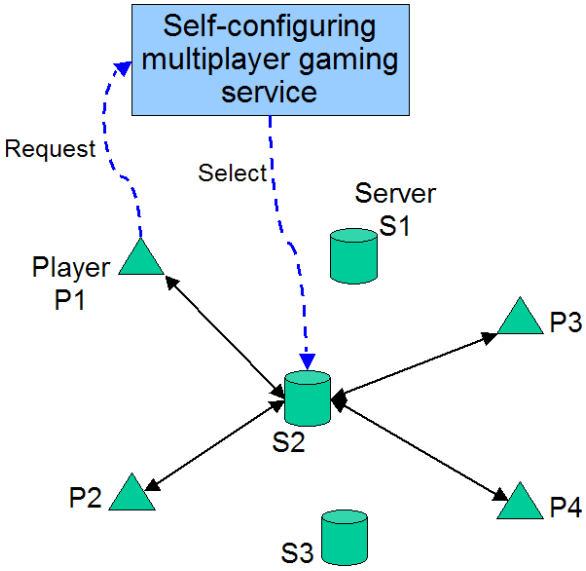


Figure 1.1: A self-configuring multiplayer online gaming service.

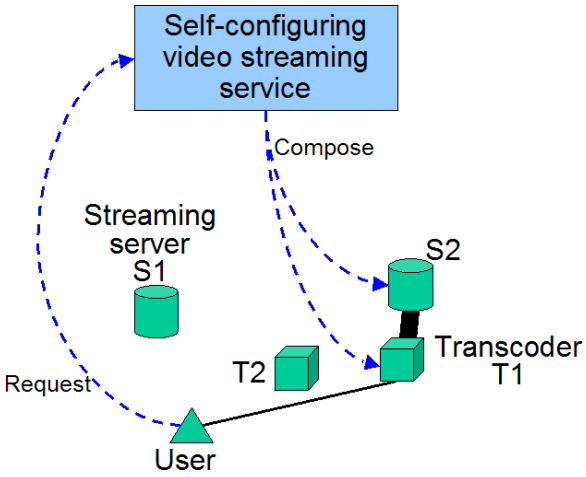


Figure 1.2: A self-configuring video streaming service.

latency from the server to any player in the session. At run time, i.e., after the players start using the composed service configuration, the service may need to modify the configuration to adapt to changes in user requirements and the environment. For example, if one of the players who originally is using a desktop computer wants to switch to a handheld console, the service can insert a computation proxy for the player to compensate for the loss of computation power. If the gaming server becomes overloaded, the service may decrease the synchronization resolution to reduce the load, potentially at the expense of player-perceived quality.

The second example is the self-configuring video streaming service in Figure 1.2. At

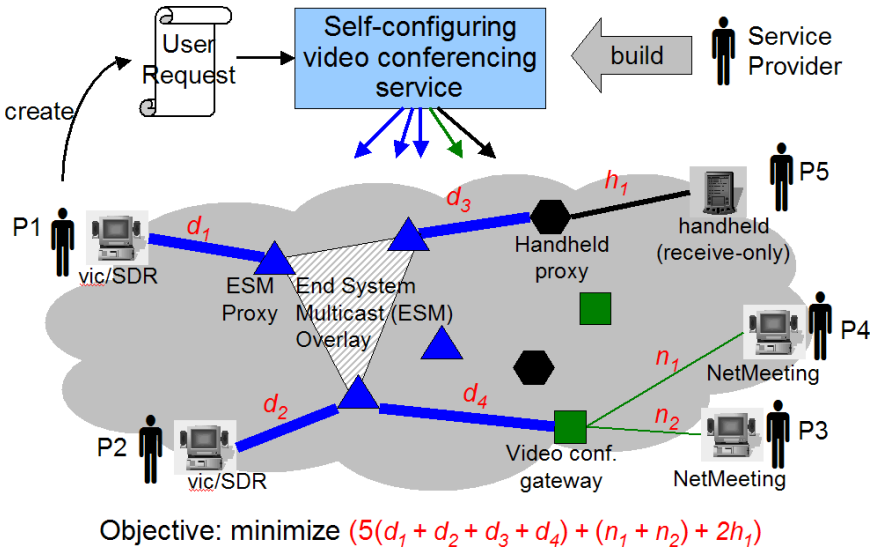


Figure 1.3: A self-configuring video conferencing service.

invocation time, if a user requests for a low-bitrate MPEG-4 video stream, the video streaming service can satisfy the request by composing a service configuration that consists of a high-bitrate MPEG-2 streaming server and an MPEG-2-to-MPEG-4 transcoder. Furthermore, in order to reduce the network resource usage, the service can select a server and a transcoder such that the bitrate-weighted network distance between the server and the user is minimized. At run time, if the transcoder becomes overloaded, the service can either decrease the video codec quality to reduce the load or replace the transcoder with another one that has a higher capacity.

Finally, Figure 1.3 depicts a self-configuring video conferencing service. Five users want to hold a video conference: P1 and P2 have MBone conferencing applications vic/-SDR (VIC), P3 and P4 use NetMeeting (NM), and P5 uses a receive-only handheld device (HH). After receiving the request, the service determines that to support this conferencing session, a video conferencing gateway (VGW) is needed for protocol translation and video forwarding between VIC and NM, and a handheld proxy (HHP) is needed to join the conference for P5. In addition, since IP multicast is not available between P1 and P2, an End System Multicast (ESM) [23] overlay consisting of three ESM proxies (ESMPs) is needed to enable wide-area multicast among P1, P2, VGW, and HHP. To select all these components, the service tries to minimize the shown objective function in order to reduce the network resource usage. After the conferencing session is started, the service monitors the load on the different components, and if a component is overloaded, the service will try to reduce the load or replace the component if necessary.

These examples show that self-configuring services can be built to dynamically compose and change the service configuration at invocation time and run time to accommodate the user requirements and environment characteristics. Of course, there are a broad range of services to which self-configuration can be applied. Therefore, previous studies have

looked at services that are different from the above examples. In this dissertation, we focus on self-configuring services that are *component-based*, *session-oriented*, and *network-sensitive*. These properties can be seen in the example services illustrated above, and below is a summary of these properties.

- **Component-based**: In this dissertation, we want to support services that can configure itself not only at the *parameter level* but also at the *component level*, for example, in addition to changing the run-time parameters (e.g., bit rate) of a component, a service should also be able to insert a new component into the configuration or remove an old one. This allows much richer services to be built than what are possible in previous studies that focus on parameter-level self-configuration, for example, adjusting the allocation of limited resources among different service components, changing the service fidelity to cope with network problems, and so on.

- **Session-oriented**: We focus on services where each user request is for a *session* that will last for a relatively long time, e.g., on the order of minutes or hours. This is different from services that use a request-response model, i.e., each user request is seeking a response from the service, and thus the lifetime of a "service configuration" is likely on the order of milliseconds or seconds. In particular, session-oriented services raise several interesting issues. At invocation time, a session-oriented service can afford to spend more computation power and time to find an optimal configuration for the user request since such a configuration will be used for the entire session. Secondly, as a session can last for a long time, the user requirements and environment characteristics are likely to change during the session. Therefore, mechanisms are needed to adapt the configuration to such changes at run time. Finally, such run-time adaptations should try to avoid dramatic changes that would disrupt the on-going session.

- **Network-sensitive**: We target services that are sensitive to network performance, i.e., such a service not only needs to find a feasible configuration that can provide the required functionalities, but it also needs to optimize the configuration to improve the network performance. Specifically, the selection of the needed components in the configuration needs to be based on the network performance of the candidates for the components.

In this dissertation, we provide a framework for service providers to build self-configuring services with the above properties. From the discussions above, we can see that building a self-configuring service involves a broad range of expertise such as service semantics, component discovery, optimization, and run-time adaptation, and therefore, it can be a complex task for the service provider. The goal of this dissertation is to enable service providers to build *effective* self-configuring services with *low development cost*. Therefore, we are concerned with the following two issues.

- **Effectiveness**: The effectiveness of a self-configuring service has two aspects. (1) How optimal is the resulting configuration composed by the self-configuring service?

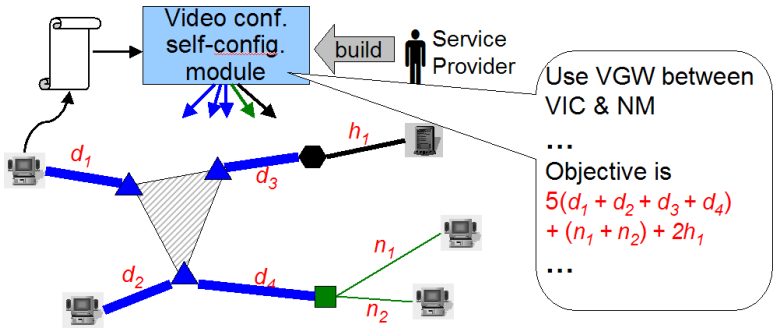


Figure 1.4: Service-specific approach.

(Note that the optimality is measured against some service-specific criteria.) (2) How efficient is the process of finding the optimal service configuration?

- **Development cost**: The cost of building a self-configuring service is measured by, for example, what expertise the provider needs to build the service, the number of lines of code the provider needs to implement, and so on.

Before presenting our solution, let us first look at previous approaches for building self-configuring services.

1.2 Previous approaches

Previously, two approaches have been proposed for building self-configuring services. However, each only addresses one of the issues above at the expense of the other. We now briefly describe these two approaches. (A more detailed discussion of related work is in Section 1.5.)

- **Service-specific approach**: To build a self-configuring service, a service provider can implement a service-specific self-configuration module that composes a service configuration for each user request. An example of this approach is shown in Figure 1.4: to build a self-configuring video conferencing service, a provider can develop a self-configuration module that is able to handle different types of users in a video conferencing session, e.g., using a VGW if there are both VIC users and NM users.

  Using this approach, the provider has complete control over how self-configuration is performed. Therefore, this approach can achieve highly effective self-configuration. However, the problem with this approach is that, as we mentioned above, self-configuration involves a broad range of expertise. Having a service provider implement its own complete self-configuration solution requires the provider to acquire the expertise in all the areas and to spend the extra efforts in integrating them all together. Therefore, this approach requires high development cost.

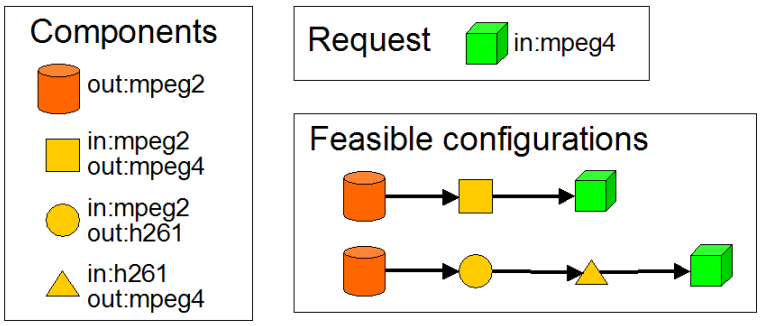


Figure 1.5: Generic approach.

- **Generic approach**: In contrast to the service-specific approach, some researchers have proposed a generic approach using *type-based service composition*. In such an approach, a generic self-configuration framework is developed and can be shared by providers of different services. All service components have typed input/output interfaces, and a user request asks for an input interface or a set of interfaces. By matching the input/output interfaces, the generic framework can find a composition of components that provides the requested interface(s). Figure 1.5 shows an example. Suppose a user requests MPEG-4-in from a video streaming service. A generic self-configuration framework can automatically determine that this request can be satisfied by two different combinations of the components, for example, a combination of a streaming server with MPEG-2-out and a video transcoder with MPEG-2-in and MPEG-4-out.

  Since the framework is shared by all services, this approach greatly reduces the development cost. However, the effectiveness of this approach is limited by the fact that a provider cannot control most aspects of self-configuration. For example, even if a video streaming service provider knows that the particular combination of server and transcoder should be used, the generic framework has to look at many different and potentially irrelevant types of components, resulting in a large search space of feasible configurations. Furthermore, when selecting each of the needed components, the generic framework uses a generic selection criteria instead of the service provider's service-specific metrics, resulting in sub-optimal configurations.

1.3 Challenges

The key difference between the above two approaches is how much they can take advantage of a provider's *service-specific knowledge*. In the service-specific approach, a service provider can hard-wire the service-specific knowledge into the self-configuration module, but additional expertise and efforts are required to integrate or implement the lower-level mechanisms, resulting in high development cost. In the generic approach, the generic

framework can be reused by different services, but self-configuration is performed using only the generic knowledge embedded in the framework, resulting in limited effectiveness. Therefore, in order to achieve our goal of both high effectiveness and low cost, we need a solution that can not only make use of providers' service-specific knowledge but also provide a generic, shared framework. This presents the following two major challenges.

- *Abstracting service-specific knowledge from low-level mechanisms*: To make use of the service-specific knowledge with a generic framework, the first necessary step is to separate the service-specific knowledge from the low-level self-configuration mechanisms. For example, the knowledge that "a conferencing gateway is needed" is service-specific, but how to format a service discovery query is not. We need to identify the different aspects of service-specific knowledge that need to be abstracted, and we need to define a representation that is sufficiently expressive for service providers to express their service-specific knowledge.

- *Designing a general framework that supports service-specific self-configuration operations*: Given the abstracted service-specific knowledge, we then need a general framework that can interpret the knowledge representation to perform self-configuration in a service-specific way. For example, given the needed component types and the service-specific criteria for component selection, the framework needs formulate and solve an optimization problem to select the actual components in the composed configuration. We need to develop mechanisms for interpreting the abstracted service-specific knowledge, techniques for formulating optimization problems, algorithms for solving such problems, infrastructures that provide low-level functionalities such as service discovery and network measurement, and so on.

1.4 Proposed approach

In this dissertation, we show that service-specific knowledge can be abstracted from the lower-level self-configuration mechanisms such that service providers can build effective self-configuring services using a general, shared self-configuration framework.

To achieve this, we propose a *recipe-based self-configuration* architecture. Figure 1.6 shows a high-level view of this architecture. The key elements of our architecture are the *service recipe*, which expresses a service provider's service-specific knowledge, and the *synthesizer*, which performs self-configuration. A service provider transforms its service-specific knowledge into a service recipe and submits the recipe to the synthesizer to create a self-configuring service. The synthesizer has two tasks. At invocation time, the synthesizer uses the knowledge in the recipe to perform *global configuration*, i.e., it automatically finds the necessary components to compose an optimal global configuration according to the specified user requirements and the environment characteristics. At run time, the synthesizer uses the knowledge to perform *local adaptation*, i.e., it monitors the configuration and makes incremental changes to the configuration if necessary in order to adapt to user or

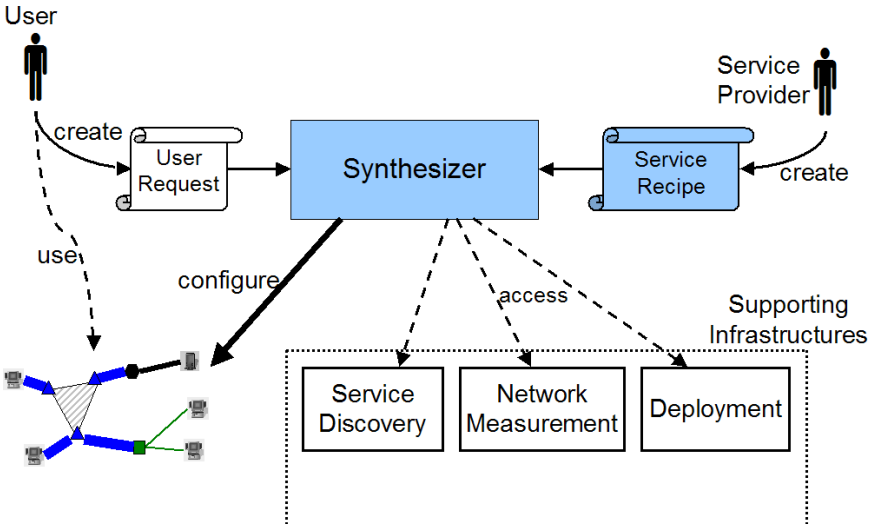


Figure 1.6: A high-level view of the recipe-based self-configuration architecture.

environment changes. The low-level functionalities required for global configuration and local adaptation are provided by the *supporting infrastructures*.

Unlike the service-specific approach, our architecture provides generic self-configuration functionalities that can be reused by providers of different services. In contrast to the generic approach, service providers can express their service-specific knowledge such as configuration heuristics and optimization criteria in the recipes to customize how self-configuration is performed. In other words, our approach achieves effective self-configuration through the use of providers' service-specific knowledge, and furthermore, it does not require providers to worry about the lower-level mechanisms, resulting in lower development cost.

1.5 Related work

Enabling technologies. A self-configuring service requires a broad range of supporting infrastructures and technologies. Service discovery infrastructures such as Service Location Protocol (SLP) [61], Service Discovery Service (SDS) [27], and Jini [79] allows the discovery of a particular component based on its functional attributes, e.g., finding video transcoders that support a particular video codec. Middleware infrastructures such as Globus [42] and the Open Grid Services Architecture (OGSA) [43] provide mechanisms for dynamic discovery, monitoring, and deployment of distributed resources, e.g., computation and storage servers. Measurement infrastructures such as Global Network Positioning (GNP) [97] and Remos [59] can provide performance information such as latency, available bandwidth, and topology, which are needed for selecting the optimal resources and components. Distributed component framework such as the Common Object Request Broker Architecture (CORBA) [101], Distributed Component Object Model (DCOM) [32],

and JavaBeans [77] and Web service technologies such as Simple Object Access Protocol (SOAP) [119] and Web Services Description Language (WSDL) [133] simplify the development of and interoperation between components.

To realize self-configuring services, the key issue is how to automate the process of using the above technologies to find an optimal configuration consisting of distributed components and resources according to user requirements and environment characteristics. Next, we discuss the previous *service-specific* and *generic* approaches for building self-configuring services.

Service-specific approach. One approach to building a self-configuring service is to develop a specialized framework that can perform self-configuration for a particular service or a particular type of services. In other words, a service provider's self-configuration logic is hard-coded into the service. For example, in [26, 87], a resource allocation framework is proposed that is able to allocate distributed resources dynamically for resource-intensive Grid applications. Gu and Nahrstedt propose a service aggregation model that uses shortest path algorithms to select components of a given service path in peer-to-peer Grids [60]. The focus in [108] is on component selection and resource allocation for resource-aware multi-fidelity applications. A framework is proposed in [88] to address the issue of resource selection for path-based resource-intensive applications. As discussed earlier, these service-specific frameworks can make effective use of service-specific knowledge, but the required development cost is higher.

Generic approach. In contrast to the service-specific approach, some previous efforts adopt a generic approach that can support different services using *type-based service composition*. In other words, all components have typed input/output (or requires/provides) interfaces, and a user request is also represented by the requested input type. Therefore, a generic self-configuration module can look for combinations of components that result in an output type matching the user-requested type. Some of these previous efforts, e.g., [57, 115, 48], focus on a "path-based" model, i.e., a service configuration is represented by a series of components that form a "service path" from a server to the user. Others support a more general graph-based model, e.g., [109, 75]. As discussed earlier, using such a generic framework to build self-configuring services can reduce the development cost, but the effectiveness is limited since it does not take advantage of service-specific knowledge.

1.6 Contributions

The main contributions of this dissertation are as follows.

- **Recipe-based self-configuration architecture.** We identify service-specific knowledge as the key to the effectiveness of self-configuration, and we propose a new architecture that abstracts the service-specific aspects of self-configuration from the lower-level mechanisms. This recipe-based self-configuration architecture is general

and supports different services, and it allows service providers to customize both invocation-time and run-time self-configuration using their service-specific knowledge with low development cost.

- **Network-Sensitive Service Discovery infrastructure.** We observe that component selection based on service-specific network performance criteria is the fundamental operation of self-configuration, which is not supported by traditional service discovery solutions. We propose a simple API that can be used for such service-specific operations without knowledge of the lower-level mechanisms. We design and implement the network-sensitive service discovery infrastructure, which combines the functionalities of service discovery and network measurement to support this API. We have presented results for this part of the thesis research at USITS '03 [69] and in the Journal of Grid Computing [70].

- **Synthesizer and recipe representation.** We divide the process of composing a complete service configuration into two steps: abstract mapping, i.e., determining what types of components are needed, and physical mapping, i.e., selecting the actual components. We design APIs that can be used by service providers to specify their service-specific knowledge, both for abstract and physical mappings, in the form of a recipe. We develop the synthesizer that composes a service configuration for a given user request according to the knowledge embedded in such a recipe. The recipe representation and synthesizer have been presented at HPDC-13 [71].

- **Local adaptation support.** In order to adapt to constantly changing user requirements and environment characteristics, self-configuration is necessary at both invocation time and run time. At run time, simply composing a brand new service configuration to replace the existing one may cause service disruption and incur high cost; therefore, local adaptations may be more effective in such cases.

  We identify three important aspects of a provider's service-specific adaptation knowledge: *adaptation strategies* specify when and how to adapt, *customization knowledge* specifies how to customize the strategies according to the user requirements and environment, and *coordination knowledge* specifies how to coordinate the strategies to avoid conflicts. We extend the recipe APIs and the synthesizer to allow a provider to specify such adaptation knowledge in a recipe. We design and implement coordination mechanisms that detect and resolve conflicts between strategies according to the coordination knowledge. Preliminary results for this part of the thesis research have been presented at WICSA '04 [20].

1.7 Evaluation methodology

From the previous section, we can see that our contributions have many different aspects. Some are quantitative such as the performance of network-sensitive service discovery or

the efficiency of the synthesizer in finding configurations. Some are qualitative such as the feasibility of the architecture or the expressiveness of the recipe representation. Therefore, we evaluate different parts of our work using different methods.

We validate the thesis by developing a prototype self-configuring video conferencing service using our recipe-based approach. We assume that existing component technologies and middleware infrastructures can be leveraged to handle component interoperation and deployment issues. Therefore, we focus on how the service-specific knowledge can be represented in a recipe and how the synthesizer uses a recipe to perform self-configuration. We will show that the service-specific knowledge for the video conferencing service can be captured using our recipe representation, and we will present experimental results that demonstrate that the synthesizer can interpret the recipe to generate an optimal global configuration with minimum overhead. We also demonstrate that our heuristic for the global optimization problem is effective and efficient for finding optimal configurations for the video conferencing service.

In addition, we evaluate the flexibility of the NSSD approach using a simulated multiplayer online gaming service deployed on the PlanetLab [106] wide area testbed. We demonstrate the expressiveness of our recipe representation using a video streaming service and an interactive search service as examples. Finally, we use the video conferencing service and a massively multiplayer online gaming service to demonstrate the flexibility of our local adaptation support in capturing the three aspects of the service-specific adaptation knowledge, and we perform simulations using the massively multiplayer online game scenario to show that our adaptation coordination mechanisms do not introduce unreasonable overhead.

1.8 Road map

The remainder of this dissertation is organized as follows. In Chapter 2, we present an overview of the recipe-based self-configuration architecture. We identify the important aspects of service-specific knowledge, describe the architectural components, and discuss how a self-configuring service is built and its operations.

In Chapter 3, we describe the network-sensitive service discovery (NSSD) infrastructure that supports component selection based on service-specific network performance criteria. We present the NSSD API and the design and implementation of a prototype NSSD infrastructure. We use both simulations and Internet-testbed experiments to demonstrate the flexibility and effectiveness of NSSD in supporting component selection based on service-specific optimization criteria. We further discuss how to support service-specific global optimization in coordinated component selection.

In Chapter 4, we present the recipe representation that allows service providers to express their service-specific self-configuration knowledge. We describe the design and implementation of the synthesizer and how it uses a recipe to compose service configurations. We also discuss the trade-off between cost and optimality when solving the global optimization problem of physical mapping and how a provider can specify a service-specific

trade-off.

In Chapter 5, we discuss how to support service-specific local adaptations at run time. We present extensions to the recipe APIs and the synthesizer that allow providers to specify their service-specific adaptation knowledge. We use simulations to demonstrate the flexibility of our customization mechanisms and also to show that our coordination mechanisms behave as expected and do not introduce unreasonable overhead.

Finally, we summarize the dissertation in Chapter 6 and discuss future research directions.

Chapter 2

Recipe-based Self-configuration Architecture

In this chapter, we present our recipe-based self-configuration architecture. In Section 2.1, we first outline a general architecture for self-configuring services and describe the required architectural elements. The core element of the architecture is the self-configuration module, which requires both service-specific and generic knowledge to perform self-configuration. We then discuss two previous approaches for building self-configuring services in Section 2.2. The two approaches differ in how they handle the two types of knowledge, and they present a trade-off between effectiveness and development cost. To achieve both high effectiveness and low cost, we present the recipe-based self-configuration approach in Section 2.3. The basic idea is to abstract the service-specific self-configuration knowledge from the generic infrastructure knowledge such that the service-specific knowledge can be used with a general, shared self-configuration framework. In Section 2.4, we present our recipe-based self-configuration architecture and describe the key pieces required to support recipe-based self-configuration, namely, the synthesizer and recipe representation, the Network-Sensitive Service Discovery infrastructure, and the local adaptation support. These elements are the focus of this dissertation. Finally, we discuss the scope of our work in Section 2.5 and look at related work in Section 2.6.

2.1 A general architecture for self-configuring services

The goal of this dissertation is to provide a framework for service providers to build self-configuring services, and the first step towards this goal is to identify what functionality is required for a self-configuring service. In this section, we identify the required architectural elements of a self-configuring service.

Consider the self-configuring video conferencing service example from the previous chapter (Figure 1.3). For this service to work, clearly components such as the VGW, HHP, and ESMP must exist in the network. The service must also be able to find the suitable components, e.g., when a VGW is needed, the service needs to be able to get a list of

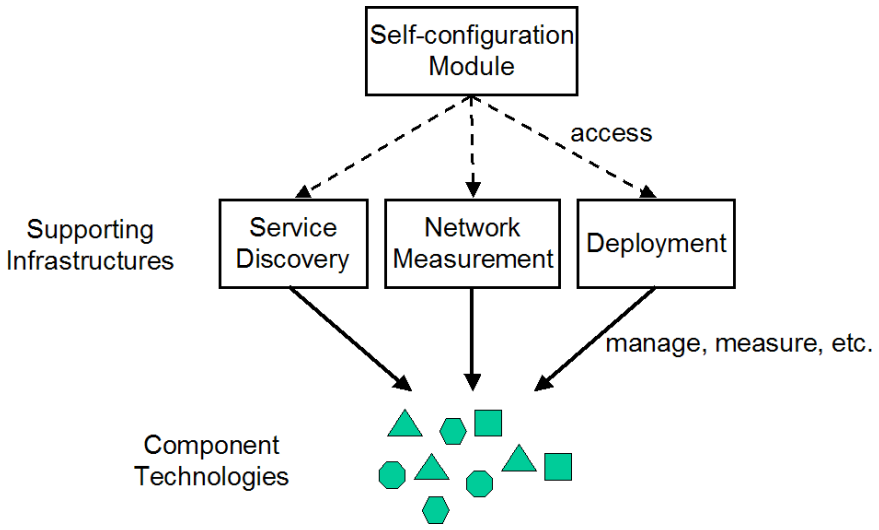


Figure 2.1: Required architectural elements for self-configuring services.

available VGWs. The service also needs mechanisms to obtain network performance information in order to select the best candidate using some criteria, e.g., selecting the VGW that has the lowest latency to NM clients. To actually use the selected components, the service needs to deal with deployment issues such as authentication, billing, remote execution, management, etc. Of course, the deployed components must be able to interoperate with one another. At run time, i.e., after the configuration is composed, the service needs to continue monitoring the various components and connections and modify the configuration to adapt to changes in user requirements and environment. For example, the service may need to adjust the video quality to control the load on the VGW or replace the existing VGW with a new one in case of a failure.

From the example above, we sketch out in Figure 2.1 the required architectural elements for self-configuration services, and we now describe these elements.

- **Component technologies**: For each request, a self-configuring service composes and maintains a service configuration consisting of distributed components, for example, specialized servers such as a VGW or a transcoder, generic resources such as a computation server or a storage server, and so on. Therefore, the components must be implemented using technologies that allow them to be reusable and interoperable. For example, components can export well-defined interfaces so that their functionalities can be accessed and combined easily even if they are implemented by different sources. Most of the required functionalities for reusability and interoperability can be provided by existing technologies.

- **Service discovery infrastructure**: The second requirement for a self-configuring service is that it must be able to find the components it needs. A component provides a specialized “service” such as transcoding or a generic service such as CPU cycles.

Therefore, finding the needed components can be formulated as the more general service discovery problem. There have been many previous studies addressing this problem, and they mostly vary in the flexibility of service description and in system properties such as scalability.

- **Network measurement infrastructure**: The network measurement infrastructure serves two roles. First, it provides network performance information that is necessary for a self-configuring service to select one of the candidates for a needed component. Secondly, it allows a self-configuring service to monitor network properties in an existing configuration to detect potential performance problems. These functionalities can be provided by many existing infrastructures and technologies. Although we only discuss the network measurement infrastructure here, we must note that component properties such as load, computation power, and so on are also needed by self-configuring services. We do not discuss a separate infrastructure for measuring and monitoring component properties since components implemented using new component technologies will likely provide a control interface for obtaining such information.

- **Deployment infrastructure**: The deployment infrastructure encompasses a broad range of functionalities required by a self-configuring service. For example, after a component is located, a self-configuring service will need mechanisms to reserve the component (or the resource needed to run the component) and to remotely initialize or execute the component. It may also need to deploy components across administrative domains, which would require authentication, billing, and other mechanisms. Many existing infrastructures and frameworks can be leveraged to provide these deployment functionalities.

- **Self-configuration module**: Finally, the self-configuration module is the core element of a self-configuring service. It uses the functionalities provided by the above elements to perform the following two self-configuration tasks.

  – *Global configuration*: At invocation time, the module composes an optimal global configuration for a request according to the particular user requirements and environment.

  – *Local adaptation*: At run time, the module make incremental changes to the service configuration in order to adapt to changes in user requirements or environment. Local adaptation is necessary because global configuration is likely expensive in terms of required computation power and time, and furthermore, switching to a new global configuration at run time may be disruptive to the users.

We observe that in order for the self-configuration module to perform global configuration and local adaptation, it requires the following two types of knowledge.

- *Service-specific self-configuration knowledge*: To automatically perform self-configuration, the module needs to know what configuration is "best" for a given set of user requirements and environment characteristics. This knowledge includes, for example, what types of components are needed in the global configuration, what optimization criteria should be used for selecting the best among multiple candidates for a particular component, when local adaptation should be performed, what incremental changes are needed for a particular adaptation, and so on. Such knowledge is service-specific since, for example, even for the same service, the best configuration for one provider may not be the best for a different provider.

- *Generic infrastructure knowledge*: In addition to the service-specific knowledge, the self-configuration module also needs to know how to use the other four elements above to actually carry out the self-configuration tasks. For example, if the service-specific knowledge specifies that "a VGW with low latency to a user is needed", then the module needs to know which service discovery infrastructure can provide information of available VGWs, which network measurement infrastructure can provide the necessary latency information, and also how to construct queries for these infrastructures to obtain the needed information. Such knowledge is generic since the underlying infrastructures and frameworks can be shared by different services/providers. For example, a latency measurement infrastructure can provide latency information to different services.

Previous solutions for building self-configuring services mostly differ in how they realize the self-configuration module or, more specifically, how they handle the two types of knowledge required by the self-configuration module. In the next section, we examine the two main approaches adopted by previous solutions.

2.2 Self-configuring services: previous approaches

Previous efforts in building self-configuring services can be categorized into two main approaches. The first one is the *service-specific* approach where a service provider builds a service-specific self-configuration module that integrates both the provider's service-specific self-configuration knowledge and the necessary generic infrastructures. As an example, the self-configuration module of a video streaming service may have the following pseudo-code segment.

- If user input format is MPEG4, do the following.

    1. Two components are required, one with type MPEG2-Server and the other with type MPEG24-Transcoder.
    2. Use infrastructure I1 to find candidates for MPEG2-Server and MPEG24-Transcoder.
    3. Use infrastructure I2 to obtain latency information for all candidates.

4. Select a candidate for each component using the latency information such that the bitrate-weighted sum of latencies is minimized.

In other words, using the service-specific approach, the service provider needs to have not only the service-specific self-configuration knowledge (e.g., "use MPEG2-Server and MPEG24-Transcoder" and "optimize bitrate-weighted sum of latencies") but also the generic infrastructure knowledge (e.g., "I1 and I2 are needed" and how to access them). The provider then hard-wires the two types of knowledge into the service-specific self-configuration module.

Since in the service-specific approach the provider has complete control over how self-configuration is performed, this approach can achieve high effectiveness. However, this approach requires a provider to have sufficient knowledge about the generic infrastructures that involves many areas, and it also requires extra efforts to integrate the generic infrastructures and technologies. Furthermore, since the provider's knowledge is hard-wired into the service-specific module, it will be difficult to reuse one provider's efforts for a different service. Therefore, the downside of this approach is the resulting high development cost.

Another approach for building self-configuring services is the *generic* approach. In this approach, a service provider builds a self-configuring service by leveraging a generic self-configuration framework that can be shared by different services. Many previous studies have developed such generic self-configuration frameworks, and most of them are based on "type-based service composition". The basic idea of type-based composition is that components have well-defined input/output types, and a user request specifies the required input type. Therefore, without any service-specific efforts, a generic self-configuration module can automatically find feasible configurations that can satisfy a particular request by looking for combinations of components that result in the requested type. Such a generic self-configuration module may have the following pseudo-code segment.

1. Find the set $C = \{c \mid c =< t_1, \ldots, t_n >$ such that $(t_1.\text{in} = \text{null})$, $(\forall 1 \leq j \leq (n-1), t_j.\text{out} = t_{j+1}.\text{in})$, and $(t_n.\text{out} = \text{user.in})\}$. I.e., C is the set of feasible configurations.

2. Select the configuration c from C that has the lowest number of component.

3. Use infrastructures I3 and I4 to find a candidate for each component type in c so that the generic optimization criteria O is optimized.

We can see that, aside from the component types and the user requested type, the generic self-configuration module does not require any service-specific knowledge. Therefore, as long as components and requests have well-defined types, such a generic self-configuration module can be shared by any services without extra efforts from the providers, resulting in low development cost.

On the flip side, this approach has limited effectiveness since it does not take advantage of the service-specific knowledge. For example, when the self-configuration module needs to find feasible configurations, the search space can potentially be huge since there can be

a large number of component types, many of which may be irrelevant to the target service. Moreover, in step 2 above, the number of components may not be a good selection metric for all services. Finally, when selecting the best candidate for each component, the generic module can only use the embedded generic optimization criteria. Therefore, the generic approach may be inefficient in finding service configurations, and the result may also be sub-optimal according to service-specific criteria.

To summarize, these two previous approaches present a trade-off between effectiveness and cost. Since our goal is to achieve both high effectiveness and low development cost, we need a solution that can achieve the best of both worlds. In the next section, we present our solution.

2.3 Proposed solution

From the above discussions, we can see that the main reason that the service-specific approach can achieve high effectiveness is that it is able to make use of the provider's service-specific self-configuration knowledge. On the other hand, the main reason that the generic approach can achieve low development cost is because the generic infrastructure knowledge is implemented in the generic self-configuration module shared by all service providers. To achieve the best of both worlds, we propose a *recipe-based self-configuration* approach for building self-configuring services. In our approach, we abstract the service-specific self-configuration knowledge from the generic infrastructure knowledge. The generic knowledge is implemented in a general self-configuration module named *synthesizer* shared by different services, and a service provider can express its service-specific knowledge in the form of a *recipe*. Self-configuration is performed by the synthesizer using the service-specific knowledge in the recipe and the generic knowledge embedded in the synthesizer.

Continuing the above video streaming example, if such a service is built using recipe-based self-configuration, the recipe may include the following service-specific knowledge.

- If user input format is MPEG4, then
    1. Two components are required, one with type MPEG2Server and the other with type MPEG24Transcoder.
    2. The two components should be selected such that the bitrate-weighted sum of latencies is minimized.

On the other hand, the following generic knowledge may be embedded in the synthesizer.

1. Use infrastructure I1 to find candidates for each required type of component.
2. If the component selection criteria involves latency, use infrastructure I2 to obtain latency information for all candidates.
3. Select a candidate for each component using the service-specific criteria.

Using a combination of these two types of knowledge, the synthesizer can then compose an optimal service configuration for a particular user request.

To realize this recipe-based self-configuration approach, one key issue that needs to be addressed is: what service-specific knowledge should be abstracted from the generic infrastructure knowledge? We identify the following three important aspects of service-specific self-configuration knowledge.

- *Component type determination*: This aspect of the knowledge specifies what types of component are needed to satisfy the user requirements in a particular user request and to accommodate the environment characteristics. A service provider needs to be able to specify service-specific logic in the recipe to determine the types of needed components.

- *Component selection criteria*: For each type of component needed for a user request, there may be multiple eligible candidates, and the synthesizer needs to select a candidate for each component. The synthesizer should not simply select an arbitrary one since the service provider will likely have its own service-specific criteria for component selection. Therefore, a service provider needs to be able to specify such selection criteria in the recipe.

- *Local adaptation knowledge*: After the initial configuration is composed, it starts serving the users. As discussed earlier, local adaptation is then needed at run time to adapt the configuration to changes in user requirements and environment characteristics. We have identified three types of service-specific knowledge for local adaptation.

  - *Adaptation strategies* specify when and how to perform local adaptation. Note that adaptation strategies that operate at the component level will also involve component type determination and component selection criteria as described above.
  - *Customization knowledge* specifies how the strategies should be customized according to the actual configuration and environment.
  - *Coordination knowledge* specifies how the strategies should be coordinated if the adaptations they try to perform conflict with one another.

We now use the multiplayer gaming service and the video conferencing service examples from the previous chapter to illustrate these aspects of service-specific self-configuration knowledge.

Examples. The most basic form of a self-configuring service is one that needs to select a single component to optimize some service-specific criteria. The multiplayer online gaming service shown in Figure 2.2 is an example of such a service. In this example, four

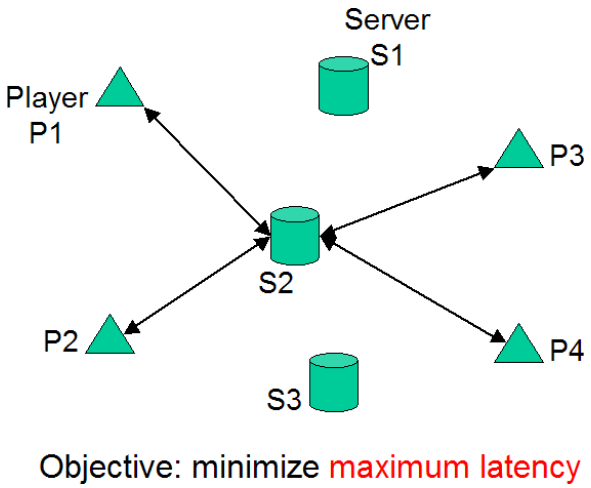


Figure 2.2: A self-configuring multiplayer online gaming service.

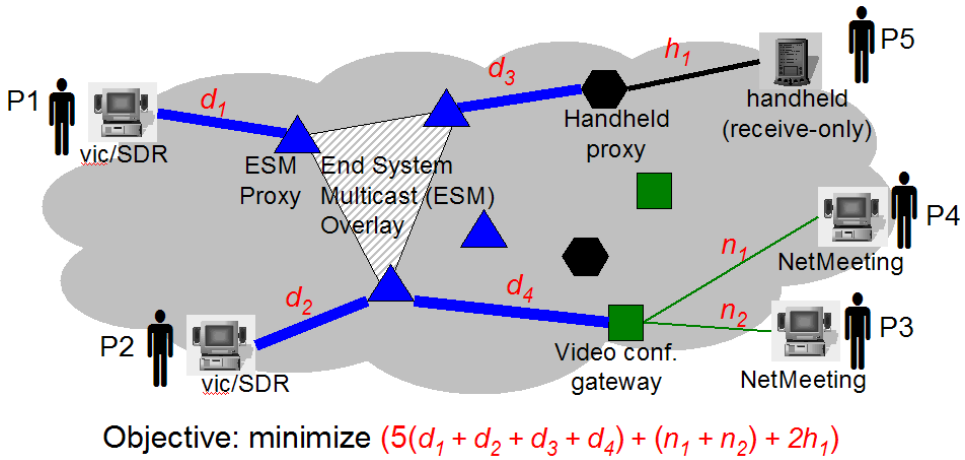


Figure 2.3: A video conferencing service example.

players want to play a multiplayer online game together, so they send a request to a self-configuring gaming service asking for a game server to host their gaming session. Determining the needed type of component is straightforward since all requests can be satisfied using a single game server. However, the provider may have many servers at its disposal, and in order to provide good gaming performance, the service cannot just use a random one for the session. In this case, the provider's service-specific component selection criterion is to minimize the maximum latency experienced by any players in the session.

The video conferencing service in Figure 2.3 is a more sophisticated example where the service configuration may need to change according to the particular user requirements and environment, and the selection of multiple components may need to be "coordinated". In this example, five users want to hold a video conference: P1 and P2 have MBone con-

ferencing applications vic/SDR (VIC), P3 and P4 use NetMeeting (NM), and P5 uses a receive-only handheld device (HH). The service provider has its service-specific logic for determining the types of components needed to satisfy a particular request. For example, if there are both VIC and NM users, a video conferencing gateway (VGW) should be used for protocol translation and video forwarding; if a HH user participates in the session, a handheld proxy (HHP) should be used; and an End System Multicast (ESM) [23] overlay consisting of ESM proxies (ESMPs) can be used to enable wide-area multicast. Given a user request, these service-specific rules can be applied accordingly to determine what types of components are needed.

After determining the component types, the service needs to select the actual components. In order to optimize user-experienced performance, the service-specific component selection criterion in this case is the objective shown in Figure 2.3. The main difference between this example and the gaming service example is that in order to find the optimal configuration based on the given objective in this case, the selection of the multiple components needs to be "coordinated". If the service only perform "local optimization" to select each component independently, the resulting configuration may be substantially sub-optimal with respect to the objective function. Therefore, in such cases, "global optimization" is needed, i.e., all components need to be selected in a coordinated fashion according to the global objective.

The above examples illustrate the service-specific knowledge for global configuration. To illustrate the service-specific knowledge for local adaptation, we continue the video conferencing example above. At run time, i.e., after the participants have begun using the composed global configuration, the service needs to be able to adapt the configuration to changes in user requirements and environment. The service provider may have service-specific adaptation strategies, for example, "S1: if a new NM user wants to join the conferencing session, connect the new user to the VGW", "S2: if the VGW becomes overloaded, replace it with a higher-capacity VGW", and so on. The provider may also have service-specific rules for customizing the strategies according to the actual configuration and environment. For example, the above strategy S2 may be applied under normal circumstances; however, when the configuration cost exceeds a service-specific threshold, the provider may want to change the strategy such that instead of replacing the VGW, it degrades the service quality. Finally, the provider may have service-specific policies for coordinating the different strategies. For example, if a new NM user joins (i.e., S1 is invoked) and the VGW becomes overloaded (i.e., S2 is invoked) at the same time, S1 should wait until S2 is finished (which replaces the VGW).

To summarize, these examples illustrate the three aspects of service-specific self-configuration knowledge that we have identified, namely, component type determination, component selection criteria, and local adaptation knowledge. To allow service providers to build effective self-configuring services with low development cost, we propose that such knowledge should be abstracted from the generic infrastructure knowledge and be expressed in a recipe, and the synthesizer, a general self-configuration module, implements the generic infrastructure knowledge to access the supporting infrastructures and interprets

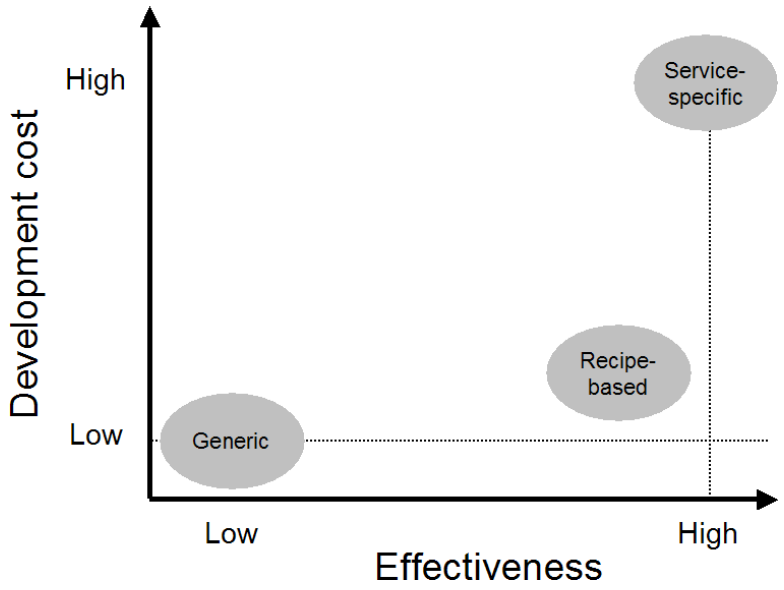


Figure 2.4: Trade-off between effectiveness and development cost.

the recipe to perform self-configuration. Figure 2.4 shows a comparison of recipe-based self-configuration and the two previous approaches in terms of effectiveness and development cost. By providing a general self-configuration module with embedded generic infrastructure knowledge, our approach relieves service providers from having to worry about the supporting infrastructures and technologies so that they can concentrate on the higher-level, service-specific issues. By allowing a service provider's service-specific knowledge to be expressed in a recipe, our approach can use such knowledge to perform self-configuration. Therefore, the effectiveness of our approach is close to that of the service-specific approach, and the development cost is close to that of the generic approach. In the next section, we present the recipe-based self-configuration architecture.

2.4 Recipe-based self-configuration architecture

Figure 2.5 illustrates the recipe-based self-configuration architecture. Like the previous generic and service-specific approaches, the recipe-based self-configuration architecture is basically an instance of the general self-configuration architecture depicted in Figure 2.1. The synthesizer, playing the role of the self-configuration module, performs self-configuration using functionalities provided by the supporting infrastructures and technologies. First, let us look at how a self-configuring service is built using the recipe-based architecture and how it operates. The operations of a recipe-based self-configuring service can be divided into three stages, as shown in Figure 2.6.

At *design time* (Figure 2.6(a)), the synthesizer exports a *recipe representation* that can be used by service providers to express their service-specific self-configuration knowledge.

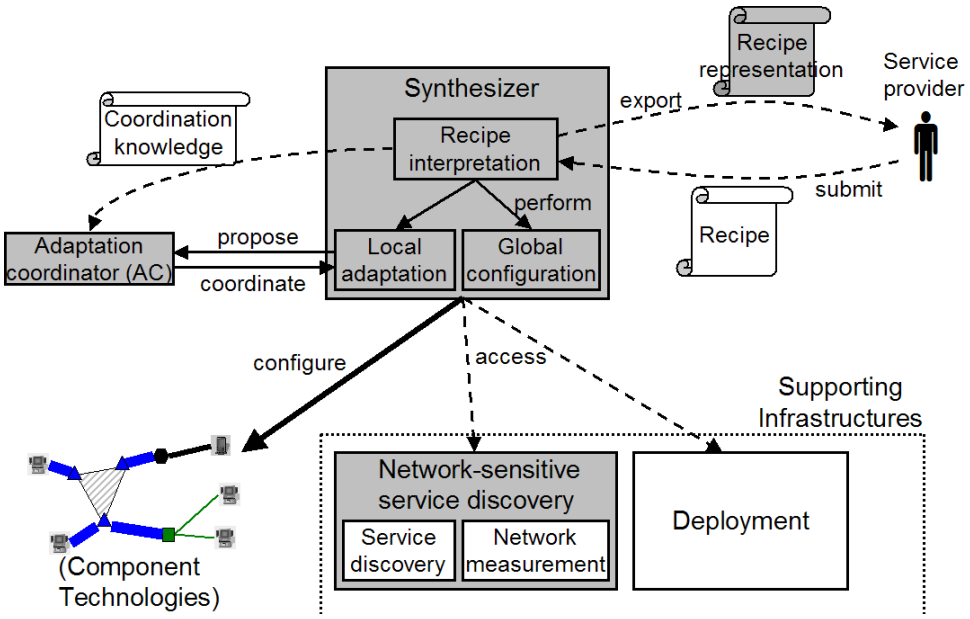


Figure 2.5: Recipe-based self-configuration architecture.

To build a self-configuring service, a service provider will transform its service-specific knowledge, including component type determination, component selection criteria, and local adaptation knowledge into a *recipe* using the recipe representation. The recipe is submitted to the synthesizer to complete the self-configuring service.

At *invocation time* (Figure 2.6(b)), a user invokes this service by sending a request (not shown in the figure) to the synthesizer. The synthesizer performs global configuration using the service-specific knowledge in the recipe to find an optimal service configuration for the request. This requires functionalities provided by the supporting infrastructures and technologies. During this stage, the synthesizer also uses the customization knowledge to customize the adaptation strategies needed later. In addition, the synthesizer extracts the coordination knowledge and passes it on to the adaptation coordinator (AC). The synthesizer then informs the requesting user that the composed configuration is ready, and the service session starts.

At *run time* (Figure 2.6(c)), i.e., after the service session is started, the synthesizer monitors the service configuration and uses the adaptation strategies in the recipe to modify the configuration when it is necessary to adapt to changes in user requirements and environment. When an adaptation strategy is executed, the changes that it wants to make are proposed to the AC. If the proposal does not conflict with other proposals, the changes can be carried out. If a conflict exists between multiple proposals, the AC will use the coordination knowledge to determine which proposals should be accepted and which should be rejected.

As discussed earlier, many of the required lower-level infrastructures and technologies for self-configuration already exist. In this dissertation, we assume that previous work can

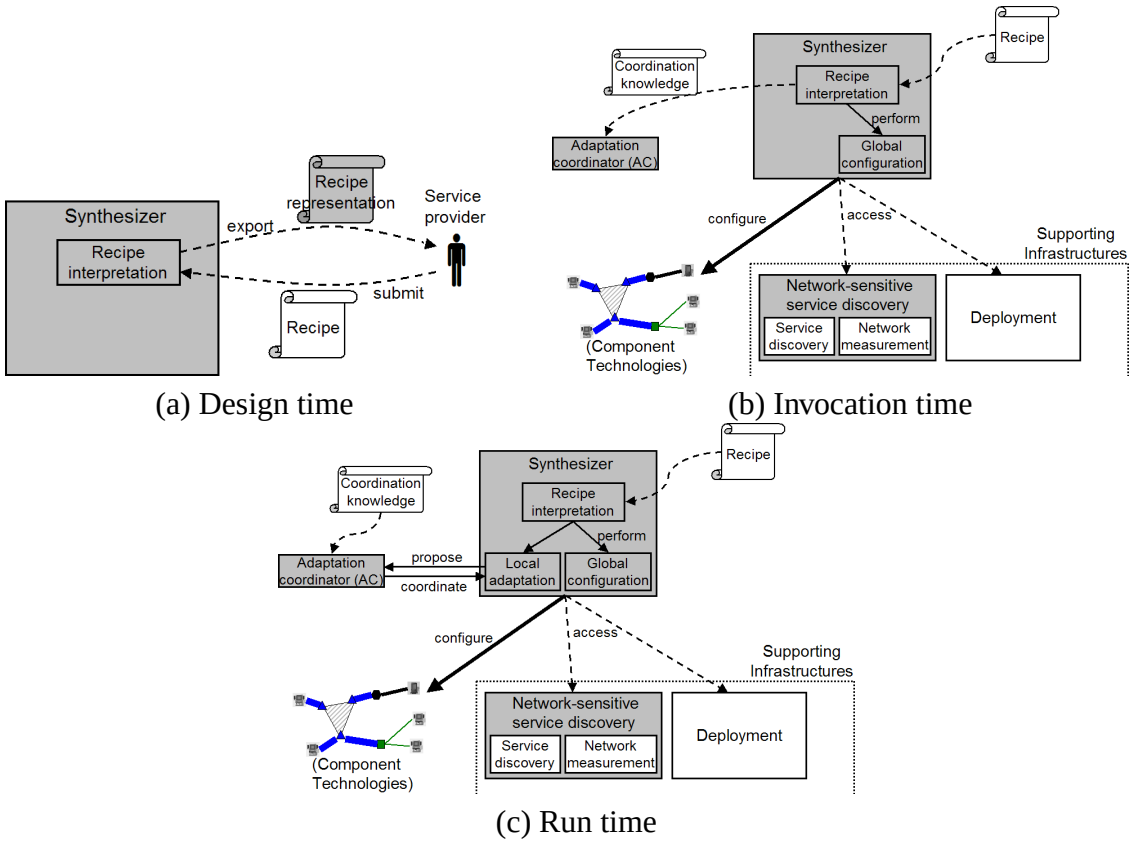


Figure 2.6: Operations of a recipe-based self-configuring service.

be leveraged to provide the lower-level mechanisms, and therefore, we do not address issues such as component reusability and interoperability, standardized service description, network measurement and monitoring, resource reservation, billing/authentication for component deployment, and so on. Instead, we focus on the missing elements that are needed to realize recipe-based self-configuration. Specifically, we focus on the shaded elements in Figure 2.5. We now briefly describe these elements.

Synthesizer and recipe representation. In our architecture, the synthesizer plays the role of the self-configuration module for a self-configuring service. The synthesizer provides two main functionalities. First, it allows a service provider to submit the service-specific knowledge in the form of a recipe that will be used to guide the self-configuration tasks, i.e., the synthesizer must be able to interpret the service-specific knowledge in the recipe. Secondly, the synthesizer implements the generic infrastructure knowledge that is necessary for performing self-configuration and can be shared by different services. Using the recipe and the infrastructure knowledge, the synthesizer performs global configuration and local adaptation.

In order to allow service providers to express their service-specific knowledge, we de-

fine a recipe representation in the form of APIs exported by the synthesizer. A provider can use these APIs to write a recipe that specifies the service-specific component selection criteria, component type determination rules, and local adaptation knowledge. The synthesizer also implements heuristics and algorithms for component selection using information obtained from the supporting infrastructures. In Chapter 4, we describe the recipe representation and the design and implementation of the synthesizer for supporting global configuration.

Network-Sensitive Service Discovery infrastructure. Given the service-specific component selection criteria specified in the recipe by the provider, the synthesizer needs to find the combination of components that optimizes the criteria. This results in two types of component selection operations, single component selection using local optimization criteria and coordinated selection of multiple components using global optimization criteria. Although such operations can be supported using the functionalities provided by existing service discovery and network measurement infrastructures, previous solutions either cannot make use of the service-specific criteria or incurs high overhead. To efficiently support component selection based on service-specific criteria, we propose the Network-Sensitive Service Discovery (NSSD) approach.

NSSD integrates the functionalities of the service discovery and network measurement infrastructures and supports service lookups based on both functional and network performance attributes. For single component selection, a self-configuring service can simply specify the local optimization criteria in a query to the NSSD infrastructure, and the NSSD infrastructure will return the best candidate accordingly. For coordinated selection of multiple components, we propose a heuristic that greatly reduces the size of the search space of the global optimization problem by selecting a small number of candidates for each component using a local objective, and global optimization is performed on the reduced search space. To support this heuristic, the NSSD infrastructure is able to return the best n candidates for a component using a local objective. The details of the NSSD infrastructure are presented in Chapter 3.

Local adaptation support. The synthesizer design and recipe representation described above support global configuration at invocation time. As discussed earlier, local adaptation at run time is also an important part of self-configuration. Therefore, we extend the architecture to support run-time local adaptation as follows.

Performing local adaptation requires three types of service-specific knowledge: adaptation strategies, customization, and coordination. Although most previous efforts for adaptation support are able to handle adaptation strategies in some form, they do not address the customization and coordination issues. We extend our recipe representation to allow service providers to specify their service-specific adaptation strategies, customization rules, and coordination policies in their recipes. We also extend the synthesizer to customize the strategies accordingly and execute the strategies at run time when appropriate. After an adaptation strategy is executed, a proposal is sent to the adaptation coordinator (AC). The

proposal specifies the changes it wants to make to the configuration. When there are multiple proposals, the AC uses the coordination knowledge in the recipe to detect conflicts and resolve them by rejecting or delaying the appropriate proposals. The details of the local adaptation support are presented in Chapter 5.

2.5 Scope

In this section, we discuss the scope of this dissertation. As mentioned earlier, we focus on self-configuring services that are component-based, session-oriented, and network-sensitive. Therefore, we do not explicitly address parameter-level self-configuration, which has been studied previously, for example, to dynamically allocate resources among different components according to user requirements. Such self-configuration mechanisms are complementary to our approach and can be integrated into our architecture. In addition, because of the session-oriented nature, we emphasize the problem of finding the optimal global configuration. As a result, our self-configuration mechanisms may be too heavy-weight for services based on the request-response model. However, as demonstrated later, our approach is flexible enough that in such cases, a provider can choose to sacrifice the optimality to increase the efficiency of self-configuration.

For run-time adaptation, we focus on performance-related local adaptations that try to change the current service configuration incrementally in response to performance problems. We also simplify the coordination problem by assuming that conflicts between adaptations only occur due to the direct effects of adaptations. In the more general case, the effect of a simple adaptation can potentially propagate throughout the entire configuration, i.e., every adaptation can potentially conflict with all other adaptations "indirectly". In this dissertation, we do not explicitly address such indirect conflicts, and we leave it to the providers to identify them at a higher level.

2.6 Related work

In the previous chapter, we have discussed previous efforts for building self-configuring services that adopted the generic approach or the service-specific approach. We now look at previous work that focuses on infrastructures and technologies that provide lower-level mechanisms required for self-configuration.

Component technologies. Many existing technologies and framework address the issue of component reusability and interoperability. For example, Microsoft's Component Object Model (COM) [24] and Sun's JavaBeans [77] allow developers to implement software modules with well-defined interfaces and provide mechanisms for combining such components into an integrated system. Technologies such as the Common Object Request Broker Architecture (CORBA) [101], Distributed COM (DCOM) [32], Java Remote Method Invocation (RMI) [78], and Simple Object Access Protocol (SOAP) [119] provide mechanisms

for using components that are distributed on different network hosts. Enterprise JavaBeans (EJB) [38], CORBA Component Model (CCM) [13], and the .NET framework [91] provide low-level mechanisms such as location, transaction, and security model and make them transparent to users of distributed components.

Traditionally, distributed components are combined manually by developers to provide integrated functionalities. For example, to make use of distributed Java components, one can implement a Java [81] program that instantiates and invokes the distributed objects. Similarly, one can describe how various components should interact using a language such as the Business Process Execution Language for Web Services (BPEL4WS) [5]. A self-configuring service can accomplish this automatically without human intervention by, for example, generating a BPEL description of the composed configuration, which can then be executed using existing mechanisms. Previous research efforts such as SWORD [109], Ninja [57], and Partitionable Services Framework [75] also include similar functionalities for deploying and invoking components in a composed configuration.

Other previous research efforts have looked at how to make components "adaptive". For example, Rover [80], Puppeteer [33], and Odyssey [100] provide interfaces to control the components and adjust their run-time parameters in response to run-time changes. In [128], Q-RAM [85], and [108], adaptations are performed by adjusting the resource allocation among different components according to utility functions.

Service discovery. Many solutions have been proposed for the service discovery problem, and they mostly vary in the flexibility of the service description and in system properties such as scalability. For example, a service can be described using attribute-value pairs that represent the functional properties of the service. To find a service, one can construct a query that includes attribute-value pairs representing the properties of the desired service. Such a query can then be matched against the descriptions of available services to find the suitable ones. This approach is used in the Service Location Protocol (SLP) [61], which has a central directory that stores available service information and handles service queries. The Service Discovery Service (SDS) [27] proposes a hierarchical system to address the scalability issue. The Metacomputing Directory Service (MDS) [41] uses a similar approach to describe and lookup resources available in a Grid environment.

Another approach is to describe a service using its input/output or requires/provides interfaces. For example, Jini [79] allows a user to find Java components that have the desired interfaces. Similarly, Web services can be described using the Web Services Description Language (WSDL) [133], and Universal Description Discovery & Integration (UDDI) [125] supports the discovery of such services.

Network measurement. A network measurement infrastructure provides network performance information for the target network host(s). Many network measurement frameworks have been proposed before. For example, IDMaps [46] allows one to query the network latency between two network hosts. Global Network Positioning (GNP) [97] can be used to compute a set of coordinates for each network host so that the network latency

between two hosts can be easily calculated. Remos [59] provides a rich set of network information such as available bandwidth measurement, topology, and so on.

Deployment. Component deployment involves a wide range of issues, for example, resource reservation, admission control, remote execution, authentication, and so on. Most of these issues have been studied in previous work. For example, Legion [95] and Globus [42] provide component deployment mechanisms that work across administrative domains. Darwin [14], Globus [26], and Legion [17] provide mechanisms for resource reservation and enforcement.

Remote initialization and execution of components can be provided by component technologies such as Java RMI. Ninja [57] provides component deployment mechanisms in a cluster environment, and Sahara [111] provides fault-tolerant capabilities. The security models in Java, .NET, and other component technologies address the security issue in component deployment.

2.7 Chapter summary

In this chapter, we first identified the required elements of a self-configuring service and described two previous approaches for building such services. We observe that the service-specific approach requires high development cost while the generic approach has limited effectiveness. To address these problems, we propose a recipe-based self-configuration architecture that abstracts the service-specific knowledge from the generic infrastructure knowledge. Therefore, a provider's service-specific knowledge can be expressed in a recipe, and the generic knowledge is implemented in the synthesizer, which is shared by different services. We used examples to illustrate the key aspects of service-specific knowledge needed for self-configuration. We then identified the missing elements that are needed for building such recipe-based self-configuring services. The design of these elements, namely the Network-Sensitive Service Discovery (NSSD) infrastructure, the synthesizer and the recipe representation, and the local adaptation support, are the focus of this dissertation.

Chapter 3

Network-Sensitive Service Discovery

As discussed in the previous chapter, component selection based on service-specific network performance criteria is the fundamental operation for self-configuring services. In order to find the optimal service configuration, the self-configuration module needs to find the best candidate for each required component according to some service-specific optimization metrics. Two types of component selection operations need to be supported: single component selection and coordinated selection of multiple components. Although the required functionalities, service discovery and network measurement, already exist, previous approaches to supporting such operations either do not allow service-specific criteria or incur unnecessary overhead that may result in scalability problems.

In this chapter, we present the design and implementation of the *network-sensitive service discovery* (NSSD) infrastructure that supports the component selection operations. Our approach is based on the observation that most of the tasks required for component selection are generic and can be performed by the infrastructure. Therefore, NSSD integrates the functionalities of service discovery and network measurement and provides a simple API for component discovery based on not only the required functional properties but also the desired network properties. We discuss how NSSD can be used by a self-configuration module to perform both single component selection and coordinated selection of multiple components.

The remainder of this chapter is organized as follows. We formulate the problem and elaborate on the challenges in the next section. In Section 3.3 we describe our network-sensitive service discovery solution, including the design of the API and the mechanisms used for network-sensitive service discovery. We describe a prototype system in Section 3.4, and Section 3.5 presents an evaluation using the PlanetLab testbed. Finally, we discuss related work and summarize.

3.1 The network-sensitive service selection problem

In this section, we first look at the component selection problem in the self-configuring service examples from the previous chapters, and then we formulate such component selection

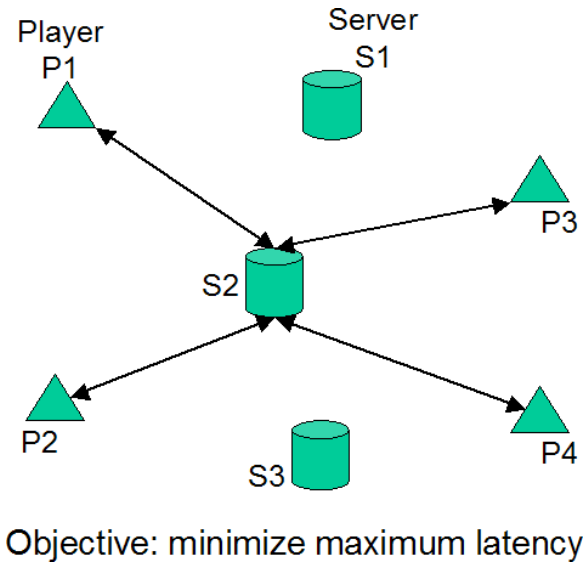


Figure 3.1: A multiplayer online gaming service example.

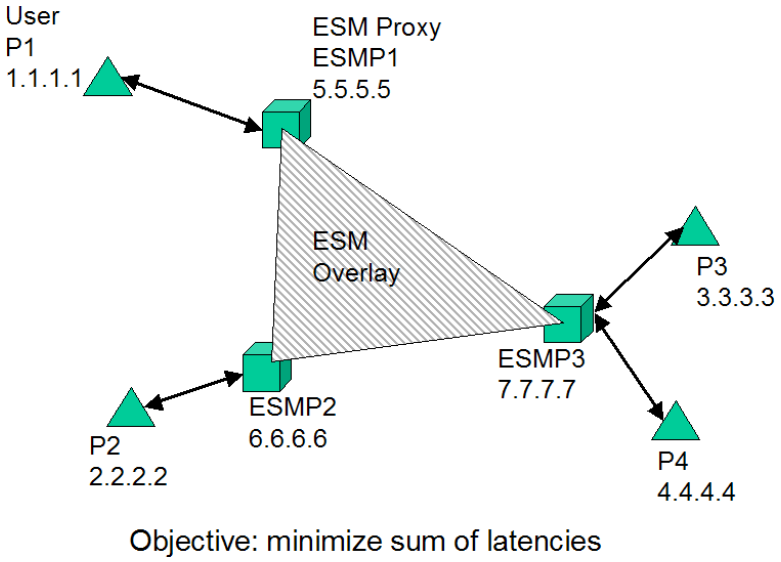


Figure 3.2: A proxy-based End System Multicast (ESM) example.

problems as the more general network-sensitive service selection problem.

3.1.1 Application examples

In Chapter 2, we described several examples of self-configuring services, and we identified that one of the key aspects of service-specific self-configuration knowledge is the optimization criteria for component selection. In other words, component selection based on such

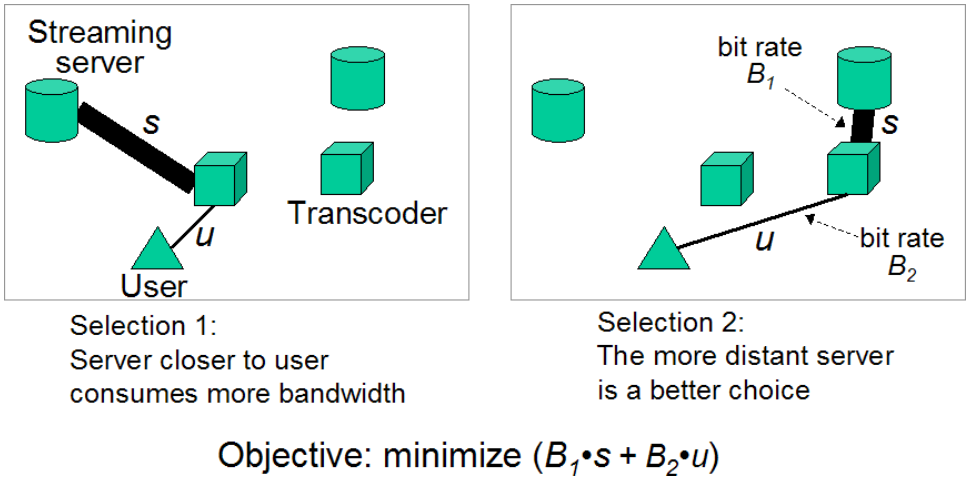


Figure 3.3: A video streaming service example.

service-specific criteria is a fundamental operation for most (if not all) self-configuring services. Now let us revisit the examples from the previous chapter and take a closer look at the component selection problem. Figure 3.1 shows a scenario for the multiplayer online gaming service: four users want to start a gaming session, and they ask the gaming service for a game server to host their session. Specifically, they need a game server that not only satisfies certain functional properties, e.g., supporting a particular game and having certain anti-cheating features, but also delivers good "performance". As shown in the figure, for this particular service, the provider's service-specific selection criteria for the server component is to minimize the maximum latency from the server to any players in the session.

The second example in Figure 3.2 is a scenario for an End System Multicast (ESM) [23] service, which is used as part of the video conferencing service in Chapter 2. In this proxy-based ESM service, each multicast participant sends its packets to an ESM proxy, and all proxies serving this group construct an ESM overlay to efficiently deliver the packets to all other participants. In order to reduce the total network resource usage of the service configuration and improve the performance for the users, the provider determines that the service-specific selection criteria for the components should be to minimize the sum of the latencies between each user and the corresponding ESM proxy.

Figure 3.3 shows a scenario for the video streaming service example in Chapter 1. Suppose a user wants to receive low bit rate MPEG-4 video streams, and the service determines that the request can be satisfied by by putting together a high bit rate MPEG-2 video streaming server and an MPEG-2-to-MPEG-4 video transcoding server. As shown in the figure, if the provider's goal is to minimize the total bandwidth usage, then "minimizing the bitrate-weighted sum of latencies" may be used as the component selection objective. Using such an objective presents an interesting situation: selecting a streaming server close to the user can reduce bandwidth usage, but a more distant streaming server may turn out to be a better choice if we can find a transcoder that is very close to the server. In other words, this problem requires the coordinated selection of multiple components.

3.1.2 Problem formulation

From the examples above, we categorize component selection operations into two types:

- **Single component selection**: The self-configuration module only needs to select a single component at a time. One possibility is that the self-configuring service only needs one component, e.g., the multiplayer gaming service only needs a single gaming server that minimizes the maximum latency. Another possibility is that the service needs multiple components, but each component can be selected independently, e.g., a service needs a transcoder that is close to the server and a proxy that is close to the client. Therefore, each component can be selected individually. In either case, the component selection is a *local optimization* problem

- **Coordinated selection of multiple components**: The self-configuration module needs to select multiple components together, for example, in the above video streaming service, selecting each component individually may result in sub-optimal configurations. Therefore, to select the optimal set of components, the self-configuration module needs to select all components in a coordinated fashion, i.e., it needs to solve a *global optimization* problem.

We now formulate the component selection problem as a more general *network-sensitive service selection* (NSSS) problem: a *user* needs to find a *service* (or a set of services) offered by a *provider* (or providers) according to not only the required *functional properties* such as service type but also the desired *network properties* such as latency to a certain network host. In other words, the selection of services is based on *user-specific selection criteria*. There may be multiple providers for a particular service, e.g., video transcoding, and each provider may have a set of distributed *servers* to deliver the service, e.g., a set of video transcoders at different locations. In the next section, we look at currently available solutions for the NSSS problem.

3.2 Current solutions

Existing research related to the NSSS problem falls in two categories.

- **Service discovery**: traditional service discovery infrastructures allow a user to find a set of servers with certain functional properties, where the properties are typically described as a set of attribute-value pairs. Existing solutions, for example, [61] and [27], differ in the flexibility of naming and matching and in their scalability properties. Similar schemes such as the Metacomputing Directory Service (MDS) [41, 25] have also been proposed in the context of computational Grids.

- **Network-sensitive server selection**: many previous studies have looked at the problem of how to select among a given set of servers the one that best satisfies the user's network performance requirements. Basically, these techniques obtain the

relevant network performance information of all the candidate servers from a network measurement infrastructure such as Remos [59] or GNP [97] and then select the best candidate. For example, some approaches perform active probing to select the best server [12, 36], while others use network properties that have been computed, measured, or observed earlier [4, 114, 116, 117]. Other researchers have also looked at similar resource selection problems in Grid environments [59, 87]. Most of these techniques were developed for specific applications, for example, identifying the Web server that is "closest" to a client, i.e., having the highest-bandwidth or lowest-latency path to the client [4, 12, 83, 116].

Clearly, the NSSS problem can be solved using the existing service discovery and network measurement infrastructures. More specifically, a user first uses a service discovery infrastructure to obtain a list of servers that can provide the required service and then queries a network measurement infrastructure to obtain the relevant network performance information of each candidate. The best server can then be selected according to the particular network performance requirements.

As an example, in the self-configuring multiplayer gaming service, the self-configuration module plays the role of the user in the above approach. Therefore, it will first get a list of gaming servers from a service discovery infrastructure and then obtain each server's latencies to all players in the target session. Finally, it selects the server that minimizes the maximum latency.

Although this *user-side* approach may be suitable for some applications, e.g., Web mirror server selection, there are several problems. The first problem is efficiency. Consider a scenario where there are many self-configuring services, each of which has a self-configuration module that is trying to perform component selection, and many of these services require a commonly used component. There are likely a large number of candidates for this component, and, using the above approach, the same long list of candidates will be returned to all those self-configuration modules asking for the component. Furthermore, each self-configuration module will then ask for network performance information of each of the candidates. Therefore, this approach is inefficient and can potentially result in scalability problem. The second problem is complexity. Many of the tasks performed by the user in this approach are in fact generic and can be performed by the infrastructure. Requiring every user to re-implement these generic functionalities unnecessarily increases the complexity on the user side. Finally, one requirement of this approach is that the user is able to get a list of all servers offered by each provider. However, commercial providers may not be willing to expose all their individual servers, in which case this user-side approach cannot be performed.

An alternative is a *provider-side* approach, in which a user first selects a provider, and the provider then internally applies a network-sensitive selection technique to select one of its servers for the user. Although this approach may be suitable for some applications, e.g., cache selection in content distribution networks, its main problem is that the user cannot specify the selection criteria and does not know what level of performance can be expected.

To address the NSSS problem in the context of self-configuring services, we need an infrastructure that (1) provides the generic functionalities for network-sensitive selection to reduce the overhead and (2) allows the use of user-specific optimization criteria. In this chapter, we describe our solution, *network-sensitive service discovery* (NSSD), which integrates service discovery and network-sensitive server selection functionalities. When looking for a service, a user can specify both functional and network properties through a simple API provided by NSSD, which then returns the optimal service(s) according to the user-specific criteria. In effect we have moved most of the complexity of server selection from the user to the NSSD infrastructure. Providers do not need to expose all their server information to all users since server selection is handled by NSSD, which is trusted by the providers. Moreover, by integrating service discovery and server selection, NSSD can amortize overhead such as collecting network information across multiple users and can also apply distributed solutions. Therefore, we can solve the NSSS problem more efficiently and in a more scalable way.

3.3 Network-Sensitive Service Discovery

In this section, we define the API used for formulating NSSD queries and describe several possible NSSD designs.

3.3.1 NSSD API

The purpose of the NSSD API is to let a user specify the required functional properties and the desired network properties of the target service. NSSD uses the functional properties to find the candidates and then uses the network properties as optimization criteria to select the best candidate using some optimization algorithms. Since traditional service discovery infrastructures already provide APIs for specifying the functional properties, we focus on the specification of the network properties.

To define the NSSD API, we need to determine how much complexity we want to move into the NSSD infrastructure, and how much should remain at the user side. Let us revisit the three self-configuring service examples in Section 3.1.1 and how NSSD may support each scenario. Note that the "user" of NSSD is the self-configuration module of each self-configuring service.

- Multiplayer online gaming (Figure 3.1): One service, i.e., gaming server, is required. The selection criteria is "minimize the maximum latency to players P1, P2, P3, and P4". To support this scenario, NSSD needs to (after getting a list of gaming servers) obtain the latencies between each server and all players, sort all servers according to their maximum latency, and select the best one.

- End system multicast (ESM) (Figure 3.2): A single service, ESM, is required. However, this service requires three "identical servers", i.e., three ESM proxies. The selection criteria is "minimize the sum of latencies between each of P1, P2, P3, and

P4 and the corresponding ESM proxy". To support this scenario, NSSD needs to first recognize that this can be formulated as the *p*-median optimization problem [29]. Then NSSD can obtain the necessary latency information and use, for example, an approximate algorithm to solve the optimization problem.

- Video streaming (Figure 3.3): Two different services, video streaming and video transcoding, are required. The selection criteria is to minimize $(B_1 s + B_2 u)$, where B_1 and B_2 are, respectively, the output bit rates of the streaming service and the transcoding service, and s and u are the latency between the streaming service and the transcoding service and the latency between the transcoding service and the client, respectively. To support this scenario, NSSD needs to have an API that allows a user to specify an arbitrary objective function. In addition, NSSD needs to be able to formulate an optimization problem from the objective and then solve the problem using, for example, a linear programming solver.

We can see that supporting the first scenario is fairly straightforward since most of the tasks involved are generic and do not require a lot of computation power, and the selection criteria only involves a single metric and therefore can be specified using a simple API. On the other hand, supporting the video streaming scenario is difficult and undesirable because (1) a more complex API is required, (2) solving arbitrary global optimization problems for every service is expensive, and (3) if a user has heuristics for the global optimization, NSSD cannot make use of them. Therefore, we decided to draw the line between the first and the third scenarios, i.e., NSSD will allow users to specify a local optimization metric for a single service but not a global objective function for multiple services.

Note that the second scenario is a special case. It is a global optimization problem, but it may be a common form of NSSD queries, so it will be convenient if NSSD supports it. Furthermore, the selection criteria can be specified using the same simple API required for specifying the local optimization metric. Therefore, in our design, NSSD also supports selecting a single service that requires multiple identical servers.

Of course, coordinated selection of multiple components will be necessary for many self-configuring services. Therefore, NSSD should at least provide some "hints" that can help users perform global optimization. Next, we present the basic NSSD API that supports local optimization, and in Section 3.3.2, we discuss how we extend NSSD to support global optimization.

Figure 3.4 shows the basic NSSD API that supports service selection based on local optimization metrics. The argument "num_servers" specifies how many identical servers are required in the returned solution. The solution is returned, for example, in the form of IP addresses. When multiple identical servers are needed in a solution, "mapping" specifies which server is used for each user. Note that the API here only shows the latency, bandwidth, and load metrics. Of course, in practice it may be necessary to support a richer set of metrics, for example, CPU speed, memory size, and other metrics that are useful for Grid applications [87].

Let us use the End System Multicast (ESM) example in Figure 3.2 to illustrate how the

```
Input
  service_properties // service attributes
  target_list        // optimization targets
  num_servers        // num. of identical servers needed
  latency_type       // MAX/AVG/NONE
  latency_constraint // constraint/MINIMIZE/NONE
  bw_type            // MIN/AVG/NONE
  bw_constraint      // constraint/MAXIMIZE/NONE
  load_constraint    // constraint/MINIMIZE/NONE

Output
  solution         // the best solution
  mapping          // target-server mapping
```

Figure 3.4: The NSSD API.

```
Input
  service_properties: "(type=ESMProxy)(protocol=Narada)
(version=1.0)"
  target_list:        "1.1.1.1,2.2.2.2,3.3.3.3,4.4.4.4"
  num_servers:        3
  latency_type:       AVG
  latency_constraint: MINIMIZE
  bw_type:            NONE
  bw_constraint:      NONE
  load_constraint:    NONE

Output
  solution:     "5.5.5.5,6.6.6.6,7.7.7.7"
  mapping:      "0,1,2,2"
```

Figure 3.5: Using the API in the ESM example.

NSSD API is used. The top part of Figure 3.5 shows the NSSD query in this scenario. We first specify that we want to find ESM proxies that are using the Narada protocol version 1.0. The next parameter specify that the selection should be optimized for P1, P2, P3, and P4 in this scenario. The "num_servers" parameter specifies that we want three identical ESM proxies in the returned solution. The remaining input parameters specify that we want to minimize the average latency, and we do not have constraints or preferences on bandwidth and load. Assuming that the best solution is the configuration in Figure 3.2, the result returned by the API is shown in the bottom part of Figure 3.5. NSSD returns the best solution, which consists of three ESM proxies, and the "mapping" specifies that P1

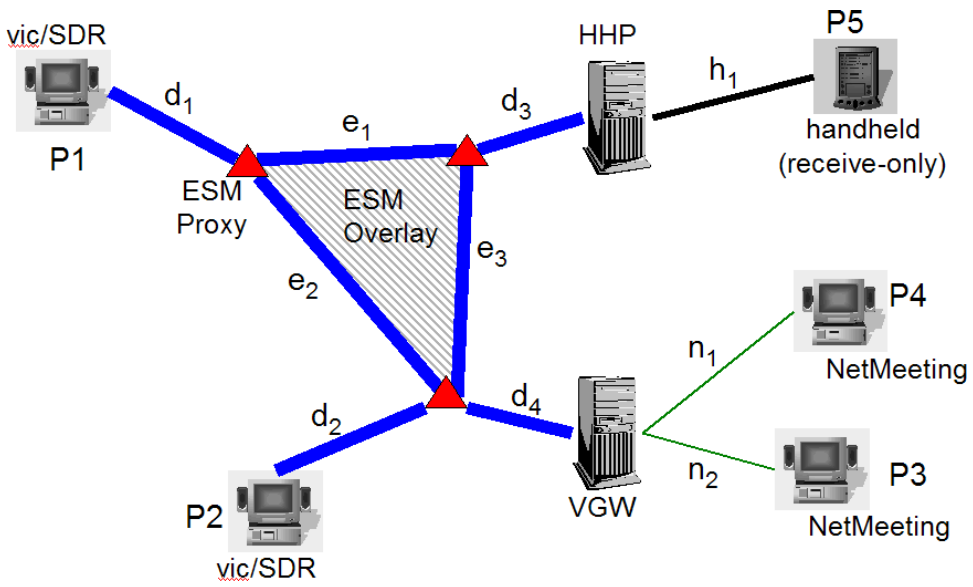


Figure 3.6: Coordinated selection of multiple components for a video conferencing service.

is assigned to ESMP1 at IP address 5.5.5.5, P2 is assigned to ESMP2, and P3 and P4 are assigned to ESMP3.

3.3.2 Supporting coordinated selection

As discussed above, NSSD only supports service selection based on local optimization metrics. However, since coordinated selection of multiple components will be necessary for many self-configuring services, we now discuss how we extend the basic NSSD design to provide "hints" that are useful for users to perform global optimization. First, let us revisit the self-configuring video conferencing service example from Chapter 2 and take a closer look at the challenges of coordinated selection.

Suppose users P1 to P5 in Figure 3.6 want to establish a video conferencing session: P1 and P2 have MBone conferencing applications vic/SDR (VIC), P3 and P4 use NetMeeting (NM), and P5 uses a receive-only handheld device (HH). The self-configuring service determines that the following components are needed: a video conferencing gateway (VGW) for protocol translation and video forwarding between VIC and NM, a handheld proxy (HHP) for P5, and an End System Multicast (ESM) overlay consisting of three ESM proxies (ESMPs) for wide-area multicast. As discussed earlier, the provider of this self-configuring video conferencing service may want to minimize the total network resource usage of the service configuration. Therefore, the provider may specify in its service recipe that the service-specific optimization objective for component selection is to minimize the

following function:

$$W_1(d_1+d_2+d_3+d_4+\frac{2}{3}(e_1+e_2+e_3)) + W_2h_1 + W_3(n_1+n_2) \quad (3.1)$$

where W_1, W_2, and W_3 are weights for the three flow types (multicast, handheld, and NetMeeting), reflecting the difference in bandwidth consumption, and the other variables are the latencies between nodes as depicted in Figure 3.6.

This example raises the following challenge: the selection of the different components is *mutually dependent*, i.e., to find the optimal solution, all components need to be selected together. Unfortunately, selecting all the components together may be too expensive. For example, suppose there are n VGWs, n HHPs, and n ESM proxies available. To find the optimal configuration, we need to look at roughly n^5 possible configurations, which is only feasible when n is small.

Since NSSD supports local optimization, a simple heuristic for solving this optimization problem using NSSD is to select each component using local optimizations and then combine the locally optimal solutions into a global solution. For example, the self-configuring service can use NSSD to find the VGW that minimizes (n_1+n_2), the HHP that minimizes h_1, and then the ESMPs that minimizes $(d_1+d_2+d_3+d_4)$ given the previously found VGW and HHP.

Of course, the problem with this heuristic is that a combination of locally optimal solutions may not be globally optimal. To improve the performance of coordinated component selection, we propose a *hybrid* heuristic in which the local optimization supported by NSSD is used to reduce the search space of the optimization problem, and then global optimization is performed on the reduced search space. We describe the hybrid heuristic below.

Hybrid heuristic. The hybrid heuristic is based on the observation that in optimization problems similar to the above case, although the locally optimal candidate for a component may not be the globally optimal one, selecting a "locally good" candidate is more likely to improve the global optimality than selecting a "locally bad" candidate. The reason is that the metric used in the local optimization is part of the global objective, and as a result, for example, if we minimize the local metric, the global objective is likely to decrease as well. Therefore, the hybrid heuristic is that when solving a global optimization problem, instead of looking at all possible combinations, we can reduce the search space by considering only a small number of "locally good" candidates for each component. For example, in the video conferencing scenario above, we can first the best n VGWs (i.e., minimizing n_1+n_2) and the best m HHPs (i.e., minimizing h_1). Then, for each of the nm possible VGW/HHP combinations, we find the optimal set of ESMPs, and we get a global solution. Therefore, we have a set of nm global solutions, and we can use Function (3.1) to evaluate them and select the best global solution in this set.

Input

```
service_properties // service attributes
target_list        // optimization targets
num_servers        // num. of identical servers needed
num_solutions      // num. of solutions needed
latency_type       // MAX/AVG/NONE
latency_constraint // constraint/MINIMIZE/NONE
bw_type            // MIN/AVG/NONE
bw_constraint      // constraint/MAXIMIZE/NONE
load_constraint    // constraint/MINIMIZE/NONE
```

Output

```
solution   // the best candidate(s)
mapping    // target-server mapping(s)
fitness    // the ‘‘score(s)’’ of the candidate(s)
```

Figure 3.7: The extended NSSD API.

In order to support this hybrid heuristic, NSSD needs to provide the “best-n-solutions” feature, i.e., it must be able to return *n* locally good candidates instead of only the best one. To provide this feature, first, the NSSD API needs to be extended, and the extended API is shown in Figure 3.7. The additional parameters include “num_solutions” and “fitness”. The “num_solutions” parameter lets a user specify how many locally good solutions should be returned, and “fitness” represents the values of the local metric for returned solutions. The “fitness” parameter is necessary for the user to perform global optimization on the reduced search space.

Next, we discuss how the NSSD infrastructure supports the best-n-solutions feature.

Supporting best-n-solutions. In order to provide this best-n-solutions feature, NSSD needs to determine the best set of *n* candidates among all candidates. One simple algorithm here is to sort all candidates according to the local optimization metric and return the best *n* candidates. One potential problem with this algorithm is that the returned candidates may be “redundant” for the later global optimization. For example, suppose that NSSD needs to return the best two VGWs among the three in Figure 3.8. Using the simple algorithm above, NSSD returns VGW1 and VGW2, which are closest to P3 and P4. However, since our ultimate goal is to minimize Function (3.1), VGW3 (which is very close to ESMP1) is likely to be a better candidate especially if the weight W_1 is relatively high. One important observation here is that, since VGW1 and VGW2 are very close to each other, returning both is redundant because they will have similar impact on the global optimality measured by Function (3.1). Therefore, a second algorithm for selecting the best *n* candidates to return is to select a set of candidates that are more evenly distributed so that they are more “representative” of the network characteristics of all candidates.

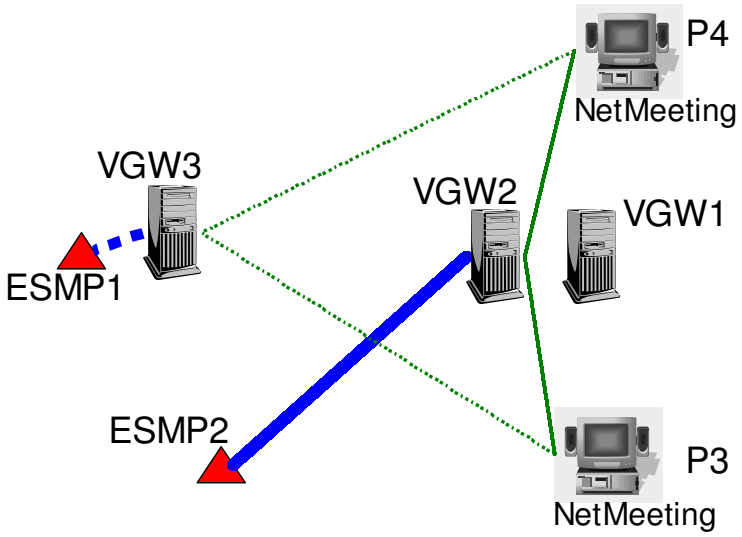


Figure 3.8: An example of "redundant" candidates.

We believe using the hybrid heuristic for global optimization can yield a reasonably good global solution and allow us to adjust the trade-off between global optimality and optimization cost by controlling the size of the search space for the final global optimization. In the video conferencing example, if *n* and *m* (the numbers of VGWs and HHPs returned by NSSD, respectively) are set to 1, the resulting solution is simply a combination of locally optimal solutions. On the other hand, if *n* and *m* are the total numbers of VGWs and HHPs, respectively, we are in fact performing an exhaustive search in the complete search space, and we can find the globally optimal solution at a higher cost. In Section 3.4.4, we discuss how the best-n-solutions feature is implemented, and in Section 3.5.4, we use the video conferencing example to evaluate the effectiveness of the hybrid heuristic and compare the different algorithms for selecting the best *n* candidates.

3.3.3 A Simple NSSD Query Processor

In this section, we describe a simple NSSD query processor that formed the basis for our prototype implementation. We discuss several alternative designs in the next section.

As shown in Figure 3.9, a simple NSSD query processor (QP) can be built on top of a service discovery infrastructure (e.g., the Service Location Protocol [61]) and a network measurement infrastructure that can provide network performance information such as latencies between nodes. When the QP module receives an NSSD query (step 1 in Figure 3.9), it forwards the functional part of the query to the service directory (step 2). The directory returns a list of candidates that match the functional properties specified in the query (step 3). Then the QP module retrieves the necessary network information (e.g., the latency between each of the candidate and user X) from the network measurement infrastructure (step 4). Finally, the QP module computes the best solution as described below

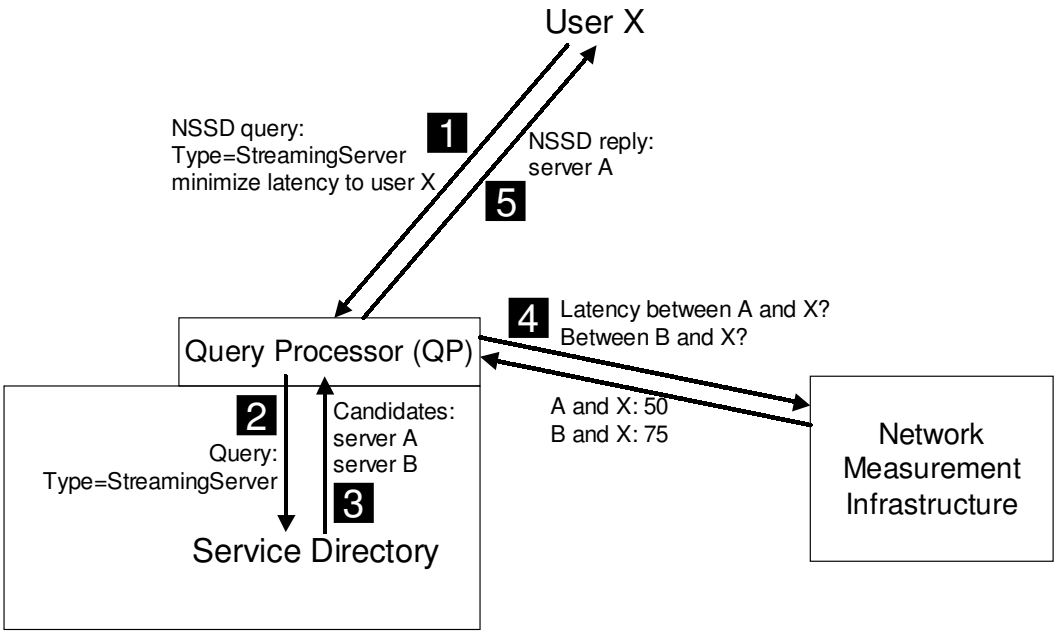


Figure 3.9: Handling an NSSD query.

and returns it to the user (step 5).

One benefit of integrating service discovery and network-sensitive server selection into the infrastructure is that caching can be used to improve performance and scalability. When NSSD gets requests from many users, the cost of collecting eligible candidates, network performance information, and server load information can be amortized if the same information is needed for multiple requests. Furthermore, network nodes that are close to each other should have similar network properties. Therefore, when handling requests, NSSD can aggregate the nodes from different requests to reduce the amount of network performance information required. For example, the same latency information can be used for all users in an address prefix, all servers in a single cluster should have similar network properties, and so on. Such aggregation can increase the effectiveness of caching/amortization. Finally, the caching/amortization is more effective when network performance information that is more expensive to obtain is required. For example, eliminating the need for 100 users to obtain latency information to the same server using "ping" is obviously a much smaller saving than eliminating the need for 100 users to obtain bandwidth information to the same server using active bandwidth probing.

Selecting the best solution(s) requires NSSD to solve a fairly general optimization problem. Through the NSSD API described earlier, a user can specify many different combinations of (1) constraints and preferences on three metrics (latency, bandwidth, and load), (2) how many identical servers are needed in a solution, and (3) how many solutions are needed. The QP computes the solution for a query as follows. First, the QP applies any constraints in a query to eliminate any ineligible candidates. Then, the preferences in a

query can be formulated as an optimization problem. If there is only a single preference, the QP can simply sort the candidates accordingly. If there are multiple preferences (e.g., minimize load and minimize latency), there may not be any candidates that satisfy all preferences. One possible solution is to have the QP define an order among the preferences (e.g., in the order they appear in the query) and sort the candidates accordingly. Finally, if a query requests multiple identical servers in a solution (e.g., requesting 3 ESM proxies for 4 users), the optimization problem can be cast as *p*-median, *p*-center, or set covering problems [30], which are more expensive to solve.

3.3.4 Alternative Implementations

The NSSD design outlined above is a simple integration of service discovery and network measurement. It could be implemented as a directory service that gathers all service information from services around the Internet and uses an internal or external network measurement mechanism to support network-sensitive queries. Today's web search engines such as Google [56] show that this approach is probably feasible albeit expensive. However, if service attributes are highly dynamic, it may be difficult to keep a centralized large-scale directory up-to-date.

Alternatively, NSSD can also be built on a distributed architecture. Here we list a number of possible approaches.

- In application-layer anycasting [136], each service is represented by an anycast domain name (ADN). A user submits an ADN along with a server selection filter (which specifies the selection criteria) to an anycast resolver, which resolves the ADN to a list of IP addresses and selects one or more from the list using the filter. To support the NSSD functionalities, the ADN and resolvers can potentially be extended to allow users to specify the desired service attributes, and the filter can be generalized to support more general metrics.

- Distributed routing algorithms are highly scalable, and they can, for example, be used to find a path that satisfies certain network properties and also includes a server with certain available computational resources [68]. A generalization of this approach can be combined with a service discovery mechanism to handle NSSD queries.

- A Content Discovery System (CDS) based on hashing is described in [51]. The system uses a distributed hash table (such as Chord [124]) to allow publishers and subscribers to find each other in rendezvous points based on common attribute-value pairs, which may be dynamic. Therefore, it can also be used as a service discovery infrastructure, and one can incorporate network sensitivity into the query resolution phase of the system so that the returned matches satisfy certain network properties specified in the query.

```
(|(&(game=Half-Life)(mod=Counter-Strike)
    (version>=1.5)(load<=10))
  (&(x-NSSD-targets=1.01,2.02;3.03,4.04)
    (x-NSSD-maxlatency=minimize)))
```

Figure 3.10: A sample filter for the game example.

3.4 Implementation

We describe a prototype NSSD based on Service Location Protocol (SLP) [61] and Global Network Positioning (GNP) [97]. We have experimented with versions of our NSSD implementation on the ABone [1], Emulab [39], and PlanetLab [106] testbeds.

3.4.1 Extending SLP

Our prototype implementation of NSSD is based on OpenSLP [102], an open-source implementation of the Service Location Protocol. In SLP, available services register their information such as location, type, and attributes with a Directory Agent (DA), and users looking for services send queries to the DA specifying the type and attributes of the desired services. Service types and attributes are well known so that a user knows what to ask for. SLP query specification is quite general: attributes can be specified as an LDAPv3 search filter [65], which supports, for example, logical operations, inequality, and substring match. Therefore, a user query includes a service type (e.g., "GameServer") and a filter, e.g., "(&(game=Half-Life)(mod=Counter-Strike)(version>=1.5))". We believe this query representation is sufficiently general to support NSSD queries as defined by the API in Section 3.3.1.

In order to support NSSD queries, we extended the semantics of the SLP filter to include a set of special attributes, representing the parameters described in Figure 3.4. For example, suppose a user wants to find a game server that (1) matches certain service attributes, (2) is serving at most ten sessions, and (3) minimizes the maximum latency for the two players whose GNP coordinates are "1.01,2.02" and "3.03,4.04", respectively. These parameters can be specified by the filter shown in Figure 3.10.

The original SLP API returns a list of "service URLs" [62]. To return additional information about each returned solution such as the "fitness" and "mapping", we append the information to the end of the URLs. For example, to return the mapping, the DA can return the following service URL: "service:GameServer://192.168.0.1;x-NSSD-mapping=0,0,0,0".

Since server load is also a service attribute, each server's registration includes the load attribute (e.g., "load=0.5"). When conducting "live" experiments (i.e., involving applications running on actual network hosts), we need mechanisms to dynamically update the load value of each server. Our live experiments are developed and conducted on the PlanetLab wide-area testbed [106], which allows us shell access on about 70 hosts at nearly 30

sites. We implemented a "push-based" load update mechanism: servers push their load information (in the form of a registration update) to the DA. In our evaluation, we look at how the frequency of load update affects the performance of the server selection techniques.

3.4.2 Network Measurement

A network measurement infrastructure provides a way for users to obtain network information. A number of such infrastructures have been proposed, for example, IDMaps [46] and Global Network Positioning (GNP) [97]. Since it is in general much harder and more expensive to estimate the bandwidth between two network nodes, most of these infrastructures only provide latency information. Therefore, currently, in our prototype NSSD infrastructure, we focus on dealing with the latency metric. As mentioned earlier, more metrics can be supported by adding the appropriate measurement infrastructures that provide such metrics and extending the API to support the specification of such metrics, and the benefits of caching/amortization will be more evident when the required metrics are more expensive to obtain.

In our implementation, we use GNP as the network measurement infrastructure to provide latency (round trip time) information between two network nodes. The key idea behind GNP is to model the Internet as a geometric space using a set of "landmark nodes", and each network host can compute its own coordinates by probing the landmarks. It is then straightforward to compute the distance (latency) between two hosts given their coordinates. Since the PlanetLab testbed is our current platform, we use the GNP approach to obtain a set of coordinates for every PlanetLab node.

In the GNP model, each node computes its own coordinates. Therefore, in our implementation, we use GNP coordinates as a service attribute, i.e., when a server register with the SLP DA, the registration includes the coordinates of the server node. When a user specifies the list of optimization targets in a query (see the API in Section 3.3.1), each target is specified in the form of GNP coordinates. When a QP asks for a list of candidates, the DA returns the list along with the coordinates of each candidate. The advantage of this design is that since the coordinates are computed off-line, and the latency information can be derived from the coordinates directly, the cost of querying the network measurement infrastructure at runtime is eliminated. A QP can simply use the candidates' and the targets' coordinates to solve the particular optimization problem specified by a user.

3.4.3 Selection Techniques

The NSSD API defined in Section 3.3.1 can be used to specify a wide range of combinations of constraints and preferences, and these combinations translate into different techniques for network-sensitive selection. Below we list the selection techniques that are supported by the prototype Query Processor and used in our evaluation (k denotes the number of identical servers needed in a solution, and n denotes the number of locally good solutions needed).

- "$k = 1$, $n = 1$, minimize load" (**MLR**): find the server with the lowest load. If there are multiple servers with the same load, select one randomly.

- "$k = 1$, $n = 1$, minimize maximum latency" (**MM**): find the server that minimizes the maximum latency to the specified target(s).

- "$k = 1$, $n = 1$, load constraint x, minimize maximum latency" (**LCxMM**): among the servers that satisfy the specified load constraint (load $\leq x$), find the one that minimizes the maximum latency to the specified target(s).

- "$k = 1$, $n = 1$, minimize load, minimize maximum latency" (**MLMM**): find the server with the lowest load, and use maximum latency as the tiebreaker.

- "$k = p$, $n = 1$, minimize average latency" (**PMA**): find a set of p servers and assign each target to a server so that the average latency between each target and its assigned server is minimized (i.e., the p-median problem [29]).

- "Random" (**R**): randomly select a server.

Note that these techniques all return only the best solution. Next, we describe the algorithms we implemented that return the best n solutions.

3.4.4 Best-n-Solutions

As discussed in Section 3.3.2, we implemented two algorithms for selecting the best n solutions.

- "Pure-local": NSSD sorts the candidates according to the local optimization metric and then return the best n. This algorithm is simple but may return candidates that are "redundant" for the global optimization.

- "Clustering-f": In order to avoid redundant candidates and to return those that can better "represent" the network characteristics of all candidates, we implemented the clustering-f algorithm (f is the *clustering factor*) as follows. Let N be the number of all candidates, and $N_c = \frac{N}{f}$. First, NSSD applies a clustering algorithm to classify all candidates into N_c clusters based on the GNP coordinates of the candidates. We use the direct k-means clustering algorithm [3] in our prototype. On the other hand, NSSD also sorts the candidates using the local optimization metric, and the resulting list is $\{c_1, c_2, \ldots, c_N\}$, where c_1 is the best. Then NSSD selects the n candidates in two phases. (1) Starting with c_1, NSSD examines each candidate in the list and selects a candidate if it belongs to one of the N_c clusters from which no other candidate has been selected. If $(n \leq N_c)$, the second phase is not necessary. (2) If $(n > N_c)$, the NSSD again starts with c_1 and selects each candidate that has not been selected in the first phase until n candidates have been selected.

3.5 Evaluation

We present the results of experiments that quantify how well our NSSD prototype can support two sample applications. First, we use a multiplayer gaming service to illustrate the importance of network-sensitive server selection and to show the relative performance of the different selection mechanisms in our prototype. Next we use the video conferencing example of Section 3.3.2 to show the trade-offs between local and global optimizations, and we compare the performance of the two algorithms described in Section 3.4.4. Finally, we also present the computational overhead of the NSSD prototype.

The experiments for the gaming service scenario are conducted on the PlanetLab wide-area testbed [106]. The experiments using the video conferencing scenario consist of two sets of simulations. The first set is based on latency measurement data from the Active Measurement Project at NLANR [99], and the second is based on data obtained from the GNP project.

3.5.1 Importance of Network Sensitivity

In the first set of experiments, we look at how network sensitivity can help improve application performance. We implemented a simple "simulated" multiplayer game client/server: a number of game clients join a session hosted by a game server, and every 100ms each client sends a small UDP packet to the hosting server, which processes the packet, forwards it to all other participants in the session, and sends an "ACK" packet back to the sender. Our performance metric for the gaming service is the maximum latency from the server to all clients since we do not want sessions in which some players are very close to the server while others experience high latency. Note that the latency metric is measured from the time that a packet is sent by a client to the time that the ACK from the server is received by the client.

The goal of this set of experiments is to evaluate the role of network performance in the server selection process. Therefore, we minimized the computation overhead in the game server to ensure that the server computation power, including the number of servers, does not affect game performance. We compare four different scenarios. First, we consider two scenarios, each of which has 10 distributed servers selected from PlanetLab nodes. The **MM** and **R** techniques are applied in the two scenarios, respectively. For comparison, we also consider two centralized cluster scenarios, **CAM** and **CMU**, each of which has a 3-server cluster, and a random server is selected for each session. In the CAM scenario, the cluster is located at the University of Cambridge; in the CMU scenario, the cluster is at Carnegie Mellon University. The user machines are randomly selected from 50 PlanetLab nodes.

Figure 3.11 shows the average maximum latency (of 100 sessions) as a function of the number of participants in each session. MM consistently has the lowest maximum latency, confirming that network-sensitive service discovery can successfully come up with the best solution. More specifically, the results illustrate that being able to choose from a set of distributed servers (MM outperforms CMU and CAM) in a network-sensitive fashion (MM

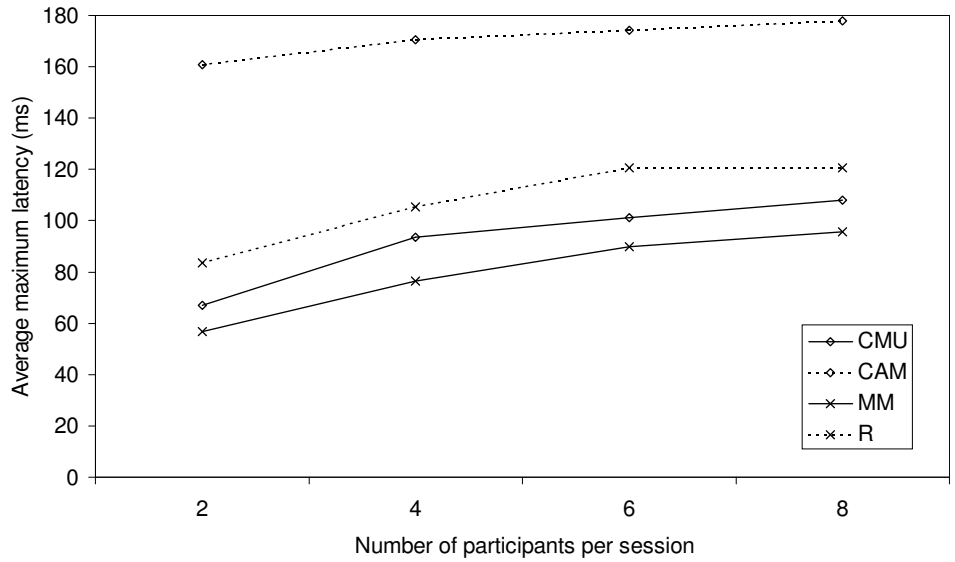


Figure 3.11: Central cluster vs. distributed servers in a multiplayer game application.

outperforms R) is a win. Interestingly, having distributed servers is not necessarily better than a centralized solution. In our case, a centrally located cluster (CMU) outperforms a distributed network-"insensitive" solution (R).

3.5.2 Different Selection Techniques

While the focus of this paper is not on new selection techniques, we next show how NSSD can easily support different techniques. We use the simulated game applications to compare the effectiveness of different selection techniques in NSSD. We look at the techniques **MLR**, **MM**, **LC3MM**, **MLMM**, and **R** described in Section 3.4.3. (LC3MM has load constraint 3, which means that only servers serving no more than three sessions are eligible.) Ten nodes at 10 different sites are used as game servers, 50 other nodes are game clients, and there are four randomly selected participants in each session. There are on average 10 simultaneous sessions at any time, and the server load information in the DA's database is updated immediately, i.e., each server's load value is updated whenever it changes (we relax this in the next section). We ran measurements for different per-packet processing overhead, which we controlled by changing the number of iterations of a computationally expensive loop. The unit of the processing overhead corresponds roughly to 1% of the CPU when hosting a 4-participant session. For example, on a server that is hosting a single 4-participant session with 10 units of per-packet processing overhead, the CPU load is about 10%.

Figure 3.12 shows the average maximum latency (of 200 sessions) as a function of

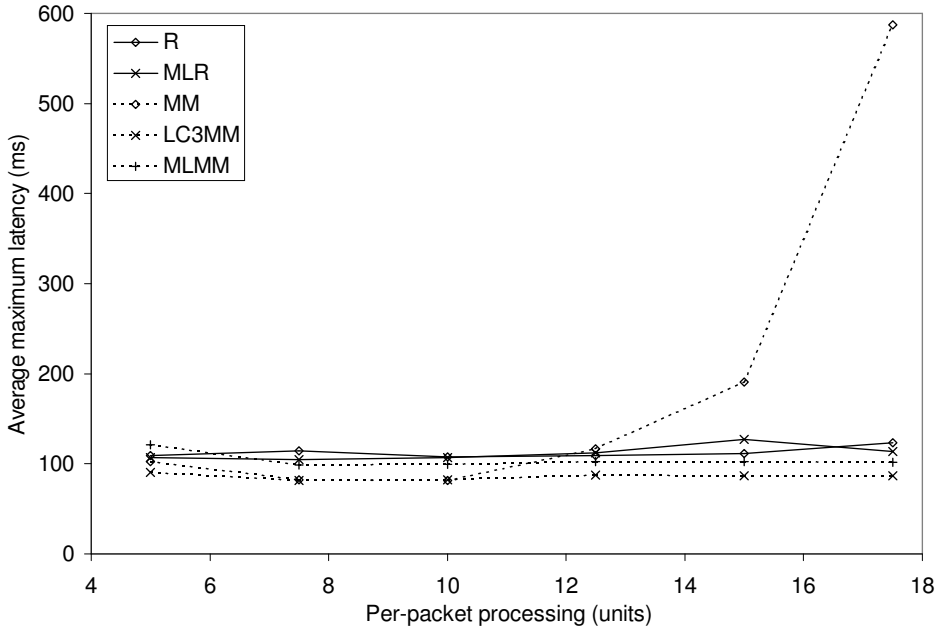


Figure 3.12: Effect of per-packet processing: average maximum latency.

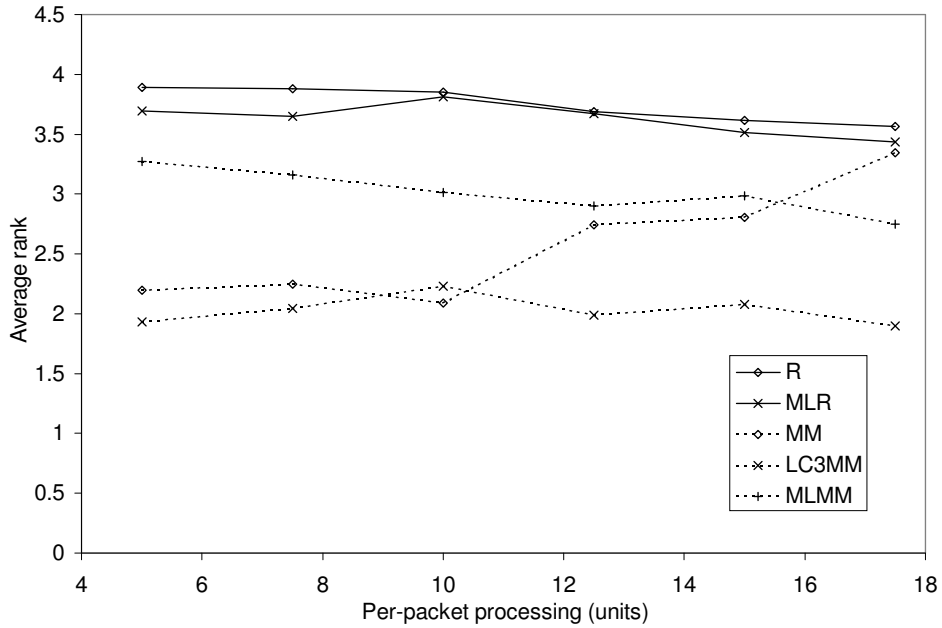


Figure 3.13: Effect of per-packet processing: average rank.

the per-packet processing cost. R and MLR have almost identical performance, which is expected since in this case MLR evenly distributes sessions based on the number of gaming

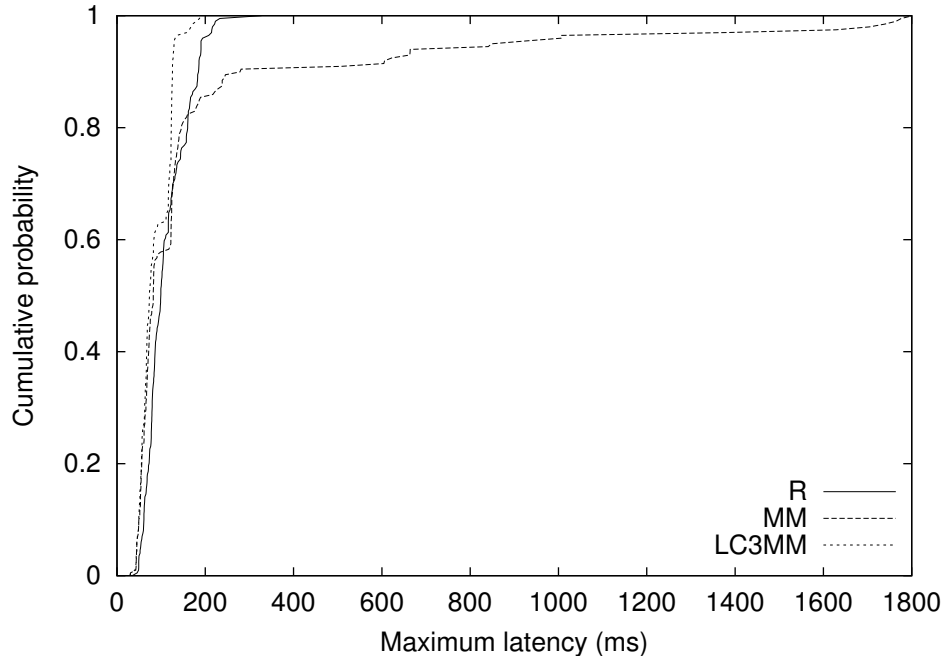


Figure 3.14: Cumulative distribution under 15 units per-packet processing.

sessions on each server. MM and LC3MM are also almost identical when the per-packet processing is low. However, the performance of MM degrades rapidly at higher loads, while LC3MM remains the best performer throughout this set of experiments.

Figure 3.13 shows the results from a different perspective: it shows the average "rank" of the five techniques, i.e., for each session, we rank the techniques from 1 to 5 (best to worst), and we then average over the 200 sessions. We see that although the rank of MM gets worse for higher loads, it is still better than R and MLR at the highest load, despite the fact that its average is much worse than those of R and MLR. The reason can be seen in Figure 3.14, which compares the cumulative distributions of the maximum latency of R, MM, and LC3MM for the case of 15 units per-packet processing. It shows that MM in fact makes many good selections, which helps its average rank. However, the 10% really bad choices (selecting overloaded servers) hurt the average maximum latency significantly. In contrast, LC3MM consistently makes good selections, which makes it the best performer both in terms of average maximum latency and rank.

3.5.3 Effect of Load Update Frequency

In the previous set of experiments, the server load information stored in the DA's database is always up-to-date, which may not be feasible in practice. We now look at how the load update frequency affects the performance of the selection techniques. The experimental set up is the same as that in the previous section, except that we fix the per-packet processing to

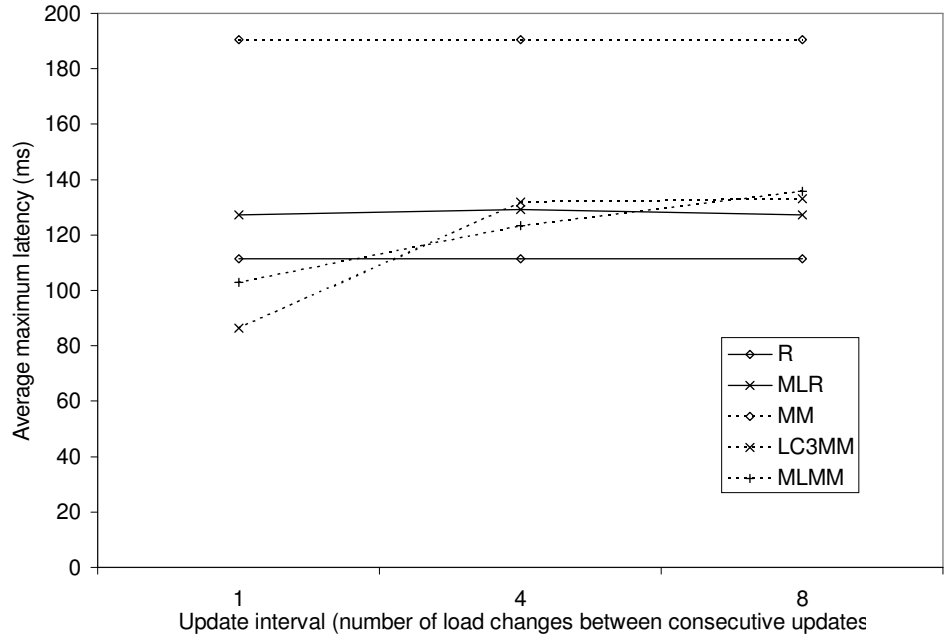


Figure 3.15: Effect of load update frequency: average maximum latency.

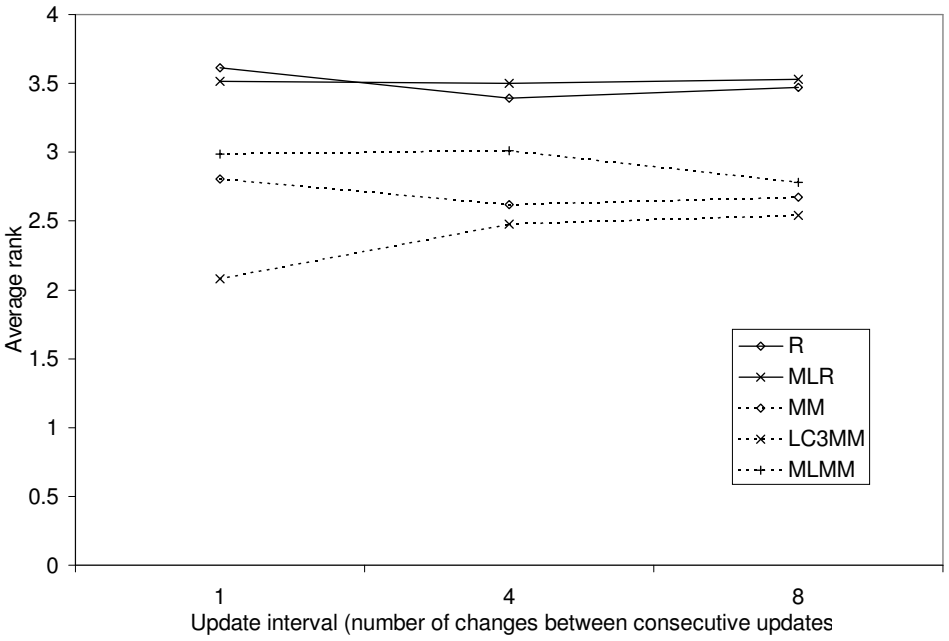


Figure 3.16: Effect of load update frequency: average rank.

15 units and vary the load update interval. The load update interval is defined as the number of system-wide "load changes" between two consecutive updates of the load information

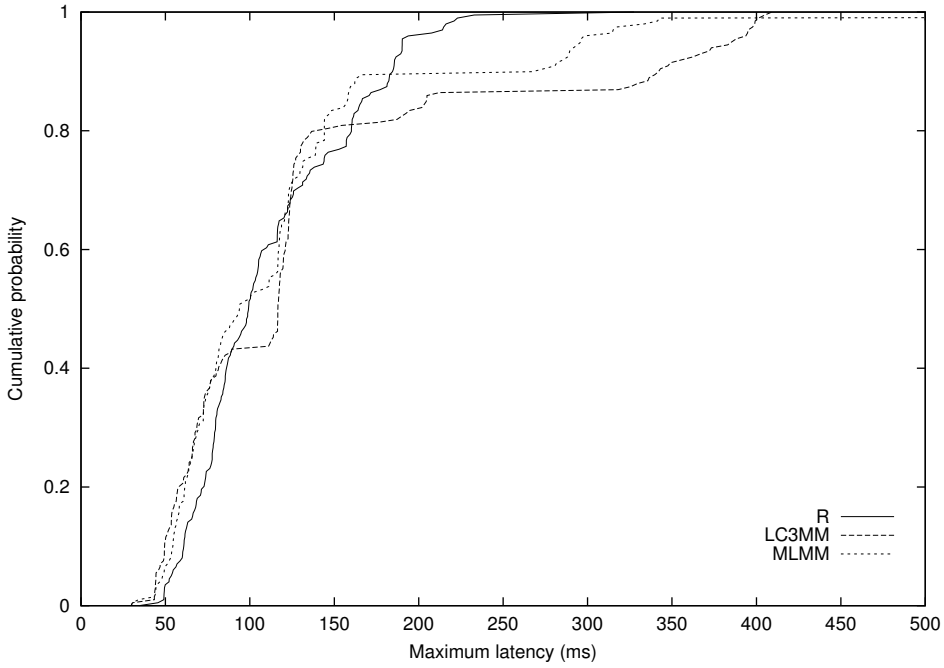


Figure 3.17: Cumulative distribution of maximum latency with update interval of 8.

on the DA. Note that load changes include both establishing a new session and terminating an old session. We experimented with three different update intervals: 1 (i.e., immediate updates, as in the previous set of experiments), 4, and 8. Since there are on average 10 simultaneous sessions at any time, an update interval of 8 means that when an update occurs, 40% of the active sessions have been replaced since the previous update.

Figure 3.15 shows the average maximum latency for the different techniques using different update intervals. Since R and MM are not affected by load, we show the data from the previous set of experiments for comparison. We see that as the update interval becomes longer, the performance of both LC3MM and MLMM degrades significantly since they make decisions based on stale load information. They are worse than R under longer update intervals. However, in Figure 3.16, LC3MM and MLMM consistently have a better average rank than R. The reason is that when the load update interval is long, LC3MM and MLMM occasionally make really bad choices, which greatly affect the average maximum latency. For example, Figure 3.17 shows that when the update interval is 8, LC3MM and MLMM outperform R in most of the sessions, but in the worst 10% to 20% of the sessions, LC3MM and MLMM perform significantly worse than R.

3.5.4 Local Optimization vs. Global Optimization

In this set of experiments, we use NSSD in the video conferencing scenario described in Section 3.3.2. As depicted in Figure 3.6, we need to find a video conferencing gateway

(VGW), a handheld proxy (HHP), and a set of ESM proxies. We use the heuristic described in Section 3.3.2: first, we ask NSSD to select the best *n* VGWs using the pure-local algorithm with average latency as the local optimization metric (see Section 3.4.4). Similarly, we ask NSSD to select the best *m* HHPs. For each of the *nm* VGW/HHP combinations, we ask NSSD to find the optimal set of ESM proxies using the PMA technique (see Section 3.4.3). Finally, we evaluate the resulting *nm* global solutions using Function 3.1 and select the best one.

Since we are interested in the global optimality (as defined by Function 3.1) of the resulting service instances, we do not need to run the various components on actual machines. Therefore, our experiments are based on simulations using the latency measurement data from the NLANR Active Measurement Project [99]. The data consists of round trip time measurements from each of 130 monitors (mostly located in the U.S.) to all the other monitors. We process the data to generate a latency matrix for 103 sites, and then in the simulations we randomly select nodes from these sites to represent users and various components. Next, we present the results from three sets of experiments and compare them.

Weighted-5

In the weighted-5 set, we use weights 5.0, 2.0, and 1.0 for W_1, W_2, and W_3, respectively. We select 40 random nodes as client nodes, 5 random nodes as VGWs, 5 as HHPs, and 5 as ESM proxies (these 4 sets are disjoint). We then generate 100 sessions by selecting 5 participants (2 vic/SDR, 1 handheld, and 2 NetMeeting) from the client nodes for each session. For each session, we vary the values of *n* and *m* from 1 to 5 and compute the corresponding global solutions. This process (node selection/session generation/simulation) is repeated 20 times, resulting in 20 simulation configurations. The performance metric is "relative global optimality", which is defined as the value of Function 3.1 for a solution divided by the value for the globally optimal solution. For example, a solution with relative global optimality 1.25 is 25% worse than the globally optimal solution.

Let us first look at all 100 sessions in a typical simulation configuration. We experimented with four different settings for (n,m): (1,1), (2,2), (3,3), and (4,4), and the results from a typical configuration are plotted in Figure 3.18. For each (n,m) setting, the sessions are sorted according to their relative global optimality (rank 1 to 100, i.e., best to worst). When (n,m) is (1,1), we are able to find the globally optimal solution for 27 sessions, and the worst-case relative global optimality is 1.19. We can see that as we increase the size of the search space using the best-n-solutions feature of NSSD, we not only increase the chance of finding the globally optimal solution but also improve the worst-case performance. Therefore, the result demonstrates the effectiveness of the hybrid heuristic for the global optimization problem in coordinated component selection.

Next, we want to look at the results for all sessions in all 20 simulation configurations using all (n,m) settings. The average relative global optimality result is shown in Figure 3.19. Each data point is the average of 20 configurations, each of which is the average of 100 sessions. We decided to present the average (mean) relative global optimality for each (n,m) setting instead of the median value because as we can see in Figure 3.18, the

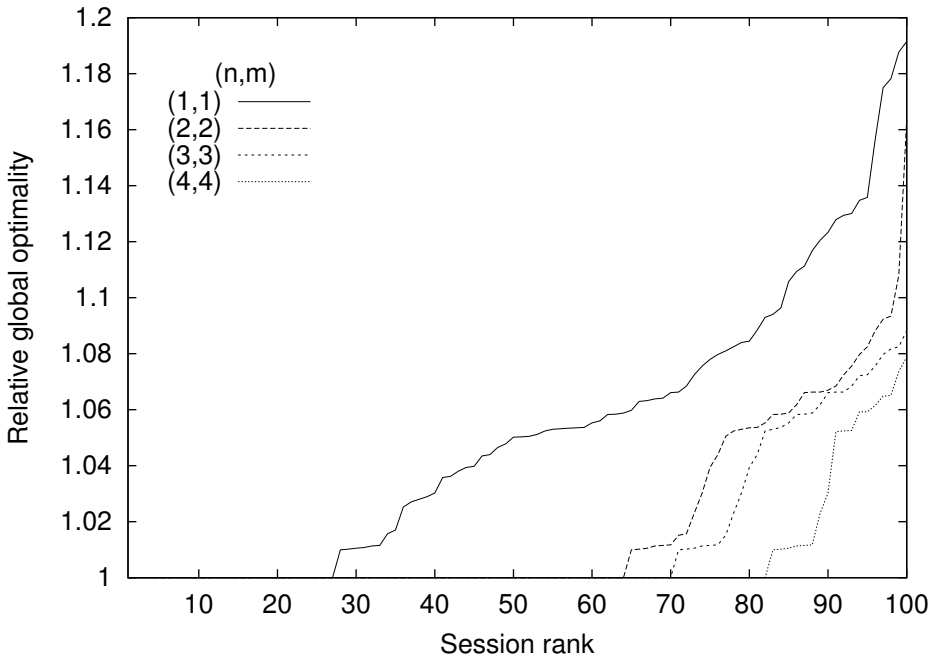


Figure 3.18: Relative global optimality for sessions in a typical weighted-5 configuration.

median value often cannot show the difference in the performance of different settings. When (n,m) is (1,1), i.e., when we are simply using the combination of locally optimal solutions as the global solution, the resulting solution is on average 13.7% worse than the globally optimal solution. When we use (2,2) for (n,m), we are performing an exhaustive search in 16% of the complete search space, and the resulting solution is 5.3% worse than the globally optimal solution.

Weighted-25

The weighted-25 setup is the same as weighted-5 above except that we use 25 VGWs, 25 HHPs, and 25 ESM proxies for this set. Figure 3.20 shows the average relative global optimality for this set of experiments. When we set (n,m) to (1,1) and (10,10), the average relative global optimality of the resulting solution is 1.279 and 1.035, respectively (i.e., 27.9% and 3.5% worse than the globally optimal solution).

Unweighted-25

The unweighted-25 set is the same as weighted-25 above except that the weights W_1, W_2, and W_3 in the global optimization function (Function 3.1) are all set to 1.0. The average relative global optimality result is shown in Figure 3.21. When we set (n,m) to (1,1) and (10,10), the resulting solution is on average 5.9% and 0.1% worse than the globally optimal solution, respectively.

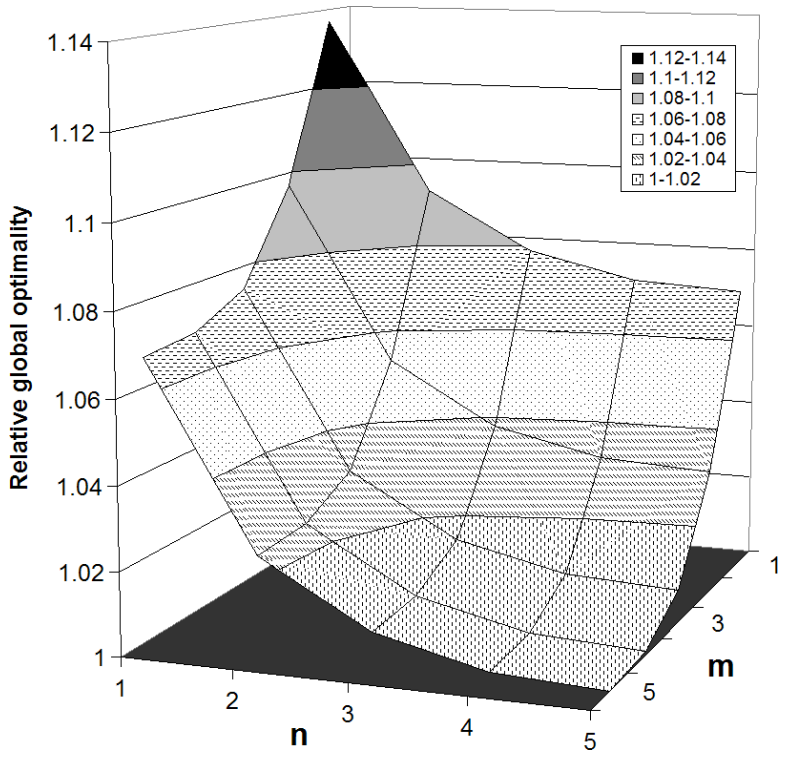


Figure 3.19: Relative global optimality for weighted-5.

Comparison

A comparison between weighted-5 and weighted-25 illustrates a few points. First, although using the combination of locally optimal solutions can greatly reduce the cost of solving the selection problem, it can lead to bad solutions (in terms of global optimality). Second, using the best-n-solutions feature of NSSD is effective, as we can significantly improve the global optimality of the resulting solution by searching in a relatively small space. Third, the performance at (1,1) seems to degrade as the complete search space becomes larger (1.137 in weighted-5 and 1.279 in weighted-25). On the other hand, the effectiveness of the hybrid heuristic seems to increase with the size of the complete search space, e.g., when searching only 16% of the complete search space (i.e., when we set (n,m) to (2,2) and (10,10) in weighted-5 and weighted-25, respectively), the improvement in weighted-25 is greater than in weighted-5 (27.9%→3.5% vs. 13.7%→5.3%).

When comparing weighted-25 with unweighted-25, we see that in unweighted-25, the performance at (1,1) is much better than that in weighted-25 (1.059 vs. 1.279), and increasing n and m improves the global optimality much faster than it does in weighted-25, e.g., 5.9%→1.0% vs. 27.9%→12.6% when (n,m) is (4,4). An intuitive explanation of this significant difference between the weighted and the unweighted configurations is that, in this video conferencing example, the unweighted global optimization function is actually quite close to the sum of the local optimization metrics used to select the individual services. As

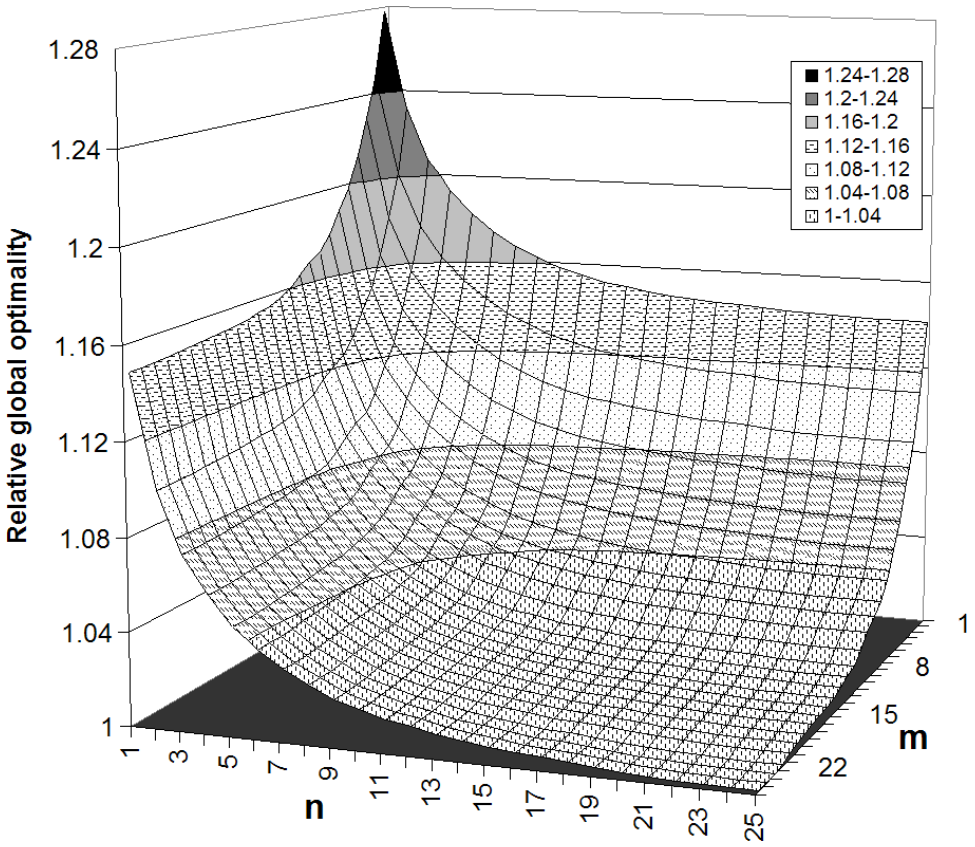


Figure 3.20: Relative global optimality for weighted-25.

a result, a "locally good" candidate is very likely to also be "globally good". On the other hand, in the weighted configuration, the global optimality of a solution is less dependent on the local optimality of each component, and therefore, we need to expand the search space more to find good global solutions.

To verify this explanation, we look at how likely each particular VGW/HHP combination results in the globally optimal solution. Specifically, for each of the 2000 sessions, we look at which VGW/HHP combination (according to their local ranks, e.g., the combination of the i-th ranked VGW and the j-th ranked HHP) results in the globally optimal solution. Then we aggregate the results and present, for all $1 \leq i \leq 25$ and $1 \leq j \leq 25$, the fraction of time that the combination of the i-th ranked VGW and the j-th ranked HHP results in the globally optimal solution. Figure 3.22 shows that in unweighted-25, nearly 30% of the time simply using the best VGW (rank 1) and the best HHP (rank 1) results in the globally optimal solution. Similarly, about 11% of the time using the 2nd-ranked VGW and the 1st-ranked HHP results in the globally optimal solution, and so on. In fact, in unweighted-25, the vast majority of globally optimal solutions involve the best few VGWs and HHPs. On the other hand, Figure 3.23 shows the results for weighted-25. Although using the combination of the 1st-ranked VGW and the 1st-ranked HHP is still more likely to result in

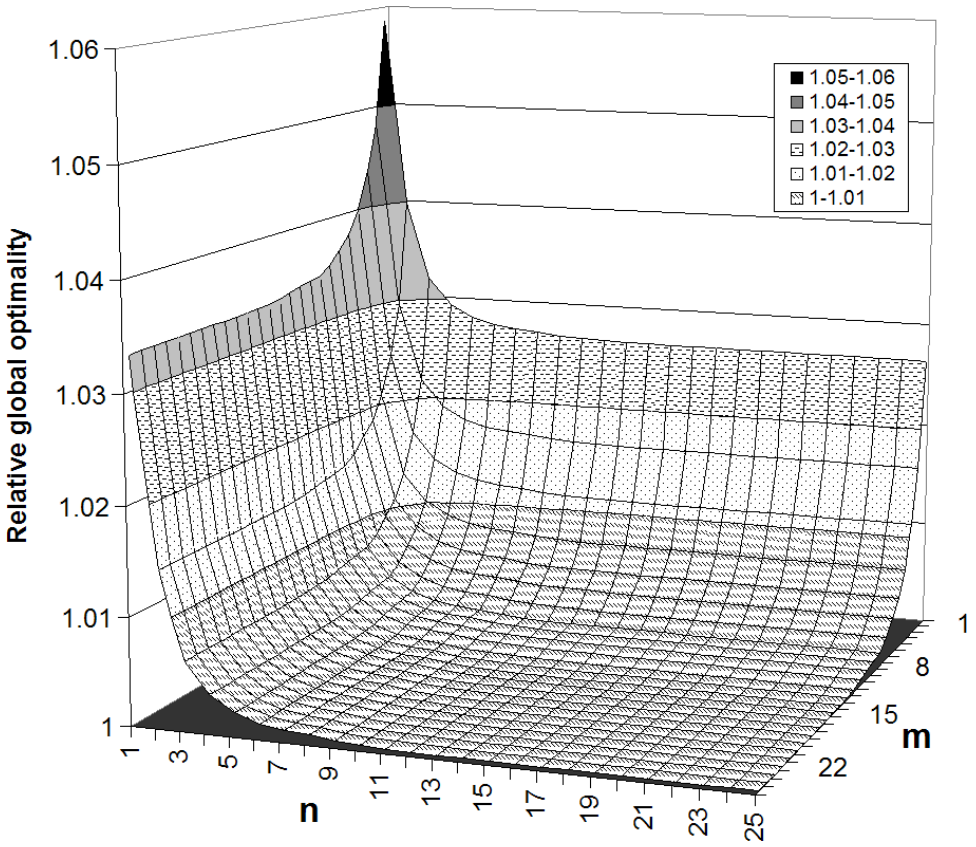


Figure 3.21: Relative global optimality for unweighted-25.

the globally optimal solution than any other VGW/HHP combinations, the fraction is now only 4.3%. Furthermore, the distribution is much more dispersed. Therefore, these results match our intuition.

3.5.5 Algorithms for Best-n-Solutions

So far we have used the pure-local algorithm for selecting the best n local candidates. In this section, we compare the pure-local algorithm with the cluster-f algorithm (see Section 3.4.4). The simulations are based on a data set obtained from the GNP project, and the data set includes the coordinates of 869 Internet nodes. In each simulation, we randomly select 200 nodes as VGWs, 200 as HHPs, and 200 as ESMPs, and the remaining setup is the same as that in the previous section. Therefore, using the previous definition, this set of simulations is called "weighted-200". We experimented with two values for the clustering factor f: 3 and 10. Therefore, we present results for three algorithms: pure-local, clustering-3, and clustering-10.

First, Figure 3.24 shows the performance of pure-local in this weighted-200 scenario. When (n,m) is (1,1) (i.e., using the combination of locally optimal solutions), on average

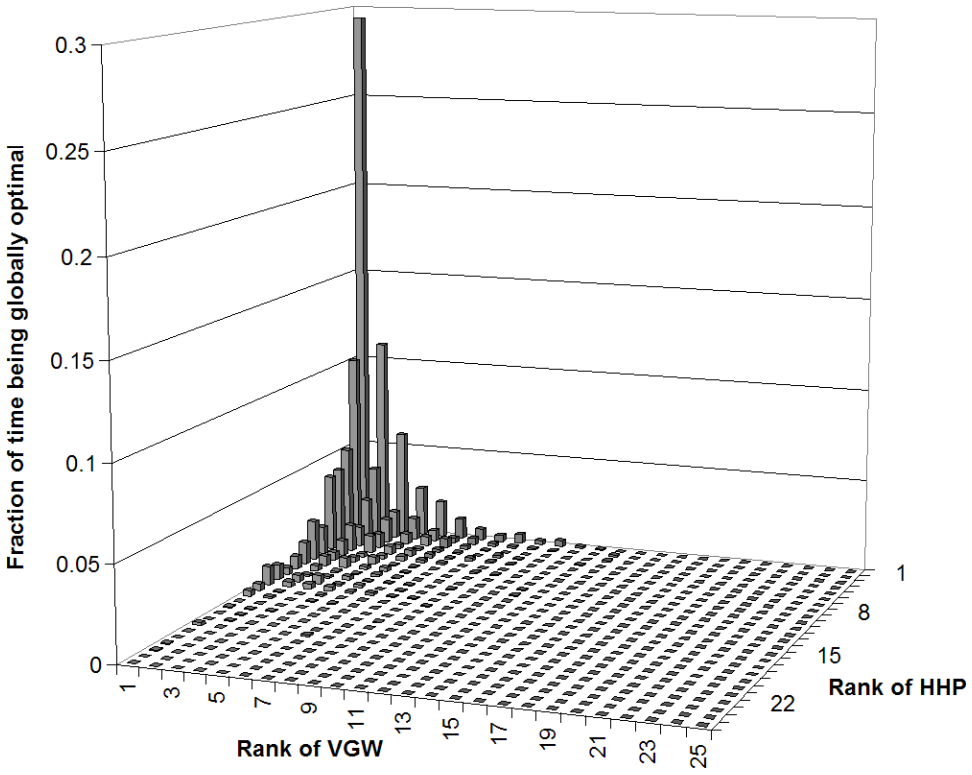


Figure 3.22: Fraction of time each VGW/HHP combination is globally optimal (unweighted-25).

the solution computed by pure-local is 47.3% worse than then globally optimal solution. When searching 16% of the complete search space (i.e., (n,m) is (80,80)) using the candidates returned by the best-n-solutions functionality, the resulting solution is on average 3.6% worse than the globally optimal one. Comparing this result (47.3%→3.6%) with the results from the weighted-5 (13.7%→5.3%) and weighted-25 (27.9%→3.5%) scenarios, we can confirm the trend we observed in the previous section that the effectiveness of the hybrid heuristic increases with the size of the complete search space.

Now we compare the results of pure-local with clustering-3 and clustering-10. Figure 3.25 shows the difference between the clustering-3 algorithm and the pure-local algorithm. Each point (n,m) in the graph is computed by subtracting the relative global optimality (at (n,m)) of the clustering-3 algorithm from that of the pure-local algorithm. Therefore, for example, if the relative global optimality of pure-local and clustering-3 are 1.16 and 1.10, respectively, then the difference is 0.06. The maximum and minimum values in the graph are 0.0835 and -0.0015, respectively. The result shows that using clustering-3 further improves the global optimality of the best-n-solutions approach with the simple pure-local algorithm. Most of the improvements are in the areas of $(2 \leq n \leq 32)$ and $(4 \leq m \leq 48)$. This is expected since clustering-3 divides the candidates into 67 clusters, so when n or m is higher than 67, the solutions generated by the two algorithms become

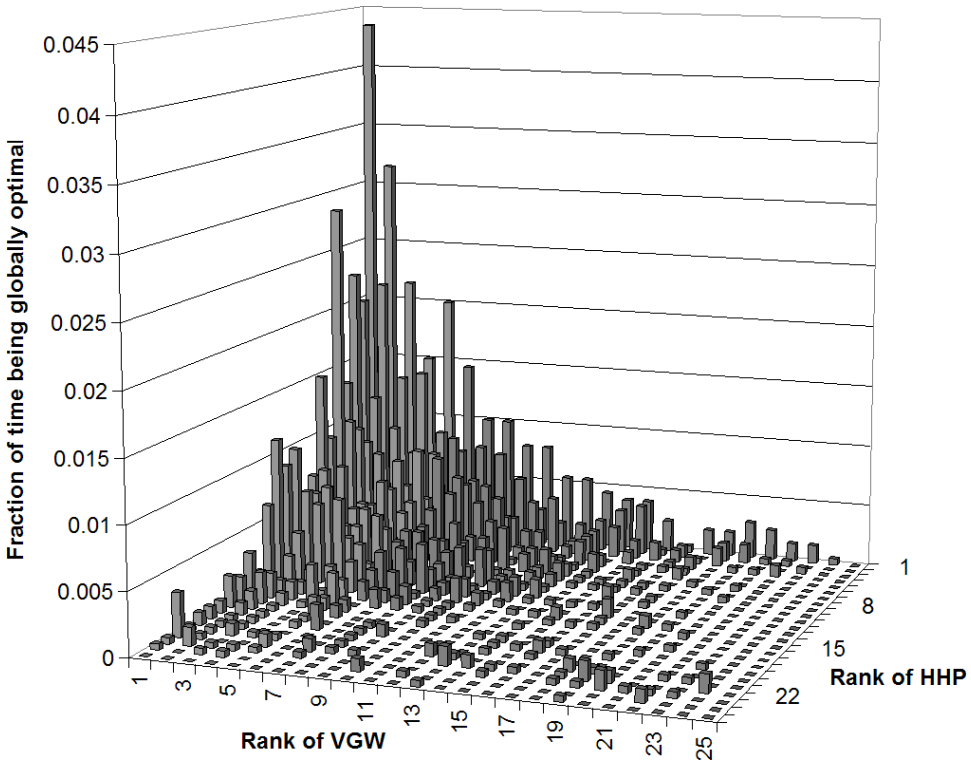


Figure 3.23: Fraction of time each VGW/HHP combination is globally optimal (weighted-25).

more and more similar.

Similarly, Figure 3.26 shows the difference between clustering-10 and pure-local. The maximum and minimum values in the graph are 0.1184 and -0.0016, respectively. The result shows that the difference between clustering-10 and pure-local is even higher than that between clustering-3 and pure-local, and most of the improvements are in the areas of $(2 \leq n \leq 16)$ and $(2 \leq m \leq 32)$. Again, this is because when n or m is higher than the number of clusters (20 for clustering-10), the algorithms become more and more similar.

From Figures 3.25 and 3.26, we see that the location of the area resulting in the most improvement is directly related to the number of clusters in the algorithm used. Therefore, we can extrapolate that as we increase the clustering factor f (i.e., decrease the number of clusters in the algorithm), the area of the most improvement will move closer and closer to $(1,1)$. In other words, a higher clustering factor will result in more significant performance improvement for the requests asking for a smaller number of candidates.

In addition, in both graphs we see that the "improvement band" at the lower m values is both wider and higher than that at the lower n values. This is because the local optimization metric for selecting the HHP is simply the latency to a single target, the handheld user. Therefore, it is an "easier" optimization problem than the selection of the VGW, which needs to consider the latencies to two targets. As a result, the clustering-f algorithms can

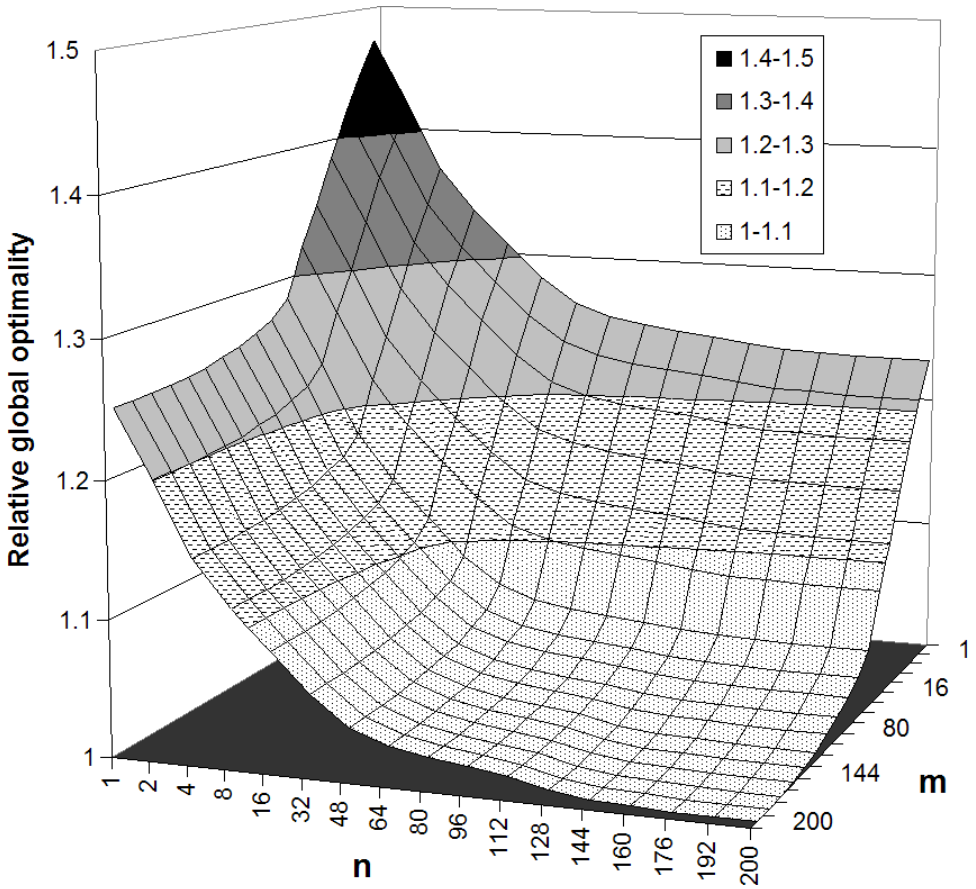


Figure 3.24: Relative global optimality of pure-local in weighted-200.

improve the global optimality of the HHP selection better than that of the VGW selection.

Finally, we compare the three algorithms by computing the distributions of the relative global optimality of all sessions. Figure 3.27 is a summary of the distributions for 3 different (n,m) combinations. It shows the mean and median of each distribution, along with its 95th-percentile and 5th-percentile values. Both clustering-f algorithms have better mean, median, and 95-% values than those of pure-local. Between the clustering-f algorithms, when n and m are low, clustering-10 is clearly better than clustering-3. However, when (n,m) is (16,16), from Figure 3.25 and Figure 3.26 we know that the effectiveness of clustering-10 has begun to decrease, while clustering-3 is at its peak effectiveness. Therefore, we see that clustering-3 has better mean and median values than those of clustering-10. Interestingly, however, clustering-10 still has a better 95-% value. In other words, clustering-10 is much better at improving the global optimality of the worst cases. This is because clustering-10 divides the candidates into fewer clusters (20 compared with 67), and therefore we need a lower n or m to get to candidates that are worse in terms of the local optimization metric.

From this result, we can again extrapolate that using a higher clustering factor will

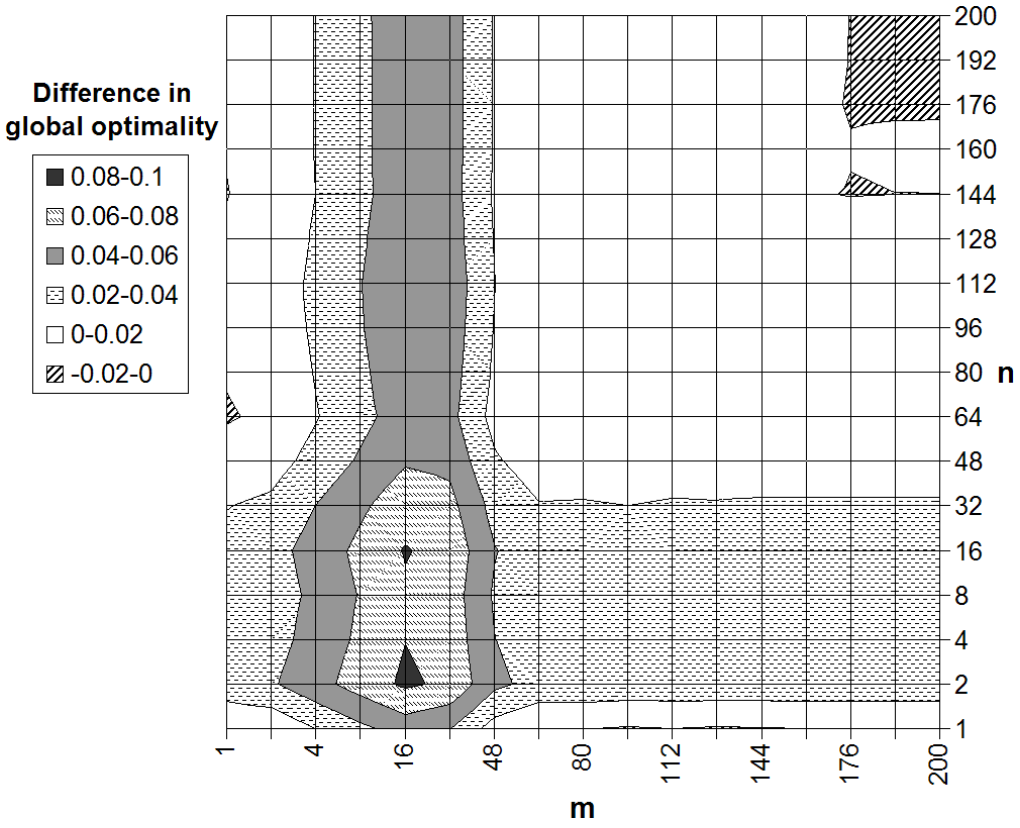


Figure 3.25: Difference in global optimality between pure-local and clustering-3.

improve the mean/median optimality at lower (n,m) (e.g., lower than (8,8)), and at higher (n,m) (e.g., higher than (8,8)) although the mean/median may become slightly worse than that resulting from a lower clustering factor, a higher clustering factor is likely to improve the optimality of the worst cases.

3.5.6 NSSD Overhead

We used the ESM scenario of Figure 3.2 to evaluate the overhead of the NSSD implementation. We register 12 ESM proxies (and 24 other services) with NSSD. Each query asks NSSD to select three ESM proxies for four random participants while minimizing the average latency between each participant and its assigned ESM proxy. The queries are generated using a Perl script and sent (through a FIFO) to another process that uses the SLP API to send the queries to the service directory. The replies flow in the reverse direction back to the Perl script. All the processes are running on a single desktop machine with a Pentium III 933MHz CPU and 256MB of RAM.

We measure the total time of generating and processing 4000 back-to-back queries, and the average (over 10 runs) is 5.57 seconds (with standard deviation 0.032), which shows that in this set up NSSD can process roughly 718 queries per second. We believe this is a

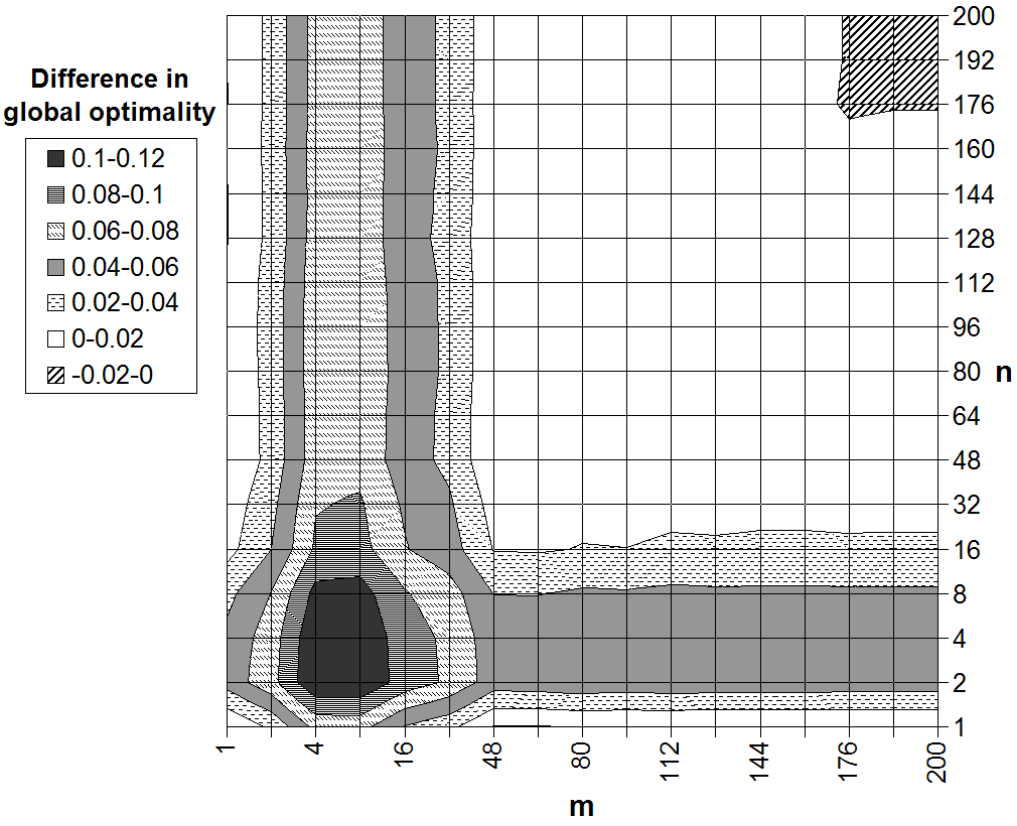


Figure 3.26: Difference in global optimality between pure-local and clustering-10.

reasonable number given the complexity of selecting the ESM proxies and the fact that the time also includes the overhead of query generation and IPC.

3.6 Related Work

There have been many proposals for service discovery infrastructures. For example, Service Location Protocol [61], Service Discovery Service [27], and Java-based Jini [79]. A distributed hashing-based content discovery system such as [51] can also provide service discovery functionality. These general service discovery infrastructures only support service lookup based on functional properties, not network properties. Naming-based approaches for routing client requests to appropriate servers can also provide service discovery functionality. Application-layer anycasting [136] performs server selection during anycast domain name resolution. In TRIAD [58], requests are routed according to the desired content and routing metrics. The Intentional Naming System [2] resolves intentional names, which are based on attribute-value pairs, and routes requests accordingly. Active Names [126] allows clients and service providers to customize how resolvers perform name resolution. NSSD can potentially be built on top of the service discovery and server selec-

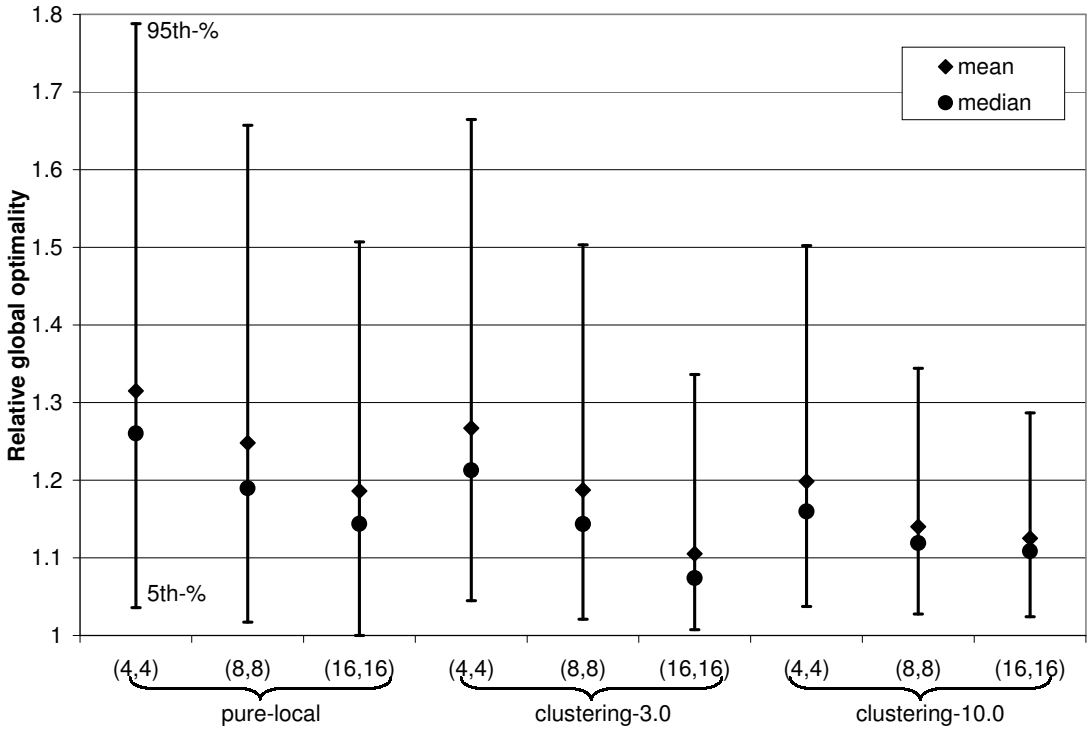


Figure 3.27: Distributions of relative global optimality.

tion mechanisms in these approaches.

An important part of NSSD is the network measurement infrastructure, which provides estimates of network properties such as latency between hosts. A number of research efforts focus on such an infrastructure. For example, in IDMaps [46], the distance between two hosts is estimated as the distance from the hosts to their nearest "tracers" plus the distance between the tracers. GNP [97] uses a coordinates-based approach. Remos [59] defines and implements an API for providing network information to network-aware applications.

Many network-sensitive server selection techniques have been studied before. For example, in [12] a number of probing techniques are proposed for dynamic server selection. Client clustering using BGP routing information [83] or passive monitoring [4] has been applied to server selection. Similarly, distributed binning [114] can be used to identify nodes with similar network properties. In SPAND [116], server selection is based on passive monitoring of application traffic. The effectiveness of DNS-based server selection is studied in [117]. Network-layer anycast [105] handles network-sensitive selection at the routing layer. The Smart Client architecture [135] uses a service-specific applet to perform server selection for a user (primarily for load-balancing and fault-transparency). The performance of various selection techniques is evaluated in [63] and [36]. These studies provide new ways of collecting and using network information for server selection and are complementary to our work. Some other efforts address the problem of request distribution

in a server cluster, for example, [45] and [104]. They are also complementary to NSSD since all nodes within the same cluster have similar network properties.

In the context of computational Grids, researchers have also been working on the resource/service discovery and selection problems. The Metacomputing Directory Service (MDS) [41, 25] implements the Grid information service architecture [25] that defines data models, mechanisms, and protocols for Grid applications to access both resource characteristics and network performance information. A resource selection framework is proposed to provide user-side resource selection functionality using information provided by MDS [87]. Another hierarchical service discovery framework [134] extends SDS by incorporating user preference feedback into service selection. The Remos network monitoring/measurement infrastructure can be used by applications to perform network-sensitive service selection in a Grid environment [59]. In fact, these previous efforts provide the building blocks for realizing network-sensitive service discovery in a Grid environment: in the Grid information service architecture, the NSSD functionalities can be built into a specialized *aggregate directory service* that extracts and combines information from both "service information providers" (e.g., a traditional service directory) and "monitoring information providers" (e.g., Remos). Through the NSSD API, this directory can provide useful information for self-configuring services and other adaptive applications in a computational Grid.

3.7 Chapter summary

In this chapter, we described the architecture and implementation of the Network-Sensitive Service Discovery (NSSD) infrastructure. NSSD integrates the functionalities of traditional service discovery and network-sensitive server selection and allows users to benefit from network-sensitive selection without having to implement their own selection mechanisms. It also does not require providers to expose all their server information to users. We defined the NSSD API that supports service lookups based on both functional and network properties required by users. For coordinated selection of multiple components, we designed the hybrid heuristic that performs global optimization on a reduced search space. NSSD supports the hybrid heuristic by providing additional information through the best-n-solutions feature.

In the context of self-configuring services, the self-configuration module is a "user" of NSSD. The local optimization techniques supported by NSSD can be used to perform the common self-configuration operation of single component selection based on service-specific network performance criteria. For self-configuring services where coordinated selection of multiple components is necessary, a self-configuration module can apply the hybrid heuristic using the best-n-solutions feature of NSSD to approximate the desired global optimization.

In our evaluation, we show that our prototype implementation of NSSD has reasonably good query processing performance. Experimental results of the multiplayer online gaming service show that by using the local optimization functionality provided by NSSD, the simulated service can achieve significant performance improvements. Simulation re-

sults for coordinated selection of multiple components in the video conferencing scenario demonstrate that the flexibility of the best-n-solutions feature allows a user to perform global optimization in a reduced search space and greatly improve the performance of the resulting solution.

Chapter 4

Synthesizer And Recipe Representation

The key element in our recipe-based self-configuration architecture is the *synthesizer*, which plays the role of self-configuration module. To build a self-configuring service, a service provider transforms its service-specific self-configuration knowledge into a *recipe* using the *recipe representation* exported by the synthesizer. The recipe is submitted to the synthesizer and is used by the synthesizer to guide the self-configuration operations. As discussed earlier, the synthesizer performs two types of self-configuration operations, *global configuration* and *local adaptation*. In this chapter, we discuss the definition of the recipe representation and the design and implementation of the synthesizer to support global configuration. In the next chapter, we extend the recipe representation and the synthesizer to support local adaptation.

The remainder of this chapter is organized as follows. In the next section, we give a high-level overview of the synthesizer architecture. In Section 4.2, we define the recipe representation as a set of APIs that can be used by service providers to design their service recipes. A service recipe contains a service provider's service-specific knowledge in two aspects of self-configuration, abstract mapping and physical mapping. We then look at the synthesizer design in Section 4.3 and focus on how the synthesizer solves the physical mapping problem. We discuss how the complexity of the problem depends on the objective function and describe optimization algorithms for solving different problems. We also discuss how the synthesizer can select the appropriate algorithm according to a provider's cost and optimality constraints. We describe our prototype implementation in Section 4.4 and present our evaluation results in Section 4.5.

4.1 Overview of the synthesizer architecture

Figure 4.1 presents an overview of the synthesizer architecture. A *service recipe* is written by a service provider and contains an operational description of the service-specific knowledge. The *synthesizer* plays the role of the self-configuration module and consists of two modules. The *facility module* implements the generic infrastructure knowledge that is common and reusable for different services. The *facility interface* exports the facility module

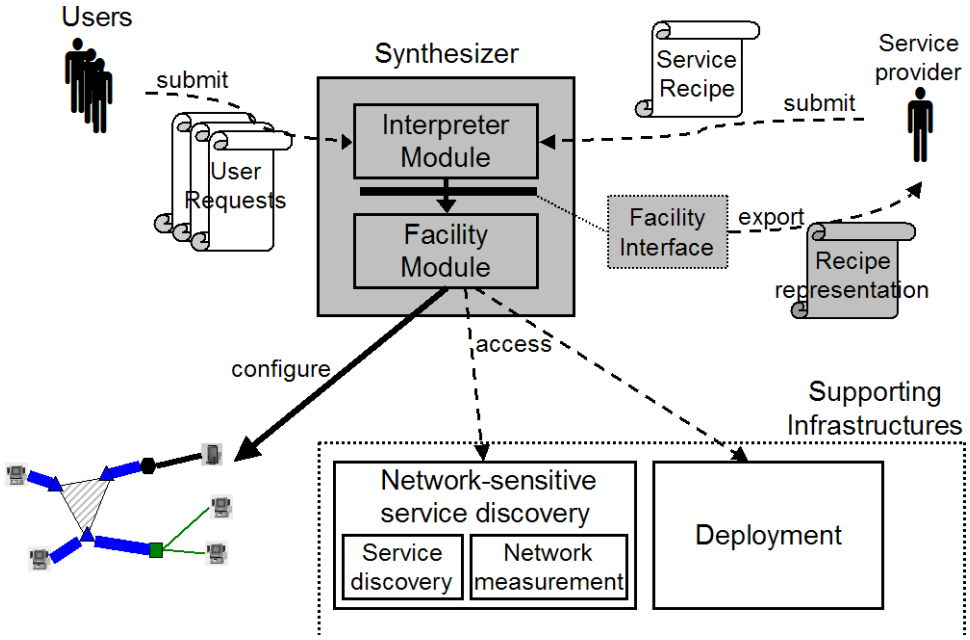


Figure 4.1: Synthesizer architecture.

functionalities in the forms of APIs and libraries that can be used by service providers to write their recipes. The *interpreter module* executes a recipe submitted by a provider to compose a global configuration for each user request.

To build a self-configuring service, the service provider gives the recipe to the synthesizer. At run time, a user can send a request to the synthesizer specifying the user requirements. For example, in a video conferencing scenario, a user request specifies the IP addresses and conferencing applications of the participants. When the synthesizer receives the request, it extracts the user requirements from the request, and the interpreter module access the functionalities provided by the facility module to compose an optimal global configuration for the request according to the recipe.

Note that the previous generic and service-specific approaches can be viewed as two extreme cases of this architecture. In the generic approach, the "recipe" is an abstract specification containing only the highest-level service-specific knowledge, e.g., the required input type. All other tasks are automatically performed by the generic "synthesizer" using only generic self-configuration knowledge. On the other hand, the "recipe" in the service-specific approach is in fact a service-specific self-configuration module that takes the place of the synthesizer. Implementing such a monolithic module requires extra efforts from the provider who may not have the necessary infrastructure knowledge.

In contrast, in our approach, service providers design their recipes using a general-purpose programming language, e.g., Java is used in our prototype. On one hand, this provides a much richer and more flexible representation of service-specific knowledge than the representation in the generic approach, and therefore, it allows providers to specify a much broader range of service-specific knowledge than the input/output types, for example.

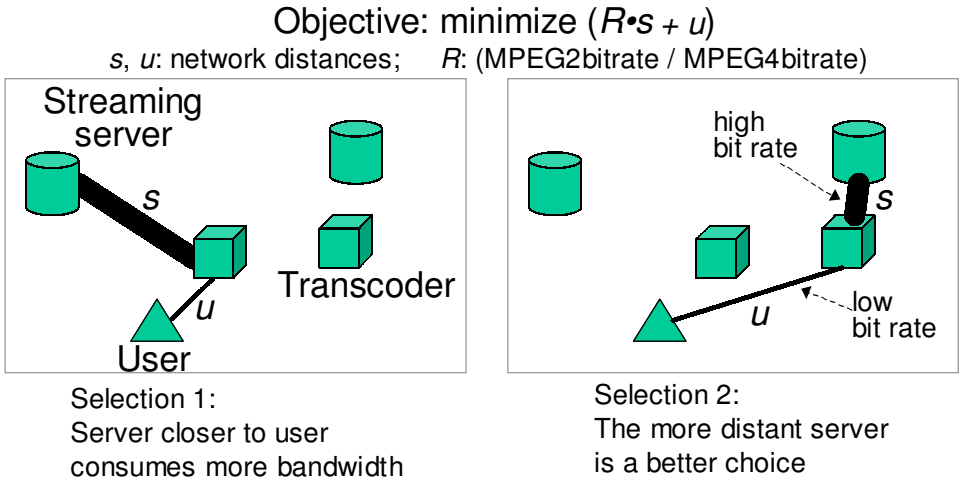


Figure 4.2: A video streaming service.

On the other hand, since only the service-specific knowledge needs to be encoded into the recipe, service providers do not have to worry about the generic infrastructure knowledge, making this recipe representation much easier to use (i.e., requiring much less efforts) than the "representation" in the service-specific approach. Note that the synthesizer can take the form of a toolkit/library that can be used by service providers to build self-configuring services, or it can also be a standalone entity whereby a provider builds a self-configuring service by submitting a recipe to the synthesizer.

In the next two sections, we discuss how we define the recipe representation for specifying the different aspects of the service-specific knowledge and how the synthesizer uses such knowledge to perform global configuration.

4.2 Recipe representation

To define the recipe representation, we first look at some self-configuring service examples to identify the knowledge required for global configuration.

4.2.1 Self-configuring service examples

First, let us consider the self-configuring video streaming service example from Chapter 1, shown in Figure 4.2. There are two pieces of service-specific knowledge in this scenario. First, for a user who requests a low-bitrate MPEG-4 video stream, the global configuration should be a combination of an MPEG-2 server and an MPEG-2-to-MPEG-4 transcoder. Secondly, the provider's goal is to minimize the network resource consumption. Therefore, among the many candidates for MPEG-2 server and MPEG-2-to-MPEG-4 transcoder, the provider's objective for component selection is to select the pair of server and transcoder

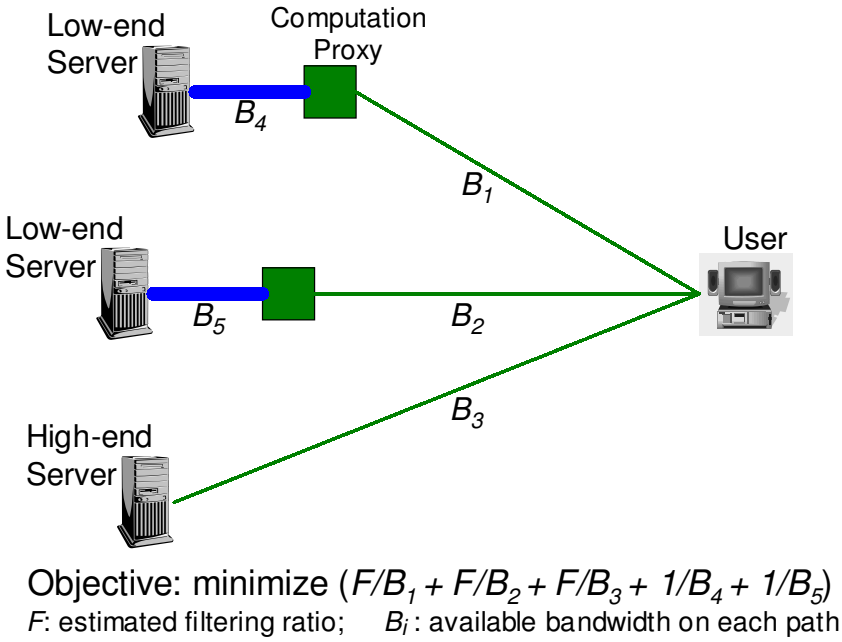


Figure 4.3: An interactive search service.

that minimizes the bitrate-weighted network distance represented by the objective in the figure.

Another example is the self-configuring interactive search service shown in Figure 4.3 that supports application-specific filtering, similar to the Diamond system [72]. A user wants to find a particular picture from an image collection distributed across three storage servers. For efficiency, this service allows the user to upload application-specific filters, e.g., "blue with water-like texture", to the servers so that irrelevant pictures can be discarded early to reduce the bandwidth consumption. The service provider's service-specific knowledge dictates that if a storage server does not have sufficient computation resources to handle the application-specific filters, a computation proxy can be used to run the filters in front of the server. Furthermore, to reduce the overall communication time, the selection of the computation proxies should minimize the objective shown in the figure.

Finally, let us revisit the video conferencing service example from Chapter 2, shown in Figure 4.4. Five users want to hold a video conference: P1 and P2 have MBone conferencing applications vic/SDR (VIC), P3 and P4 use NetMeeting (NM), and P5 uses a receive-only handheld device (HH). The service provider's service-specific knowledge indicates that a configuration supporting these users can be composed as follows. A video conferencing gateway (VGW) is needed for protocol translation and video forwarding between VIC and NM users. A handheld proxy (HHP) is needed to join the conference for P5. An End System Multicast (ESM) [23] overlay consisting of three ESM proxies (ESMPs) can be used to enable wide-area multicast among P1, P2, VGW, and HHP, Furthermore, to reduce the network resource usage, the necessary components in the global configura-

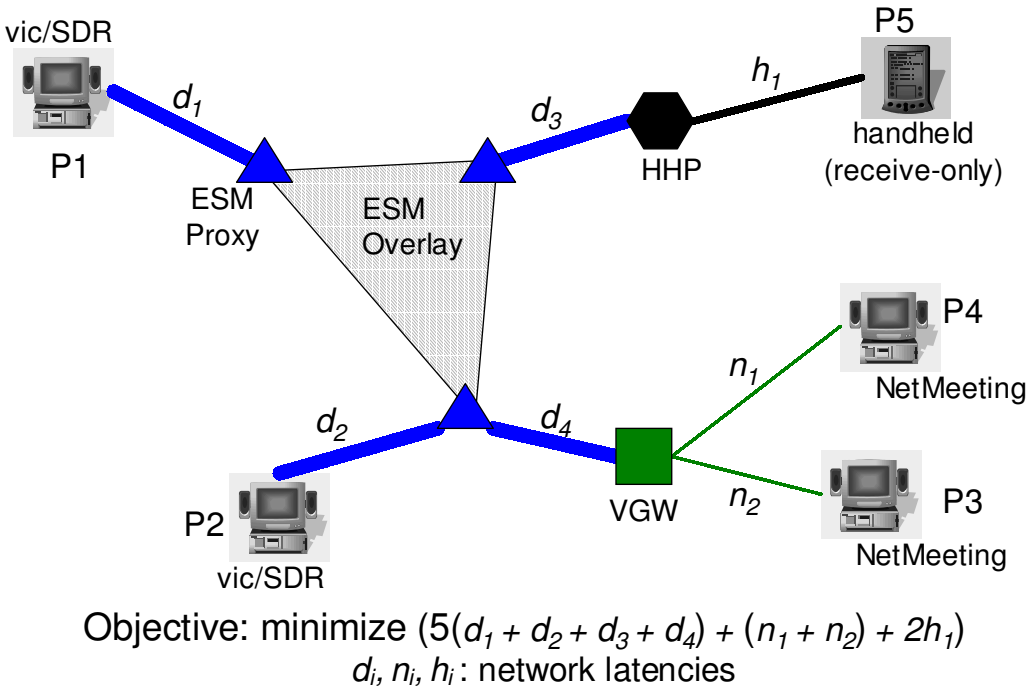


Figure 4.4: A video conferencing service.

tion should be selected such that the objective function shown in the figure is minimized. The variables in the objective function are network latencies, and the weights reflect the bandwidth usage of different conferencing applications, e.g., NetMeeting only receives one video stream, and the handheld cannot receive all streams.

From these examples, we can see a common theme in the process of composing a global configuration. First, it requires some service-specific knowledge to determine the types of components needed to serve a particular request. It then needs some service-specific criteria for selecting an actual component for each of the required component types. Therefore, we divide the global configuration process into two steps: *abstract mapping* and *physical mapping*, This division is similar to a number of previous self-configuration frameworks, for example, [57, 115, 109, 48, 75, 60].

As shown in Figure 4.5, in the abstract mapping step, the synthesizer generates an *abstract configuration* that specifies what types of components are needed to satisfy a request, for example, VGW, HHP, and ESMP in the video conferencing scenario. Then the synthesizer performs physical mapping to generate a *physical configuration* that maps each *abstract component* in the abstract configuration to a *physical component* in the network, for example, abstract component VGW is mapped to the physical component with IP address 192.168.1.1, HHP is mapped to 192.168.2.2, and so on.

To allow service providers to specify the service-specific mapping knowledge, one design decision we need to make is what type of "language" should be used for the speci-

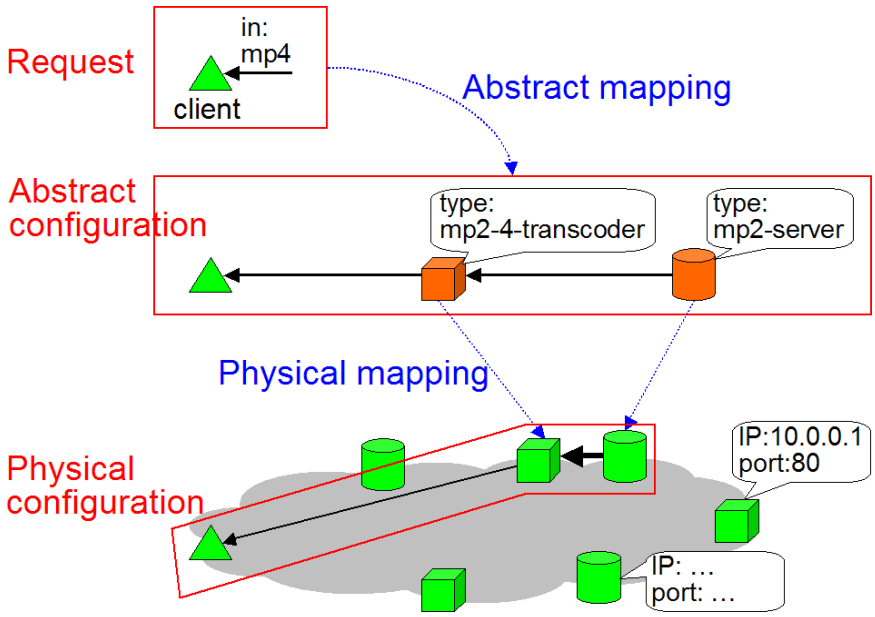


Figure 4.5: Abstract mapping and physical mapping.

fication. There has been a large body of work on the design of Architecture Description Languages (ADLs) (e.g., Acme [53]) that can be used to formally describe the architecture of a component-based system, which is roughly equivalent to our definition of abstract configuration. These ADLs allow the specification of components and connectors and how they fit together in the system. Therefore, a natural question is: can we use such languages for our purpose, or do we need a language that is closer to a general-purpose programming language with, for example, control flow features?

For the purpose of specifying the mapping knowledge, the main difference between using a more restricted "specification language" such as an ADL and using something closer to a general-purpose "programming language" is that such a specification language is designed for specifying *a configuration* while a programming language can be used to describe the *process of constructing a configuration*. One advantage of using a restricted specification language is that it requires less efforts from the service provider, and formal analysis or verification can be applied more easily. However, as we see in earlier examples, the actual configuration very often depends on the particular user requirements in the target request. In other words, the provider must be able to specify how to construct a configuration instead of specifying a particular configuration instance. As a result, when manipulation of the architecture is required, previous work has extended the notion of ADLs to include features such as control flow from general-purpose programming languages. For example, Rainbow [19] extends the Acme ADL to allow the specification of adaptation strategies and tactics that can diagnose run-time problems and adapt the configuration accordingly. An architecture reconfiguration language is proposed in [131] that allows the manipulation of the configuration. Such languages are close to general-purpose program-

ming languages in terms of the features they provide, and they are more suitable for our purpose than a restricted specification language.

Since the focus of this dissertation is on what service-specific knowledge should be abstracted and how the abstracted knowledge is used for self-configuration, we do not address specific language design issues. Therefore, we use a general-purpose programming language, Java [81], for specifying the service-specific knowledge in our proof-of-concept prototype. The recipe representation is defined as a set of APIs that can be used by providers to write their recipes. In practice, if a language similar to those in [19, 131] is adopted for the specification, our representation can be transformed into new features of the language. Even if a restricted specification language is to be used, most of the required functionalities we have identified in our APIs will still be applicable.

We now discuss how we define the recipe representation to capture the service-specific abstract mapping and physical mapping knowledge in a general way.

4.2.2 Abstract mapping knowledge

An abstract configuration is basically a "graph" consisting of nodes representing the types of components needed, i.e., *abstract components*, and links representing the connections between the abstract components. In other words, in the abstract mapping step, the synthesizer needs to generate a data structure that represents such a graph. Although the knowledge of what types of components are needed and how they are connected is service-specific, the task of constructing and maintaining the data structure is generic across different services. Therefore, service providers should be able to specify how the abstract configuration should be constructed in a generic fashion, i.e., they should not have to design their own service-specific data structures and functions to handle abstract configurations.

To support the construction and maintenance of abstract configurations, the facility module of the synthesizer exports an abstract configuration API that provides generic data structures and functions through the facility interface. A service provider can use these data structures and functions in its recipe to construct and manipulate the abstract configuration. Figure 4.6 lists the major data structures and functions that are made available to service providers through the facility interface. The data structures represent the abstract configuration and the nodes in the configuration. `AbsComp` represents an abstract component, i.e., it specifies a required component type; `PhyComp` is a physical component, i.e., a fixed node in the abstract configuration, for example, the video conferencing participants in the video conferencing service, the user and servers in the interactive search service, and so on. The functions listed are used to manipulate these data structures.

We now use the video conferencing service example to illustrate how the service-specific abstract mapping knowledge can be expressed in a recipe using this API. Figure 4.7 shows that the abstract mapping knowledge for this service can be roughly divided into three pieces: (1) a VGW should be connected to all participants who use NetMeeting, (2) an HHP should be used for each handheld participant, and (3) an ESM overlay consisting of three ESMPs should be used to provide multicast among VIC users, VGW, and HHPs.

Data structure	
`AbsConf`	Represent abstract configurations; contains components and connections.
`AbsComp`	Represent abstract components; contains component properties (also sub-components, if any).
`PhyComp`	Represent physical components (i.e., fixed nodes such as the users); contains component properties.

Function	
`addComp(spec)`	Add an abstract component with the given specifications `spec` to the abstract configuration.
`addConn(c1,c2)`	Add a connection between components c1 and c2 to the abstract configuration.
`addSubComp(n,spec)`	Add *n* identical sub-components (with `spec`) to a component in the abstract configuration.
`getProperty(prop)`	Get the value of the property named `prop` of a component in the abstract configuration.

Figure 4.6: Abstract configuration API.

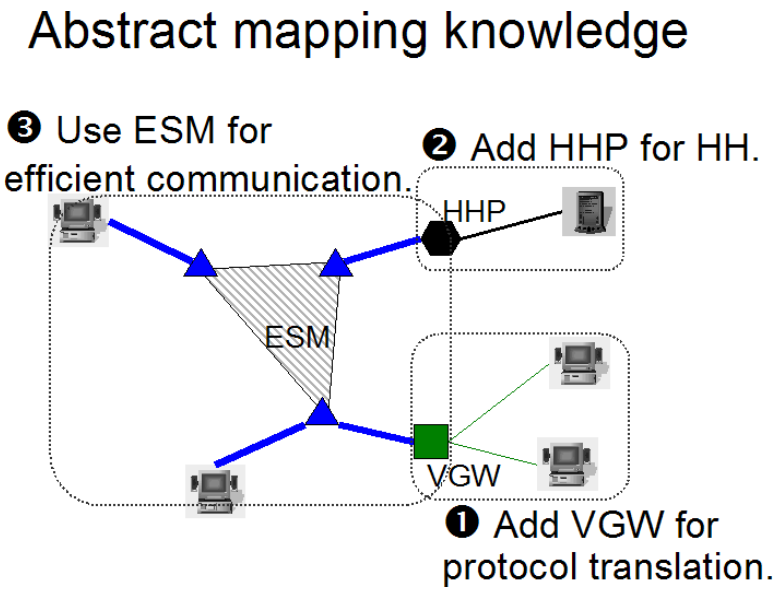


Figure 4.7: Abstract mapping knowledge for the video conferencing service.

Figure 4.8 shows a recipe that expresses the above abstract mapping knowledge using the abstract configuration API. The recipe shown is written in the Java programming language, which is used in our prototype. The three segments of the recipe correspond to the

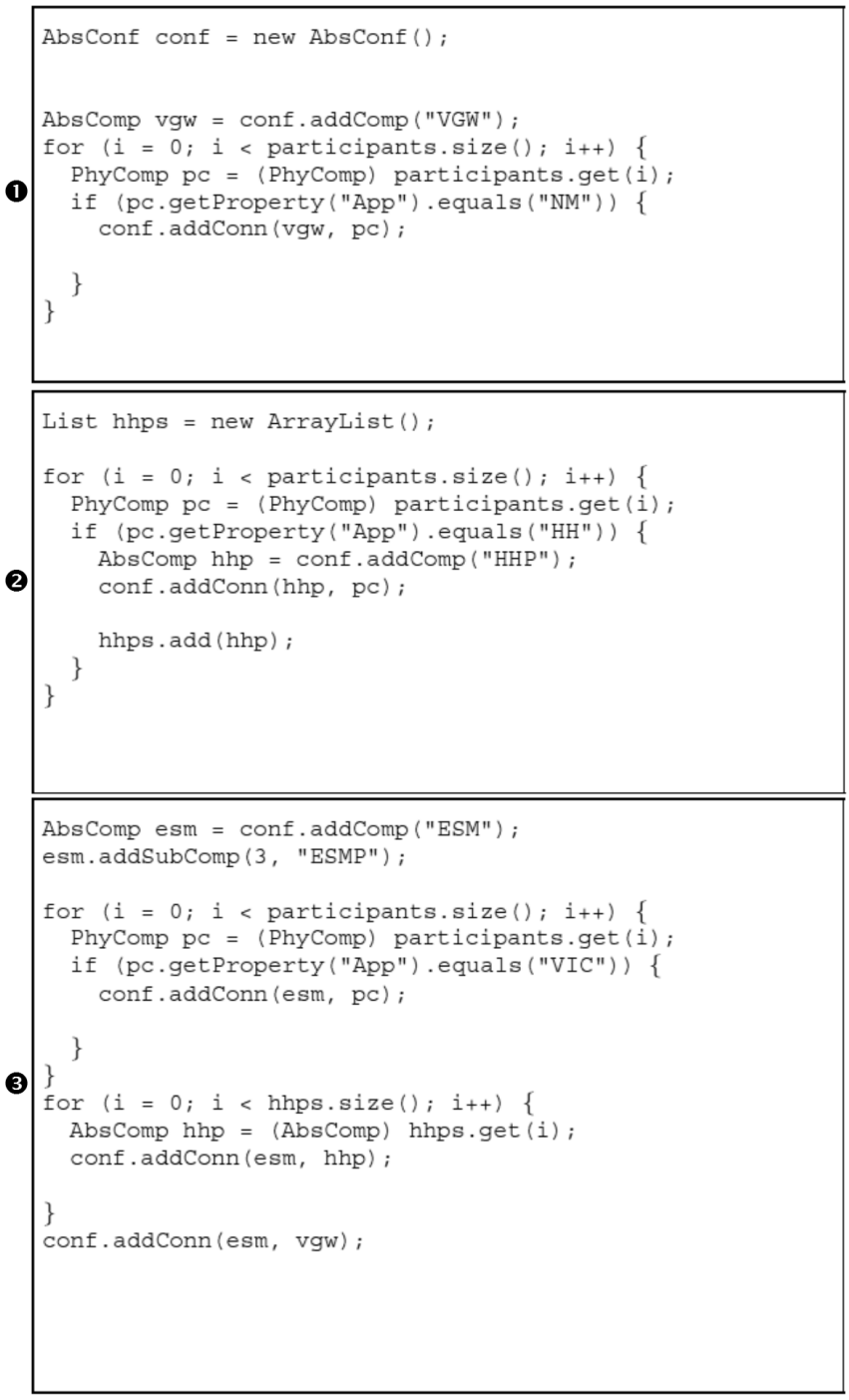


Figure 4.8: A video conferencing recipe: abstract mapping.

three pieces of knowledge above. Basically, an empty configuration is initialized first, and then the needed components are inserted into the configuration and are connected to the appropriate nodes. Note that `participants` is the list of participants (specifically, a

`List` of `PhyComp` objects) extracted from a user request by the synthesizer and is used as an input during the execution of the recipe. For simplicity, we use "symbolic names" to represent complete specifications of component type and attributes, for example, "VGW" represents "(serviceType = VideoGateway) (protocols = H323,SIP)...".

Given this recipe, generating an abstract configuration is straightforward. When a user request is received, the interpreter module of the synthesizer executes the recipe to construct the data structure `conf` that represents the abstract configuration.

4.2.3 Physical mapping knowledge

After generating an abstract configuration, the synthesizer needs to perform physical mapping, i.e., the synthesizer needs to generate a physical configuration that maps each of the abstract components in the abstract configuration to a physical component. In particular, the selection of the physical components is based on some service-specific criteria. From the examples in Section 4.2.1, we see that the component selection problem can generally be formulated as an optimization problem in which the component selection criteria is represented as an *objective function* that needs to be optimized. Therefore, the task of physical mapping can be divided into three steps: identifying the objective function, formulating the optimization problem, and solving the problem.

An important observation is that while the optimization objective is is part of a service provider's service-specific knowledge, the provider may not have the expertise to formulate and solve optimization problems. Moreover, these latter two steps are in fact relatively generic, i.e., problem formulation techniques and optimization algorithms for solving the problems can often be reused for different services. Therefore, the key service-specific knowledge for physical mapping is the objective function for component selection, and the synthesizer should provide an API that allows a service provider to specify such an objective function. Given the objective function and the abstract configuration generated in the abstract mapping step, the synthesizer can then formulate and solve the optimization problem of component selection. In this section, we focus on the API for specifying objective functions, and in Section 4.3, we discuss how the synthesizer performs component selection.

An objective function is a function in which each term is a *metric* or a function of multiple metrics. Metrics represent properties of components or properties of connections between components. For example, the objective function for the video streaming service in Figure 4.2 has two metric terms, the latency between the server and the transcoder and the latency between the transcoder and the user. The metrics in the objective function in Figure 4.3 are the available bandwidths on the different connections. The metrics for the video conferencing service in Figure 4.4 are the latencies on the connections between components and participants.

In order to optimize component selection according to an objective function, the synthesizer must be able to obtain the values of all metrics in the objective function. For example, given the objective function for the video streaming service, the synthesizer needs to ac-

Data structure	
`LatencyM`	Represent the network latency between two components.
`BandwidthM`	Represent the available bandwidth between two components.
`Function`	Represent an objective function; contains a `Term`.
`Term`	Contain either a metric or a floating point number; also provides member functions listed below for appending other instances of `Term` to "this" instance.

Function (member functions of `Term`)	
`add(t)`	Add `t` (an instance of `Term`) to "this" instance.
`subtract(t)`	Subtract `t` from this instance.
`multiplyBy(t)`	Multiply this instance by `t`.
`divideBy(t)`	Divide this instance by `t`.
`pow(t)`	Raise this instance to the `t`-th power.

Figure 4.9: Objective function API.

cess a network measurement infrastructure that can provide latency information between the streaming server candidates and the transcoder candidates and between the transcoder candidates and the user. Therefore, what metrics can be used in objective functions depend on what are provided by the supporting infrastructures. In our current prototype, we define two metrics that can be used in objective functions: `LatencyM` and `BandwidthM`, representing the network latency and the available bandwidth between two nodes, respectively. Other important metrics such as CPU speed, memory size, cost, and so on can be added by extending the supporting infrastructures to provide these properties.

Given a set of metrics, we define an API for constructing objective functions that consists of the data structures and functions shown in Figure 4.9. Note that since our prototype uses the Java programming language, all the data structures are defined as classes. All functions listed are member functions of the `Term` class, e.g., if `A` and `B` are both instances of `Term` and represent a and b, respectively, then after calling `A.add(B)`, `A` represents $(a+b)$. To summarize, this interface can be used to construct a tree-like data structure representing the objective function. This tree can then be traversed to evaluate the value of the function given a particular selection of components. This API is fairly general and allows a service provider to construct non-trivial objective functions in recipes.

As an example, we show how the objective function for the video conferencing service in Figure 4.4 is constructed in the video conferencing service recipe. As we can see in Figure 4.10, the objective function is basically a weighted sum of three partial objectives: (1) the sum of the latencies between the VGW and the NM users, (2) the sum of the latencies between HH users and their corresponding HHPs, and (3) the sum of the latencies between the ESMPs and the multicast end points.

Figure 4.11 shows a recipe that constructs this service-specific objective function using

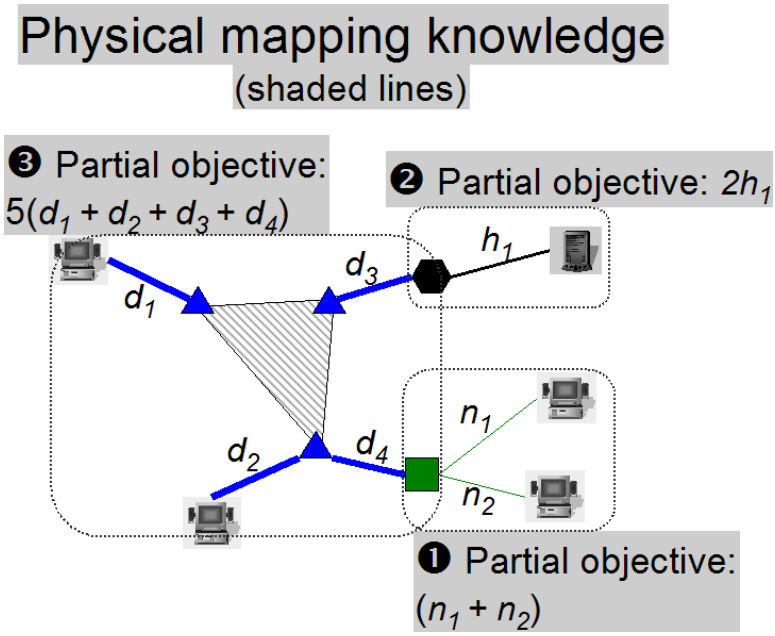


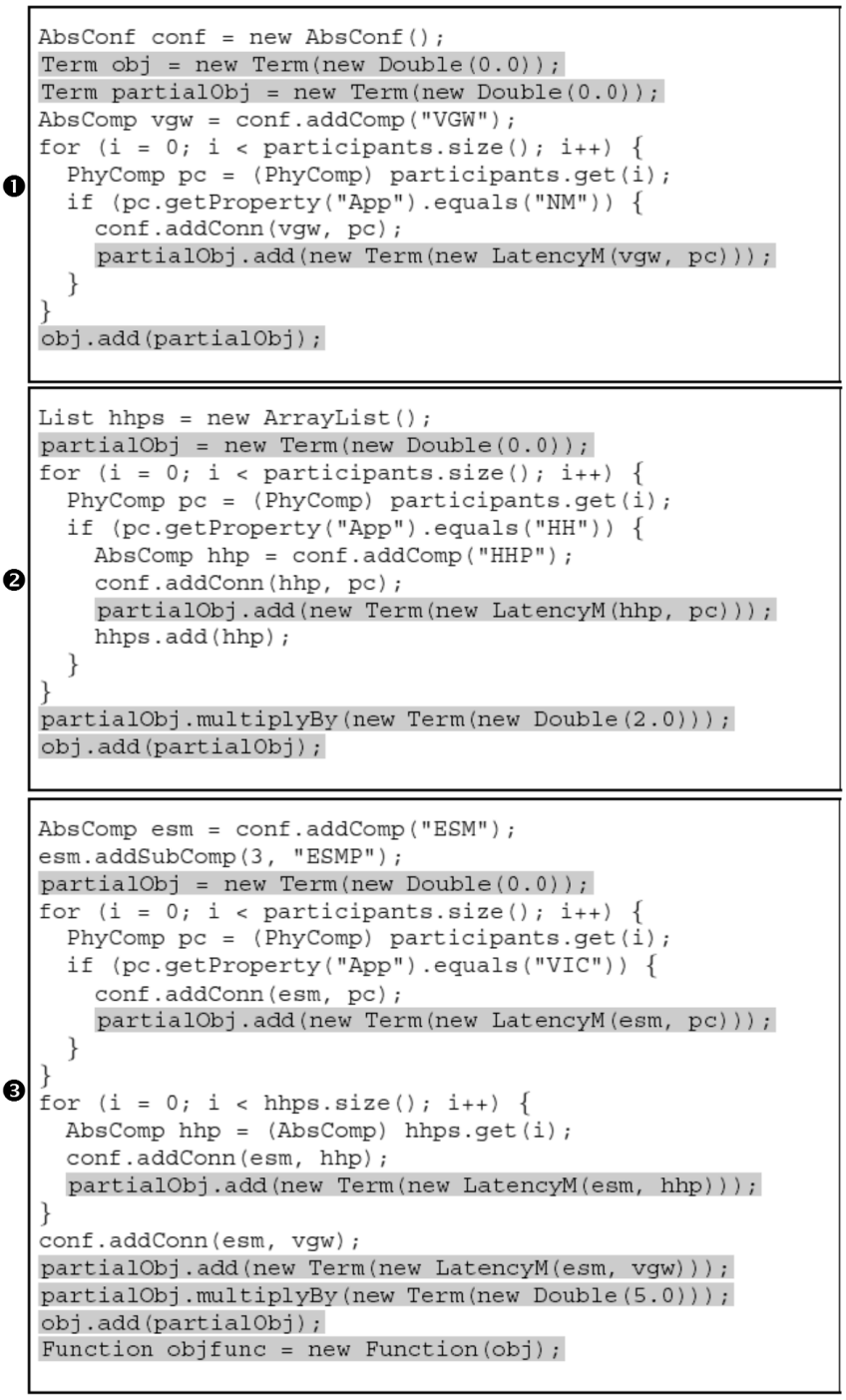
Figure 4.10: Physical mapping knowledge for the video conferencing service.

the objective function API above. The unshaded lines in the recipe are unchanged from Figure 4.8 and represent the abstract mapping knowledge. The shaded lines construct the objective function, and the three segments of the recipe construct the three partial objectives, respectively. These partial objectives are added together in the term `obj`, which is finally used to construct the function `objfunc`.

To summarize, in this section, we present a recipe representation that allows service providers to express their service-specific abstract mapping and physical mapping knowledge in a recipe. In the next section, we describe how the synthesizer finds the optimal global configuration for each user request using such knowledge.

4.3 Synthesizer

As described in the previous section, the synthesizer needs to perform two tasks, abstract mapping and physical mapping, to find the optimal global configuration for a user request. When a user request is received, the interpreter module of the synthesizer executes the recipe with the user requirements extracted from the request. During the execution of the recipe, an `AbsConf` data structure is constructed that represents the abstract configuration, and a `Function` data structure is constructed that represents the objective function, i.e., the abstract mapping step is performed during the execution of the recipe. On the other hand, for physical mapping, given the generated objective function, the synthesizer still needs to formulate and solve an optimization problem in order to select the best candidates for the needed components. Therefore, we focus on how the synthesizer formulates and



```
AbsConf conf = new AbsConf();
Term obj = new Term(new Double(0.0));
Term partialObj = new Term(new Double(0.0));
AbsComp vgw = conf.addComp("VGW");
for (i = 0; i < participants.size(); i++) {
  PhyComp pc = (PhyComp) participants.get(i);
  if (pc.getProperty("App").equals("NM")) {
    conf.addConn(vgw, pc);
    partialObj.add(new Term(new LatencyM(vgw, pc)));
  }
}
obj.add(partialObj);
```

❶

```
List hhps = new ArrayList();
partialObj = new Term(new Double(0.0));
for (i = 0; i < participants.size(); i++) {
  PhyComp pc = (PhyComp) participants.get(i);
  if (pc.getProperty("App").equals("HH")) {
    AbsComp hhp = conf.addComp("HHP");
    conf.addConn(hhp, pc);
    partialObj.add(new Term(new LatencyM(hhp, pc)));
    hhps.add(hhp);
  }
}
partialObj.multiplyBy(new Term(new Double(2.0)));
obj.add(partialObj);
```

❷

```
AbsComp esm = conf.addComp("ESM");
esm.addSubComp(3, "ESMP");
partialObj = new Term(new Double(0.0));
for (i = 0; i < participants.size(); i++) {
  PhyComp pc = (PhyComp) participants.get(i);
  if (pc.getProperty("App").equals("VIC")) {
    conf.addConn(esm, pc);
    partialObj.add(new Term(new LatencyM(esm, pc)));
  }
}
for (i = 0; i < hhps.size(); i++) {
  AbsComp hhp = (AbsComp) hhps.get(i);
  conf.addConn(esm, hhp);
  partialObj.add(new Term(new LatencyM(esm, hhp)));
}
conf.addConn(esm, vgw);
partialObj.add(new Term(new LatencyM(esm, vgw)));
partialObj.multiplyBy(new Term(new Double(5.0)));
obj.add(partialObj);
Function objfunc = new Function(obj);
```

❸

Figure 4.11: A video conferencing recipe: abstract and physical mappings.

solves the optimization problem of physical mapping given the generated abstract configuration and objective function. Let us first discuss the complexity of such problems and review algorithms for solving them.

4.3.1 Complexity and algorithms

In general, the synthesizer cannot simply select each component independently. For example, if the objective function includes a single metric $M(c_1, c_2)$ where c_1 and c_2 are abstract components with n candidates each, then the selection of c_1 *depends* on that of c_2, i.e., the synthesizer needs to select them together by searching through n^2 possible combinations to find the optimal selection. Furthermore, this dependency is *transitive*, e.g., if the objective is $M_1(c_1, c_2) + M_2(c_2, c_3)$, then all three components are mutually dependent. In addition, some physical mapping problems involve *semi-dependent* components, e.g., in $M_1(c_1, c_2) + M_2(c_2)$, c_2 is semi-dependent, i.e., independent in the second term but dependent on c_1 in the first term. Similarly, VGW and HHP in Figure 4.4 are both semi-dependent.

If some or all components are mutually dependent, physical mapping is a *global optimization* problem with a worst-case problem size of n^m where m is the number of abstract components. If every abstract component is independent, e.g., if the objective is $M_1(c_1) + M_2(c_2) + \ldots + M_m(c_m)$, physical mapping becomes a series of *local optimization* problems with a total problem size of mn. Since solving local optimization problems is more straightforward, here we focus on how to solve the more general physical mapping problems involving global optimization.

There are many techniques that can be used to solve a general physical mapping problem. One possibility is to use a general optimization algorithm that can solve generic optimization problems. For example, *exhaustive search* solves any optimization problems by enumerating all possible solutions in the problem space. An optimization problem can also be solved using the *simulated annealing* heuristic based on the physical process of annealing [66], which only searches in a small part of the problem space. One issue with using such general algorithms is that they may not be efficient for the particular target problem. For example, the optimization cost of exhaustive search grows linearly with the problem size, which can be huge for global optimization problems. Simulated annealing may not be able to find good solutions if the parameters are not optimally tuned.

Another possible technique is to use some heuristics that take advantage of the "structure" of the physical mapping problem to reduce the problem size. The hybrid heuristic described in Chapter 3 is an example of this technique: it reduces the size of a global optimization problem by selecting a small number of candidates using a local optimization objective and then uses a general optimization algorithm to solve the reduced global optimization problem. From the discussion of complexity above, we can see that the hybrid heuristic can be easily applied to physical mapping problems that involve semi-dependent components by using the independent metric as the local optimization objective for each semi-dependent component. As demonstrated by the simulation results in Chapter 3, this approach is effective since the local optimization objectives are actually parts of the global objective, and therefore, the locally good candidates are likely to be also globally good. However, if components in the target physical mapping problem all depend on one another, the effectiveness of the hybrid heuristic may be limited since it may not be clear what local optimization objective can be used for each component.

Finally, for physical mapping problems that have a particular structure, there may be specialized algorithms developed previously that can solve such special problems more efficiently than general optimization algorithms. For example, to map components to nodes along a selected route with the objective of maximizing the overall throughput, a dynamic programming algorithm is used in [48]. Choi et al. uses a shortest-path algorithm on a transformed network graph to select intermediate processing sites between two end points [21]. Gu and Nahrstedt uses a shortest-path algorithm to find a service path that minimizes the resource usage [60]. The Matchmaking framework [112] maps a computation task to an appropriate resource that optimizes user-specified criteria. Extending the Matchmaking framework, Liu et al. [87] and Raman et al. [113] propose a number of heuristic algorithms to solve the multi-resource and the resource co-allocation problems, respectively. In the ESM example in Figure 3.2 from Chapter 3, the problem of selecting ESMPs to minimize the sum of latencies can be formulated into a *p*-median problem [29], and many previous studies have presented algorithms for solving this and other variations of the facility location problem [31], for example, [118, 110]. Many of these algorithms are approximate algorithms, and therefore, they are much more efficient than generic algorithms, and they often provide theoretical bounds on optimality.

4.3.2 Algorithm selection

The algorithms discussed in the previous section are suitable under different circumstances. Therefore, for each physical mapping problem, the synthesizer needs to select the most appropriate algorithm. Ideally, when solving a physical mapping problem, the synthesizer should be able to select the best optimization algorithm automatically since the service provider may not be an optimization expert and does not know the properties of different algorithms well enough to specify which algorithm should be used.

The best choice of optimization technique depends on two major factors. The first is the properties of the problem itself, for example, when the problem size is small, an expensive algorithm can be used to achieve better optimality. Similarly, specialized algorithms can be used for problems of particular forms. For example, path-based component selection can be performed using variations of the shortest-path algorithm, problems with semi-dependent components can use the hybrid heuristic, and so on. These properties are not service-specific and can be analyzed by the synthesizer automatically to choose an algorithm. For example, using the video conferencing recipe, the synthesizer generates an abstract configuration and an objective function. By looking at the number of candidates for each component, the synthesizer can determine the problem size and estimate the feasibility of using an expensive algorithm such as exhaustive search. In addition, the synthesizer can also analyze the objective function and discover that the mapping problem involves semi-dependent components. Therefore, the hybrid heuristic algorithm can be used.

The second factor is the desired trade-off between the optimality of the resulting configuration and the cost of optimization. The optimality is measured by comparing the configuration produced by an algorithm with the actual optimal configuration. The cost is

basically the computation time required to perform the optimization, given the particular synthesizer implementation. The desired trade-off for each service is service-specific and is determined by the provider. For example, a provider may have a constraint of 2 seconds per request on the maximum optimization cost, but it may not require near-optimal configurations. Another provider may want the composed configuration for every request to be at most 20% worse than the actual optimal solution. To use the service-specific trade-off for algorithm selection, the synthesizer needs to allow a service provider to specify the desired trade-off between optimization cost and optimality. This can be achieved by providing the following simple API for specifying the trade-off in a service recipe.

```
setAlgCostConstraint(CostSpec maxCost);
setAlgOptimalityConstraint(OptSpec minOptimality);
```

Using this interface, providers can specify two constraints in their recipes: the maximum optimization cost and the minimum optimality of the resulting configuration. In general, it is difficult to obtain the exact optimality property (and, to a lesser degree, the cost property) of an optimization algorithm relative to the actual optimum. Therefore, the synthesizer may not be able to guarantee that the constraints will be met. In other words, the constraints serve as guidelines for algorithm selection, and the synthesizer selects an algorithm that, to the best of its knowledge, can satisfy the provider's constraints.

Now that we have discussed what information is necessary for the synthesizer to perform algorithm selection (i.e., properties of the mapping problem and the desired trade-off), let us look at how the synthesizer uses such information to select the best algorithm. First, we define the following notations.

- F is the objective function for the physical mapping problem.
- $A = \{a_1, a_2, \ldots, a_n\}$ is the set of built-in optimization algorithms of the synthesizer.
- For each a_i, $C(a_i, s)$ is the average optimization cost of a_i when the problem size is s. As discussed above, the cost refers to the optimization time of a_i given the particular synthesizer implementation.
- For each a_i, $P(a_i, s)$ is the average optimality of a_i when the problem size is s, i.e., how much worse a_i is than the actual optimum. For example, if $P(a_i, s)$ is 1.3, then when the problem size is s, the solution produced by a_i is on average 30% worse than the actual optimum. As discussed above, it is in general difficult to obtain a function $P()$ that provides the exact optimality property. Later we discuss how a synthesizer can obtain estimates of the optimality property.
- T_o and T_c represent the optimality and cost constraints specified in the recipe by the provider, respectively. For example, if T_o is 1.3, it means that the provider needs solutions that are at most 30% worse than the actual optimum.

Given these notations, the algorithm selection process can be divided into the following five steps.

- **Determining eligible algorithms**: General optimization algorithms such as simulated annealing can be applied to all mapping problems, but specialized algorithms only apply to special cases. In this step, the synthesizer analyzes F to determine what algorithms can be used. For example, if there are semi-dependent components in F, the Hybrid and HybridSA algorithms described earlier become eligible. If F is of a form that can be formulated into the p-median problem, then a special algorithm can be used, for example, the approximate algorithm in [18].

  One issue that needs to be addressed is that it is difficult to define a general way to express the required features in F for a particular specialized algorithm. For example, for Hybrid/HybridSA, it needs to express "components with both dependent and independent metrics". For the p-median problem, the required features are completely different. One possible solution to this issue is to implement every specialized algorithm as a "plugin" to the synthesizer, and each plugin also includes the logic for determining whether a particular objective function can be formulated into the special problem that can be handled by the corresponding algorithm.

  After examining all algorithms, e.g., by invoking the eligibility determination logic in each plugin, the synthesizer has a set of eligible algorithms $A' \subseteq A$. Note that if a particular specialized algorithm is clearly superior than other built-in algorithms, the corresponding plugin can also instruct the synthesizer to skip the three steps below and simply select the algorithm.

- **Determining problem size**: To determine the problem size, the synthesizer looks at the number of candidates of each required component. Based on how F is formed, the synthesizer can then calculate the problem size S. For example, if F requires three components to be selected together, and each component has m candidates, then $S = m^3$.

- **Applying cost constraint**: The synthesizer finds a set of algorithms A_1 such that $A_1 \subseteq A'$ and for all $a \in A_1$, $C(a,S) \leq T_c$.

- **Applying optimality constraint**: The synthesizer finds a set of algorithms A_2 such that $A_2 \subseteq A_1$ and for all $a \in A_2$, $P(a,S) \leq T_o$.

- **Selecting an algorithm**: After the above four steps, if A_2 contains a single algorithm a, the synthesizer will use a to solve the optimization problem. If A_2 contains more than one algorithm, any of them can satisfy both the cost and optimality constraints specified by the provider. Therefore, the synthesizer can use any algorithm in A_2 to perform the optimization. For example, the synthesizer can select the one with the lowest cost. However, if A_2 is empty, it means that none of the synthesizer's built-in algorithms can satisfy both constraints. One possible approach in this case is that the synthesizer can select an algorithm whose properties are closest to the constraints, for example, by relaxing the constraints and repeating the steps above. Another possibility is that the synthesizer can notify the provider so that the provider can relax the constraints.

Given the above algorithm selection process, one remaining question that needs to be addressed is how does the synthesizer obtain the functions $P()$ and $C()$, which represent the optimality and cost properties of the built-in algorithms, respectively? Below we outline two possible approaches.

- **Offline**: The properties of the built-in algorithms can be collected offline, i.e., before the synthesizer becomes operational and starts handling user requests. For example, the synthesizer can use simulated requests to measure the optimality and cost of the built-in algorithms at different problem sizes. Such data can then be used to estimate the cost and optimality when the synthesizer is handling real requests. Of course, for some algorithms, the cost and optimality properties depend a lot on the actual optimization problem, for example, how many variables are there in the objective function, etc. Therefore, the cost and optimality data needs to be collected specifically for the service that the synthesizer is handling.

- **Online**: Another possibility is that the synthesizer can "learn" about the cost and optimality properties of the built-in algorithms from the experiences of handling actual user requests. I.e., after handling some user requests using different optimization algorithms, the synthesizer will have accumulated some cost and optimality data similar to those collected using the offline method above. Such data can then be used to guide the algorithm selection process for later requests. One problem with this approach is that the synthesizer cannot handle the initial requests well since it does not have enough data to know which algorithm is the best.

Of course, it is also possible to combine the two approaches. For example, the offline method can be used to obtain some baseline data that can be used by the synthesizer to handle the initial user requests. After the synthesizer has handled a sufficient number of requests, the online data can then be used to improve the baseline estimates.

Later in Section 4.5, we use the video conferencing service as an example to illustrate how to collect the cost and optimality data using the offline method and how to use such data to perform algorithm selection as described above.

4.4 Implementation

We have implemented the synthesizer, including the facility and interpreter modules, in the Java programming language to take advantage of the class loading capability for our recipe implementation. The facility interface consists of a set of Java classes and interfaces exported by the facility module. A service recipe can be written by designing a Java class that "implements" the `Recipe` interface, which provides a method for the synthesizer to invoke the recipe. A recipe will use the classes and methods described earlier to construct the abstract configuration and the objective function.

To interpret a recipe, the interpreter module dynamically loads the recipe class using Java class loading capability and executes the recipe through the invocation method. During

the execution, the recipe class accesses the facility module as described above. After the execution, the facility module has both the abstract configuration and objective function. The interpreter module then invokes a method in the facility module to start the physical mapping phase.

As described in Chapter 3, we have designed and implemented the Network-Sensitive Service Discovery (NSSD) infrastructure to support physical mapping. In addition, to support the `LatencyM` metric used in objective functions, we use the Global Network Positioning (GNP) [97] approach to provide estimates of network latencies between nodes. The `BandwidthM` metric is not supported since we have not integrated a bandwidth measurement infrastructure.

To solve the optimization problem of physical mapping, we have implemented the following optimization algorithms in the synthesizer

- *SA(R)*: This means simulated annealing with temperature reduction ratio *R*, which affects the optimization cost and the optimality of the resulting configuration. The optimization cost of this algorithm grows slowly with the problem size since the number of iterations is independent of the problem size.

- *Hybrid(m)*: The hybrid heuristic is applied to reduce the problem size by choosing *m* "locally good" candidates for each semi-dependent component. The exhaustive search algorithm is then used to solve the reduced problem. Although the complexity of this algorithm is much lower than that of exhaustive search (e.g., $O(n^3)$ vs. $O(n^5)$ in the video conferencing scenario), it still grows rapidly with the problem size and thus becomes infeasible quickly.

- *HybridSA(m)*: This is the same as Hybrid(*m*) except that simulated annealing is used instead of exhaustive search. Therefore, the cost property is similar to generic simulated annealing. Note that we increase the temperature reduction ratio with *m* to achieve better optimality (at higher costs).

In addition, we have implemented the generic exhaustive search algorithm, which is expensive but is able to find the actual optimal solution. We have also implemented a heuristic algorithm for the *p*-median problem, which appears in the selection of ESM proxies, for example.

4.5 Evaluation

In this section, we evaluate several aspects of the recipe-based self-configuration architecture. In Section 4.5.1, we illustrate the expressiveness of our recipe representation using several examples. In Section 4.5.2, we look at the effectiveness and development cost of recipe-based self-configuration. In Section 4.5.3, we evaluate the effectiveness of algorithms based on the hybrid heuristic. Finally, we use an example in Section 4.5.4 to illustrate how the synthesizer collects algorithm properties offline and performs algorithm selection accordingly.

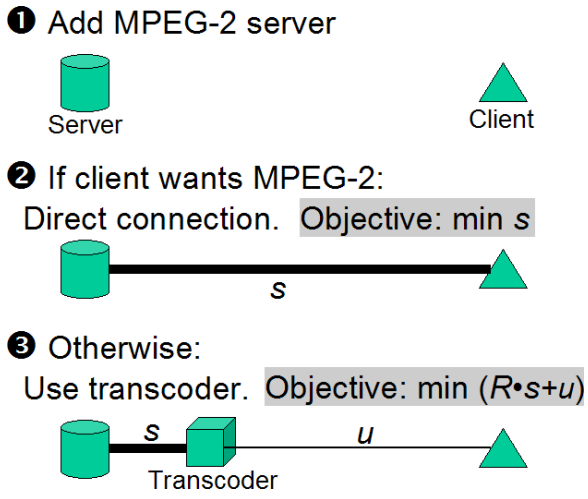


Figure 4.12: Service-specific knowledge for the video streaming service.

```
❶ AbsConf conf = new AbsConf();
  Term obj = new Term(new Double(0.0));
  AbsComp vserver = conf.addComp("MPEG2VideoServer");

❷ if (client.getProperty("VideoIn").equals("MPEG2")) {
    conf.addConn(vserver, client);
    obj.add(new Term(new LatencyM(vserver, client)));

❸ } else {
    AbsComp transcoder = conf.addComp("Transcoder");
    conf.addConn(vserver, transcoder);
    Term partialObj = new Term(new LatencyM(vserver, transcoder));
    partialObj.multiplyBy(new Term(new Double(MPEG2BitRate)));
    obj.add(partialObj);
    conf.addConn(transcoder, client);
    partialObj = new Term(new LatencyM(transcoder, client));
    partialObj.multiplyBy(new Term(new Double(MPEG4BitRate)));
    obj.add(partialObj);
  }
  Function objfunc = new Function(obj);
```

Figure 4.13: A video streaming recipe.

4.5.1 Expressiveness of recipe representation

To evaluate the expressiveness of the recipe representation, we apply the recipe representation to the other two services in Section 4.2.1 in addition to video conferencing. Figure 4.12 illustrates the service-specific knowledge for the video streaming service in Figure 4.2, and Figure 4.13 shows a recipe based on the knowledge. Similarly, Figure 4.14 illustrates the service-specific knowledge for the interactive search service in Figure 4.3, and the corresponding recipe is shown in Figure 4.15.

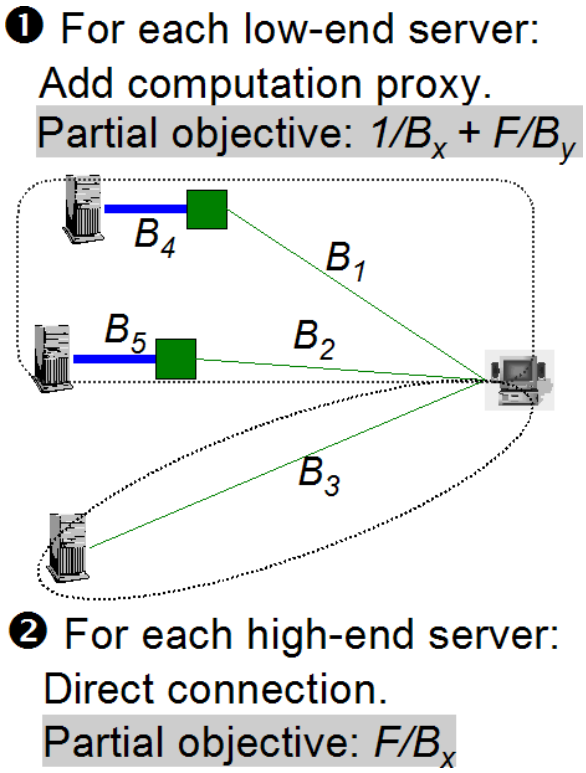


Figure 4.14: Service-specific knowledge for the interactive search service.

The video streaming recipe constructs different abstract solutions depending on whether the user is requesting MPEG-2 or MPEG-4 video. The user is represented by the physical component `client` extracted from the user request. The objective function in Figure 4.2 is easily constructed using the objective function API.

In the interactive search service recipe, `client` (the user) and `storageServers` (the storage servers) are fixed physical components extracted from the user request. For each server, if it does not have sufficient computation resources, the recipe adds a computation proxy and its "contribution" to the global objective in Figure 4.3. In these examples, the recipe representation allows us to express the service-specific knowledge easily and flexibly.

Of course, different providers building the same service may design different recipes. For example, instead of the recipe in Figure 4.13, a provider of a video streaming service may come up with the simpler recipe in Figure 4.16 that constructs an objective function involving only local optimization. Using this "local" recipe, the synthesizer may not produce the globally optimal configuration, but the synthesizer will spend significantly less time in configuring the service. In this particular recipe, the objective function does not involve the transcoder, so the synthesizer can select a random candidate. However, a recipe may also construct objective functions for individual components so that each component is selected using its own local objective.

Figure 4.15: An interactive search service recipe.

```
AbsConf conf = new AbsConf();
AbsComp vserver = conf.addComp("MPEG2VideoServer");

if (client.getProperty("VideoIn").equals("MPEG2")) {
  conf.addConn(vserver,client);
} else {
  AbsComp transcoder = conf.addComp("Transcoder");
  conf.addConn(vserver,transcoder);
  conf.addConn(transcoder,client);
}
Term obj = new Term(new LatencyM(vserver,client));
Function objfunc = new Function(obj);
```

Figure 4.16: A simpler video streaming recipe.

To see the effect of the different video streaming recipes, we perform a set of simple simulations. We obtain the GNP coordinates of 869 Internet nodes from the GNP project [97], so the latencies between the nodes are realistic. We randomly select 600 nodes to represent clients, servers, and transcoders (200 each). For each client, we find the best pair of server and transcoder using either the “global” recipe or the “local” recipe. This

is repeated 20 times for a total of 4000 scenarios. We compute the resource consumption (i.e., the global objective) of the resulting configuration in each scenario and also measure the time it takes to find the configuration. The average resource consumption is 90.79 for global and 203.41 for local. The time measurements (total of 4000 scenarios) are 158.20 seconds for global and 2.05 seconds for local. The results clearly show the trade-off between optimality and optimization cost in the two recipe designs. Therefore, providers can write different recipes according to their service-specific requirements.

4.5.2 Effectiveness and cost of recipe-based self-configuration

To evaluate the effectiveness of our recipe-based self-configuration architecture, we perform a set of experiments where we use the video conferencing service recipe described earlier to create a self-configuring video conferencing service. Since our focus is on how the synthesizer uses the recipe to perform global configuration, in our experiments the synthesizer does not actually deploy the components. Instead, we look at the efficiency with which the synthesizer composes the optimal configuration and the optimality of the selection of components in the composed configuration. The efficiency is measured by the *configuration time per request* metric, which is the time needed by the synthesizer to compose the optimal configuration. We measure the optimality using the *relative optimality* metric, which compares the optimality of the resulting configuration to the actual optimum, for example, 1.3 means 30% worse than the actual optimum. The actual optimum is obtained using the exhaustive search algorithm. When the problem size is large, exhaustive search becomes infeasible, and we compute the relative optimality by comparing the result to the best configuration in the particular experiment.

Our experiments use the video conferencing scenario in Figure 4.4, i.e., we assume each user request represents five participants (2 NMs, 2 VICs, and 1 HH), and it requires the self-configuring service to find a VGW, a HHP, and 3 ESMPs to optimize the shown objective. As with the simulations described previously, we use the set of 869 nodes measured by the GNP project so that the latencies between nodes are realistic. For each experiment, we randomly select (from the 869 nodes) n candidates for each of VGW, HHP, and ESMP, and we generate 20 requests (each of which consists of 5 participants). The synthesizer uses the recipe to compose the optimal configuration for each of the requests by selecting the best candidates of VGW, HHP, and ESMP. In our experiments, n ranges from 5 to 200. For each n, the above experiment is repeated 10 times, resulting in 10 different candidate distributions and a total of 200 conferencing sessions.

For each request, the synthesizer executes the recipe in Figure 4.11 to compose the optimal configuration. As described earlier, the synthesizer selects the best candidates by querying the NSSD infrastructure. In our experiments, the synthesizer and the NSSD infrastructure are both run on the same desktop machine with a Pentium III 933 MHz CPU, 512 MB of RAM, and Red Hat Linux 7.1. The synthesizer is run using J2SE 1.4.2, and it communicates with NSSD through TCP sockets.

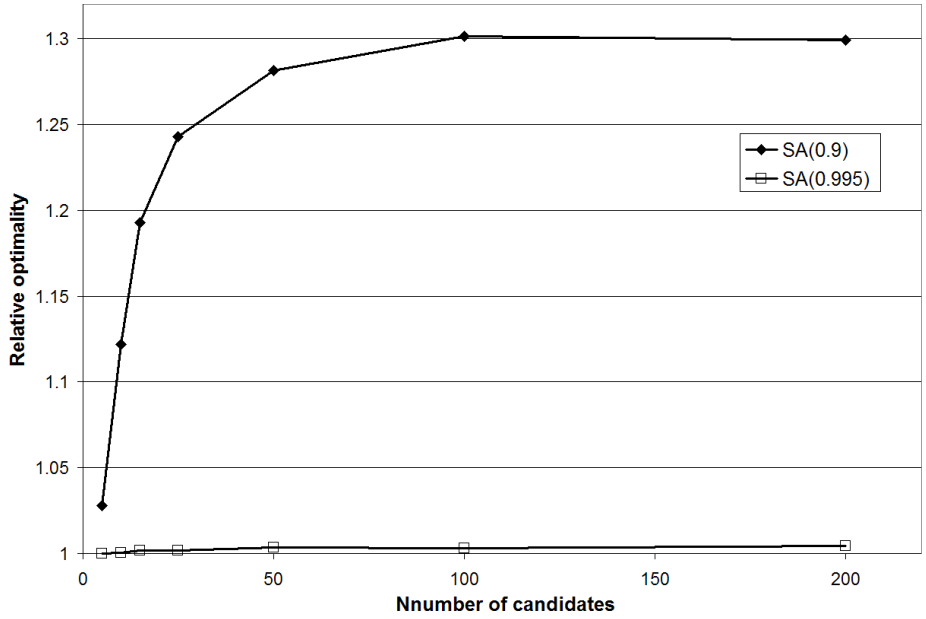


Figure 4.17: Relative optimality: generic simulated annealing.

Efficiency and optimality. We now present results from the first set of experiments where the synthesizer uses the generic simulated annealing optimization algorithm with two different temperature reduction ratios, 0.9 and 0.995. We measure the configuration time for each request and compute the relative optimality of the configurations produced by the two algorithms. The results are averaged over the 200 requests of 10 different candidate distributions.

The average relative optimality of the two algorithms can be seen in Figure 4.17. Since exhaustive search algorithm becomes infeasible with the larger n in these experiments, we compute the relative optimality by comparing the two algorithms with each other. We can see that the relative optimality of SA(0.995) is always close to 1.0. This is because in most sessions, the configuration composed by SA(0.995) is better than that composed by SA(0.9). Therefore, in most sessions, SA(0.995) has relative optimality 1.0, and thus its average relative optimality is close to 1.0. On the other hand, when n is large, on average the configuration found by SA(0.9) is about 30% worse than that found by SA(0.995), which is expected since SA(0.9) reduces the "temperature" much more quickly and therefore is more likely to fall into local optima.

Figure 4.18 shows the average configuration time per request for the two algorithms. We can see that because of the properties of the simulated annealing algorithm, the configuration time only increases slowly with the problem size. Even with the more expensive SA(0.995), the configuration time is about 5 seconds per request when n is 200. Given that we are targeting session-oriented services with session duration on the order of minutes or hours, this can be considered a reasonable price to pay in order to find a better configuration. On the other hand, the configuration time can be reduced by a factor of roughly 3 to 10

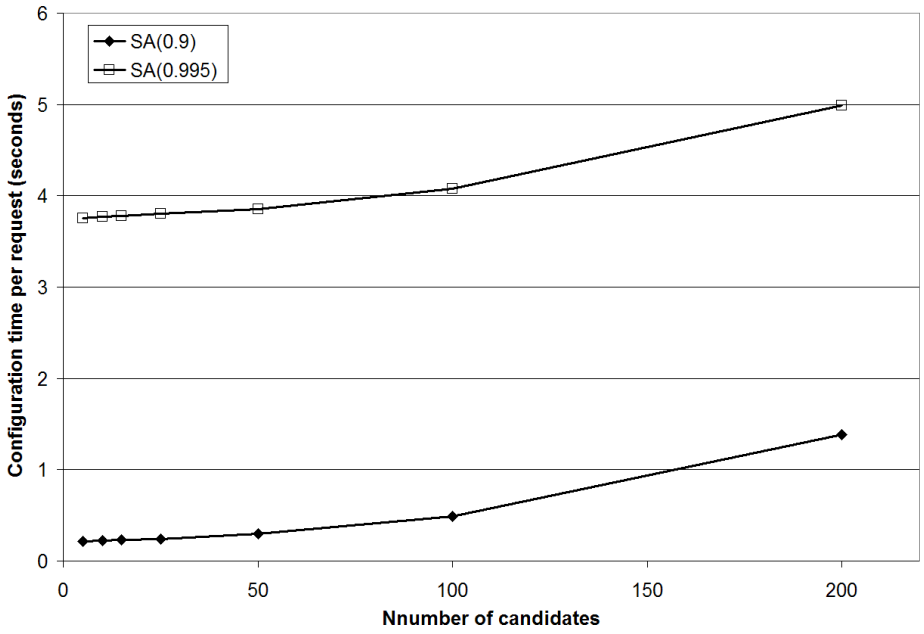


Figure 4.18: Configuration time per request: generic simulated annealing.

by using the SA(0.9) algorithm if the decrease of optimality as shown earlier is acceptable.

Breakdown of configuration time. To look more closely at how much time the different self-configuration tasks consume, we categorize the configuration time into three parts:

- **Objective evaluation.** This is the total time that the synthesizer spends in evaluating the value of the objective function for a particular feasible configuration.
- **Service discovery.** This is the total time the synthesizer spends in querying the NSSD infrastructure to obtain the eligible candidates, including the communication overhead.
- **Synthesizer.** This is the total time the synthesizer spends on everything else, for example, executing the recipe, applying optimization algorithms, finding feasible configurations, and so on.

Figure 4.19 shows a breakdown of the configuration time for n from 50 to 100. We can see that service discovery takes a fixed amount of time for a particular problem size, and the time increases with the problem size. This is because properties of the current NSSD implementation, the underlying SLP implementation, and the Java implementation of the SLP API cause the query/reply time to increase with the number of candidates returned. The generic simulated annealing algorithm requires all candidates, and therefore the service discovery time increases with n.

Excluding the service discovery time, we can see that the synthesizer spends almost all the configuration time in the evaluation of objective values. Recall that our objective

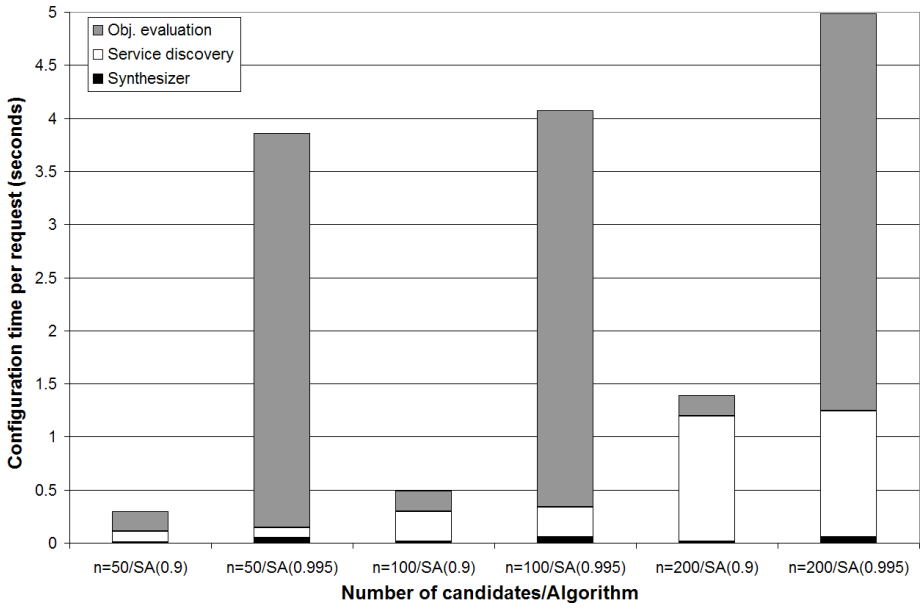


Figure 4.19: Breakdown of configuration time per request: generic simulated annealing.

function API can be used in a recipe to construct a simple tree-like data structure representing the objective function. Given a feasible configuration, its objective value is evaluated by traversing this tree. Since the data structure is not efficient, and the evaluation needs to be done for every feasible configuration considered by the optimization algorithm, the resulting overhead is significant compared to other tasks that need to be performed by the synthesizer. Such overhead can be greatly reduced by, for example, using a more efficient data structure to represent the objective function, implementing the objective evaluation mechanisms in a lower-level language, and so on.

To demonstrate this, we conducted a set of simple experiments that measure the overhead of traversing the data structure representing the objective function in the video conferencing example. We extract the objective function data structure and traversal code from the synthesizer Java implementation into a standalone program, and we compare this program with a comparable implementation written in C++. We measure the time it takes for either program to perform 10^6 traversals. Note that this measured overhead does not include the overhead of evaluating the metric terms, i.e., the latencies, which are replaced by random numbers in the experiments. The experiments are conducted on a desktop machine with a Pentium 4 3.0GHz CPU, 512 MB of RAM, and Fedora Core 2 Linux. The Java program is compiled and run with J2SE 1.4.2, and the C++ program is compiled with gcc 3.3.3 without compiler optimizations. The results show that the Java version, representing our prototype implementation, takes 61.62 seconds to perform 10^6 traversals, and the C++ version takes 1.09 seconds. In other words, by porting the traversal code to C++, the speed of traversing the objective function can be increased by a factor of more than 50.

Excluding the service discovery and objective evaluation time, we can see that the overhead incurred by the synthesizer is insignificant. In other words, in our recipe-based self-configuration architecture, very little overhead is incurred by the abstraction of service-specific knowledge into recipes and the execution of recipes to compose optimal configurations.

Development cost. Finally, we look at the required development cost for a service provider using the recipe-based self-configuration approach. As mentioned previously, the development cost includes different aspects such as the expertise required in different areas, the lines of code that the provider needs to implement, and so on. Here we only look at the more concrete metric, that is, the number of lines of code in the implementation.

We compare our recipe-based approach with the service-specific approach. Previously, we built a self-configuring video conferencing service using the service-specific approach [122], i.e., we built a service-specific self-configuration module that integrates the supporting infrastructures and is able to compose service configurations for video conferencing sessions. This service-specific self-configuration module consists of 4141 lines of C++ code, including the hard-wired self-configuration knowledge and the integration of supporting infrastructures. Note that this does not include the infrastructure code such as NSSD and SLP, the required libraries such as libxml2 [86] for XML processing, and the actual video conferencing components such as the video conferencing gateway [67]. In contrast, using the recipe-based self-configuration architecture described here, a service provider can build a similar self-configuring service by designing the recipe in Figure 4.11. Including the necessary extra lines (e.g., "import"), the actual recipe consists of 80 lines of Java code.

Of course, our recipe representation cannot possibly cover all conceivable service-specific tasks that a provider may want the self-configuring service to do. For example, if a provider has a special algorithm that is only applicable for the particular physical mapping problem in its service, our recipe-based approach will not be able to take advantage of the algorithm since the recipe representation does not allow the specification of an algorithm. However, our assumption is that most service providers do not have expertise in the area of optimization, and the built-in algorithms should provide reasonable performance for most providers. Moreover, if the provider does have a specialized algorithm, it may be implemented as a plugin to the synthesizer to extend the capability of the synthesizer. The same principle extends beyond the algorithm example, i.e., our aim is to capture the aspects of service-specific knowledge that are important for most service providers, and if a provider has additional expertise, our framework may be used as a basis for further customization.

The comparison above between recipe-based and service-specific approaches demonstrates that our recipe-based self-configuration architecture enables the use of providers' service-specific knowledge in self-configuration with dramatically lower development cost than what is required in the service-specific approach.

On the other hand, when compared with the generic approach, by paying a slightly

higher development cost (e.g., 80 lines vs. 0), a service provider can customize the self-configuration process using a broad range of service-specific knowledge. As mentioned earlier, since the generic type-based approach cannot take advantage of the service-specific knowledge, its effectiveness is limited in two aspects: (1) it is inefficient due to the large search space for feasible abstract configurations, and (2) the resulting configuration is sub-optimal because of generic component selection criteria. Below we briefly discuss these two issues.

In our approach, the abstract configuration for a request is constructed by executing the recipe. Therefore, it is very efficient since a "search" is not necessary. In contrast, in the generic type-based approach, the generic self-configuration must look for combinations of of components that can satisfy the user-requested type. For example, let us first consider the simple path model, i.e., all components in a configuration form a path from a sender to a receiver. Suppose that there are k hops on the path excluding the sender and the receiver. At each hop, there are n standardized input types and n output types, and there are n^2 standardized component types, i.e., one component type for each input/output combination. Suppose that only r of the n input types and r of the n output types at each hop are relevant to the target service, e.g., video streaming. Therefore, at the first hop from the receiver, the generic approach needs to look at the n^2 component types to find the r candidates that output the receiver's input type. At the second hop, it needs to look at n^2 components to find r matching components for each of the r components at the first hop, and so on. As a result, the number of feasible abstract configurations is $O(r^k)$, and the complexity of the search at each hop is $O(rn^2)$.

Of course, the assumptions in the above simple calculations may not be realistic, and optimizations such as caching may be applied to make the search more efficient such that its performance is acceptable for small scale problems. However, if we go beyond the simple path model and consider configurations involving multi-point communication, then the complexity can go up significantly. For example, suppose that we want to find a configuration for multi-point communication among n users. We can use an ESM-like service to support all n users. We can also use ESM for $n-1$ users and connect the remaining 1 user using unicast, and so on. This results in n types of configuration even before considering input/output types. Also, if we use ESM proxies for multicast, the generic module needs to try different number of ESM proxies in the configuration. Since it has no service-specific knowledge, it may consider a wide range of numbers, say 1 to n, to be feasible. This adds yet another level of complexity to the search and results in a large number of feasible abstract configurations.

Furthermore, to select a feasible abstract configuration, the generic module can only use some generic criteria, e.g., minimizing the number of components. However, this may result in sub-optimal or even unreasonable configurations. In the above ESM example, if the generic module tries to minimize the number of components, it will select a configuration with a single ESM proxy to support n users, which defeats the purpose of multicast. Finally, similar to the selection of a feasible abstract configuration, the generic module can also only use some generic criteria to select the physical candidates for components in the abstract

configuration. Again, this may results in sub-optimal configurations. In other words, even if the complexity of the search for feasible abstract configurations can be reduced to an acceptable level, the physical configuration selected using generic optimization criteria may be unacceptable. For example, if the generic module selects the lowest-cost candidates to minimize the cost of the video conferencing configuration, the selected video conferencing gateway and handheld proxy may be far away from the NetMeeting users and the handheld users, respectively (in terms of latency), resulting in poor performance. Therefore, based on the above factors, we argue that by using the abstracted service-specific knowledge, our approach can achieve much higher effectiveness for many services than what the generic approach can.

4.5.3 Effectiveness of hybrid heuristic

In Chapter 3, we presented simulation results that demonstrate that algorithms based on the hybrid heuristic can find solutions that are much better in terms of global optimality than those found by simply combining locally optimal solutions. In this section, we evaluate the effectiveness of the hybrid heuristic in the context of the operation of the whole recipe-based self-configuration architecture by comparing the efficiency and optimality of the Hybrid and HybridSA algorithms with those of the generic exhaustive search and SA algorithms.

First, we perform experiments with small-scale problems, i.e., with n from 5 to 15. Since exhaustive search is feasible in this range, we compute the relative optimality of different algorithms by comparing the results with the actual optima obtained from the exhaustive search algorithm. Figure 4.20 shows the relative optimality of four algorithms. Exhaustive search always finds the actual optima and has relative optimality 1.0. The two algorithms based on the hybrid heuristic both outperform the generic simulated annealing algorithm, and their relative optimality also deteriorates slower than simulated annealing. As expected, hybrid(4) performs better than HSA(4), since hybrid uses exhaustive search in the reduced search space while HSA uses simulated annealing.

The average configuration time per request of the different algorithms is shown in Figure 4.21 (note the log scale on the Y-axis). Not surprisingly, the configuration time of the exhaustive search and hybrid(4) algorithms grows very rapidly with the problem size, since the sizes of their search spaces are $O(n^5)$ and $O(n^3)$, respectively. In fact, the results for the two algorithms match almost exactly with curves (not shown) for n^5 and n^3, respectively. On the other hand, the configuration time of the simulated annealing and HSA(4) algorithms grows very slowly (not evident in this graph) with the problem size. HSA(4) has a lower configuration time since it only performs simulated annealing on a reduced search space. From these results, we can see that the HSA algorithm provides relatively good optimality with very low cost, and the hybrid algorithm provides very good optimality with relatively low cost.

To look at the effectiveness of the hybrid heuristic for larger-scale problems, we compare the SA(0.995) and HSA(4) algorithms with n from 5 to 200. The relative optimality is

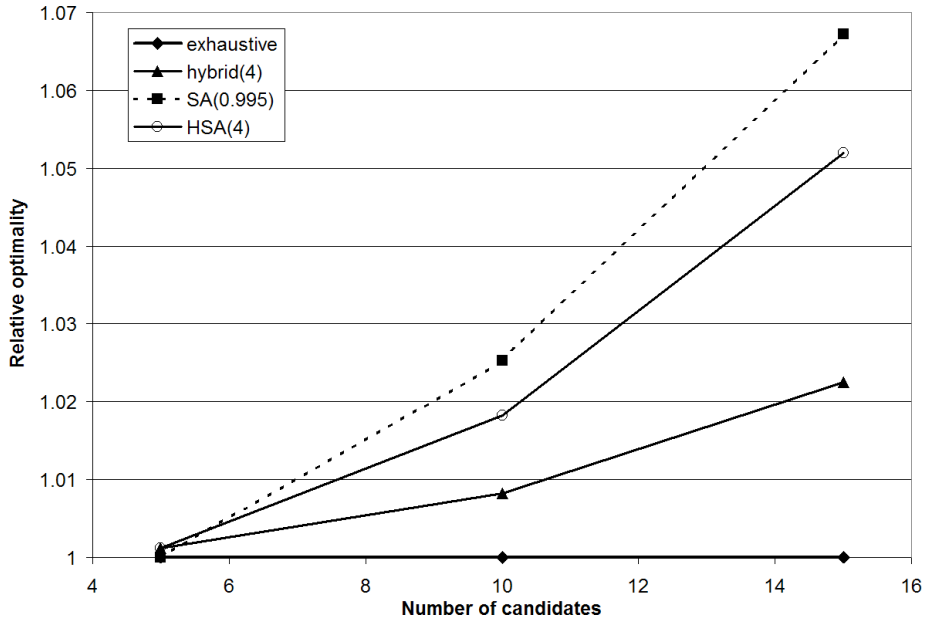


Figure 4.20: Relative optimality: comparison for small-scale problems.

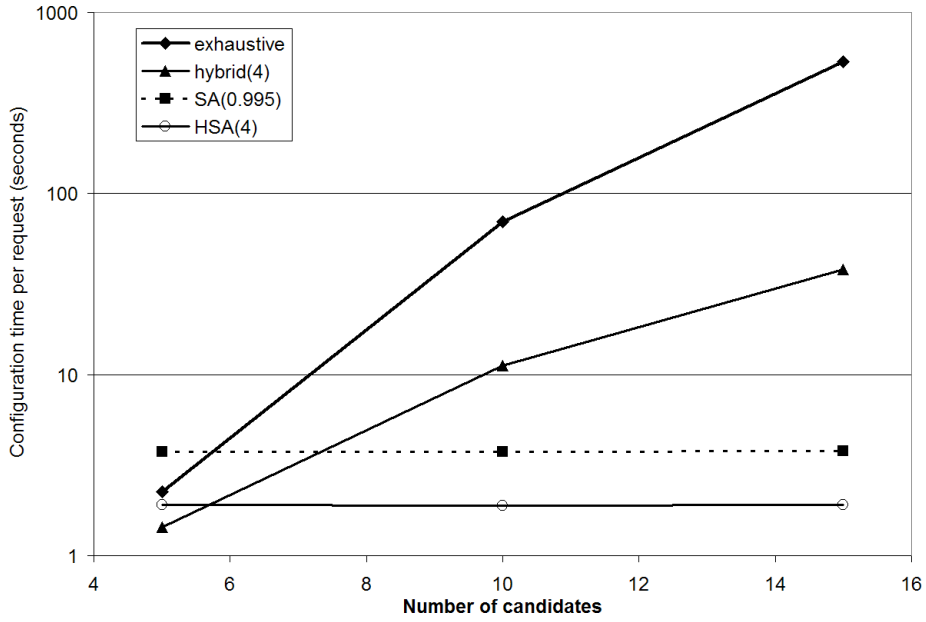


Figure 4.21: Configuration time per request: comparison for small-scale problems.

computed by comparing the two algorithms with each other. The optimality result and the configuration time per session are shown in Figures 4.22 and 4.23, respectively. The results basically follow the trend seen in the above results for small-scale problems. The HSA(4) algorithm performs better than SA(0.995) in terms of both the optimality and the configu-

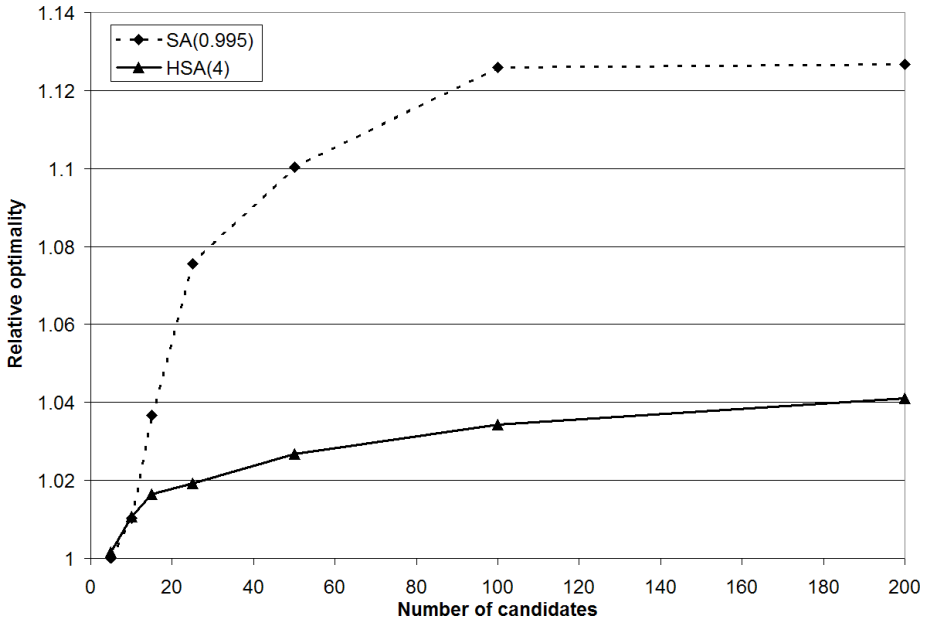


Figure 4.22: Relative optimality: comparison for larger-scale problems.

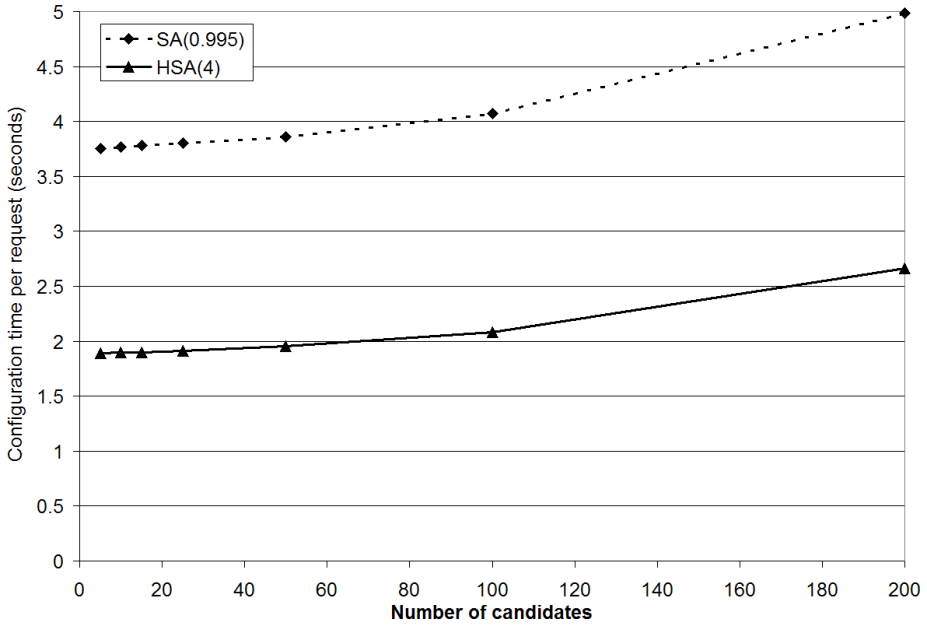


Figure 4.23: Configuration time per request: comparison for larger-scale problems.

ration time. This is because HSA uses the hybrid heuristic and focuses on a reduced search space that is more likely to contain the actual optimum than other parts of the search space. We also see that relative optimality curves of both algorithms become flat when the scale is large, which suggests that although the optimality of SA deteriorates faster at small n, both

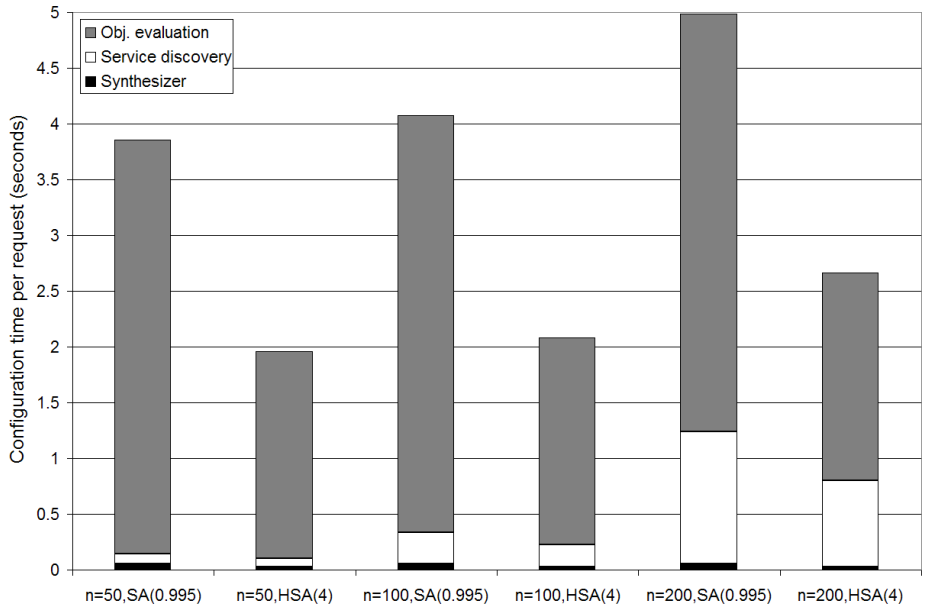


Figure 4.24: Breakdown of configuration time per request: comparison for larger-scale problems.

algorithms deteriorate at roughly the same rate after the problem size becomes sufficiently large.

Finally, we again break down the configuration into three parts and show the results in Figure 4.24. Similar to the breakdown results earlier, we see that the most significant overhead lies in the evaluation of the objective function. The overhead caused by recipe interpretation and other tasks of the synthesizer (excluding service discovery) is negligible in comparison. We also see that at the same problem size, HSA(4) has a shorter service discovery time than SA(0.995). This is because, as discussed earlier, the service discovery time is proportional to the number of candidates returned, and HSA(4) requests for fewer candidates than SA does. For example, when n is 200, SA(0.995) performs 5 service discovery operations, each of which returns 200 candidates. In contrast, HSA(4) performs 2 service discovery operations that return 4 candidates each and 3 service discovery operations that return 200 candidates each.

4.5.4 Algorithm selection using offline data

Finally, in this section we illustrate how the synthesizer can obtain the cost and optimality data offline and use the data for algorithm selection. As discussed earlier, the cost and optimality properties of the synthesizer's built-in algorithms can be measured offline using simulated requests. We perform a set of experiments to collect such data for the various algorithms we implemented in the synthesizer. The experiment setup is the same as that in the experiments discussed above. Again, because it becomes infeasible to use the more

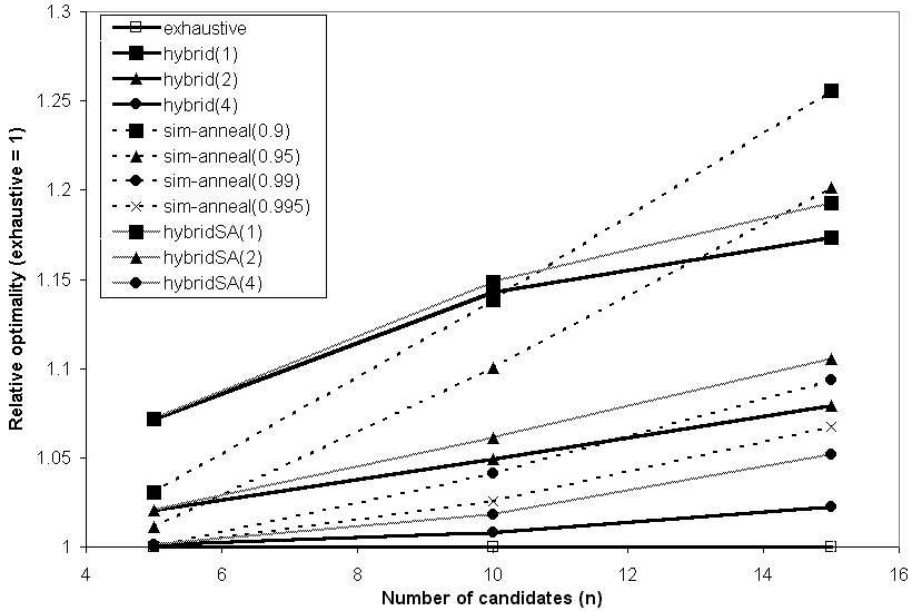


Figure 4.25: Optimality of resulting configurations: small-scale problems.

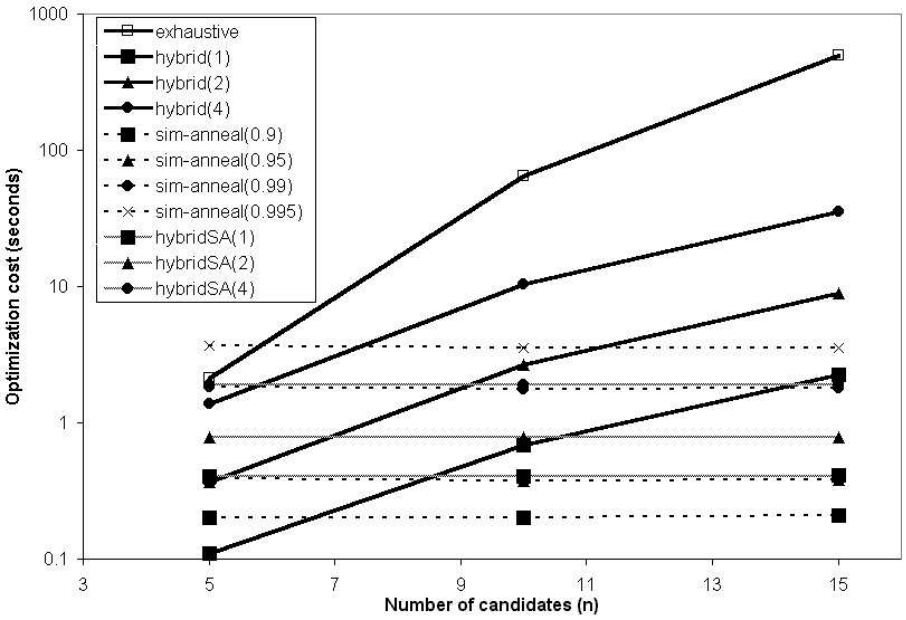


Figure 4.26: Optimization cost: small-scale problems.

expensive algorithms for larger scale problems, the results are presented in two parts.

For n between 5 and 15, Figures 4.25 and 4.26 show the optimality and cost of the different built-in algorithms, respectively. Figures 4.27 and 4.28 show the results for n between 25 and 200. The trends in these results match those we have seen in the previous

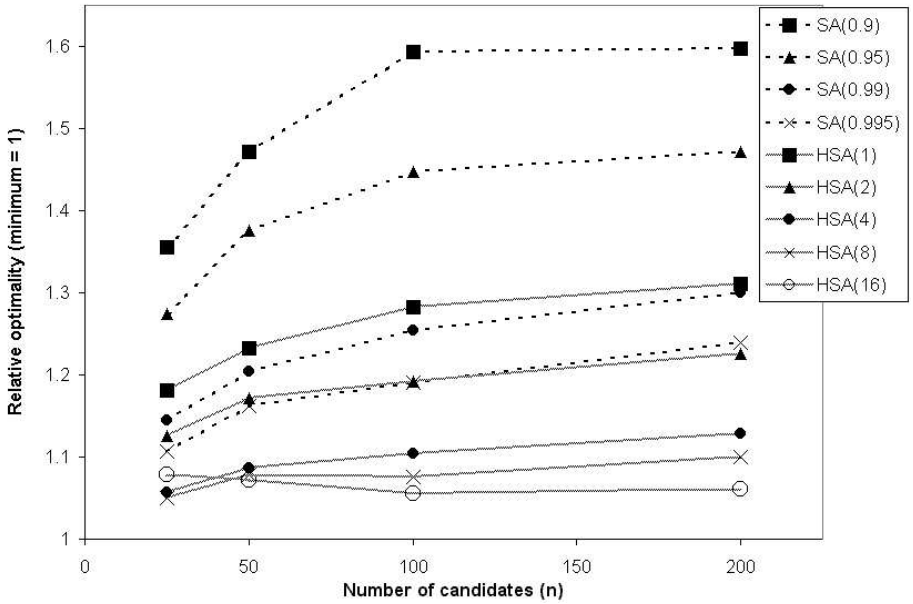


Figure 4.27: Optimality of resulting configurations: larger-scale problems.

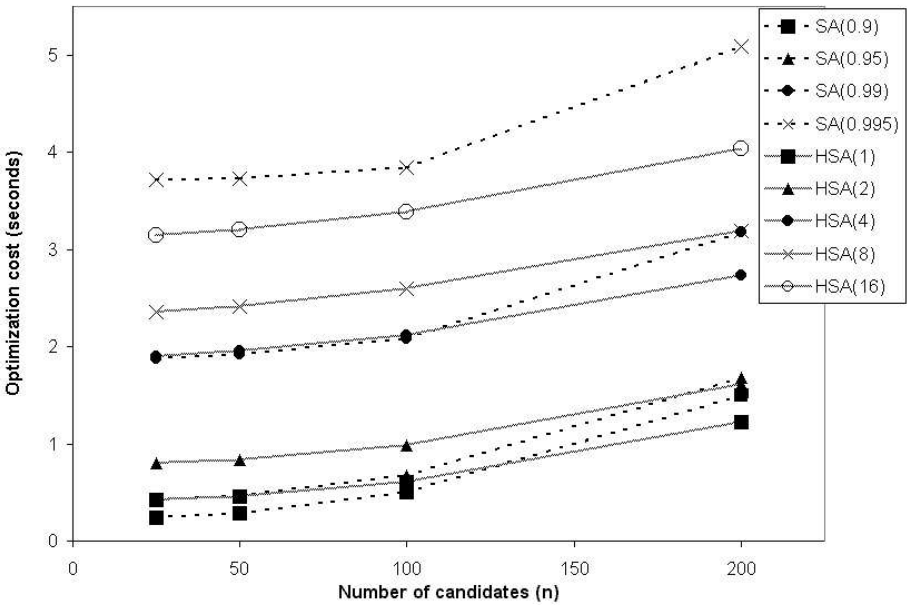


Figure 4.28: Optimization cost: larger-scale problems.

experiments.

The cost and optimality data in these graphs can be used by the synthesizer to perform algorithm selection for the video conferencing service. In other words, similar to our experiments, the synthesizer can use simulated requests to measure the cost and optimality

properties of the built-in algorithms. Such data can be summarized into a lookup table that serves the purpose of the functions $P()$ and $C()$ as described earlier. Now we use an example to illustrate how the synthesizer performs algorithm selection according to the process discussed in Section 4.3.2.

Suppose the provider's cost constraint is 1 seconds per session, and the optimality constraint is 1.15. The synthesizer obtains the functions $P()$ and $C()$ from the results in the graphs above. Suppose now the synthesizer needs to select an optimization algorithm to solve the physical mapping problem for a particular request. The synthesizer performs algorithm selection as follows.

- **Determining eligible algorithms**: The synthesizer analyzes the objective function constructed by the video conferencing recipe and discovers that there are two semi-dependent components. Therefore, the hybrid and HSA algorithms are eligible. In addition, the generic exhaustive search and simulated annealing algorithms can also be used.

- **Determining problem size**: The synthesizer queries the service discovery infrastructure and discovers that there are 15 candidates for each of the components in the abstract configuration.

- **Applying cost constraint**: Since the provider's cost constraint is 1 seconds per session, the synthesizer uses the function $C()$ (see Figure 4.26) and discovers that when n is 15, only 4 algorithms can satisfy the cost constraint: SA(0.9), SA(0.95), HSA(1), and HSA(2).

- **Applying optimality constraint**: Given the optimality constraint of 1.15, the synthesizer uses the function $P()$ (see Figure 4.25) and discovers that among the 4 algorithms above, only HSA(2) can satisfy the optimality constraint when n is 15.

- **Selecting an algorithm**: Since there is only one algorithm that satisfies both constraints, the synthesizer will use the HSA(2) algorithm to solve the physical mapping problem for the particular request.

To summarize, by providing an interface for providers to specify their desired cost/optimality trade-off, our approach allows providers to customize algorithm selection without knowing the details of the algorithms.

4.6 Related work

Recently, self-configuration is identified as one of the fundamental features of "autonomic computing systems" [82, 50]. Many previous studies have proposed different forms of self-configuring services. For example, the Xena service broker [15] in the Darwin project [14] allows a network application to submit an input graph. Such a graph is roughly equivalent to the abstract configuration composed by our synthesizer. Xena then selects the necessary

components in the graph for the application using some generic optimization criteria. In the Globus resource management architecture [26, 42], an application can specify its resource requirements using the Resource Specification Language (RSL). Application-specific brokers can translate the high-level abstract requirements into concrete resource specifications, for example, 10 nodes with 1GB memory. Such specifications are further "specialized" until the locations of the resources are completely specified, for example, 10 nodes at site A. The QoS resource manager [44] at each site then handles the resource reservation and run-time management issues. The ClassAd language in the Matchmaking framework [112] allows a user to specify the requirements of a computation tasks, and the framework provides mechanisms for mapping such requirements with the appropriate resources, which are also described using the ClassAd language. The language is extended by Liu et al. [87] and Raman et al. [113] to support mapping a single task to multiple different resources and to support resource co-allocation, respectively. In contrast to our work, these approaches focus on the task of physical mapping and does not emphasize the use of the service-specific knowledge.

Other previous efforts build self-configuring services using type-based service composition, i.e., components have well-defined input/output (requires/provides) interfaces such that a generic self-configuration module can automatically find a configuration that can satisfy a particular user request. For example, the Ninja architecture [57] provides a cluster computing environment and uses type-based service composition [90] to construct a "service path" that satisfies a particular user request, i.e., it is able to find a series of components that can convert the output type of a server to match the input type of the user. Iceberg [129] uses the Ninja platform to construct a framework for sophisticated communication services. The Panda project [115] also uses the service path model to find a series of "adaptors" that can remedy a particular set of network problems that occur between a sender and a receiver. Similar to the path model, the service morphing approach [107] supports dynamic configuration and adaptation for data streaming services. The information channel between two entities can be configured and adapted by dynamically generating and deploying code modules that process and transform the data to cope with variations in resource availability. The CANS infrastructure [48] supports dynamic composition and adaptation by customizing the data path between users and services using a series of drivers that transform the data stream. The Partitionable Services Framework [75] builds on CANS and supports a more flexible and general composition model. These approaches provide a generic framework that can be shared by different services and thus reduce the development cost. However, their effectiveness are limited by the fact that they cannot take advantage of the providers' service-specific knowledge for self-configuration.

The SWORD toolkit [109] allows service providers to quickly create composite Web services. A rule-based plan generator is used to find a feasible "plan" using type-based composition. The plan consists of the components necessary to satisfy the provider's needs. However, the toolkit does not handle the deployment of the components, and the provider will deploy the components according to the generated plan. In the Rainbow framework [19], a software system is modeled using Acme [53], an architecture descrip-

tion language. This allows the specification of the structure, properties, constraints, types, and styles of the system. Using such a formal description, Rainbow monitors the abstract model and modifies it to adapt to run-time environment changes. A compositional approach is proposed in [121] for constructing connectors between software components, i.e., a series of transformations are applied in order to connect incompatible components together. The approaches described above focus on higher-level abstract structures of the service configuration and do not address the physical mapping problem.

Rascal [49] is a resource manager for multi-agent systems in "smart spaces" where it dynamically allocates "resources" such as TV set, projector display, and other devices among different applications using utility functions. The task-driven computing approach allows a user to specify the desired task and maps the task to the appropriate software modules [130]. The Prism task manager [120, 108] in the Aura project [54] extends task-driven computing and supports task migration, run-time adaptation, and context awareness. These approaches focus on software module selection within a small locale and parameter-level configuration/adaptation instead of component selection in the wide-area network, and they emphasize user-specific concerns such as distraction and intention instead of network performance.

As discussed previously, specialized frameworks have been proposed to provide self-configuration capabilities to specific types of services. For example, the Da CaPo++ architecture [123] uses a set of protocol modules to generate a customized communication protocol that meets the requirements of a particular communication session. Tina [35] is an architecture for composing telecommunication services using components with CORBA [101] IDL interfaces. Another CORBA-based architecture, xbind [84], uses low-level components such as kernel services and peripherals to compose services. CitiTime [6] is an architecture that dynamically creates communication sessions by downloading and activating an appropriate service module according to the requirements of the caller and the callee. Gbaguidi et al. [55] propose a Java-based architecture that allows the creation of hybrid telecommunication services using a set of JavaBeans components. Other examples include using shortest path algorithms to select components of a given service path in peer-to-peer Grids [60], selecting components and resources given a set of multi-fidelity applications [108], and selecting resources for path-based resource-intensive applications [88]. As discussed earlier, although using such service-specific approaches can achieve highly effective self-configuration for the target service through the use of providers' service-specific knowledge, it requires high development cost since each provider needs to implement a complete self-configuration solution.

Finally, as mentioned earlier, researchers have proposed many optimization algorithms for solving many different forms of component/resource selection/mapping problems. For example, a dynamic algorithm for mapping components along a path [48], a shortest-path algorithm for selecting intermediate processing sites [21], a shortest-path algorithm for constructing a service path [60], and heuristic algorithms for the resource mapping and co-allocation problems [87, 113]. Each of these studies addresses a subset or a particular form of the mapping problems. In contrast, since in our approach a provider can specify a general

objective functions as the optimization criteria for the mapping problem, the synthesizer must be able to solve a broader range of mapping problems. Therefore, we focus on how the synthesizer can take a provider's service-specific trade-off between optimization cost and optimality into consideration in order to select the most suitable algorithm for each particular mapping problem.

4.7 Chapter summary

In this chapter, we described the design and implementation of the synthesizer, the core element in our recipe-based self-configuration architecture. We defined the recipe representation as a set of APIs that can be used by a service provider to write its service recipe. The recipe APIs allows a provider to express the service-specific knowledge for both abstract mapping, i.e., finding the necessary component types, and physical mapping, i.e., finding the actual components in the network. The synthesizer performs abstract mapping for a request by executing the recipe, and physical mapping is based on the objective function constructed during the recipe execution. We discussed how the complexity of the physical mapping problem depends on the objective function and what optimization algorithms can be used to solve different problems. We then described a process that the synthesizer can use to automatically select the best optimization algorithm for a particular physical mapping problem according to the cost and optimality trade-off specified by the service provider.

In our evaluation, we first illustrate the expressiveness of our recipe representation by designing service recipes for three different services. We also use two different recipe designs for the same service to show how providers can write different recipes according to their own goals. We then evaluate the effectiveness of the recipe-based self-configuration approach using the video conferencing service scenario. The results show that, using generic simulated annealing algorithms, the synthesizer can use the service-specific knowledge in a recipe to efficiently find suitable service configurations. Moreover, most of the overhead is spent in our unoptimized objective evaluation mechanisms. The overhead caused by the abstraction of service-specific knowledge is almost negligible. We also compared the development cost of our recipe-based approach with that of the service-specific approach and found that our approach requires dramatically lower implementation efforts yet still allows the use of providers' service-specific knowledge. We then evaluated the effectiveness of our hybrid heuristic and showed that the hybrid heuristic combined with simulated annealing is more effective than generic simulated annealing both in terms of optimality and cost. Finally, we used the video conferencing service scenario to illustrate how the synthesizer can use cost and optimality data collected offline to select the best optimization algorithm according to the provider's cost and optimality constraints.

Chapter 5

Run-time Adaptation

As discussed in Chapter 2, self-configuration is necessary at both invocation time and run time. So far, we have described the recipe-based self-configuration architecture that performs global configuration at invocation time. A service provider can specify the service-specific global configuration knowledge in a recipe, which is then used at invocation time by the synthesizer to compose the optimal global configuration for each user request. In this chapter, we focus on the run-time adaptation aspect of self-configuration.

One naïve approach to add such run-time adaptation to our recipe-based self-configuration architecture is to have the synthesizer periodically re-perform global configuration in order to maintain the optimality of the service configuration at run time. However, this is not a desirable approach since (1) global configuration is expensive in terms of computation resource requirements and thus cannot be performed frequently, (2) as a result, such an approach is not "agile" [100], i.e., it cannot react to frequent changes in user requirements and environment characteristics in a timely fashion, and (3) switching to a completely new global configuration at run time may cause disruption/interruption to the user session, potentially resulting in poor service quality. Therefore, we focus on local adaptations that make incremental changes to the configuration at run time.

Like the global configuration knowledge, the knowledge of how to perform adaptations is also highly service-specific. We have identified three aspects of service-specific adaptation knowledge: *adaptation strategies*, *customization*, and *coordination*. Previously, research efforts in adaptation focus on supporting different forms of adaptation strategies that specify when and how to perform adaptations, for example, using a utility function to guide the adaptation of system parameters [128], using explicit "event→action(s)"-style strategies [19], and so on. However, customization and coordination have been largely ignored.

Customization refers to the fact that there may be multiple strategies for a particular adaptation, and the best strategy may depend on the actual system configuration and environment. For example, for a self-configuring video conferencing service, some user sessions may use IP multicast configurations while others have to use end system multicast. Therefore, the adaptation strategies need to be customized according to which type of configuration is used since the two types of configuration have very different requirements.

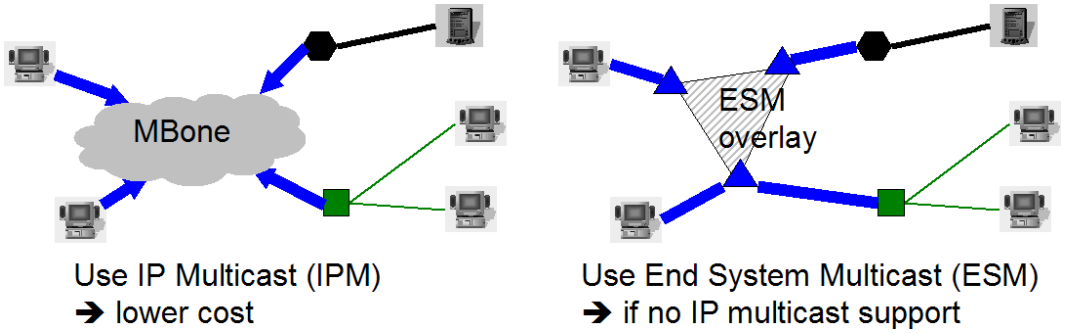


Figure 5.1: Two service configurations for a conferencing session: IPM and ESM.

On the other hand, coordination of adaptation strategies is important because, for example, two strategies may attempt to make conflicting changes to the current configuration, or the provider may accidentally design two strategies that "work at cross purpose". Therefore, coordination mechanisms are needed to detect and resolve such problem cases.

In this chapter, we describe how we extend our recipe-based self-configuration architecture to support run-time local adaptations. Section 5.1 identifies the three aspects of service-specific adaptation knowledge, and Sections 5.3, 5.4, and 5.5 describe how they are supported. The recipe APIs are extended so that a service provider can specify these three aspects of adaptation knowledge in a recipe. At invocation time, the synthesizer customizes the adaptation strategies according to the customization knowledge. At run time, the synthesizer invokes the adaptation strategies when appropriate. The results of different strategies are coordinated by the *adaptation coordinator* using the coordination knowledge in the recipe. We describe our prototype implementation in Section 5.6. In Section 5.7, we perform simulations using a massively multiplayer online gaming service scenario to illustrate the flexibility of the customization mechanisms and to demonstrate that our recipe-based approach allows service providers to easily design their service-specific coordination policies, and the coordination mechanisms work as expected and do not introduce significant overhead.

5.1 Problem statement

Let us first revisit the video conferencing scenario from Chapter 2 to illustrate the three aspects of a provider's service-specific adaptation knowledge. In Figure 4.4, five users want to hold a video conference. Two are using MBone conferencing applications vic/SDR (VIC), two use NetMeeting (NM), and one uses a receive-only handheld device (HH). The self-configuring video conferencing service can compose distributed gateway and proxies together to support such a conferencing session. As shown in Figure 5.1, there are two possible configurations for this conferencing session: (1) if IP multicast (IPM) is supported by the components and the underlying networks, then it will be used for communication among the VIC users, the video conferencing gateway (VGW), and the handheld proxy

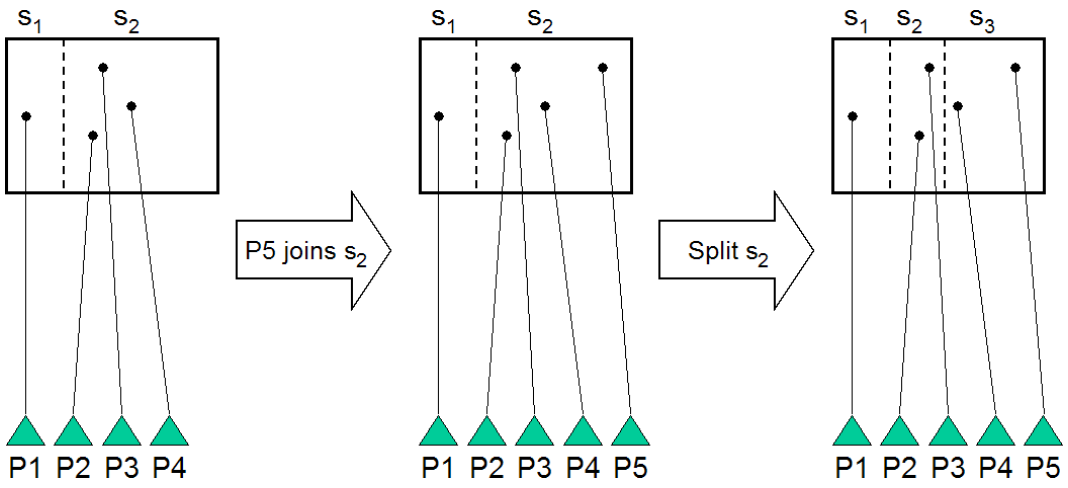


Figure 5.2: A massively multiplayer online gaming service.

(HHP); (2) otherwise, End System Multicast (ESM) [23] will be used, i.e., three ESM proxies are needed in this scenario.

At run time, the service needs to adapt to changes in user requirements or environment, e.g., the VGW becomes overloaded. The service provider needs to design adaptation strategies to handle such changes. Furthermore, the strategies may need to be customized according to the actual configuration. For example, to handle "VGW overload", the provider may have different strategies S1 and S2 for the IPM and ESM configurations.

- S1: (VGW overloaded) → (degrade VGW output video quality)
- S2: (VGW overloaded) → (replace VGW with a high-capacity VGW)

If the current configuration uses IPM, strategy S1 is used to maintain the low cost of the configuration. On the other hand, S2 is used for ESM configurations since the users are already paying a higher cost.

The various strategies also need to be coordinated. Suppose the provider designs the following strategy.

- S3: (VGW congestion) → (replace VGW with a high-bandwidth VGW)

Note that if S2 and S3 are invoked at the same time, they may want to replace the current VGW with different candidates. Therefore, only one of them should be allowed to execute its actions.

As another example, consider the self-configuring online gaming service for massively multiplayer online games, shown in Figure 5.2. The service uses a set of distributed game servers to maintain a partitioned virtual game world (each server maintains a partition). At run time, the service needs to dynamically adapt to the current load. Suppose the provider designs the following adaptation strategies.

- S4: (server s overloaded, i.e., load $> T_s$) $\rightarrow$ (adds a new server s'; migrate half of the players on s to s')
- S5: (s and s' under-utilized, i.e., load $< T_m$) $\rightarrow$ (migrate the players on s' to s; remove s')
- S6: (new user joins in the partition of s) $\rightarrow$ (connect to s)

Note that T_s and T_m are the "split" and "merge" thresholds, respectively. In addition, the provider may want to customize S4 and S5 according to the number of servers in the current configuration because, for example, (1) although adding more servers reduce the load, it increases the inter-server communication overhead, and (2) the provider has contracted a primary server pool for normal use and an emergency pool, which is more expensive, for extraordinary load. Therefore, when the number of servers in the current configuration is low (e.g., less than 80% of the primary pool), the provider may set T_s/T_m to 0.7/0.3; when the number is high, T_s/T_m may be set to 0.9/0.4. Finally, the strategies also need to be coordinated. For example, when a new user joins in A's partition (i.e., S6 is invoked), if server s is being split (S4), then the join should be blocked until S4 finishes so that the user can be connected to the correct server.

The above examples illustrate the three aspects of service-specific adaptation knowledge: adaptation strategies, customization rules, and coordination policies. The goal of this chapter is to extend our recipe representation so that providers can specify such knowledge in a recipe and to extend the synthesizer architecture to provide support for the adaptation operations. As mentioned earlier, we focus on supporting *local adaptation*, i.e., support for adaptation strategies that only involve local actions and the customization and coordination of such strategies. Specifically, we define the scope of local adaptation as follows.

- *Single component*: A local adaptation strategy adapts the configuration by executing actions that only change a single component. Such actions include, for example, changing the run-time parameters of a component and adding/removing/replacing a component. For example, the strategy S1 above changes a parameter of the VGW component, and S4 adds a new game server to the configuration.
- *No effect on adaptation strategies*: The actions of a local adaptation strategy does not affect the applicability of existing strategies and does not require new strategies to be added. For instance, in the gaming service example above, S4 can be defined to cover all components of the game server type, instead of a particular game server. As a result, a new server added by S4 does not require a new strategy (i.e., another copy of S4) to be added. However, if S4 adds a new component type, e.g., text-to-voice translator, that requires an adaptation strategy not present in the current configuration, then S4 would need to add a new strategy. Therefore, such an adaptation action is outside the scope of local adaptation.
- *No indirect effect*: The execution of a local adaptation strategy only has localized effects on the target component, i.e., we do not consider the propagation of effects

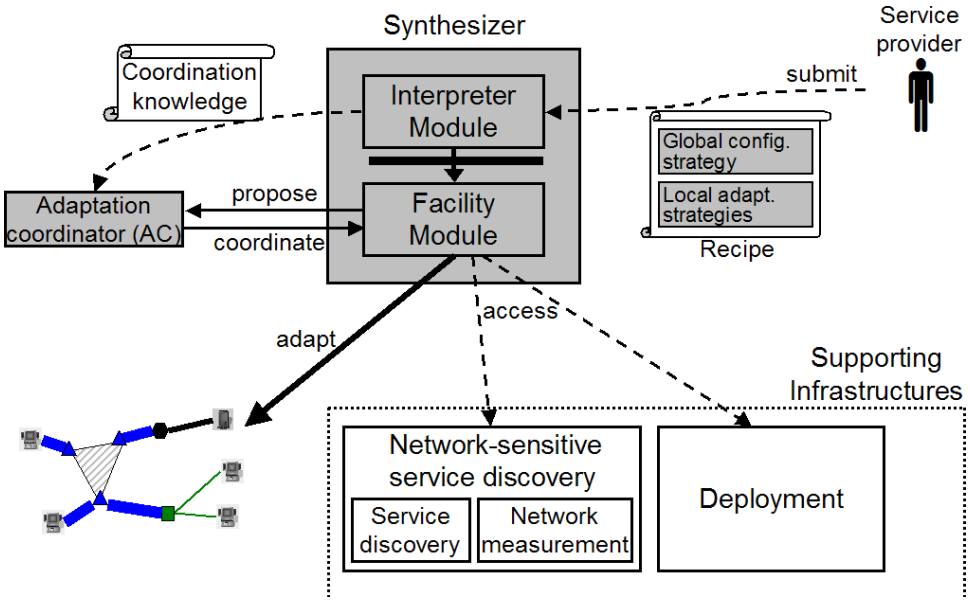


Figure 5.3: Extended synthesizer architecture for run-time local adaptation support.

in the global configuration. The main reason for this limitation is to reduce the complexity of the coordination problem. For instance, in the video conferencing example above, suppose two adaptation strategies are invoked at the same time. One wants to replace an ESMP that no longer has sufficient bandwidth to the current VGW, and the other wants to replace the current VGW because it is overloaded. If we only consider direct effects of adaptation strategies, both actions can be executed without conflict. However, if we consider indirect effects, replacing the ESMP may be unnecessary since the ESMP may have sufficient bandwidth to the new VGW.

Next, we present the extended synthesizer architecture that supports run-time local adaptation.

5.2 Architecture overview

Figure 5.3 shows the extended synthesizer architecture that supports run-time local adaptation. To build a self-configuring service, a service provider creates a *service recipe* that contains the service-specific knowledge. More specifically, a recipe consists of a *global configuration strategy* and a set of *local adaptation strategies*.

The *customization* and *coordination* knowledge for adaptation is part of the global configuration strategy. At invocation time, the synthesizer executes the global configuration strategy to compose a service configuration. Based on the resulting configuration, the synthesizer uses the customization knowledge to customize the local adaptation strategies, e.g., it is possible that only a subset of the adaptation strategies in the recipe is applicable for

the current configuration. Finally, according to the coordination knowledge, the synthesizer generates a set of *coordination policies* that can be used to detect and resolve conflicts among the customized adaptation strategies. The coordination policies are given to the *adaptation coordinator* (AC).

At run time, the synthesizer monitors the service configuration and invokes the adaptation strategies when appropriate. When an adaptation strategy is executed, it generates a *proposal* that is sent to the AC. The proposal specifies how the strategy wants to change the current configuration. If a proposal does not conflict with other proposals, the AC informs the synthesizer to change the configuration accordingly. Otherwise, the AC rejects the proposal, and the proposed changes are not performed.

In addition to the local adaptations, the synthesizer can also periodically re-executes the global configuration strategy if a more global form of adaptation is desired or required. Because of changes in the environment and user requirements, such a re-configuration may result in a different configuration, differently customized adaptation strategies (or a different set of strategies), and/or a different set of coordination policies.

Next, we describe how the three aspects of service-specific adaptation knowledge are expressed in recipes and how the adaptations are performed accordingly.

5.3 Adaptation strategies

Previous studies have proposed many different approaches for applying adaptation strategies to a system. Most of these approaches are based on "internalized" adaptation strategies, i.e., the adaptation logic and mechanisms are hard-wired into the system itself. Since the developer of the system needs to implement a complete solution for run-time adaptation, this internalized approach is equivalent to the service-specific approach for building self-configuring services discussed in previous chapters where a service provider needs to implement a complete self-configuration solution. As we have argued earlier, although such an approach will allow the service provider to have complete control over how adaptations should be performed, it will require high development cost.

The "externalized" approach adopted by some previous studies such as [52, 128] addresses this problem by separating the adaptation strategies and mechanisms from the actual system. This enables the development of a general adaptation framework that can be shared/reused by different systems, and run-time adaptation capabilities can be added to a system by designing externalized strategies without modifying the system components. As a result, the required development cost for supporting run-time adaptation is much lower than what is required in the internalized approach.

We can see that this externalized approach is equivalent to our recipe-based self-configuration approach where the provider's service-specific knowledge is externalized into a recipe. Since we have already adopted such an externalized approach for global configuration in our recipe-based self-configuration architecture, it is natural to extend our architecture to support externalized adaptation strategies for run-time adaptation.

A design decision that we need to make in order to support externalized adaptation

strategies is how to specify such strategies. As categorized in [128], an adaptation strategy can be specified as a high-level "utility" function or an explicit "event-action" rule. The event-action approach lets a service provider specify a rule that dictates what adaptation "actions" should be taken when a particular "event" occurs. For example, when the event "component X becomes overloaded" occurs, the appropriate action may be to "replace X with a higher capacity one". This approach is used in previous studies where the target system involves different, heterogeneous components, for example, [52, 37].

On the other hand, the utility approach allows a service provider to specify a utility function that indicates what configurations are more desirable. At run time, the adaptation mechanisms will automatically modify the service configuration as necessary towards higher utility. This approach is used in many previous studies that use dynamic resource allocation as the primary means for run-time adaptation, for example, [85, 108, 128]. In this context, the utility approach is feasible since the service configuration is represented by a set of resource parameters and can be directly mapped to a utility value. Adaptation is then performed by selecting the set of parameters that maximizes the utility. However, our definition of adaptation is broader and includes component-level adaptations such as replacing existing components in the configuration and adding/removing components. Even if it is possible to map every possible configuration to a simple utility value (e.g., using a 0-1 variable to represent whether each component is used in the configuration), the search space for the resulting optimization problem (which may be a nonlinear 0-1 programming problem [8]) is likely too large for the utility approach to be feasible. Therefore, based on the above factors, we adopt the externalized, event-action approach for strategy specification. Next, we introduce the format of adaptation strategies and how we extend our architecture to support such strategies.

5.3.1 Strategy format

Our support for adaptation strategies is built on the externalized approach presented in Rainbow [52]. (We will discuss the differences between Rainbow and our work in the related work section.) An adaptation strategy consists of the following three parts.

- *Constraint*: An adaptation strategy is invoked when its constraint is "violated". A constraint is specified as a condition on certain properties of the service configuration, e.g., "load(X) $< C$" where X is a server in the configuration, and C is a threshold on server load. The properties can be performance metrics, such as bandwidth and latency, or component properties. At run time, the synthesizer will monitor the constraints associated with the adaptation strategies and invoke a strategy when its condition becomes false.

  An entity referred to in a constraint can be either a particular component (as X above) or a type of component. For example, if X in the above constraint refers to the type "MPEG2Server", then the constraint is violated when any component of type MPEG2Server becomes overloaded. Furthermore, when a constraint is violated, the

"violator" is passed on to the corresponding adaptation strategy (similar to Rainbow [52]) such that the strategy can operate on the appropriate component.

- *Problem determination*: A constraint violation may be caused by multiple different *triggering problems*. When a strategy is invoked, it may need to, for example, query more specific properties of the configuration in order to to determine the actual cause of the constraint violation. For example, in the video conferencing example, a strategy triggered by "low HH video quality" needs to determine whether the actual problem is HHP failure, low quality codec used by the HHP, or congestion at the HHP.

- *Tactic*: A tactic consists of a set of *actions* and addresses a particular triggering problem. Actions range from changing a run-time parameter of a particular component to changing the service configuration by inserting/removing components. In the example above, "HHP failure" can be addressed by a tactic that replaces the failed HHP with a new one, "low quality codec" can be addressed by "increasing the codec quality", and so on.

To summarize, the synthesizer monitors the constraints of the adaptation strategies and invokes a strategy in the event of constraint violation. The strategy then determines the triggering problem and executes the appropriate tactic to perform adaptation.

5.3.2 Strategy specification

The recipe APIs described in Chapter 4 can be used to specify how to construct an abstract configuration (e.g., types of components) and an objective function for mapping the abstract configuration to a physical configuration. To allow providers to specify adaptation strategies in recipes using the above format, we extend the recipe APIs as follows.

- *Constraint*: The previous objective function API allows a provider to specify a function of properties of the configuration, for example, a function of different performance metrics. To specify a constraint, we need to add relation operators (e.g., "==", ">=", etc.) and boolean operators (e.g., "and") to the objective function API.

- *Problem determination*: Since a recipe is written in a general purpose programming language, for example, Java in our prototype implementation, a service provider should have sufficient flexibility in implementing the problem determination logic. In addition, we assume that problem determination can be done by querying more specific properties of the configuration. Such querying operations are supported by the original recipe APIs.

- *Tactic*: A tactic specifies a set of *actions* that change the current configuration. We assume adaptations are performed within a "local" scope relative to the "global configuration" operation performed by the synthesizer. Therefore, the API needs to support

local actions such as replacing a current component with a new physical candidate or changing a parameter of a current component.

Similar to how physical mapping is done for the global configuration operation as described in Chapter 4, when a tactic specifies an action that requires a new physical component, it is not necessary to specify the exact physical node. Instead, an objective function can be specified as the criteria for component selection. This objective function can be specified using the original objective function API.

- *Strategy*: Finally, since adaptation strategies are tied to the current configuration, they need to be "instantiated" at the global configuration time. In our approach, global configuration is performed by the synthesizer using the global configuration strategy. Therefore, in the global configuration strategy, the provider needs to instantiate the appropriate local adaptation strategies. Specifically, for each local adaptation strategy, the global configuration strategy needs to construct the corresponding constraint, instantiate the local adaptation strategy, and associate the constraint with the strategy.

Figure 5.4 summarizes the major extensions to the original recipe APIs.

5.3.3 Example

Now we use the video conferencing service as an example to illustrate how a provider's local adaptation strategies can be specified in the recipe using the extended APIs. Suppose the provider implements the following tactics (for simplicity, the actual code is not shown).

- `tacticNewNM`: This tactic connects a new NM user to the VGW.
- `tacticNewVIC`: Connect a new VIC user to the closest ESMP.
- `tacticNewHH`: Connect a new HH user to the HHP.
- `tacticVGWFail`: Replace a failed VGW with a high-capacity one.
- `tacticVGWOverload`: Replace an overloaded VGW with a high-capacity one.
- `tacticVGWCongest`: Replace a congested VGW with a high-bandwidth one.
- `tacticVGWLowQual`: Increase the VGW's codec quality
- `tacticESMPFail`: Replace a failed ESMP with a high-bandwidth one.
- `tacticESMPCongest`: Replace a congested ESMP with a high-bandwidth one.

The provider also implements the following strategies that use the tactics above.

- `stratNewUser`: This strategy is triggered when a new user wants to join the session. It determines the problem to address and invokes the appropriate tactic as follows.

Data structure	
`RelationOp`	Represent relation operators such as "==", ">=", etc.
`BooleanOp`	Represent boolean operators such as "AND", "OR", etc.
`Condition`	Represent a combination of "`Term RelationOp Term`".
`Constraint`	Represent a combination of "`Condition BooleanOp Condition`".
`Tactic`	Represent the base class for a tactic; providers implement tactics by creating specializations.
`Strategy`	Represent the base class for a strategy; providers implement strategies by creating specializations.

Function	
`replaceComponent(c)`	Represent an adaptation action that replaces an existing component `c` in the current configuration.
`changeParameter(pn,pv)`	Represent an adaptation action that changes the value of the parameter `pn` of a component to `pv`.
`connect(c1,c2)`	Represent an adaptation action that connects components `c1` and `c2`.
`setTacticObjective(obj)`	Set the component selection objective for a tactic to `obj`, which will be used for, e.g., the `replaceComponent` actions.
`addStrategy(S,C)`	Add the adaptation strategy `S` for run-time adaptation and associate it with constraint `C`.

Figure 5.4: Extensions to the recipe APIs for local adaptation strategies.

  - If the new user is NM → invoke `tacticNewNM`
  - If the new user is VIC → invoke `tacticNewVIC`
  - If the new user is HH→ invoke `tacticNewHH`

- `stratNMQual`: Triggered when the video quality of NM is below a threshold T_n.
  - If VGW has failed → invoke `tacticVGWFail`
  - If VGW is overloaded → invoke `tacticVGWOverload`
  - If VGW is congested → invoke `tacticVGWCongest`
  - If VGW uses low quality codec → invoke `tacticVGWLowQual`
- `stratVICQual`: Triggered when the video quality of VIC is below a threshold T_v.
  - If ESMP has failed → invoke `tacticESMPFail`
  - If ESMP is congested → invoke `tacticESMPCongest`

In the global configuration strategy, the provider will instantiate the following constraints.

- C1: `(config.numUnconnectedUsers == 0)`
- C2: `(NM.videoQuality >= Tn)`
- C3: `(VIC.videoQuality >= Tv)`

Finally, the global configuration strategy instantiates the local adaptation strategies and adds them to the current configuration as follows.

```
stratNewUser S1 = new stratNewUser(config);
config.addStrategy(C1, S1)
stratNMQual S2 = new stratNMQual(config);
config.addStrategy(C2, S2)
stratVICQual S3 = new stratVICQual(config);
config.addStrategy(C3, S3)
```

After the execution of the global configuration strategy is completed, the composed configuration contains the necessary local adaptation strategies and their associated constraints.

5.4 Customization

As seen from the examples in Section 5.1, a provider may want to customize the adaptation strategies according to the current configuration and/or environment. Our architecture supports three types of customization: *strategy selection*, *dynamic constraint*, and *dynamic tactic binding*. We now describe these customization mechanisms and discuss our design.

5.4.1 Strategy selection and dynamic constraint

One simple form of customization is strategy selection, i.e., given the current configuration and environment, only the relevant set of adaptation strategies are instantiated. For example, in the video conferencing example above, the adaptation strategy S3 is only applicable if the current configuration actually uses ESM since S3 handles problems related to the ESM proxy components. Therefore, S3 should only be instantiated if ESM is used in the current configuration.

In addition to strategy selection, a provider may want to customize the constraint associated with a strategy according to the current configuration/environment. For example, consider again the adaptation strategies for the video conferencing service. A provider may want to customize the threshold `Th` according to the quality requirement specified by the user request. If high quality is required, the threshold should be set lower to correct quality problems more aggressively. On the other hand, if the user does not require high quality (perhaps to reduce the cost), then the thresholds can be set higher.

Since a constraint is constructed and associated to the corresponding strategy by the synthesizer during global configuration as described in the previous section, a provider can easily customize the constraints dynamically in the global configuration strategy. Extending the video conferencing service example in the previous section, the global configuration strategy can customize the constraint C2 as follows.

```
if (request.qualReq == HIGH) {
    Th = 0.5
} else {
    Th = 0.7
}
```

Then `C2` can be instantiated the same way as in the previous example. Finally, the global configuration strategy instantiates the appropriate adaptation strategies as follows.

```
stratNewUser S1 = new stratNewUser(config);
config.addStrategy(C1, S1)
stratNMQual S2 = new stratNMQual(config);
config.addStrategy(C2, S2)
if (config.isESM == true) {
    stratVICQual S3 = new stratVICQual(config);
    config.addStrategy(C3, S3)
}
```

Therefore, in the above example, both the selection of applicable strategies and the constraint `C2` are customized according to the actual user requirements and/or the current configuration.

5.4.2 Dynamic tactic binding

Another form of customization is to change the actions performed by the strategy according to the actual configuration and environment. Since the actions for a particular course of adaptation are grouped into a tactic, tactics become a natural unit of customization. In other words, a provider can design multiple different tactics that can be used to address a particular triggering problem such that each of these tactics is suitable under different configurations/environments.

For example, consider the video conferencing service in Section 5.1. Each conferencing session can have two possible configurations, ESM and IPM. When the current configuration is IPM, the provider's goal is to maintain the low service cost. On the other hand, for an ESM configuration, the goal is to maintain the quality. Now consider the strategy `S2` for NM video quality from Section 5.1. The original tactics are suitable for the ESM configuration, but they may be too expensive for the IPM configuration. Therefore, the provider may design the following additional tactics.

- `tacticVGWFail_1`: Replace a failed VGW with a low-cost one.

- `tacticVGWOverload_1`: Reduce the VGW's codec quality.
- `tacticVGWCongest_1`: Reduce the VGW's bit rate.
- `tacticVGWLowQual_1`: Do nothing.

Now the provider may want to customize the strategy according to the current configuration as follows.

- ESM configuration:
  - If VGW has failed → invoke `tacticVGWFail`
  - If VGW is overloaded → invoke `tacticVGWOverload`
  - If VGW is congested → invoke `tacticVGWCongest`
  - If VGW uses low quality codec → invoke `tacticVGWLowQual`
- IPM configuration:
  - If VGW has failed → invoke `tacticVGWFail_1`
  - If VGW is overloaded → invoke `tacticVGWOverload_1`
  - If VGW is congested → invoke `tacticVGWCongest_1`
  - If VGW uses low quality codec → invoke `tacticVGWLowQual_1`

To customize a strategy using the different tactics, we can view the triggering problems as "symbolic names" for the adaptation tactics. In other words, in a strategy, after the triggering problem is determined, the strategy invokes a tactic by specifying the symbolic name. However, the connections between symbolic names and tactics are not established until the global configuration time, when the synthesizer *binds* the appropriate tactics to the symbolic names according to the actual configuration/environment. How to perform this dynamic tactic binding is specified in the global configuration strategy.

In order to support dynamic tactic binding, we extend the recipe APIs by adding the following functions to handle the symbolic names and binding.

```
invokeTactic(SymName)
bindTactic(SymName, Tactic)
```

Now we use the above adaptation strategy as an example to show how dynamic tactic binding is used. The strategy `stratNMQual` is now implemented as follows.

- If VGW has failed → `invokeTactic("VGWFail")`
- If VGW is overloaded → `invokeTactic("VGWOverload")`
- If VGW is congested → `invokeTactic("VGWCongest")`

- If VGW uses low quality codec → `invokeTactic("VGWLowQual")`

The global configuration strategy will bind the tactics to the symbolic names dynamically and then instantiates the strategy as follows.

```
if (config.isESM == true) {
    bindTactic("VGWFail", tacticVGWFail)
    bindTactic("VGWOverload", tacticVGWOverload)
    bindTactic("VGWCongest", tacticVGWCongest)
    bindTactic("VGWLowQual", tacticVGWLowQual)
} else {
    bindTactic("VGWFail", tacticVGWFail_1)
    bindTactic("VGWOverload", tacticVGWOverload_1)
    bindTactic("VGWCongest", tacticVGWCongest_1)
    bindTactic("VGWLowQual", tacticVGWLowQual_1)
}
stratNMQual S2 = new stratNMQual(config);
config.addStrategy(C2, S2)
```

Using this approach, the local adaptation strategies can be customized dynamically to invoke different tactics according to the actual configuration and environment.

5.4.3 Discussion

One key feature of the customization mechanisms described above is that the customization is performed by the synthesizer at global configuration time. An alternative design is to have each individual adaptation strategy perform customization on its own. For example, in the example in Section 5.4.1, we can put the logic for adjusting the threshold `Th` inside the strategy `C2`. We can also design the strategy `C3` so that it checks the global configuration and do nothing if it is not an ESM configuration. Similarly, the effects of tactic binding in Section 5.4.2 can be realized by having the strategy `C2` check the global configuration and invoke the appropriate tactics accordingly, or each tactic can check the configuration and perform the appropriate actions accordingly. Although this alternative approach apparently works fine in the above simple scenario, there are two potential issues.

- *Complexity*: Having each strategy perform customization on its own may increase the complexity of the adaptation strategy design. For example, in order to put the dynamic constraint mechanism inside individual strategies, a provider also needs to put constraint monitoring mechanisms inside the strategies and have the synthesizer invoke each strategy to check if its constraint is violated. Moreover, if the constraints of multiple strategies are to be customized in the same way, e.g., adjusted according to the quality requirement (high/low), the provider needs to put essentially the same logic into each strategy. Similarly, if the tactic bindings of multiple strategies are to be customized in the same way, e.g., according to whether the global configuration is

ESM, the provider needs to implement the same logic in each strategy. Furthermore, every strategy is enabled by default, and the provider needs to design each strategy so that it checks whether it should actually perform adaptation when invoked.

- *Global information*: As seen in the examples above, in some cases the customization mechanisms may require information about the global configuration or environment, for example, the current quality requirement of the user, whether the current configuration is ESM or IPM, etc. In our current design where the synthesizer handles all adaptation strategies, each strategy can obtain such information and perform customization on its own. However, an alternative architecture is to leverage multiple adaptation frameworks that can handle different aspects of run-time adaptation, e.g., architecture-based self-repair [19], utility-based resource allocation [108], tactic-based remote execution [7], and so on. In such an architecture, each adaptation strategy specified by the provider can be passed on to and handled by a different adaptation framework. As a result, the global information required for customization may not be available to each individual strategy when it is invoked. In other words, in such a scenario customization requiring global information may be infeasible.

To address these issues, our customization mechanisms move the complexity into a single location, the global configuration strategy executed by the synthesizer. Of course, one advantage of the alternative approach above is that customization can be performed with a finer granularity, i.e., it can react to changes more rapidly instead of only performing customization at global configuration time. Since the strategies/tactics in our recipe-based approach are written using a general-purpose programming language, it does not preclude customization inside individual adaptation strategies. The interaction between customization in individual strategies and customization at global configuration time is an area for further study.

5.5 Coordination

As seen from earlier examples, different adaptation strategies may attempt to change the service configuration in conflicting ways. Therefore, adaptation coordination mechanisms are needed to detect and resolve such conflicts. There are two naïve approaches for adaptation coordination, *monolithic* and *independent*. The monolithic approach requires a service provider to design a monolithic adaptation strategy that incorporates all adaptation logic and makes all adaptation decisions. This monolithic strategy can be carefully designed to avoid any conflicts. On the other hand, in the independent approach, different parts of the system adapt themselves independently without interfering with each other. One example is a system in which each individual component adapts itself independently. Another example is to partition the system into independent parts, each of which then executes its own adaptation strategy.

One problem with the monolithic approach is that it increases the complexity of the design of the adaptation strategy since at design time the provider needs to avoid possible

conflicts among all adaptation cases. Furthermore, it requires coordination mechanisms to be embedded into the strategy itself, which further increases the development cost. On the other hand, the independent approach cannot be easily applied in systems where there may be interactions between adaptations on different parts of the system. Therefore, to support adaptation coordination, our goal is to only require a service provider to specify how adaptation coordination should be performed without having to integrate all adaptation strategies into a single monolithic strategy.

In our architecture, local adaptation strategies are invoked by the synthesizer when their constraints are violated. When a strategy is invoked, it generates a proposal specifying what actions should be performed. The proposal is then sent to the adaptation coordinator (AC). If there are no simultaneous proposals and no other proposals being executed, the AC accepts the new proposal and informs the synthesizer to execute the proposed actions. If the AC detects a conflict between simultaneous proposals or between a new proposal and an ongoing one, it resolves the conflict by rejecting the appropriate proposals.

To coordinate adaptations, we have identified three issues: detecting conflicts between proposals, resolving conflicts between proposals, and identifying incompatibility between strategies. When the AC receives multiple proposals, it needs to determine whether there is a conflict between the proposals, and if there is a conflict, the AC needs to resolve it by accepting a subset of the proposals and reject the others. On the other hand, a provider may design two (or more) strategies that adapt the system towards opposite directions/goals. For example, one strategy adds a new server when an existing server is overloaded, and another strategy removes a server when existing servers are under-utilized. These two strategies can potentially cause a cycle of adding/removing a server to/from the service. Therefore, even though the AC may never see conflicting proposals coming from these two strategies, our architecture needs to be able to identify such cases where multiple strategies work at cross purpose.

As discussed earlier, since we focus on local adaptations, we only consider the "direct" effects of an adaptation. As a result, conflicts between adaptations can be clearly defined. We do not try to solve the general coordination problem where every adaptation can potentially affect the whole configuration and thus can potentially conflict with all other adaptations. Next, we discuss how we address the three coordination issues.

5.5.1 Conflict detection

Let us first define what a "conflict" is. One simple form of conflict is when two proposals want to make "conflicting changes" to the configuration. For example, if one proposal wants to replace server A with B, and another proposal wants to replace A with C, then obviously the two proposals cannot both be accepted. In other words, the actions of these two proposals attempt to make changes to the same "target" in different ways. The AC can automatically detect such conflicts by looking at the actions in the different proposals.

On the other hand, there are also cases where the actions of two proposals do not conflict, but the "intentions" of the proposals may be conflicting. For example, suppose one

strategy S1 connects a new VIC user to the closest ESMP in the current configuration, and another strategy S2 replaces a failed ESMP with a new one. Suppose there are three ESMPs (A, B, and C) in the current configuration. Now user U wants to join the video conference, and at the same time C fails; therefore, both strategies S1 and S2 are invoked. Among A, B, and C, B is closest to U, so S1 proposes to connect U to B. At the same time, S2 proposes to replace C with D. However, D is actually closer to U than B is. This example presents an interesting question: should the AC identify these two proposals as a conflict?

The answer to this question is in fact service-specific, i.e., the AC cannot automatically decide if there is a conflict or not. While one provider may want to delay the new user join until after C has been replaced with D in order to achieve a more optimal configuration, another provider may rather use the sub-optimal configuration so that the new user join will not be delayed. We can look at this from a different point of view: S1 is addressing the triggering problem of "new VIC user", and S2 is addressing the problem of "ESMP failure". Knowing that a new VIC user needs to be connected to an ESMP, the provider needs to decide that, if the two problems occur at the same time, whether they can be addressed simultaneously or should be addressed one after the other. In other words, the provider needs to decide whether there is a "problem-level conflict" when the two problems occur simultaneously.

Based on the above observations, we categorize conflicts into two types.

- *Action-level*: When two proposals P1 and P2 contain actions that change the same target in different ways, it is an action-level conflict. As mentioned above, the AC can automatically detect such conflicts by comparing the actions in the different proposals. If action A1 from P1 and action A2 from P2 have the same "target" (e.g., both want to replace the VGW), the AC will identify this as a conflict and will only accept one of the proposals.

- *Problem-level*: A problem-level conflict occurs between two proposals if they address triggering problems that conflict with each other. As discussed above, such conflicts are service-specific, and therefore, they need to be explicitly identified by the provider in its service recipe.

We extend the recipe API as follows to allow a provider to specify its service-specific problem-level conflicts. As described earlier, the triggering problems in adaptation strategies are represented as symbolic names for the appropriate tactics. Therefore, a provider can specify a conflict between two problems using their symbolic names. To support such specifications, the following function is added to the recipe APIs.

```
addProblemConflict(PName1, PName2)
```

Now we use the same video conferencing service example from previous sections to illustrate how problem-level conflicts are specified. Suppose the symbolic names for the triggering problems in the example are: "NewNM", "NewVIC", "VGWFail", "VGWOverload", "VGWCongest", "VGWLowQual", "ESMPFail", and "ESMPCongest". The provider can specify the problem-level conflicts among them by adding the following to the global configuration strategy.

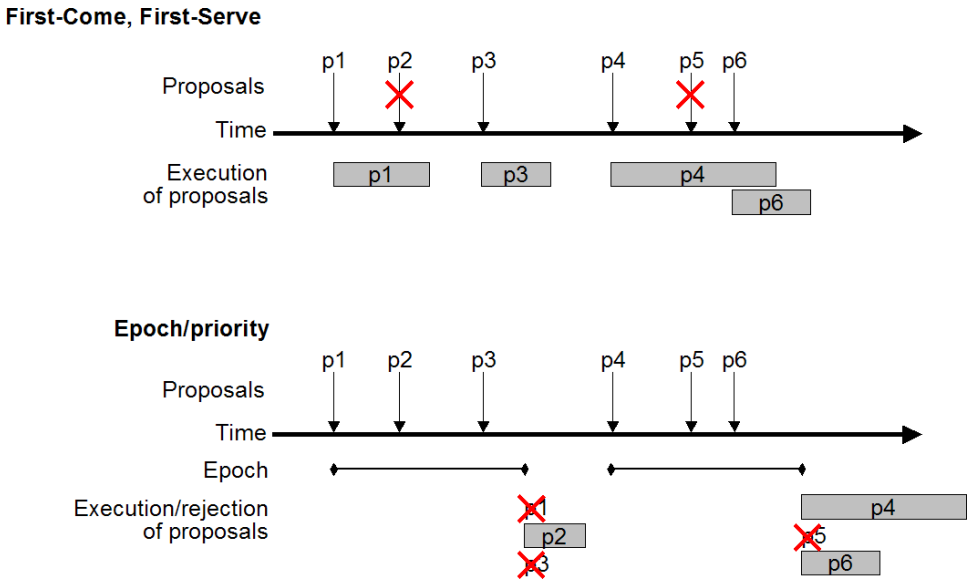


Figure 5.5: Two conflict resolution approaches.

```
config.addProblemConflict("NewNM", "VGWFail")
if (config.isESM == true) {
    config.addProblemConflict("NewNM", "VGWOverload")
    config.addProblemConflict("NewNM", "VGWCongest")
    config.addProblemConflict("NewVIC", "ESMPFail")
    config.addProblemConflict("NewVIC", "ESMPCongest")
}
```

Note that there are more conflicts when the current configuration is using ESM. This is because of the additional adaptation strategy S3 for the ESM configuration and the customization of the other strategies as described earlier.

When the synthesizer executes the global configuration strategy, the appropriate problem-level conflicts are added to the current configuration. After the execution is completed, the synthesizer constructs a coordination policy that represents the problem-level conflicts for the current configuration (e.g., as a "conflict matrix"), and the policy is sent to the AC. At run time, when an adaptation strategy proposes adaptation actions, the proposal includes the name of the problem that it is addressing. Therefore, the AC can look up the coordination policy to detect problem-level conflicts between different proposals.

5.5.2 Conflict resolution

We identify two different approaches for resolving conflicts between proposals, as shown in Figure 5.5. There are six proposals, and conflicts exist between p1 and p2, between p2 and p3, and between p4 and p5. In addition, p2 has a higher priority than p1 (p2 $>$ p1), p2 $>$ p3, and p4 $>$ p5. We now briefly describe the two different conflict resolution approaches.

- *First-Come, First-Serve (FCFS)*: This approach accepts/rejects proposals as they are received. If a proposal is received when no other proposals are being executed, then it is accepted. On the other hand, if a proposal is received when another proposal is being executed, the AC performs conflict detection between the two proposals. If a conflict is detected, the newly received proposal is rejected.

  In the figure, p2 is rejected because p1 is proposed earlier and is being executed, and similarly, p5 is rejected since p4 is in progress. On the other hand, p6 is allowed to start since there is no conflict between p4 and p6.

- *Epoch/priority*: This approach divides time into discrete periods, i.e., epochs. At the end of an epoch, the AC performs conflict detection among all proposals received within the epoch. If two proposals conflict with each other, for example, the one with the higher priority is accepted and the other rejected. Priorities are assigned to the adaptation tactics by the provider according to the service-specific coordination knowledge.

  In the figure, p1, p2, and p3 falls in the first epoch, and only p2 is allowed to be executed because of the conflicts and priority among them. Similarly, in the second epoch, p5 is rejected.

We can see that the FCFS approach supports more limited conflict resolution while the epoch/priority approach is more flexible. However, the flexibility of the epoch/priority approach is gained by sacrificing the "agility" of dynamic adaptation: all proposals within an epoch have to wait until the end of the epoch. Furthermore, in the literature, the epoch/priority approach is used in, for example, the active database domain to coordinate rules that specify actions to be executed upon the occurrence of particular events [76, 22]. In such a context, one can assume that, in our terminology, all related events occur simultaneously and trigger all related "proposals" instantaneously, the proposals are all received simultaneously, and all actions can be finished instantaneously. Based on these assumptions, the epoch/priority approach can be used effectively without delaying the actions. However, in the context of coordinating adaptations for self-configuring services, such assumptions are not realistic for the following reasons.

- Events (e.g., component failure, bandwidth fluctuation, user preference change) may occur independently at any time.

- When an adaptation strategy is triggered by an event, it needs to determine the triggering problem, which may require heavy-weight operations such as network measurement. As a result, there will be a delay between the occurrence of the event and the proposal reaching the AC.

- Executing an adaptation action, e.g., starting a server, in this context may require a relatively long period of time when compared to, e.g., the time it takes to update a database.

Therefore, to use the epoch-based approach for coordinating the adaptations in our context, complex heuristics may be needed to handle the many epoch-related issues resulting from realistic constraints. For example, we need heuristics to determine the optimal size for the coordination epochs, decide when a new epoch should be started, and so on. Furthermore, since we focus on performance-related adaptations, it may not be desirable to delay adaptation actions for coordination. Therefore, the FCFS approach is more appropriate for our purposes. Although conflict resolution in the FCFS approach may result in sub-optimal solution (e.g., a more important proposal is rejected because it is received later), it has the advantage that coordination decisions are made quickly without delaying adaptation actions.

5.5.3 Identifying incompatibility

To prevent adaptation strategies from working at cross purpose, we need to identify adaptation strategies that are incompatible with each other. This is in general a very difficult problem because a provider may intentionally design two adaptation strategies that have opposite goals. For example, one strategy adds a new server when an existing one is overloaded, and another strategy removes a server when existing servers are under-utilized. These two opposite strategies are intended to maintain the system in a particular operational region. However, if the triggering conditions for these strategies are not defined carefully, they can potentially cause a "cycle" of adding/removing a server to/from the service.

If we want to automatically determine whether such cycles exist for any given pair of strategies, we need to be able to automatically determine the exact effects of the strategies (e.g., how much load is reduced by adding a server) and analyze the triggering conditions to determine if one strategy's effects will trigger another strategy. Such analysis is difficult since it likely requires domain knowledge, and the exact run-time effects of adaptation may be difficult to determine.

Instead of trying to solve the general problem, we observe that it is usually sufficient if we can identify strategies that have opposite goals (and thus may cause undesirable cycles) and then warn the provider of such potential problems. The provider can then verify that the opposite goals are intentional and make sure that the combination of triggering conditions and effects should not cause cycles.

To automatically identify incompatible adaptation strategies, we need to capture the cause and effect of each strategy. We observe that a strategy is triggered by a "directional change" (i.e., increase or decrease) in one property of the configuration (or multiple properties) such as a particular performance metric, and the effect of a strategy can be summarized as a directional change in one property or multiple properties. If the effect of one strategy changes a particular property along the direction that may cause another strategy to be invoked and vice versa, then a potential cycle exists.

Using the gaming service in Section 5.1 as an example, the "split" strategy may be triggered when the "load" property increases ("load+"), and its effect is a decrease in load ("load-"). On the other hand, the "merge" strategy may be triggered by "load-", and it

causes “load+”. Given this information, the potential cycle between the strategies can be automatically detected and reported to the provider.

Note that so far we have discussed incompatibility at the strategy level. However, since the unit of coordination in our architecture is tactic, we need to analyze incompatibility at the tactic-level since different tactics in the same strategy may have different causes and effects.

To support such automatic analysis, the causes and effects of tactics can be “annotated” by the provider when designing the tactics. When the synthesizer instantiates and customizes the appropriate adaptation strategies and tactics during global configuration, it can use these cause/effect annotations to detect potential cycles between the tactics.

The recipe APIs are extended to support the cause/effect annotation as follows. The annotations need to identify the properties that are changed and the directions of the changes. For now we assume that the properties of interest are performance metrics, which can be specified using the objective function API. Therefore, the following functions can be used to specify the metrics whose increase and decrease may invoke a particular tactic and the metrics that may be increased and decreased by the tactic.

```
addTacticCauses(Tactic t, List increasedMetrics,
                List decreasedMetrics)
addTacticEffects(Tactic t, List increasedMetrics,
                 List decreasedMetrics)
```

In the gaming service example above, after the tactics `splitT` and `mergeT` are instantiated in the global configuration strategy, they can be annotated as follows.

```
// s1 { load }, s2 { }, s3 { }, s4 { load }
addTacticCauses(splitT, s1, s2);
addTacticEffects(splitT, s3, s4);

// m1 { }, m2 { load }, m3 { load }, m4 { }
addTacticCauses(mergeT, m1, m2);
addTacticEffects(mergeT, m3, m4);
```

After all tactics are instantiated and bound to the adaptation strategies, the synthesizer can analyze the annotations and discover the potential cycle between `splitT` and `mergeT`.

5.6 Implementation

As mentioned earlier, our prototype implementation is based on the recipe-based self-configuration architecture described in the previous chapters. The additional knowledge specifications necessary for local adaptation strategies, customization, and coordination are supported by extending the original recipe APIs, and the table in Figure 5.6 summarizes these new extensions. Correspondingly, the synthesizer is extended to support these

Data structure	
`RelationOp`	Represent relation operators such as "==", ">=", etc.
`BooleanOp`	Represent boolean operators such as "AND", "OR", etc.
`Condition`	Represent a combination of "`Term RelationOp Term`".
`Constraint`	Represent a combination of "`Condition BooleanOp Condition`".
`Tactic`	Represent the base class for a tactic; providers implement tactics by creating specializations.
`Strategy`	Represent the base class for a strategy; providers implement strategies by creating specializations.

Function	
`replaceComponent(c)`	Represent an adaptation action that replaces an existing component `c` in the current configuration.
`changeParameter(pn,pv)`	Represent an adaptation action that changes the value of the parameter `pn` of a component to `pv`.
`connect(c1,c2)`	Represent an adaptation action that connects components `c1` and `c2`.
`setTacticObjective(obj)`	Set the component selection objective for a tactic to `obj`, which will be used for, e.g., the `replaceComponent` actions.
`addStrategy(S,C)`	Add the adaptation strategy `S` for run-time adaptation and associate it with constraint `C`.
`invokeTactic(N)`	Used in an adaptation strategy to invoke the tactic that is bound to the symbolic problem name `N`.
`bindTactic(N,T)`	Bind the tactic `T` to the triggering problem represented by the symbolic name `N`.
`addProblemConflict(N1,N2)`	Specify a problem level conflict between two triggering problems represented by the symbolic names `N1` and `N2`.
`addTacticCauses(T,IL,DL)`	Annotate the tactic `T` to specify its causes. `IL` is a list of metrics that, when increased, may trigger `T`. `DL` is a list of metrics that, when decreased, may trigger `T`.
`addTacticEffects(T,IL,DL)`	Annotate the tactic `T` to specify its effects. `IL` is a list of metrics that may be increased by `T`. `DL` is a list of metrics that may be decreased by `T`.

Figure 5.6: Extensions to the recipe APIs for adaptation support.

new functionalities, e.g., associating constraints with adaptation strategies, binding tactics to triggering problems, customizing and triggering adaptation strategies, executing tactics, and so on. The new architectural element for run-time adaptation support is the adaptation

coordinator (AC). In our current prototype, the AC is implemented as a sub-module of the synthesizer so that they can communicate with each other easily.

5.7 Evaluation

To evaluate our architecture, we want to (1) show that the coordination mechanisms work correctly and do not introduce unreasonable overhead and (2) demonstrate the flexibility of adaptation customization. Our evaluation is based on simulations of the massively multiplayer gaming service described earlier. Below we summarize the key properties of the simulations.

Service components. There are two types of nodes in a service configuration: users and servers. Each user moves around randomly in the virtual game space, and each server handles a partition of the space (i.e., all users within that space). The service quality experienced by a user is a function of the current load of its server: when the load is below 0.5, the quality is 1.0; when load is higher than 1.0, the quality is 0.0; when the load is between 0.5 and 1.0, the quality decreases linearly. The service has at its disposal N_p "primary servers" and an unlimited number of "emergency servers". Each server has capacity S (i.e., can handle S users). When added to the service configuration, each primary server has a cost of 1.0, and each emergency server has a cost of C_e.

Global configuration. The global configuration strategy of the service composes a service configuration that consists of $\lceil \frac{N_u}{T_g} \rceil$ servers, where N_u is the current number of users, and T_g is a provider-specified threshold that limits the maximum number of users on any server after global configuration. The strategy partitions the game space among the servers so that each server handles (nearly) the same number of users. At run time, the global configuration strategy is invoked periodically with interval I_g, and we assume that it returns a proposal that will be coordinated with the other adaptation strategies described below.

Adaptation strategies. As mentioned earlier, the self-configuring service has five adaptation strategies: *join* connects a new user to the corresponding server, *leave* disconnects a user from its server, *cross* moves a user's state from one server to another when the user moves between servers, *split* adds a new server when an existing server's load goes above T_o, and *merge* removes a server by merging two servers when their loads are below T_u. T_o and T_u are thresholds specified by the provider. Note that each of these strategies only invokes one tactic, so the names also refer to the tactics.

Let t_p denote the time between triggering and proposal, and t_e denotes the execution time of a proposal. We assume the following in the simulations.

$$t_p = \begin{cases} 10\text{ms} & \text{for join, leave, and cross} \\ 20\text{ms} & \text{for split and merge} \end{cases}$$

$$t_e = \begin{cases} 100\text{ms} & \text{for join, leave, and cross} \\ 500\text{ms} & \text{for split and merge} \end{cases}$$

Since at run time global configuration is treated as an adaptation, we assume that t_p and t_e for global configuration are 40 and 2000 ms, respectively.

Customization. The thresholds T_o and T_u are customized at global configuration time according to the total load of the servers: $(T_o = T_{oL} + (T_{oH} - T_{oL}) \cdot F)$ and $(T_u = T_{uL} + (T_{uH} - T_{uL}) \cdot F)$. $F = ((N_u/(N_p \cdot S)) - H)/(1.0 - H)$, where H is the load threshold beyond which T_o and T_u move from T_{oL} and T_{uL} towards T_{oH} and T_{uH}, respectively.

In other words, when the synthesizer executes the global configuration strategy, it carries out the above calculations and sets the thresholds for the split and merge strategies accordingly.

Coordination. The adaptation tactics can be divided into two sets, $S_1 = \{\text{split}, \text{merge}\}$ and $S_2 = \{\text{join}, \text{leave}, \text{cross}\}$, and the provider specifies the following set of problem-level conflicts in the global configuration strategy:

$$\{(t_1 \leftrightarrow t_2) | t_1 \in S_1, t_2 \in S_2\}.$$

As discussed in Section 5.5, the tactics can be annotated with their causes and effects so that the potential incompatibility between the split and merge tactics can be detected.

Simulator. We generate traces of user arrivals and departures with inter-arrival time I_a and stay duration I_d. I_a has an exponential distribution, and I_d has a bounded Pareto distribution with minimum 30 seconds, maximum 10.8K seconds, and $\alpha = 1.5$. We implemented an event-driven simulator that takes such a trace as input and simulates the gaming service as described above. The simulator is integrated with the synthesizer, which interprets our recipe specifying the adaptation knowledge described above. The simulation resolution is 10 ms, i.e., each simulation round simulates 10 ms.

During each simulation round, the simulator processes new events from the traces, executes the appropriate tactics to produce new proposals, coordinates (i.e., accepts or rejects) proposals, and executes accepted proposals. We assume that, if applicable, rejected proposals will be re-proposed by their corresponding tactics as soon as possible. In other words, a rejection will cause the rejected adaptation to be delayed. To prevent "starvation", we also assume that the re-proposals are processed on a FCFS basis, i.e., a re-proposal that was first rejected at time t_0 has precedence over another re-proposal first rejected at time t_1 if $(t_0 < t_1)$.

Next, we present the simulation results.

5.7.1 Coordination

In this section, we show that the coordination mechanisms work as expected and do not introduce unreasonable adaptation delay under the simulation settings. We present simu-

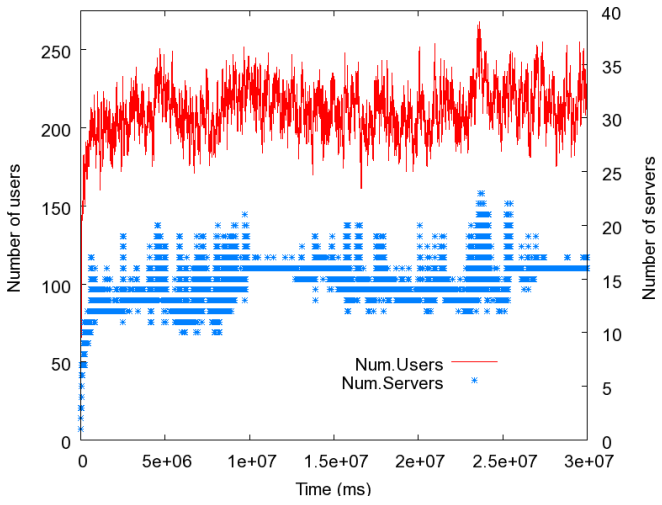


Figure 5.7: Number of users/servers during simulation: hi-load-ng.

lation results without the dynamic constraint customization, i.e., thresholds T_o and T_u are set to 40 and 10 throughout the simulations. S is set to 45, $C_e = 3.0$, and $T_g = 22$. The simulation duration for all results is 30K seconds. We experimented with two different user loads in our simulations. In the **hi-load** setting, the $E[I_a]$ is 400 ms, and it is 600 ms in the **low-load** setting. Therefore, the average number of users in the system is 213.17 and 142.12 for hi-load and low-load, respectively. In addition, N_p is set to 11 and 7 for hi-load and low-load, respectively.

Adaptation delay

In the first set of simulations, there are no periodic global configurations, i.e., the service configuration is adapted using only the local adaptation strategies. We use **ng** ("no-global") to label the results of these simulations, and we compare the results of **hi-load-ng** and **low-load-ng**. First, let us look at how the service configuration evolves as adaptations are applied to the system. Figures 5.7 and 5.8 show the numbers of users and servers during the simulation for hi-load-ng and low-load-ng, respectively. We can see that in both cases, the number of simultaneous users matches the number calculated from the average. We can also see that the local adaptation strategies are successful in maintaining the number of servers within a relatively small range. The results show that the adaptation strategies are working as expected.

Now we look at how the coordination mechanisms affects the adaptation time, which is defined as the time from when an adaptation strategy is triggered to when the adaptation is finished. In other words, if all proposals are allowed to execute without coordination, then the adaptation time is 110 ms for join/leave/cross and 520 ms for split/merge. Since the results for join, leave, and cross are very similar (and similarly for split and merge), below we present the adaptation time during simulation for join and split. Figures 5.9 and 5.10

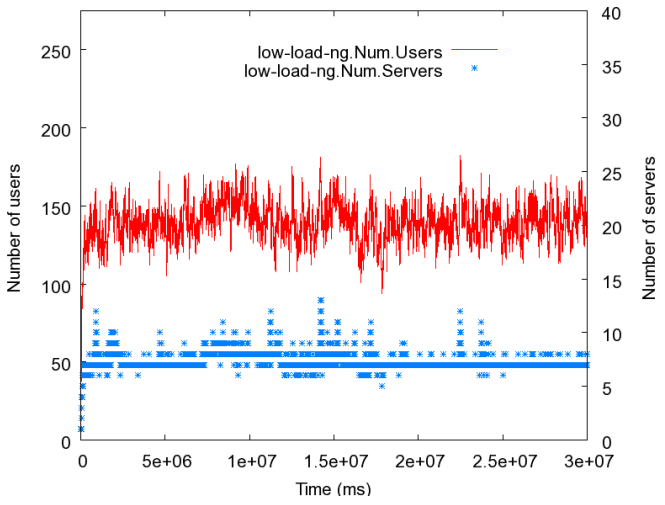


Figure 5.8: Number of users/servers during simulation: low-load-ng.

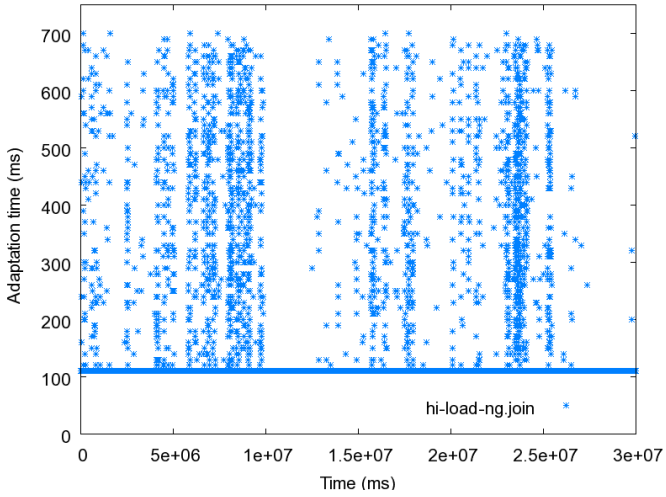


Figure 5.9: Adaptation time for join during simulation: hi-load-ng.

show the adaptation time for joins in hi-load-ng and low-load-ng, respectively. Figures 5.11 and 5.12 show the adaptation time for splits in hi-load-ng and low-load-ng, respectively. We can see that most joins (and similarly leaves and crosses) are at their minimum 110 ms, i.e., they do not encounter any conflicts and are not delayed by the coordination mechanisms. On the other hand, most splits (and similarly merges) are delayed and take more than the minimum 520 ms. To confirm this, we calculate the percentage of each type of adaptation that are delayed by coordination. The tables in Figure 5.13 shows the results for hi-load-ng and low-load-ng, and they confirm our observations. We can see that there is a large difference in the percentage of delayed adaptations between join/leave/cross and

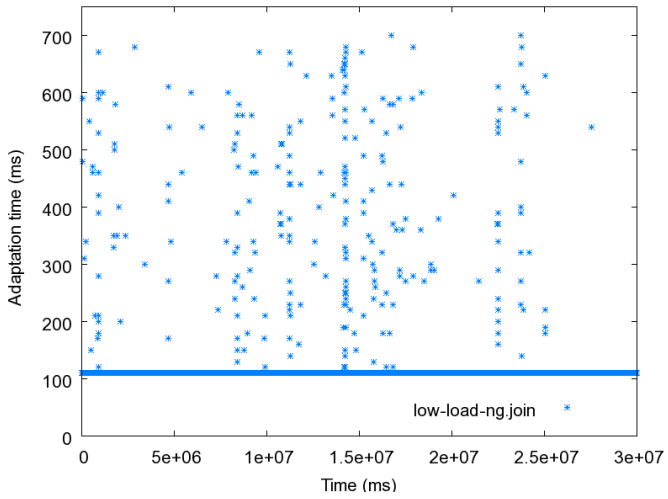


Figure 5.10: Adaptation time for join during simulation: low-load-ng.

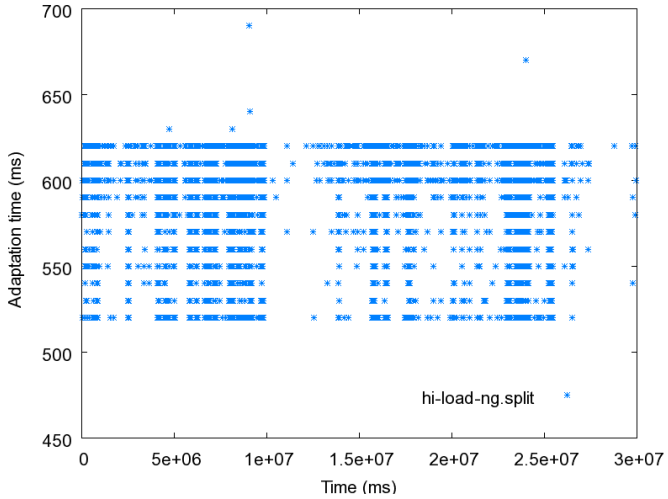


Figure 5.11: Adaptation time for split during simulation: hi-load-ng.

split/merge. The reason is that the number of split/merge adaptations is very small compared to join/leave/cross. Therefore, when a join/leave/cross is proposed, it is unlikely to conflict with an on-going split/merge. On the other hand, when a split/merge is proposed, it is likely to conflict with an on-going join/leave/cross. Another interesting result is that join/leave/cross are more likely to be delayed in hi-load-ng than in low-load-ng. This is because there are almost 10 times as many split/merge in hi-load-ng than in low-load-ng. As a result, a join/leave/cross proposal is more likely to encounter an on-going split/merge and be delayed. Finally, there are much more cross adaptations in hi-load-ng than in low-load-ng. The reason is that, as we have seen earlier, the number of servers in hi-load-ng is more

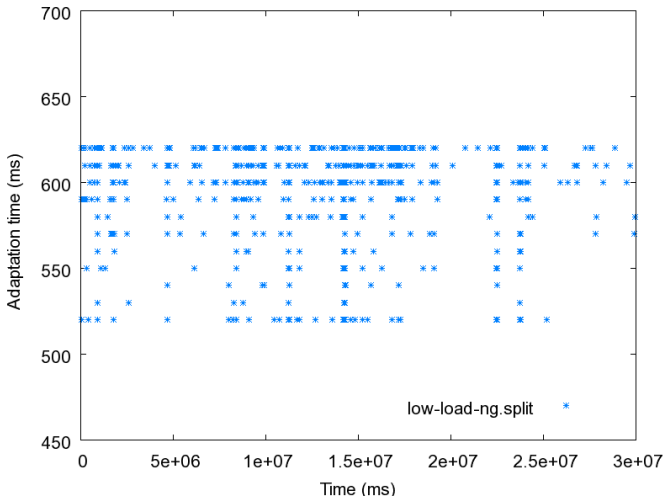


Figure 5.12: Adaptation time for split during simulation: low-load-ng.

hi-load-ng

	Num. adapt.	Adapt. delayed	Percent. delayed
join	74888	2045	2.73
leave	74674	3268	4.38
cross	15027269	421961	2.81
split	7332	6550	89.33
merge	7301	6869	94.08

low-load-ng

	Num. adapt.	Adapt. delayed	Percent. delayed
join	50149	258	0.51
leave	49992	384	0.77
cross	6194675	38146	0.62
split	789	715	90.62
merge	783	746	95.27

Figure 5.13: Number and percentage of delayed adaptations: hi-load-ng and low-load-ng.

than that in low-load-ng, and therefore, users are more likely to cross server boundaries in hi-load-ng.

To further investigate the impact of the delays caused by the coordination mechanisms, we look at the adaptation time distribution of the delayed adaptations. Again, because of the similarity between different adaptation, we only present results for join and split. Figures 5.14 and 5.15 show the adaptation time distributions for joins and splits, respectively. First, we can see that although there are more conflicts in hi-load-ng than in low-load-ng, resulting in more delayed adaptations, the distributions of adaptation time in hi-load-ng and low-load-ng are very similar. For join (and similarly for leave/cross), the adaptation delays (i.e., adaptation time minus the minimum 110 ms) are distributed roughly uniformly, and the maximum delay is around 600 ms. Although the worst-case delay is bad, the overall impact of delays of join/leave/cross is not significant because, as we see above, only a small percentage of the join/leave/cross adaptations are affected. On the other hand, although most splits/merges do encounter conflicts, the adaptation time in most cases is still below 620 ms. Given that split and merge are rare (compared to the others) and that they are

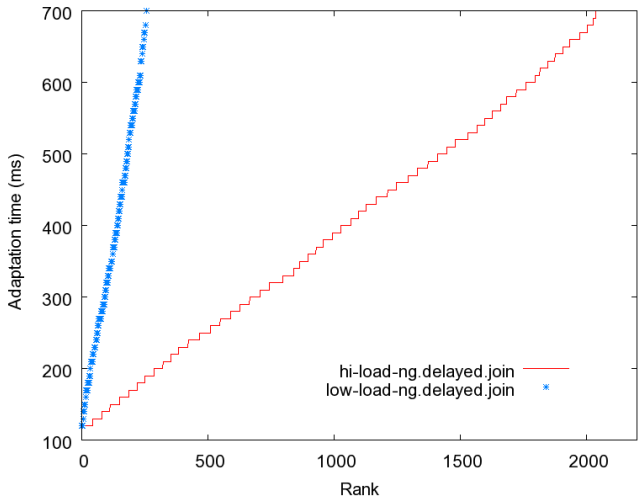


Figure 5.14: Adaptation time distribution for delayed joins.

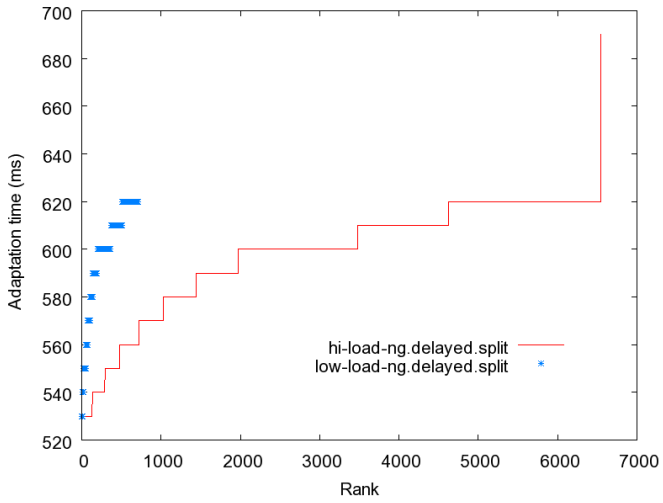


Figure 5.15: Adaptation time distribution for delayed splits.

expensive (520 ms) even without conflicts, the additional delay caused by the coordination mechanisms also does not have a significant overall impact.

To summarize, the results show that using our coordination mechanisms, the majority of adaptations will not experience delays caused by coordination as long as the adaptation actions can be completed in a relatively short time. Even for adaptations that take a long time to complete, the overall impact of the delays caused by coordination is not significant if such more expensive adaptations are not invoked very frequently. Therefore, our coordination mechanisms should work well for most services since we expect that expensive adaptations should be much less frequent than inexpensive ones in general.

	Quality	Cost
low-load-ng	0.9573	0.0567
low-load-g	0.9506	0.0598
hi-load-ng	0.9666	0.1014
hi-load-g	0.9452	0.0864

Figure 5.16: Quality and cost: no-global vs global.

Effects of periodic global configuration

In the second set of simulations, we add periodic global configurations with I_g = 1500 seconds. We use **g** ("global") to label the results of these simulations, and we compare the results of **hi-load-g** and **low-load-g**. We now look at the benefits and overhead of periodic global configuration.

Benefits of periodic global configuration. The benefit of adding global configuration is that a global view can be used to optimize the resulting service configuration. For this service, the two metrics for evaluating the service are quality and cost as defined earlier. Since quality is a metric of individual servers, it can be addressed sufficiently well by the local adaptation strategy split. On the other hand, cost is a metric of the collection of servers used. Since the merge strategy is only invoked when two adjacent servers are both underutilized, in the worst case no servers can be merged even if half of the servers are underutilized (but no two are adjacent).

While the local merge strategy may miss some opportunities for reducing the cost, global configuration can redistribute the load evenly using a smaller number of servers, i.e., global configuration can "merge" non-adjacent servers. However, how effective this is depends on the number of servers in the configuration. For example, if 2 out of n servers are underutilized, the 2 underutilized servers are $\frac{n-3}{2}$ times as likely to be non-adjacent as to be adjacent. In other words, as n increases, they are more likely to be non-adjacent. Therefore, in our simulation settings, we would expect that periodic global configurations will have a more significant benefit when the load (and therefore the number of servers) is higher.

The table in Figure 5.16 summarizes the average quality and the average cost observed in both no-global (from the previous section) and global simulations. We can see that when the load is low, adding global configuration actually slightly increases the overall cost. The reason is that since the number of servers are low, when a global configuration redistributes the load, it is more likely to increase the number of servers (so that each server has less than T_g users) than to decrease it. In the hi-load simulations (hi-load-ng and hi-load-g), adding global configuration is able to reduce the average cost by roughly 15%. This confirms our intuition that the effectiveness of global configuration increases with the load for the strategies used in the simulations.

One interesting issue that arises from combining local adaptation with periodic global

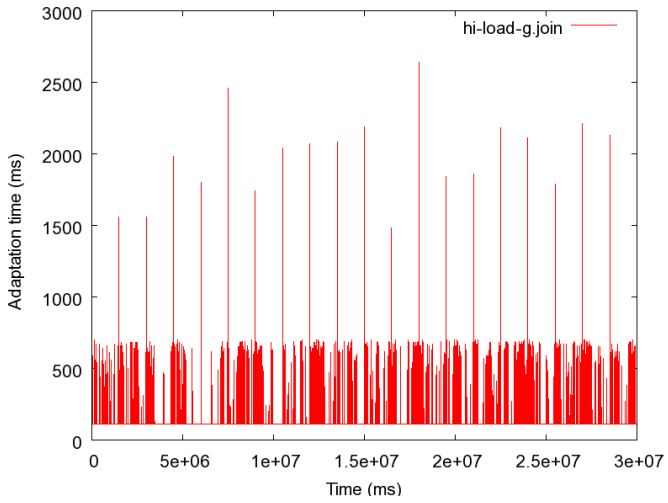


Figure 5.17: Adaptation time for join during simulation: hi-load-g.

configuration is how to "align" the "goals" of local adaptation and global configuration? Obviously, the global configuration strategy and the local adaptation strategies cannot have identical goals since they are designed for different scopes. In the above example, the global goal is to "maintain a minimum quality with the least cost", and the local goals are to "increase quality" (split) and to "decrease cost" (merge). We can see that the individual local goals in fact are contradictory to the global goal: split increases the cost, and merge decreases the quality. However, the combination of the split and merge strategies results in a rough alignment with the global goal. As we discuss later, there has been related work looking at similar issues, for example, given a global objective function how to derive local objective functions for individual system components. In this dissertation, we assume that the provider takes the alignment issue into consideration when designing the adaptation strategies. How to solve the general problem is an area for further study.

Overhead of periodic global configuration. On the other hand, adding periodic global configurations is likely to increase the delay caused by coordination since a global configuration "conflicts" with all the local adaptations. Therefore, we now look at the adaptation time for each type of adaptation during the global simulations. Note that after a global configuration is completed, any previously pending crosses/splits/merges are no longer necessary and are therefore removed in the simulations. Therefore, the adaptation time in such cases is defined as from the triggering of the adaptation to the completion of the global configuration. In contrast, joins/leaves that are delayed by a global configuration need to be re-proposed after the global configuration is completed.

First, we use join as an example to illustrate the effect of global configuration during the simulations. Figures 5.17 and 5.18 show the adaptation time for join in hi-load-g and low-load-g, respectively. We can see that the majority of the adaptations fall within the range observed in the previous hi-load-ng and low-load-ng results, i.e., 110 to 620 ms.

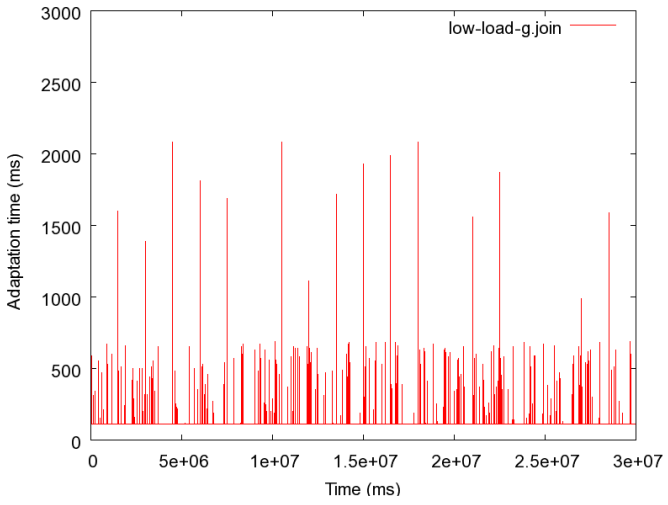


Figure 5.18: Adaptation time for join during simulation: low-load-g.

hi-load-g

	Num. adapt.	Adapt. delayed	Percent. delayed
join	74888	5231	6.99
leave	74674	8122	10.88
cross	13637992	1056223	7.74
split	18536	16182	87.30
merge	18396	17113	93.03

low-load-g

	Num. adapt.	Adapt. delayed	Percent. delayed
join	50149	475	0.95
leave	49992	657	1.31
cross	6248286	59978	0.96
split	1251	1148	91.77
merge	1233	1187	96.27

Figure 5.19: Number and percentage of delayed adaptations: hi-load-g and low-load-g.

However, there are spikes that reach around 2000 ms, which is roughly the duration of a global configuration, and if we look at the timing of these spikes, we can see that they correspond to the periodic invocations of global configuration. The results of other types of adaptations are similar. To take a closer look, we calculate the percentage of each type of adaptation that are delayed by the coordination mechanisms. The results are summarized in the tables in Figure 5.19. The main difference between the results here and the results from the no-global simulations (Figure 5.13) is that the percentage of delayed join/leave/cross in hi-load-g/low-load-g is roughly a factor of 2 to 3 higher than that in hi-load-ng/low-load-ng. To explain the difference, we compare the adaptation time distribution of the delayed join/leave/cross in hi-load-g/low-load-g and hi-load-ng/low-load-ng. Since the results of hi-load and low-load are similar, and join/leave/cross are also similar, here we present the results of adaptation time distribution of delayed joins in hi-load-g and hi-load-ng, shown in Figure 5.20. We can see that in hi-load-g, the distribution has two pieces. The lower piece corresponds to the delays caused by conflicts with split/merge adaptations since it has the same range as the distribution in hi-load-ng. The higher piece corresponds to delays caused

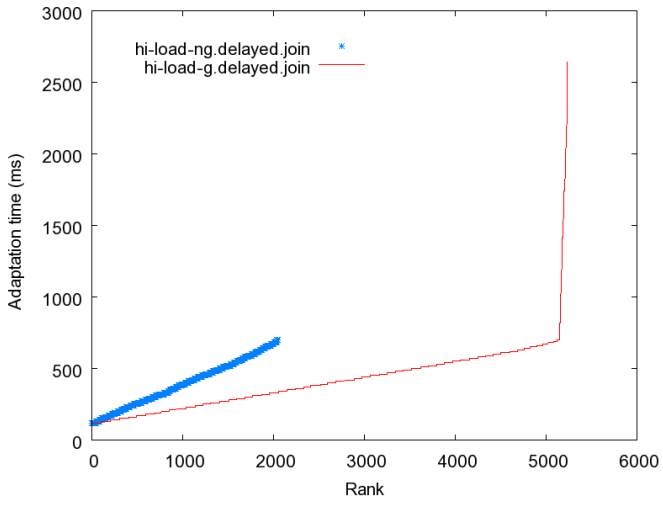


Figure 5.20: Adaptation time for delayed joins during simulation: hi-load-g and hi-load-ng.

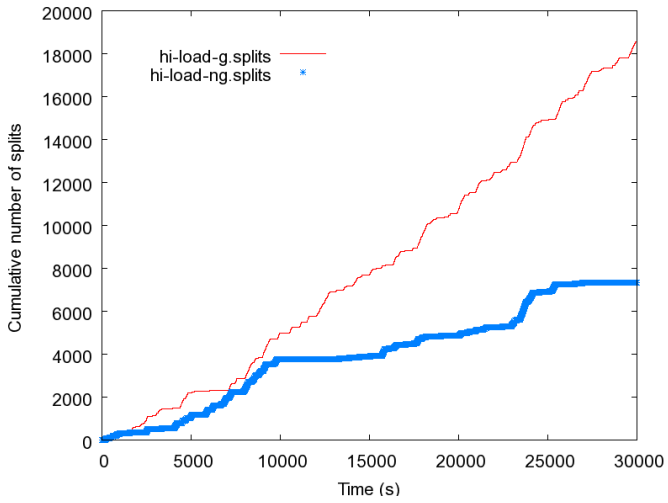


Figure 5.21: Cumulative number of splits during simulation: hi-load-g and hi-load-ng.

by conflicts with the periodic global configurations. Therefore, the increase in conflicts is a result of both the addition of conflicts with global configuration and an increase in conflicts with split/merge.

Furthermore, the reason for the increase in the conflicts with split/merge is that there are more splits/merges in the global case than in the no-global case. This is demonstrated in Figure 5.21, which shows the cumulative number of splits during simulation in hi-load-g and hi-load-ng. The results for the low-load cases and for merge are similar and are therefore not presented. We can see that the service configuration in hi-load-ng "stabilizes" after a while, i.e., the frequency of splits decreases after the initial ramp-up. On the other

hand, in hi-load-g the number of splits keeps increasing at roughly constant rate. As a result, more split/merge adaptations are proposed when period global configurations are added, resulting in more conflicts between join/leave/cross and split/merge.

To summarize, adding periodic global configuration to run-time adaptation can be beneficial when achieving a particular adaptation goal (cost reduction in the above example) requires global knowledge. On the other hand, because of the additional overhead caused by coordination between local adaptation and global configuration, a service provider should evaluate the trade-off between the benefit and the overhead to determine the frequency of periodic global configuration.

Different coordination policies

The coordination policies used in the simulations above are expressed using the recipe APIs as follows.

```
config.addConflict("JOIN", "OVERLOAD");
config.addConflict("JOIN", "UNDERUTILIZE");
config.addConflict("LEAVE", "OVERLOAD");
config.addConflict("LEAVE", "UNDERUTILIZE");
config.addConflict("CROSS", "OVERLOAD");
config.addConflict("CROSS", "UNDERUTILIZE");
config.addConflict("JOIN", "GLOBAL");
config.addConflict("LEAVE", "GLOBAL");
config.addConflict("CROSS", "GLOBAL");
config.addConflict("OVERLOAD", "GLOBAL");
config.addConflict("UNDERUTILIZE", "GLOBAL");
```

Using our recipe-based approach, implementing a different set of coordination policies is very straightforward since the provider does not need to worry about the low-level coordination mechanisms. As an example, assume that the provider of the above gaming service improves the game server implementation such that it is able to handle a leaving user in parallel with any other adaptations. In other words, the “leave” adaptation now does not conflict with any other adaptations. To take advantage of this new capability, the provider can simply modify the recipe by removing the lines specifying conflicts that involve “leave”. This results in the following policies.

```
config.addConflict("JOIN", "OVERLOAD");
config.addConflict("JOIN", "UNDERUTILIZE");
config.addConflict("CROSS", "OVERLOAD");
config.addConflict("CROSS", "UNDERUTILIZE");
config.addConflict("JOIN", "GLOBAL");
config.addConflict("CROSS", "GLOBAL");
config.addConflict("OVERLOAD", "GLOBAL");
config.addConflict("UNDERUTILIZE", "GLOBAL");
```

low-load-ng-noLC

	Num. adapt.	Adapt. delayed	Percent. delayed
join	50149	184	0.37
leave	49992	0	0
cross	6375261	28528	0.45
split	623	573	91.97
merge	613	587	95.76

low-load-ng

	Num. adapt.	Adapt. delayed	Percent. delayed
join	50149	258	0.51
leave	49992	384	0.77
cross	6194675	38146	0.62
split	789	715	90.62
merge	783	746	95.27

Figure 5.22: Number and percentage of delayed adaptations: low-load-ng-noLC and low-load-ng.

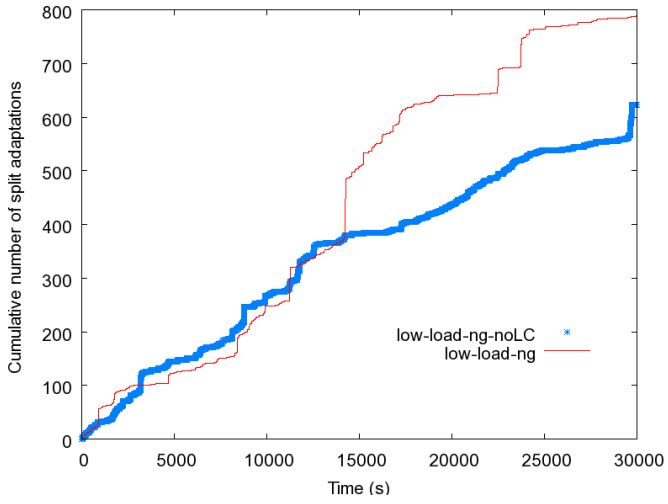


Figure 5.23: Cumulative number of splits during simulation: low-load-ng-noLC and low-load-ng.

We perform another set of simulations to verify that such changes in the recipe indeed results in expected coordination behavior at run time. The simulation parameters are the same as low-load-ng except for the recipe differences discussed above, and we label this set of results low-load-ng-noLC ("no leave conflicts"). We now look at the the percentage of delayed adaptations in low-load-ng-noLC and how it differs from the low-load-ng results. Figure 5.22 shows the comparison. Of course, as expected, no leave adaptations are delayed in low-load-ng-noLC, i.e., no conflict. Furthermore, we can see that eliminating conflicts that involve leave adaptations actually results in fewer split/merge adaptations. Figure 5.23 shows the cumulative number of split during simulation for low-load-ng-noLC and low-load-ng (result for merge is similar). We can see that in low-load-ng-noLC, the frequency of split decreases after about half-way into the simulation. In other words, the removal of conflicts with leave adaptations actually helps stabilize the service configuration, resulting in fewer split/merge adaptations. In turn, this allows more join/cross adaptations to be

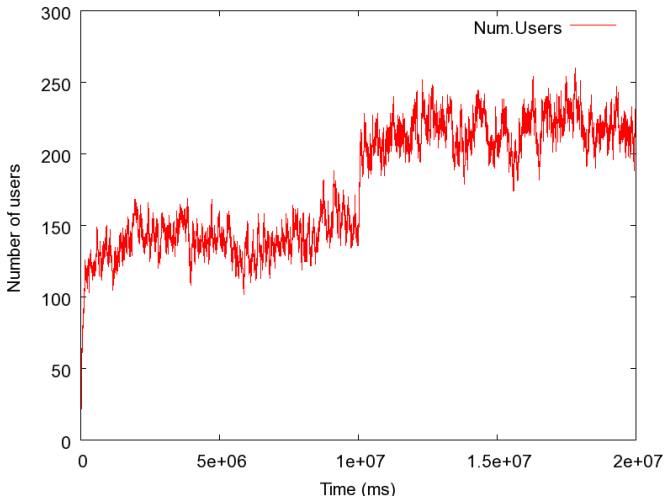


Figure 5.24: Number of users during simulations.

executed without delay, as demonstrated by the lower percentage in Figure 5.22. Finally, because the users now spend less time "waiting" (i.e., delayed by conflicts), they have more time to move around in the game space, and this results in an increase in the number of cross adaptations in low-load-ng-noLC.

To summarize, this example demonstrates that our recipe-based approach allows a service provider to easily implement service-specific coordination policies without worrying about the actual coordination mechanisms.

5.7.2 Customization

In this section, we illustrate how the synthesizer can use a provider's service-specific customization knowledge to customize the local adaptation strategies according to the actual configuration and environment, and more specifically, how the triggering constraints of strategies can be customized to achieve the provider's desired outcomes.

We perform simulations using the same gaming service scenario as in the previous section. We look at how the split and merge thresholds T_o and T_u can be customized to achieve the provider's goals in two different scenarios. In the first scenario, the provider's primary goal is to maintain the average cost at about 0.1 unit per user, and the secondary goal is to provide good quality under the cost constraint. We label this the "constant-cost" scenario. In the second scenario, the provider's primary goal is to maintain the average service quality at around 0.95, and the secondary goal is to lower the cost under the quality constraint. We label this the "constant-quality" scenario.

For the simulations, we generate a user trace with duration 20K seconds, and to simulate a change in the environment, $E[I_a]$ in the first 10K seconds is set to 600 ms, and $E[I_a]$ in the second half is 400 ms. In other words, the load in the second half of the simulations is higher than that in the first half. This change in load can be seen in Figure 5.24, which

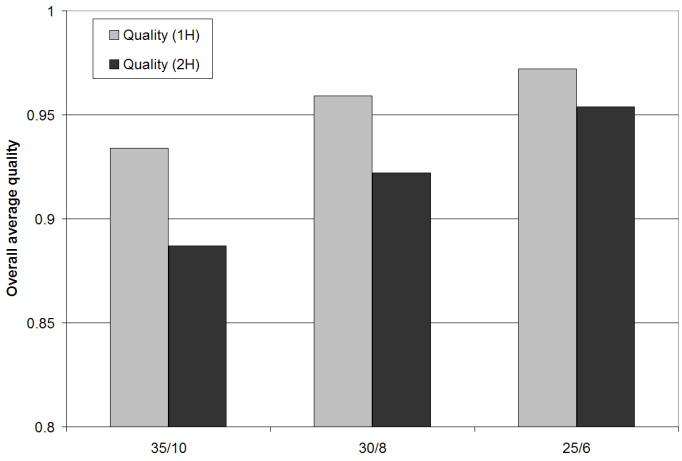


Figure 5.25: Overall average quality: no customization.

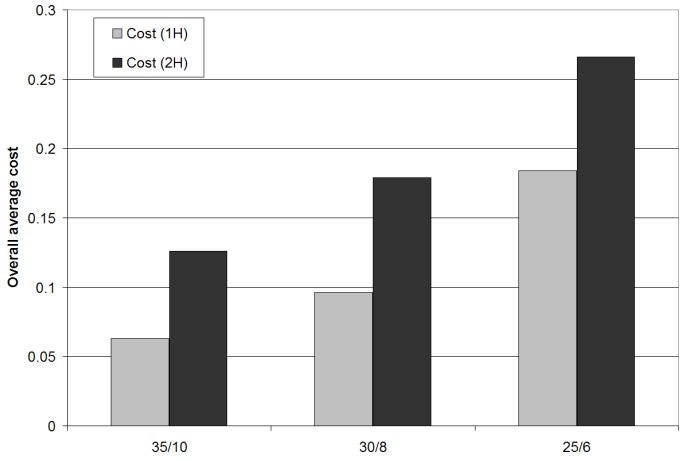


Figure 5.26: Overall average cost: no customization.

shows the number of users during one of the simulations (results from other simulations are similar). For all simulations, S is set to 40, $C_e = 3.0$, $N_p = 10$, and I_g is 1K seconds.

First, we look at the results for adaptation without constraint customization, i.e., T_o and T_u are set to fixed values throughout the simulations. We perform three simulations with three different pairs of static (T_o, T_u) values: (35, 10), (30, 8), and (25, 6), and we label these experiments 35/10, 30/8, and 25/6. T_g is set to 20, 18, and 16 for the three simulations, respectively. Note that with a lower T_o, the split strategy will be applied more aggressively. On the other hand, with a lower T_u, the merge strategy will be applied less aggressively. Therefore, using lower values for T_o and T_u should result in higher quality and higher cost.

Figure 5.25 shows the overall average quality of the three simulations. The overall average quality of a simulation is computed by first computing the average quality of all

users in each simulation round (10 ms) and then computing the average of all rounds. In the figure, we present the overall average quality for the first half and the second half of the simulations. Similarly, we present the overall average cost results in Figure 5.26. We can see that in all three simulations, the quality in the second half is noticeably worse than that in the first half due to the higher load, and yet the cost in the second half is still significantly higher than that in the first half. In terms of the constant-cost scenario, we can see that among the three different static settings, only 35/10 comes close to maintaining the overall average cost below 0.1 in both halves of the simulation. However, in the first half, it only spends about 60% of the allowed budget, resulting in unnecessarily low quality, and therefore, it does not achieve the secondary goal in this scenario. On the other hand, for the constant-quality scenario, only 25/6 can achieve the primary goal of maintaining the overall average quality around 0.95. However, in the first half, it actually achieves noticeably higher quality than the provider's constraint, resulting in unnecessarily high cost, and therefore, it does not achieve the secondary goal.

These results show that the goals in the two scenarios cannot be satisfied by the three static settings. They also suggest that in order to maintain constant cost, the split/merge strategies need to be less aggressive when the load is high. Similarly, in order to maintain constant quality, the strategies need to be more aggressive when the load is high. Based on these observations, we devise the following two ways to customize the strategies. The parameters in these customization schemes are derived through experimentation.

- **constCost**: The triggering constraints for split/merge are customized by setting T_{oL}, T_{oH}, T_{uL}, and T_{uH} to 30, 60, 8, and 20, respectively. Note that although T_{oH} is 60, the maximum value of T_o in the simulation is less than 40 since the maximum load is around 250 users, i.e., about 62.5%.

- **constQual**: T_{oL}, T_{oH}, T_{uL}, and T_{uH} are set to 30, 8, 8, and 1, respectively. Note that the thresholds need to decrease as the load increases since it is more "difficult" to maintain the same level of quality at higher load.

Now we look at the results of these two customization schemes. Figure 5.27 shows the overall average quality of the two schemes, and Figure 5.28 shows the overall average cost. When using the constCost customization scheme, we can see that the overall average cost is kept close to the 0.1 level in both halves of the simulation. Therefore, when compared with the 35/10 static setting, constCost can achieve better quality in the first half when the load is low. On the other hand, constQual is able to keep the overall average quality closest to 0.95 in both halves of the simulation. Therefore, when compared with the 25/6 static setting, constQual lowers the average cost significantly in the first half. To summarize, these customization schemes allow the service provider to achieve the goals in the two target scenarios.

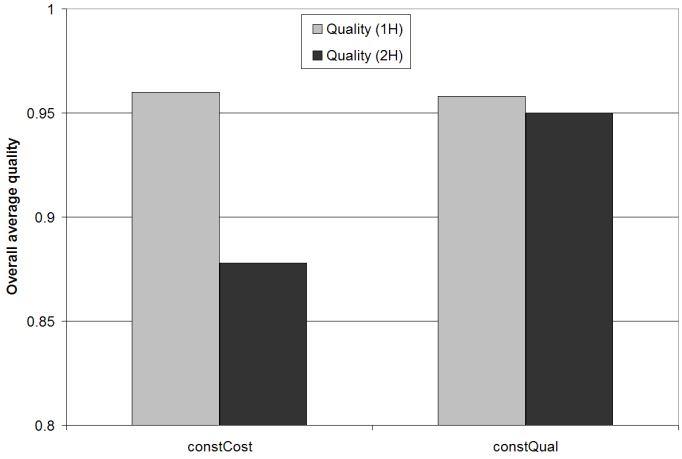


Figure 5.27: Overall average quality: with customization.

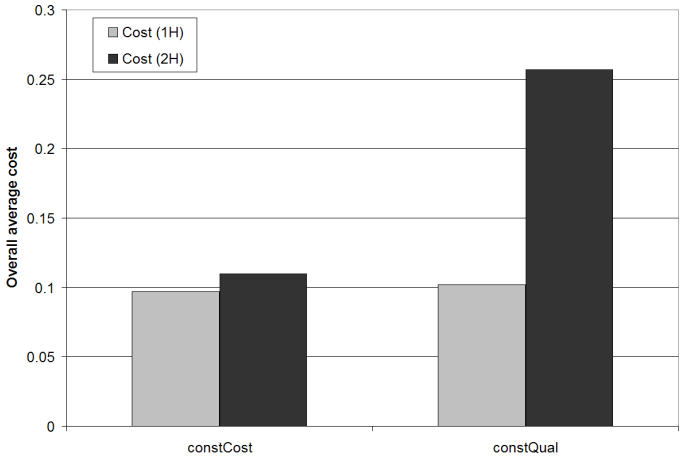


Figure 5.28: Overall average cost: with customization.

5.8 Related work

There have been many previous efforts to add run-time adaptation capabilities to various systems. As discussed earlier, most of these efforts have focused on what we call "parameter-level" adaptation. For example, some earlier work focused on adapting the communication in a client-server system. The Rover toolkit [80] facilitates the design of distributed client-server applications by providing mechanisms for dynamically adjusting communication parameters to adapt to network problems. The Quality Object (QuO) framework [89] provides a Contract Description Language (CDL) for specifying the QoS contracts between a client and a server, and its runtime system monitors the contracts and adapts the client-server communication to changing system conditions. Odyssey [100] provides APIs for applications to obtain resource availability and dynamically adjusting the fidelity level in response to changes. Puppeteer [33] uses transcoding proxies between a

client and a server to adapt the communication pattern to changing network characteristics using control APIs exported by the components. The tactic-based remote execution approach [7] uses "tactics to capture the application-specific knowledge in workload partitioning for remote execution, and it provides the supporting mechanisms such that dynamic re-partitioning capability can be easily added to a client-server application.

Some other parameter-level approaches aim for more general distributed applications. For example, the Program Control Language (PCL) framework allows developers to program distributed adaptive applications that can dynamically adapt to environment changes by modifying their run-time parameters such as communication pattern, video compression scheme, and so on [40]. An application-independent adaptation framework is proposed in [16] that can automatically determine when adaptation is needed and modify component control parameters accordingly using a tunability interface exported by the application. In [108], a framework is proposed to adapt composite services by dynamically adjusting the fidelity level of each component to optimize a utility function Although the adaptation also includes choosing suppliers of components, the selection of supplier does not change the semantics of a component, and the selection is among a small number of choices that are mapped directly into the utility space, e.g., selecting one of Word and Emacs for text processing. Therefore, we consider it a parameter-level approach. The SMART framework [10] uses a linear state feedback model to predict changes in environment (e.g., memory usage) and to modify the system component accordingly to adapt to such changes. Again, we consider this a parameter-level approach since the selection of the system component is focused on selecting implementations that have different performance trade-offs, e.g., selecting one of two matrix multiplication algorithms. A policy-driven approach for mobile adaptive systems is proposed in [37]. A policy language derived from Event Calculus is used to specify when an adaptation should be performed and what specific actions to perform. The actions are parameter-level adaptations supported by the control interface of each component.

Dynamic resource allocation is another form of run-time parameter-level adaptation. For example, in [34], a model is derived to capture the the relation between performance metrics and resources. A resource allocator can then dynamically change the resource allocation among different services in response to changing changing performance requirements and resource availability. In [128], a utility function is used to specify the preference for resource allocation. A resource arbiter can then dynamically change the resource allocation to maximize the utility in a constantly changing environment. Q-RAM [85] addresses the problem of resource reservation and admission control for multiple resources with the goal of maximizing a utility function that involves multiple QoS dimensions.

In contrast to these parameter-level efforts, some other studies look at "component-level" adaptations, i.e., adaptations that require changes in the structure of the target system. For example, Panda [115] allows a client-server system to adapt to various network problems by combining a series of adaptors that can remedy such problems between the client and the server. The CANS infrastructure [48] supports similar functionalities and also combines parameter-level adaptations in individual components. These solutions fo-

cus on a "path model" that adapts the output of a sender to the input of a receiver. In contrast, our approach targets services that have more general configurations.

Other component-level adaptation studies support more general component-based systems. For example, a compositional approach is proposed in [121] to support communication among heterogeneous components by composing a set of connectors that can transform the data to ensure compatibility. In [103], an architecture-based approach is proposed to support run-time adaptation by manipulating the architectural model of the target system, e.g., adding or removing components. An architectural model represents, at an abstract level, the components and connections in the system, and it provides explicit integrity constraints that guide the run-time adaptation. The Rainbow framework [19, 52] also uses the architecture-based adaptation approach and provides general support for different architectural styles.

Although we adopt the strategy specification approach used in Rainbow, there are several major differences between our work and Rainbow. Broadly speaking, Rainbow is an adaptation framework that can potentially support very general forms of adaptation while we make specific contributions in the area of local adaptation that addresses performance concerns. We now discuss the differences in more detail. First, in this thesis we focus on the performance concern, i.e., we specifically target adaptations that maintain the performance properties of a service configuration. In contrast, the architecture-based approach in Rainbow more generally supports adaptation based on different concerns that can be represented in an architectural model. Similarly, while we focus on supporting adaptation strategies that are within a local scope, Rainbow has a global view of the target system because of the use of an architectural model, and it can support more global adaptation strategies. Secondly, Rainbow generalizes architecture-based adaptation by enabling system designers to choose the most appropriate architectural style that provides relevant system properties and analytical methods for the target system. Therefore, adaptation is based on strategies that leverage style-specific analytical methods and adaptation operators, and different adaptation needs can be satisfied by selecting the appropriate style. In our work, we leave it to the provider to design service-specific adaptation strategies that address the particular run-time problems. Finally, Rainbow does not explicitly address the customization and coordination issues for adaptation strategies. As discussed earlier, although customization can be done individually within each strategy, a centralized customization mechanism can reduce the complexity of strategy design for providers. In addition, customization may require information about the global configuration, which may not be available to individual strategies. Moreover, in many cases, coordination of different adaptation strategies is necessary to avoid conflicting actions or to ensure the strategies do not work against each other. Therefore, our approach provides explicit support for customization and coordination.

Some of the previous efforts perform adaptations using strategies that are independent of the target system, for example, by learning when and how to perform adaptations (e.g., [16]) or by using type-based composition (e.g., [115, 48]). Other efforts support service-specific adaptation strategies to some degree. Some use "internalized" strategies, i.e., the strategies are embedded into the system and/or components, for exam-

ple, [80, 100, 40, 10]. Others supports "externalized" strategies that are separated from the adaptation mechanisms, for example, [33, 37, 34, 128, 108, 85, 7, 52]. On the other hand, as discussed earlier, adaptation strategies can be expressed in different forms. One possibility is to specify a utility function or a similar high-level objective that can be used to guide the adaptation. This approach is used in many previous studies, especially the studies focusing on dynamic resource allocation, for example, [85, 108, 34, 128, 10]. Another possibility is to specify an "event-action" rule that dictates when and how to perform adaptation, for example, [100, 33, 40, 52, 37, 7].

Compared to the previous efforts described above, our approach focus on component-level adaptation but also support parameter-level adaptation. Our work is built on the externalized approach in [52] and adopts the event-action approach for specifying adaptation strategies. As discussed earlier, this is because the externalized approach reduces the development cost, and the event-action approach is more feasible in our context. Moreover, [92] argues that one necessary condition for self-healing or self-repairing in a distributed system is that the system must possess "regularities" that are satisfied by all possible configurations. This argument can be used in favor of the event-action approach since, give this condition, it is natural to define rules that perform adaptive actions when such regularities are violated.

One main difference between our work and the previous efforts is that the previous efforts did not specifically address the issue of adaptation strategy customization based on the actual service configuration or environment. Of course, in some previous approaches, it is possible to implement ad-hoc customization schemes, for example, by embedding customization logic into every strategy. However, this is more cumbersome than our approach, which allows explicit specification of customization knowledge in an integrated fashion. Moreover, if customization logic is implemented in individual strategies, it may not have the necessary knowledge about the global configuration to perform the customization.

Another major difference between previous work and our effort is in how adaptation coordination is handled. Some previous efforts address coordination problems that are different from the problem we are addressing. For example, in [40], a single adaptation may consist of multiple simultaneous actions at different components in the distributed system, and the study addresses the problem of coordinating these actions to ensure synchronization. In [11], the focus is on selecting a common parameter, e.g., the communication protocol, among multiple components, and each component may have multiple feasible values for the parameter. Therefore, the coordination problem addressed in the study is how to select a value for the parameter such that the value is acceptable to all components and also maximizes a global utility, and the proposed solution is a combination of utility function and distributed auction. In contrast, our focus is on the coordination problem of detecting and resolving conflicts between adaptation strategies that can be triggered and executed independently. In this context, the coordination problem can be handled in different ways as described below.

One possibility is to design the adaptation framework such that conflicts cannot occur. As mentioned earlier, two possible conflict avoidance approaches are "monolithic" and "in-

dependent". In the monolithic approach, all adaptation decisions are made using a single monolithic strategy. For example, in Odyssey [100], the resource allocation among concurrent applications is performed centrally. In utility-based resource allocation schemes such as [85, 108, 34, 128], all resource parameters are controlled by the resource allocator according to the global utility function. On the other hand, the independent approach partitions the target system into independent parts, and each part then adapts independently. For example, in Odyssey [100], each application adjusts its own fidelity level to adapt to resource availability changes. In [132], the focus is on the problem of how to translate a given global utility for a composite system into a set of private utilities for the individual components of the system so that the collective behavior of all components approximately optimizes the global utility. The independent approach has also been used to address similar problems in other areas, for example, policy management for distributed systems [93, 28] As discussed earlier, either of the two approaches is only applicable in a limited context.

Instead of conflict avoidance, another solution for adaptation coordination is to detect and resolve conflicts at run time. Although such an solution has not been explicitly studied in the context of adaptation coordination, previous studies have looked at similar problems such as coordinating the execution of event-condition-action policies [22] and coordinating update rules in an active database system [76]. Both of these studies adopt the epoch/priority approach for conflict detection and resolution. As discussed earlier, the application of the basic epoch/priority approach requires a number of assumptions that are specific to the contexts targeted by these studies.

5.9 Chapter summary

In this chapter, we have described how we extend the recipe-based self-configuration architecture to support run-time local adaptations. We identified three important aspects of service-specific adaptation knowledge: adaptation strategies, customization, and coordination. We extended the recipe APIs to allow a service provider to specify service-specific adaptation strategies that are triggered by constraint violations and tactics that address different problems. The new APIs also support strategy customization based on the actual global configuration and environment, including strategy selection, dynamic constraint, and dynamic tactic binding. Finally, for adaptation coordination, we categorize adaptation conflicts into action-level conflicts, which are detected automatically, and problem-level conflicts, which are identified by the provider and specified in the recipe. We propose a first-come, first-serve approach for resolving detected conflicts, and we present a tactic annotation scheme that can help identify tactics that potentially work at cross purpose. For evaluation, we performed simulations using a massively multiplayer online gaming service scenario. The results show that the coordination mechanisms work as expected and incur only minor overhead, the recipe-based approach allows providers to easily design service-specific coordination policies, and the flexibility of the customization mechanisms allows providers to easily design customization schemes to achieve their service-specific goals.

Chapter 6

Conclusions and Future Work

Accessing services provided through the Internet has become a necessity for more and more people, and improving the performance of such services is an important research direction. As Internet connectivity and user hardware/software capabilities rapidly become more and more heterogeneous, it has become apparent that traditional statically integrated services cannot cope with the resulting variations of user requirements, resource availabilities, and other environment characteristics at both invocation time and run time. Self-configuration is an emerging approach for addressing such problems. In this dissertation, we have shown that it is feasible to abstract the service-specific self-configuration knowledge from the generic self-configuration mechanisms. As a result, self-configuration can be achieved using a general architecture while maintaining the ability to make use of the service-specific knowledge. Therefore, our recipe-based self-configuration approach reduces the development cost of building self-configuring services and increases the effectiveness of the resulting services. We now summarize our contributions and propose several future research directions.

6.1 Contributions

- **Recipe-based self-configuration architecture.** We identified that the key to improving the effectiveness of self-configuration is the use of service-specific knowledge, and the key to reducing the development cost is to provide shared generic self-configuration mechanisms. Therefore, we proposed a new architecture that abstracts the service-specific self-configuration knowledge from the lower-level mechanisms. This recipe-based self-configuration architecture achieves high effectiveness close to the previous service-specific approach by allowing service providers to express their service-specific knowledge in service recipes so that the providers can customize both invocation-time and run-time self-configuration. On the other hand, our architecture achieves low development cost close to the previous generic approach by providing shared infrastructures that are required for self-configuration.

- **Network-Sensitive Service Discovery infrastructure.** We identified that one important aspect of service-specific self-configuration knowledge is the network performance criteria for component selection, and we observed that existing solutions for component selection operations based on such knowledge are not efficient. Therefore, we built the Network-Sensitive Service Discovery (NSSD) infrastructure that integrates the functionalities of traditional service discovery and network-sensitive server selection. Such an integrated approach allows the caching and aggregation of service queries and thus reduces the overhead of network-sensitive component selection operations. Furthermore, to reduce the complexity of global optimization problems in component selection, NSSD provides the best-n-solutions feature that allows the synthesizer to approximate global optimization in a reduced search space.

- **Synthesizer and recipe representation.** In order to abstract the service-specific knowledge from the lower-level self-configuration mechanisms, we identified two key parts of self-configuration that are service-specific: the construction of the abstract configuration for a particular request and the objective function for optimizing the component selection. Therefore, we designed a recipe representation that can be used by a service provider to write a recipe expressing such service-specific knowledge. We then built the synthesizer that acts as an interface between the knowledge and the mechanisms, i.e., it interprets the recipe and accesses the lower-level mechanisms to perform self-configuration according to the abstracted knowledge.

- **Local adaptation support.** We identified three important aspects of service-specific adaptation knowledge: adaptation strategies, strategy customization, and strategy coordination. Although most previous solutions for run-time adaptation support some form of adaptation strategies, they do not address the customization and coordination issues. For customization, we observed that ad-hoc customization of individual strategies increases the complexity of strategy design, and furthermore, it may not be feasible if global configuration properties are required. Therefore, our approach allows providers to specify customization knowledge as part of global configuration. For coordination, we identified that problem-level conflicts are service-specific and cannot be detected automatically. Therefore, our solution allows providers to specify such conflicts in the recipe, and we built the adaptation coordinator that can detect and resolve conflicts at run time using a first-come, first-serve approach. In this thesis, we focused our attention on "local adaptation" that has a limited scope in terms of the effects of possible adaptation actions.

6.2 Discussion

In this section, we discuss how various factors may affect the effectiveness of our recipe-based approach.

- **Request-response vs. session-oriented**: As mentioned earlier, we target session-oriented services where each user request is for a user session that will last for a relatively long time, e.g., on the order of minutes or hours. Because of this property, we build our architecture to support more heavy-weight self-configuration mechanisms such as global optimization of component selection that could require seconds to complete, and we also develop mechanisms to support run-time adaptation. However, for a request-response service where each user request is for an "answer", i.e., a response, from the service, our architecture may be overkill. In particular, since a user "session" in a request-response service typically lasts a short time, e.g., on the order of milliseconds or seconds, any self-configuration mechanisms that take seconds to complete are not practical. Furthermore, because of the short duration, run-time adaptation mechanisms become unnecessary.

- **Network sensitivity**: One important feature of our architecture is the ability to optimize the composed service configuration based on network performance metrics. In fact, handling "network-sensitive" services incurs additional complexity that arises from the fact that most network metrics involve multiple components, i.e., component selection based on network metrics is likely a global optimization problem. However, the extra complexity may not be necessary for all services. For example, if the bottleneck of a computation-intensive service is the computation or storage capacity of the individual components, the provider of the service may be more concerned about finding the servers with the most CPU cycles or storage space than optimizing the latency or bandwidth between the servers. For such services, it may be sufficient (or even more efficient) to use a traditional resource management framework that provides the necessary functionality for dynamically allocating resources, matching available resources with requests, and so on.

- **Point-to-point vs. multi-point**: As mentioned in earlier chapters, many previous self-configuration solutions adopt a point-to-point model (path model) where the output of a sender is adapted to the input of a receiver. In this model, the complexity of finding a service configuration is low since it is limited to finding a series of components between two end points, and most of these solutions address small-scale problems, e.g., only a few hops on the path and only a small number of choices for each hop. Therefore, such a model lends itself particularly well to the use of the generic type-based self-configuration approach. In contrast, as discussed earlier, our architecture aims to support more general multi-point service configurations where the complexity of finding service configurations may be high. We address the complexity problem by allowing providers to specify the procedure of constructing the configuration. This enables more sophisticated self-configuring services. However, it cannot make use of components or construct a configuration not known by the provider at design time, which is an advantage of the generic type-based approach. Therefore, when a service has a relatively simple configuration and frequently needs to make use of previously unknown components, the generic type-based approach

may be more appropriate.

- **Parameter-level vs. component-level**: Many previous studies have proposed run-time adaptation solutions that focus on parameter-level adaptation, i.e., adaptation is performed by adjusting the run-time parameters of a system. In such cases, the adaptation strategies for a service can often be encompassed in a single utility function that guides the dynamic selection of the parameters. Such an approach also avoids the potentially complex problem of adaptation coordination. In contrast, our goal is to support both parameter-level actions and component-level operations such as adding and replacing a component. Therefore, it is infeasible to capture the adaptation strategies as a utility function. Consequently, our architecture needs to handle the issues related to strategy specification and the coordination of different strategies. If a service only requires parameter-level adaptation, or it only involves simple component selection problem that can be elegantly captured in a utility function, then using an existing parameter-level adaptation framework may be sufficient.

6.3 Future work

In this section, we identify several future research directions.

- **Hierarchical self-configuration**: In the current design of the recipe-based self-configuration architecture, the synthesizer is responsible for putting together all components needed by a service instance. However, another possible approach is to perform self-configuration in a hierarchical fashion, i.e., a component required by the synthesizer can in fact be an instance of a different service configured by another synthesizer. For instance, in the video conferencing service example, the synthesizer can find an ESM component instead of three individual ESM proxies. In this case, the ESM component will actually be a service instance composed by another synthesizer using an ESM recipe. One advantage of such a hierarchical approach is that it can reduce the complexity of the physical mapping problem for a synthesizer since it now needs to handle a smaller number of components. Since the self-configuration tasks are distributed among a hierarchy of synthesizers, the scalability should also improve. However, this hierarchical approach also raises some interesting issues. For example, in the current "flat" approach, the synthesizer uses a global objective function for component selection. If the hierarchical approach is used, how does the top-level synthesizer "partition" the global objective into sub-objectives for lower-level synthesizers? In addition, the lower-level synthesizers have their own objectives from the recipes they are using. How does the top-level synthesizer's sub-objectives interact with the lower-level objectives? These issues need to be addressed before the hierarchical approach can be realized.

- **Resource management for the synthesizer**: To perform self-configuration, the synthesizer needs to acquire certain "resources". For example, after locating the neces-

sary components for a user request, the synthesizer needs to "reserve" those components, or if the components are software modules, the synthesizer needs to find and reserve generic computation servers to run those modules. Resource reservation is necessary since the resources (e.g., components and servers) only have limited capacity. In other words, if the synthesizer uses a component in the configuration of a particular user request, then the component may not be available to any other requests. Therefore, the resource management problem here is how the synthesizer allocate the limited resources among the user requests it is handling? There are higher-level resource management issues as well. For example, if a synthesizer is used by two service providers (e.g., video conferencing and interactive search), it needs to perform self-configuration for two different services using two different recipes. How does the synthesizer allocate resources among them? Furthermore, when there are multiple synthesizers competing for the same pool of resources, how should the resource management issues be resolved? Such issues become even more interesting when the hierarchical approach described above is used.

- **More general adaptation support**: The run-time adaptation support in the current architecture has a limited scope. One area that requires further study is how factors unrelated to performance should be considered in adaptation. For example, some researchers have considered "user distraction" as a metric that should be minimized when performing adaptation. How can we factor distraction into the current adaptation mechanisms? Another area for further study is how to handle the full spectrum (from global to local) of adaptation. The current architecture focuses on the two ends of the spectrum, global configuration and local adaptation. Handling adaptations that can be anywhere in between will require additional mechanisms, for example, adaptation strategies rendered useless by a semi-global adaptation need to be removed. In the area of adaptation coordination, many issues have yet to be explored. For example, how to detect indirect conflicts between adaptations? This will require a model for the propagation of the effects of adaptations. If the epoch-based approach for conflict detection and resolution is to be used, what heuristics and mechanisms are needed to make it work for our purpose? Can we add "preemption" to the first-come, first-serve approach, i.e., can we make adaptations "preemptable"? This can address one major problem of the FCFS approach, i.e., a higher-priority adaptation can be blocked by a lower-priority one started earlier.

- **Integration with existing service frameworks**: There are many research efforts that are complementary to our recipe-based self-configuration approach. Integrating them with our architecture will allow it to provide richer functionality. For example, as discussed earlier, the generic type-based approach for self-configuration has the drawback that it cannot make use of service-specific knowledge. However, it has an advantage over our approach: since the components are put together based on their input/output types, new types of components that are unknown at design time can still be used. Therefore, one interesting direction for future work is how to enhance

our approach with type-based composition? Can the synthesizer switch to type-based composition when no feasible configurations can be found using the recipe? As another example, the current architecture focuses on using specific components, e.g., video transcoder and conferencing gateway, in a service configuration, and we design mechanisms to handle the discovery, optimization, and other issues for such components. Although generic resources such as computation server and storage server can be handled using the same mechanisms, it may not be very efficient. For example, properties of such generic resources tend to change very frequently (e.g., available memory, CPU utilization, etc.). There has been many previous studies that specifically address the discovery and optimization issues for such generic resources, for example, languages for specifying resource requirements, mechanisms for matching resource requirements to resource providers, and so on. These efforts can be leveraged to provide better support for generic resources. Other self-configuration and/or adaptation frameworks are also complementary to our efforts, for example, parameter-level self-configuration, utility-based resource allocation, tactic-based remote execution, and so on. How to integrate these efforts into our recipe-based architecture is another area for further research.

Bibliography

[1] Active Network Backbone (ABone). http://www.isi.edu/abone/.

[2] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 186–201, December 1999.

[3] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. An Efficient K-Means Clustering Algorithm. In *Proceedings of the First IPPS/SPDP Workshop on High Performance Data Mining*, March 1998.

[4] Matthew Andrews, Bruce Shepherd, Aravind Srinivasan, Peter Winkler, and Francis Zane. Clustering and Server Selection using Passive Monitoring. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[5] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services Version 1.1. http://www-128.ibm.com/developerworks/library/ws-bpel/.

[6] Farooq Anjum, Francesco Caruso, Ravi Jain, Paolo Missier, and Adalberto Zordan. CitiTime: a system for rapid creation of portable next-generation telephony services. *Computer Networks*, 35(5), April 2001.

[7] Rajesh Krishna Balan, Mahadev Satyanarayanan, SoYoung Park, and Tadashi Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, pages 273–286, May 2003.

[8] Egon Balas and Joseph B. Mazzola. Nonlinear 0-1 programming: I. Linearization techniques. *Mathematical Programming*, 30:1–21, 1984.

[9] Battle.net. http://www.battle.net/.

[10] João W. Cangussu, Kendra Cooper, and Changcheng Li. A Control Theory Based Framework for Dynamic Adaptable Systems. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 1546–1553, March 2004.

[11] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. A Micro-Economic Approach to Conflict Resolution in Mobile Computing. In *Proceedings of the 10th ACM SIGSOFT Foundations of Software Engineering Conference (FSE-10)*, pages 31–40, November 2002.

[12] Robert L. Carter and Mark E. Crovella. Server Selection Using Dynamic Path Characterization in Wide-Area Networks. In *Proceedings of IEEE INFOCOM '97*, April 1997.

[13] CORBA Component Model, v3.0. http://www.omg.org/technology/documents/formal/components.htm.

[14] Prashant Chandra, Yang-hua Chu, Allan Fisher, Jun Gao, Corey Kosak, T.S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Customizable Resource Management for Value-Added Network Services. *IEEE Network*, 15(1), January 2001.

[15] Prashant Chandra, Allan Fisher, Corey Kosak, and Peter Steenkiste. Network Support for Application-Oriented Quality of Service. In *Proceedings of the Sixth IEEE/IFIP International Workshop on Quality of Service (IWQoS '98)*, May 1998.

[16] Fangzhe Chang and Vijay Karamcheti. Automatic Configuration and Run-time Adaptation of Distributed Applications. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, pages 11–20, August 2000.

[17] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. Resource Management in Legion. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99)*, April 1999.

[18] Moses Charikar, Sudipto Guha, Éva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k-median problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC 1999)*, pages 1–10, May 1999.

[19] Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, and Peter Steenkiste. Using Architectural Style as a Basis for System Self-repair. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA 2002)*, pages 45–59, August 2002.

[20] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. An Architecture for Coordinating Multiple Self-Management Modules. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 243–254, June 2004.

[21] Sumi Choi, Jonathan Turner, and Tilman Wolf. Configuring Sessions in Programmable Networks. In *Proceedings of IEEE INFOCOM 2001*, April 2001.

[22] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR '00)*, pages 121–132, April 2000.

[23] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A Case for End System Multicast. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, June 2000.

[24] COM: Component Object Model Technologies. http://www.microsoft.com/com/.

[25] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

[26] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, March 1998.

[27] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of the ACM Annual International Conference on Mobile Computing and Networking (MobiCom '99)*, August 1999.

[28] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and T. Tonouchi. Tools for Domain-based Policy Management of Distributed Systems. In *IEEE/IFIP Network Operations and Management Symposium (NOMS2002)*, pages 213–218, April 2002.

[29] Mark S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*, chapter 6, pages 198–238. John Wiley & Sons, Inc., 1995.

[30] Mark S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*. John Wiley & Sons, Inc., 1995.

[31] Mark S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*, chapter 1.4, pages 10–18. John Wiley & Sons, Inc., 1995.

[32] DCOM Technical Overview. http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomtec.asp.

[33] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *USENIX Symposium on Internet Technologies and Systems (USITS 2001)*, March 2001.

[34] Ron Doyle, Jeff Chase, Omer Asad, Wei Jin, and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[35] Fabrice Dupuy, Gunnar Nilsson, and Yuji Inoue. The TINA Consortium: Toward Networking Telecommunications Information Services. *IEEE Communications Magazine*, 33(11), November 1995.

[36] Sandra G. Dykes, Clinton L. Jeffery, and Kay A. Robbins. An Empirical Evaluation of Client-side Server Selection Algorithms. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[37] Christos Efstratiou, Adrian Friday, Nigel Davies, and Keith Cheverst. Utilising the Event Calculus for Policy Driven Adaptation on Mobile Systems. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, June 2002.

[38] Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/.

[39] The Emulab testbed. http://www.emulab.net/.

[40] Brian Ensink and Vikram Adve. Coordinating Adaptations in Distributed Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[41] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 365–376, August 1997.

[42] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[43] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6), June 2002.

[44] Ian Foster, Alain Roy, and Volker Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8th International Workshop on Quality of Service (IWQoS 2000)*, June 2000.

[45] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 78–91, October 1997.

[46] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Transactions on Networking*, 9(5):525–540, October 2001.

[47] Friends – The Official Site. http://friends.warnerbros.com/.

[48] Xiaodong Fu, Weisong Shia, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March 2001.

[49] Krzysztof Gajos. Rascal - a Resource Manager For Multi Agent Systems In Smart Spaces. In *Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS 2001)*, pages 111–120, September 2001.

[50] Alan G. Ganek and Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[51] Jun Gao and Peter Steenkiste. Rendezvous Points-Based Scalable Content Discovery with Load Balancing. In *Proceedings of the Fourth International Workshop on Networked Group Communication (NGC '02)*, pages 71–78, October 2002.

[52] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10), October 2004.

[53] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[54] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, April 2002.

[55] Constant Gbaguidi, Jean-Pierre Hubaux, Giovanni Pacifici, and Asser N. Tantawi. Integration of Internet and Telecommunications: An Architecture for Hybrid Services. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[56] Google. http://www.google.com/.

[57] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *IEEE Computer Networks, Special Issue on Pervasive Computing*, 35(4), March 2001.

[58] Mark Gritter and David R. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March 2001.

[59] Thomas Gross, Bruce Lowekamp, Roger Karrer, Nancy Miller, and Peter Steenkiste. Design, Implementation, and Evaluation of the Remos Network Monitoring System. *Journal of Grid Computing*, 1(1), May 2003.

[60] Xiaohui Gu and Klara Nahrstedt. A Scalable QoS-Aware Service Aggregation Model for Peer-to-Peer Computing Grids. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[61] Erik Guttman, Charles Perkins, John Veizades, and Michael Day. Service Location Protocol, Version 2. RFC 2608, June 1999.

[62] Erik Guttman, Charles E. Perkins, and James Kempf. Service Templates and Service: Schemes. RFC 2609, June 1999.

[63] James D. Guyton and Michael F. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proceedings of ACM SIGCOMM '95*, pages 288–298, August 1995.

[64] The Official Half-Life Web Site. http://half-life.sierra.com/.

[65] Tim Howes. The String Representation of LDAP Search Filters. RFC 2254, December 1997.

[66] Juraj Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*, chapter 6.2, pages 389–400. Springer, 2001.

[67] Jia-Cheng Hu and Jiann-Min Ho. A Conference Gateway Supporting Interoperability Between SIP and H.323 Clients. Master's thesis, Carnegie Mellon University, March 2000.

[68] An-Cheng Huang and Peter Steenkiste. Distributed Load-Sensitive Routing for Computationally-Constrained Flows. In *Proceedings of the 38th IEEE International Conference on Communications (ICC 2003)*, May 2003.

[69] An-Cheng Huang and Peter Steenkiste. Network-Sensitive Service Discovery. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, pages 239–252, March 2003.

[70] An-Cheng Huang and Peter Steenkiste. Network-Sensitive Service Discovery. *Journal of Grid Computing*, 1(3):309–326, 2003.

[71] An-Cheng Huang and Peter Steenkiste. Building Self-configuring Services Using Service-specific Knowledge. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 45–54, June 2004.

[72] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, March 2004.

[73] Internet Domain Survey, Internet Systems Consortium, July 2004. http://www.isc.org/.

[74] ITU-T Recommendation H.323. Packet-based Multimedia Communications Systems, November 2000.

[75] Anca-Andreea Ivan, Josh Harman, Michael Allen, and Vijay Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[76] H. V. Jagadish, Alberto O. Mendelzon, and Inderpal Singh Mumick. Managing Conflicts between Rules. In *Proceedings of the 15th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems (PODS '96)*, pages 192–201, June 1996.

[77] JavaBeans. http://java.sun.com/products/javabeans/.

[78] Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/.

[79] Jini Network Technology. http://wwws.sun.com/software/jini/.

[80] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 156–171, December 1995.

[81] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[82] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[83] Balachander Krishnamurthy and Jia Wang. On Network-Aware Clustering of Web Clients. In *Proceedings of ACM SIGCOMM 2000*, August 2000.

[84] Aurel A. Lazar, Shailendra K. Bhonsle, and Koon Seng Lim. A Binding Architecture for Multimedia Networks. *Journal of Parallel and Distributed Systems*, 30(2):204–216, November 1995.

[85] Chen Lee, John Lehoczky, Dan Siewiorek, Ragunathan Rajkumar, and Jeff Hansen. A Scalable Solution to the Multi-Resource QoS Problem. Technical Report CMU-CS-99-144, Carnegie Mellon University, May 1999.

[86] The XML C parser and toolkit of Gnome. http://www.xmlsoft.org/.

[87] Chuang Liu, Lingyun Yang, Ian Foster, and Dave Angulo. Design and Evaluation of a Resource Selection Framework for Grid Applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[88] Julio López and David O'Hallaron. Evaluation of a resource selection mechanism for complex network services. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing*, August 2001.

[89] Joseph P. Loyall, Richard E. Schantz, John A. Zinky, and David E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Proceedings of the First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, April 1998.

[90] Z. Morley Mao and Randy H. Katz. Achieving Service Portability in ICEBERG. In *Proceedings of IEEE GlobeCom 2000, Workshop on Service Portability (SerP-2000)*, March 2000.

[91] Microsoft .NET Technical Resources. http://www.microsoft.com/net/technical/.

[92] Naftaly H. Minsky. On Conditions for Self-Healing in Distributed Software Systems. In *Proceedings of the Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services (AMS 2003)*, pages 86–92, June 2003.

[93] Jonathan D. Moffett and Morris S. Sloman. Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications*, 11(9):1404–1414, December 1993.

[94] Napster.com. http://www.napster.com/.

[95] Anand Natrajan, Marty Humphrey, and Andrew Grimshaw. Capacity and Capability Computing using Legion. In *Proceedings of the 2001 International Conference on Computational Sciences*, May 2001.

[96] Microsoft Windows NetMeeting. http://www.microsoft.com/windows/netmeeting/.

[97] T. S. Eugene Ng and Hui Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[98] Global Internet Index: Average Usage, Nielsen//NetRatings, September 2004. http://www.nielsen-netratings.com/.

[99] Active Measurement Project (AMP), National Laboratory for Applied Network Research. http://watt.nlanr.net/.

[100] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[101] Object Management Group. Common Object Request Broker Architecture: Core Specification, Version 3.0.3, March 2004. http://www.omg.org/technology/corba/corba3releaseinfo.htm.

[102] OpenSLP Home Page. http://www.openslp.org/.

[103] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 177–186, April 1998.

[104] Vivek Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October 1998.

[105] Craig Partridge, Trevor Mendez, and Walter Milliken. Host Anycasting Service. RFC 1546, November 1993.

[106] PlanetLab Home Page. http://www.planet-lab.org/.

[107] Christian Poellabauer, Karsten Schwan, Sandip Agarwala, Ada Gavrilovska, and Greg Eisenhauer. Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems. In *Proceedings of the Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services (AMS'03)*, June 2003.

[108] Vahe Poladian, João Pedro Sousa, David Garlan, and Mary Shaw. Dynamic Configuration of Resource-Aware Services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 604–613, May 2004.

[109] Shankar R. Ponnekanti and Armando Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proceedings of the Eleventh World Wide Web Conference (WWW 2002), Web Engineering Track*, May 2002.

[110] Ravi Ramamoorthi and Amitabh Sinha. Integrated Logistics: Approximation Algorithms Combining Facility Location and Network Design. In *Proceedings of the 9th International Conference on Integer Programming and Combinatorial Optimization (IPCO 2002)*, pages 212–229, May 2002.

[111] Bhaskaran Raman, Sharad Agarwal, Yan Chen, Matthew Caesar, Weidong Cui, Per Johansson, Kevin Lai, Tal Lavian, Sridhar Machiraju, Z. Morley Mao, George Porter, Timothy Roscoe, Mukund Seshadri, Jimmy Shih, Keith Sklower, Lakshminarayanan Subramanian, Takashi Suzuki, Shelley Zhuang, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. The SAHARA Model for Service Composition Across Multiple Providers. In *Proceedings of the First International Conference on Pervasive Computing (Pervasive 2002)*, August 2002.

[112] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, July 1998.

[113] Rajesh Raman, Miron Livny, and Marvin Solomon. . In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[114] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[115] Peter Reiher, Richard Guy, Mark Yarvis, and Alexey Rudenko. Automated Planning for Open Architectures. In *Proceedings of the Third IEEE Conference on Open Architectures and Network Programming (OPENARCH 2000) – Short Paper Session*, pages 17–20, March 2000.

[116] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proceedings of the First USENIX Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[117] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM 2001*, April 2001.

[118] David B. Shmoys, Éva Tardos, and Karen Aardal. Approximation algorithms for facility location problems. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC 1997)*, pages 265–274, May 1997.

[119] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[120] João Pedro Sousa and David Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA 2002)*, pages 29–43, August 2002.

[121] Bridget Spitznagel and David Garlan. A Compositional Approach for Constructing Connectors. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, August 2001.

[122] Peter Steenkiste, Prashant Chandra, Jun Gao, and Umair Shah. An Active Networking Approach to Service Customization. In *Proceedings of DARPA Active Networks Conference and Exposition (DANCE'02)*, pages 305–318, May 2002.

[123] Burkhard Stiller, Christina Class, Marcel Waldvogel, Germano Caronni, and Daniel Bauer. A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience. *IEEE Journal on Selected Areas in Communication, Special Issue on Middleware*, 17(9), September 1999.

[124] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001*, August 2001.

[125] UDDI Version 3.0.1. http://uddi.org/pubs/uddi_v3.htm.

[126] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99)*, October 1999.

[127] Valve Corporation. http://www.valvesoftware.com/.

[128] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility Functions in Autonomic Systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, pages 70–77, May 2004.

[129] H. J. Wang, B. Raman, C-N. Chuah, R. Biswas, R. Gummadi, B. Hohlt, X. Hong, E. Kiciman, Z. Mao, J. S. Shih, L. Subramanian, B. Y. Zhao, A. D. Joseph, and R. H. Katz. ICEBERG: An Internet-core Network Architecture for Integrated Communications. *IEEE Personal Communications Special Issue on IP-based Mobile Telecommunications Networks*, August 2000.

[130] Zhenyu Wang and David Garlan. Task-Driven Computing. Technical Report CMU-CS-00-154, Carnegie Mellon University School of Computer Science, May 2000.

[131] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proceedings of the 8th European Software Engineering Conference/the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2001)*, pages 21–32, September 2001.

[132] David Wolpert, Kevin Wheeler, and Kagan Tumer. Collective Intelligence for Control of Distributed Dynamical Systems. *Europhysics Letters*, 49(6):708–714, March 2000.

[133] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[134] Dongyan Xu, Klara Nahrstedt, and Duangdao Wichadakul. QoS-Aware Discovery of Wide-Area Distributed Services. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, May 2001.

[135] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Tom Anderson, and David Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 105–118, January 1997.

[136] Ellen W. Zegura, Mostafa H. Ammar, Zongming Fei, and Samrat Bhattacharjee. Application-Layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service. *IEEE/ACM Transactions on Networking*, 8(4):455–466, August 2000.