Partitionable Services Framework: Seamless Access to Distributed Applications

by

Anca-Andreea Ivan

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2004

Approved: _____

Research Advisor: Vijay Karamcheti

Pentru Mau

Acknowledgment

TBD

Abstract

A key problem in contemporary distributed systems is how to satisfy user quality of service (QoS) requirements for distributed applications deployed in heterogeneous, dynamically changing environments spanning multiple administrative domains.

An attractive solution is to create an infrastructure which satisfies user QoS requirements by automatically and transparently adapting distributed applications to any environment changes with minimum user input. However, successful use of this approach requires overcoming three challenges: (1) Capturing the application behavior and its relationship with the environment as a set of compact local specifications, using both general, quantitative (e.g., CPU usage) and qualitative (e.g., security) properties. Such information should be sufficient to reason about the global behavior of the application deployment. (2) Finding the "best" application deployment that satisfies both application and user requirements, and the various domain policies. The search algorithm should be complete, efficient, scalable with regard to application and network sizes, and guarantee optimality (e.g., resources consumed by applications). (3) Ensuring that the found deployments are practical and efficient, i.e., that the efficiency of automatic deployments is comparable with the efficiency of hand-tuned solutions.

This dissertation describes three techniques that address these challenges in the

context of component-based applications. The modularity and reusability of the latter enable automatic deployments while supporting reasoning about the global connectivity based on the local information exposed by each component. The first technique extends the basic component-based application model with information about conditions and effects of component deployments and linkages, together with interactions between components and the network. The second technique uses AI planning to build an efficient and scalable algorithm which exploits the expressivity of the application model to find an application deployment that satisfies user QoS and application requirements. The last technique ensures that application deployments are both practical and efficient, by leveraging language and run-time system support to automatically customize components, as appropriate for the desired security and data consistency guarantees. These techniques are implemented as integral parts of the Partitionable Services Framework (PSF), a Java-based framework which flexibly assembles component-based applications to suit the properties of their environment. PSF facilitates on-demand, transparent migration and replication of application components at locations closer to clients, while retaining the illusion of a monolithic application.

The benefits of PSF are evaluated by deploying representative component-based applications in an environment simulating fast and secure domains connected by slow and insecure links. Analysis of the programming and the deployment processes shows that: (1) the code modifications required by PSF are minimal, (2) PSF appropriately adapts the deployments based on the network state and user QoS requirements, (3) the run-time deployment overheads incurred by PSF are negligible compared to the application lifetime, and (4) the efficiency of PSF-deployed applications matches that of hand-crafted solutions.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

1.1 Motivation

Over the last few years, the Internet has evolved from a distributed data repository to an underlying infrastructure running complex services. An increasingly large number of clients use these services to perform sophisticated actions (e.g., accessing e-mail, executing sensitive bank transactions, playing music, or watching movies), while expecting that the accessed services satisfy required quality expectations. For example, e-mail and banking applications should guarantee that transactions are efficient and secure when crossing insecure environments. Similarly, users playing music or video over the Internet expect that they will receive clear sounds/images in real time.

Unfortunately, satisfying clients' expectations is a hard problem because the Internet is a highly heterogeneous environment. Sources of heterogeneity include *various devices* ranging from super-computers and PCs to hand-held devices, running *diverse software* in terms of both operating software and middleware, and being connected by

Table 1.1: Maximum available bandwidth of different types of links

Link type	**Max. bandwidth**	**Link type**	**Max. bandwidth**
56K modem	56Kbps	T-1	1.544 Mbps
Frame relay	56 Kbps-1.544 Mbps	E-1 (Europe)	2.048 Mbps
WiFi	11Mbps	T-3 (or DS3)	44.736 Mbps
Ethernet	100-1000 Mbps	E-3 (Europe)	34.368 Mbps
ADSL	1.5-8.2 Mbps down-stream, 1 Mbps upstream	OC-3	155.52 Mbps
		OC-12	622.08 Mbps
SDSL	1.544/2.048 Mbps	OC-48	2.488 Gbps

Table 1.2: Maximum available CPU and memory for different types of nodes – 2004

System	**CPU**	**Memory**
Sun Fire E25k	UltraSPARC III 1.2-GHz	$<$ 576 GB
Dell Dimension 8300	Intel Pentium IV 3.4GHz	$<$ 2 GB
Compaq Presario R3140US	AMD Athlon 64 3GHz	512 MB
HP iPAQ h1945 Pocket PC	Samsung 266MHz	64MB RAM
Palm Tungsten T3	400MHz Intel XScale	64MB RAM

links with *different properties* that differ in available bandwidth, latency, and security. Tables 1.1, 1.2, and 1.3 are simple illustrations of the diversity of the Internet. Table 1.1 shows the maximum available bandwidth of various types of links. Table 1.2 enumerates the characteristics of a small subset of devices, ranging from small Pocket PCs to Sun Fire servers. Table 1.3 lists some possible link latencies, measured from a host in New York University to hosts in other universities.

Table 1.3: RTT from New York University, measured in March 2004.

Site	Min (ms)	Avg (ms)	Max (ms)	Dev (ms)
New York University, US	0.657	1.022	1.895	0.316
Carnegie Mellon University, US	15.501	15.795	16.694	0.322
UC Berkeley, US	89.932	90.167	90.414	0.268
Stanford University, US	91.266	91.906	92.414	0.353
Georgia Insitute of Technology, US	38.353	38.796	39.731	0.344
Massachusetts Institute of Technology, US	7.208	7.948	11.112	0.761
University of Michigan, US	37.624	37.893	38.285	0.312
Academy Economical Studies, RO	137.846	139.264	141.894	0.878
Delft University of Technology, NL	90.358	90.590	91.246	0.398
University of Barcelona, SP	130.791	131.132	131.617	0.343
INRIA, FR	96.242	102.964	135.845	10.280
Tsinghua University, CN	288.429	325.047	348.602	13.931
Panjab University, IN	347.607	380.531	431.460	12.579
UTN National Technological Univ., AR	194.931	256.685	532.755	96.925

In addition, the resource availability in most real-world computing environments changes dynamically. This happens because applications sharing resources are continuously changing their workloads and preferences. For example, the available bandwidth of a link can increase or decrease depending on what other applications are running in the network (e.g., bandwidth-consuming applications such as Kazaa [81]). Similarly, the availability of software on nodes changes as a result of software updates or, more catastrophically, node failures.

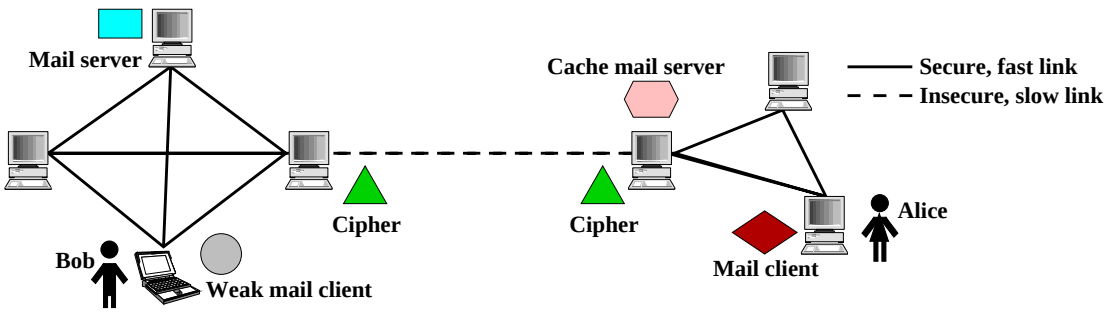


Figure 1.1: Example of using a distributed application in a heterogeneous environment.

Often, the network heterogeneity and the continuous changes prevent applications from providing the expected quality. In principle, applications could be written such that they are aware of their environment and adjust their behavior accordingly. However, such applications need to take several factors into account.

Figure 1.1 illustrates a situation when a simple e-mail application should take into consideration the state of the environment in order to provide the required service. Let's assume that there are two domains connected by slow and insecure links, while the intra-domain links are both secure and fast. The problem is allowing two users, Alice and Bob, to securely and efficiently access a mail server. The static structure of a classic client-server architecture cannot satisfy both Alice's and Bob's requirements. A simple connection from the client to the mail server satisfies Bob's requirements because Bob connects to the mail server from within the same domain. However, Alice accesses the same mail server from another domain, through an insecure and slow link. Thus, a direct connection to the mail server will not satisfy Alice's QoS requirements. A possible approach is to add cryptographic functions to the original client and server code. In this case, Alice's security requirements are satisfied, while

Bob ends up paying the cost of executing cryptographic functions even if unnecessary. The natural solution to this second problem is to design the mail application as a component-based application from components that implement basic functionality (e.g., mail server, mail client, cache mail server, cryptographic modules). The main advantage of such structured applications is that various component compositions can be created to suit the environment characteristics. For example, the cache can be deployed to offset the high latency of a link, while the cipher modules running at each end of insecure links can guarantee message privacy and correctness. Further benefits can be gained if the application structure is dynamically built, as this allows the application to automatically adapt its structure to changes in the network state.

1.2 Different adaptation approaches

As the above example shows, it is desirable for applications to automatically adapt their behavior depending on the user QoS requirements and the current state of the environment. Depending on the implemented adaptation technique, current systems can be divided into three classes: (1) *network level* systems that satisfy QoS requirements by controlling data flows and their usage of network resources, (2) *application level* systems, where the applications are responsible for taking all adaptation decisions, and (3) *middleware level* systems that provide a layer capable of taking and executing all adaptation decisions. Each of these classes is discussed in the following paragraphs.

Network level solutions. The Internet architecture, as described in the IP network protocol, provides only point-to-point best-effort services [47]. However, an increasing number of applications developed in the recent years have QoS requirements that cannot be satisfied by the current Internet model. Examples of such applications are video conferencing and remote video, where multiple clients connecting to multiple servers are very sensitive to the provided QoS. In order to accommodate such applications, new techniques were developed to (1) reserve resources and control the load admitted into the network (e.g., Integrated Services (IS) [9] and Differentiated Services (DS) [5]) or (2) adjust the behavior of the data flowing through the transport layer (e.g., active networks [63]).

In the Integrated Services architecture, applications must first set up paths and reserve resources. RSVP [98] is an example of a signaling mechanism used to perform these operations. Unfortunately, the IS architecture requires applications to perform extra steps before starting, thus becoming aware of the low-level protocol. The Differentiated Services protocol does not require the application to reserve in advance the necessary resources. Instead, DS marks packets with tags and divides them into several classes. Depending on the class they belong to, packets receive different services. There are three main disadvantages of these low-level approaches: (1) they can satisfy only a limited set of QoS requirements (e.g., available bandwidth, delay), compared to more general sets of QoS required by applications (e.g., security, frame rate, image quality), (2) they require a drastic change of the existing Internet infrastructure, and (3) because of their limited suite of mechanisms, i.e., routing packets, application QoS metrics are not always satisfied.

The second class of techniques – *active networks* – partially solves the last prob-

lem by performing rich transformations on data packets; these transformations depend on the current state of the link and the nature of the application. Examples of packet forwarding using active networks include Transforming Tunnels [82], Protocol Boosters [20], and RON [2]. All these systems allow applications to specify both application-dependent and -independent transformations to be executed on all the packets crossing a set of links. Transforming Tunnels enable applications to define tunnels by deploying transforming functions at both ends of a tunnel. However, they need to be configured by users, thus requiring comprehensive knowledge of the underlying network. Protocol Boosters allow more flexibility than Transforming Tunnels by allowing users to inject any number of entities in the protocol graph, instead of pairs of transformations at the level of links. Unfortunately, both techniques require the modification of existing infrastructures. RON is an example of an overlay network built on top of the existing Internet infrastructure, where each overlay node monitors the quality of Internet paths to other overlay nodes. RON can be used by distributed applications to recover from path failure and improve their communication performance. The common shortcoming of all these approaches is the limited use of application-specific information to control the data-packet adaptations.

Application level solutions. Traditionally, classic client-server applications were developed to provide only the required minimum functionality. Thus, they were unable to satisfy users' requirements when deployed and accessed under various conditions. For example, a banking application with no security guarantees could not be used over an insecure network. One solution is to add the extra functionality to the existing application. For example, the privacy of all messages involved in a banking transac-

tion could be protected by adding cryptographic functionality to both the client and the server code.

Systems designed to support such applications (e.g., EPIQ [29], ErDos [28], Active Harmony [44], Odyssey [70], Application Tunability framework [15], Rover [52], QuO [87]) assume that the application structure is more or less fixed. Such frameworks provide a set of basic services that transmit to the application the information necessary to make adaptation decisions. For example, as part of the framework, a network monitoring system can send alerts to the application every time the available bandwidth drops below a given threshold. Adaptation is achieved by the application, by altering the internal behavior of one or more of the components (e.g., changing an internal algorithm). The advantage is that the adaptation is fine-grained because it uses information about the application internals. The main disadvantages are the fixed application structure and the large effort required from programmers to design adaptation mechanisms that deal with all possible faults in the network.

Middleware level solutions. In order to alleviate these disadvantages, middleware infrastructures are built to provide a virtual machine layer capable of taking adaptation decisions. In this way, programmers can focus on the basic application functionality and let the middleware systems monitor the environment, trigger the application adaptation, and execute the adaptation steps. Some of the most successful middleware systems are component-based frameworks (e.g., Globus Grid [33], CORBA [71], CANS [40], Ninja [76]), where scalable distributed applications are constructed by integrating reusable component services spanning multiple administrative domains. Such component-based applications are defined as sets of components, where each

component exposes some minimal local information about its functionality and connectivity. The modularity and reusability of these applications enable automatic deployments while reasoning about the application structure based on the local information exposed by each component.

Grid frameworks like Globus Grid [33], or component frameworks like DCE [75], CORBA [71], and DCOM [88] provide infrastructural support to ease the construction of component-based applications, allowing services to register with a common substrate that provides basic services — discovery, resource management, security. Most such frameworks rely on static component linkages; thus, application adaptation is possible only by re-deploying the same components on various nodes and connecting them as specified by the static linkages. Unfortunately, this restricted type of adaptation is not always successful. In order to increase the chances of successful adaptation, a growing number of systems (e.g., Ninja [76], Active Frames [66], Eager Handlers [99], Active Streams [10], CANS [40], Conductor [64] and a recent version of Globus Grid [33]) advocate a more dynamic model, where components are combined at run-time, based on the current state of the environment and the client QoS requirements. This dynamic model enables applications to flexibly and dynamically adapt to changes in resource availability and client requests. As in the example scenario described before, low bandwidth can be masked by deploying a cache component close to clients. Similarly, security-aware applications can deploy cipher modules to protect sensitive data crossing insecure links through the use of encryption and signatures. Because of this flexibility, dynamic component-based frameworks are an attractive solution to the problem of satisfying user QoS requirements by automatically adapting distributed applications to the state of the environment.

However, a successful implementation of this solution requires addressing the following three challenges: (1) Capturing the application behavior and its relationship with the environment as a set of compact local specifications, using both general, quantitative (e.g., CPU usage) and qualitative (e.g., security) properties. Such information should be sufficient to reason about the global behavior of the application deployment. (2) Finding the "best" application deployment that satisfies both application and user requirements, and the various domain policies. The search algorithm should be complete, efficient, scalable with regard to application and network sizes, and guarantee optimality (e.g., resources consumed by applications). (3) Ensuring that the found deployments are practical and efficient, i.e., that the performances of automatic and hand-tuned deployments are comparable. Current systems only address a subset of these challenges, such as working only with standard QoS properties, dynamically creating only chains of components, or considering that all entities belong to only one administrative domain.

1.3 Thesis and methodology

This dissertation broadens the applicability of dynamic component-based frameworks by describing general, extensible solutions to the three challenges above. These solutions rely upon three techniques. The first technique extends the basic component-based application model with information about conditions and effects of component deployments and linkages, together with interactions between components and the network. The second technique uses AI planning to build an efficient and scalable algorithm which exploits the expressivity of the application model to find an application

deployment that satisfies user QoS and application requirements. The last technique ensures that application deployments are both practical and efficient by leveraging language and run-time system support to automatically customize components, as appropriate for the desired security and data consistency guarantees.

These techniques are implemented as integral parts of the Partitionable Services Framework (PSF), a Java-based framework which flexibly assembles component-based applications to suit the properties of their environment. PSF facilitates on-demand transparent migration and replication of application components at locations closer to clients while still retaining the illusion of a monolithic application. The PSF run-time system is responsible for registering applications with the framework and serving incoming client requests. Whenever a client wants to access an application, the run-time system performs the necessary security checks (authentication and authorization), decides which level of service the client has the right to access, and asks a planning module to compute a valid component deployment. Once a valid deployment is found, the PSF run-time system installs, instantiates, connects, and executes the components on the appropriate nodes.

The benefits of PSF are evaluated by deploying representative component-based applications in an environment simulating fast and secure domains connected by slow and insecure links. The characteristics of the component-based applications used to test PSF include: (1) a set of components rich in both number and properties, which allows the creation of a large number of component deployments, and (2) requirements that change during the execution and directly impact the application efficiency. Analysis of the programming and the deployment processes shows that: (1) the code modifications required by PSF are minimal, (2) PSF appropriately adapts the deploy-

ments based on the network state and user QoS requirements, (3) the run-time deployment overheads incurred by PSF are negligible compared to the application lifetime, and (4) the efficiency of PSF-deployed applications matches the efficiency of hand-crafted solutions.

1.4 Contributions

The high-level contribution of this thesis is an integrated set of techniques that allows automatic deployment of component-based distributed applications in heterogeneous environments. These techniques include: (1) defining suitable component and network *models*, (2) building a scalable *planner* which exploits the expressivity of the component model to efficiently find a valid plan, and (3) building a practical and efficient application *deployment* process.

Application models. The application model extends the basic model of component-based applications [72] by allowing user-specified arbitrary expressions to capture the conditions and the effects of component deployments and linkages, and the interactions between the deployed components and the network. These expressions are defined using component and interface properties as parameters. Unlike other models (CORBA, Web Services, OGSA) which use only standard pre-defined properties such as node CPU and link bandwidth, this model allows the specification of general, user- or application-determined, qualitative (e.g., privacy) and quantitative (e.g., frame rate, trust level) properties [48].

AI planning based algorithm. Given such an application model, it now becomes possible to automatically reason about which components should make up the application and where they should be located such that the user QoS requirements are satisfied (i.e., finding valid application configurations). A contribution of this thesis is an AI planning-based algorithm for solving this application configuration problem. The novel feature of this algorithm is that it combines and solves in one step both the component composition and the component mapping problems, instead of dealing with them separately. What differentiates this algorithm from similar algorithms is its ability to scale with the size of the network and its support for general resource functions and application structures [58].

Because the chances of finding a valid plan increase with the diversity of the component set, the initial set of application components can be enriched by dynamically creating custom components with a larger range of property values than the original components. The technique used to achieve this process is based on *views*, as explained in the following paragraph.

Practical and efficient deployment process. Given a deployment plan, additional challenges must be overcome to ensure that the plan can be deployed in a practical and efficient manner. Two challenges that this thesis focuses on are: (1) providing *security guarantees* in an environment spanning multiple administrative domains, and (2) reducing the cost of inter-component *data consistency* traffic.

The challenges of providing *security guarantees* in an environment spanning multiple administrative domains include finding a deployment that satisfies application-specific security guarantees and dealing with cross-domain authentication and autho-

rization of entities. Solutions must work in environments where there may not exist any third-party trusted by all domains to modulate their security relationships, and security decisions may need to be taken without domains exposing complete information to all parties. In such conditions, a secure application deployment requires not only that all entities are properly authenticated and authorized before being allowed to perform an action and that all communication is protected against eavesdroppers, but also that there exists a way to translate any local properties and constraints governing component deployment or resource usage in one domain into properties meaningful to other domains.

The cost of maintaining *data consistency* impacts application performance when multiple instances of the same component are simultaneously active in the wide-area network environment. The challenge is to minimize the data consistency traffic between components without making any assumptions that may not be valid across all component-based applications (i.e., these assumptions need to be application-neutral).

The key insight of the solution presented in this thesis is that frameworks can satisfy both application security and data consistency requirements by deploying only component *views*, instead of entire components. Views are customizations of original components; they allow the application developer to specify the appropriate access control granularity and capture the application information necessary in efficiently maintaining data consistency.

In order to support the dynamic deployment of views, this thesis proposes a runtime infrastructure built atop a *decentralized role-based access control and trust management system* and an application-neutral *data consistency protocol*.

The role-based trust management system provides an integrated solution to cross-

domain authentication, access control, and translation between local constraint specifications. The latter represents a novel use of a classic technique: properties are viewed as credentials belonging to a local domain and the translation between two properties is equivalent to finding a chain of credentials starting from the former and finishing with the latter.

The data consistency protocol maintains consistency between different views of the same component. The novel feature of the consistency protocol is that it satisfies the consistency requirements of component-based applications (*application-neutral*) deployed in various configurations (*flexible*), while using application-specific information embodied in the view specification. The data consistency traffic is minimized by allowing the application to specify (1) *data properties* to characterize the shared data, (2) *triggers* to indicate when updates need to be pushed or pulled between views, and (3) *merge/extract methods* to merge/extract updates from/into views and original components [50], all using an application-neutral protocol.

Additional contributions include implementing these techniques in the context of a Java-based component-based framework called the Partitionable Services Framework (PSF), and evaluating them using several representative component-based applications. PSF relies on several modules, such as DisCo [38] and Javassist [85]. DisCo is a middleware infrastructure that provides services such as: (1) a discovery mechanism that allows entities to discover running instances of PSF, (2) dRBAC [23], a decentralized role-based access control and trust management system that provides cross-domain authentication and authorization, and (3) Switchboard [24], a communication abstraction build on top of dRBAC to guarantee secure, and continuously

monitored communication channels. Javassist is a bytecode modifier tool that dynamically generates views based on Java bytecode and a set of rules.

1.5 Thesis organization

Chapter 2 describes the problems that need to be addressed in order to automatically deploy dynamically configured component-based applications in heterogeneous environments, and introduces two examples of component-based applications: a security sensitive web-based e-mail application and an airline reservation system. Chapter 3 introduces the necessary background and discusses solutions employed by current systems with similar goals. The three techniques making up the main contribution of this thesis are described in the following three chapters. The application and environment specification models are presented in Chapter 4. Chapters 5 and 6 describe the planning algorithm that creates the appropriate application configurations and the application deployment process. Chapter 7 introduces the Partitionable Services Framework and describes how the three techniques help to automatically and transparently deploy component-based applications in heterogeneous environments. Chapters 8, 9, and 10 highlight the qualitative and quantitative benefits of each technique, both considered in isolation and as part of PSF, by evaluating the performance of the applications deployed by the framework. This document ends by presenting conclusions and ideas for future work.

Chapter 2

Problem Description

This chapter describes the basic features common to dynamic component-based frameworks and highlights the challenges that must be overcome by such frameworks in order to satisfy the QoS requirements specified by clients. It also introduces two representative component-based applications that will be used throughout this document to explain and evaluate the three techniques introduced in Section 1.3.

2.1 Dynamic component-based frameworks

Dynamic component-based frameworks satisfy user QoS requirements by automatically deploying component-based applications into highly heterogeneous environments. Component-based applications are defined as sets of components, where each component exposes local information about its behavior and linkages. This information is used by frameworks to dynamically compute and deploy various component compositions, given the current state of the heterogeneous environment.

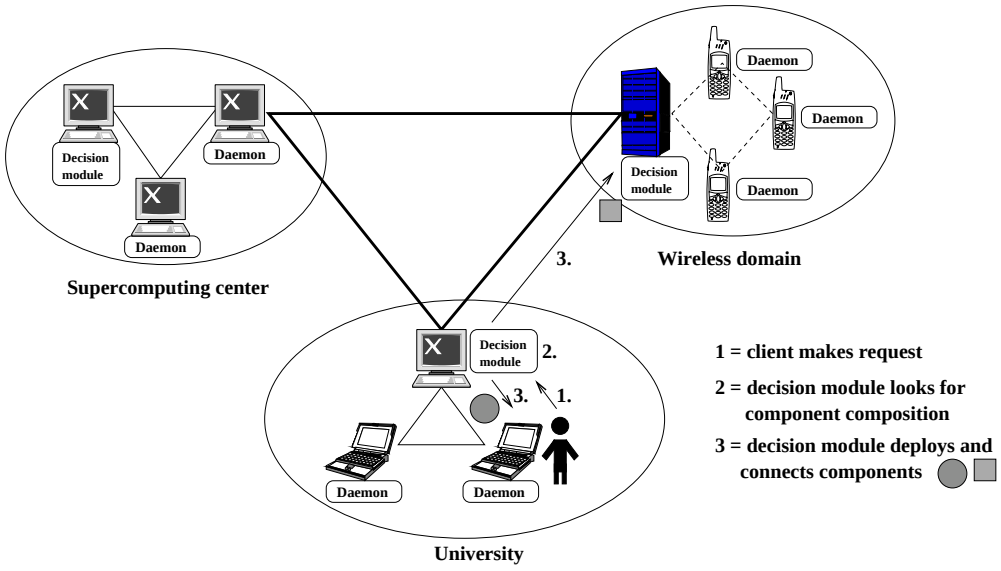


Figure 2.1: Basic architecture of a dynamic component-based framework.

General dynamic component-based frameworks rely on several modules that provide basic functionality – e.g., registering applications, monitoring the network and the application, computing the application configuration (i.e., component composition), and securely deploying the components. During the application registration step, the application provides to the framework all the information necessary to achieve the desired deployments. Examples of such information include the component functionality and linkages. Whenever a client makes a request to access the component-based application (Step 1 in Figure 2.1), the decision-making module searches for a valid component composition based on the information provided by the application, the QoS requirements specified by users, and the current state of the network (Step 2). A component composition is valid if it satisfies both the application and the user constraints. If such a composition is found, the framework is responsible

for securely deploying the components on the appropriate nodes and creating the necessary connections (Step 3). This step is based on the assumption that all nodes are aware of the framework by running a thin layer acting as a proxy between the node and the framework (e.g., the *daemon* in Figure 2.1).

In general, entities involved in this process (i.e., users, nodes, links, applications) may belong to different administrative domains. Satisfying users' QoS requirements in such conditions becomes even more challenging because each administrative domain can define its own policies. Examples of such policies include controlling the information exported outside about its entities and the access of applications to its internal resources.

2.2 Challenges

In the context of the applications and the environment described above, several challenges must be addressed to support fully automatic deployments. These challenges include:

- How to specify the information about the application, the environment, and the interactions between the two, necessary to achieve automatic application deployments?
- How to use this information in order to find the "best" application configuration that satisfies the user and application requirements, given the current state of the environment?
- How to create a practical and efficient deployment process that maps the found

application configuration onto the heterogeneous environment?

Each challenge is briefly discussed in the following sections.

2.2.1 Specifying the information necessary for automatic deployments

The first challenge is identifying the types of information necessary to automatically find and deploy application configurations that satisfy user QoS requirements. This information should be minimal, yet sufficient to achieve such automatic deployments.

In the context of component-based applications, the information can be divided into two classes: (1) *application-specific information*, and (2) *environment-specific information*. The former class defines the application behavior and its relationship with the environment. The latter describes the current state of the environment.

Application-specific information The application specification should capture information about the component functionality, the conditions and the effects of deploying a component into the network, and the conditions and the effects of connecting two or more components. A classic component-based application provides only information about the component functionality. This information is local (i.e., per component) and indicates what services are provided by the component. In general, the functionality is expressed at various levels of granularity – implemented interfaces (Java [83] and CORBA [72]) or methods (WSDL [89]). Choosing the right granularity is an important task because it directly influences the creation of application configurations. In order to automate the process of deploying components on nodes and connecting them, applications should also provide local information about the *deployment conditions* and *effects*, and *linkage conditions* and *effects*.

The *component deployment requirements* describe the conditions that need to be satisfied in order for the component to be installed on a node. For example, a node may need to have sufficient available computational power (e.g., in terms of CPU, memory, OS, libraries) before a compute-intensive component can be executed.

The *deployment effects* indicate both the way the component installation affects the environment (e.g., that node CPU and memory resources are consumed) and vice-versa (e.g., that the insecurity of a node can affect the security of sensitive data processed by the component).

The *linkage conditions* describe both the logical and the resource constraints that need to be satisfied before creating a connection between two components running on either the same or two different nodes. They are necessary to reason about the way connections can be created between two components. For example, a client of a video streaming application needs to be connected to a server producing the video stream, ideally crossing a link with sufficient available bandwidth. If the connection is established over a link with insufficient resources, the quality of the image received by the client might degrade.

The *linkage effects* describe the effects of the newly created connection on the network, and vice-versa. For example, a connection between a server producing video stream and a client consumes some amount of available bandwidth. In addition, the quality of the video stream received by the client is influenced by the available bandwidth and the latency properties of the crossed links.

In order to capture such complex conditions and effects, the application specification may require support for various types of expressions, which may differ in their complexity (e.g., linear vs. non-linear), or in their support for various kinds of rea-

soning (e.g., whether the functions are reversible vs. non-reversible). In addition, the standard QoS properties (i.e., node CPU and link bandwidth) are not always sufficient to capture the application constraints and effects; they should be extended to a general set comprising both qualitative (e.g, security) and quantitative (e.g., frame rate, trust level) properties.

One additional problem is caused by the fact that application components may be developed by different programmers; thus, the terms used to denote properties might belong to different vocabularies. This is a problem because it hinders reusability and co-operation between various applications, particularly those that need to run across multiple administrative domains.

The information described above is sufficient to automatically search for application configurations satisfying user QoS requirements. However, more information might be required to make the deployment process both practical and efficient. Two of the factors that directly influence the efficiency of the application deployment process are the application security and data consistency requirements. In order to address these issues, the application specification might need to include *component customization rules* to enrich the initial set of components and increase the chances of successfully finding the best application deployment, *security specifications* to control the deployment process, and *data consistency specifications* to minimize the data consistency traffic between components.

Environment-specific information. The environment description should provide information about the environment structure and properties; the challenges are (1) extracting the relevant information from large amounts of irrelevant data, and (2) coping

with the fact that information can span multiple administrative domains. Depending on the policies defined by each domain, dynamic component-based frameworks might use active (e.g., Remos [30]) or passive (e.g., WREN [97]) tools to monitor the networks.

This work assumes that an external network monitoring layer exists and provides the necessary information.

2.2.2 Finding valid application deployments

Finding an application configuration involves two aspects: (1) finding a set of logical component compositions and (2) mapping the "best" composition onto the network. In general, these two aspects cannot be separated out because of the inter-dependence between them. The mapping of components on nodes and the connections between components depend on the logical compositions; a logical composition depends on whether the mapping of a component on the network satisfies its resource constraints.

A possible solution is to consider the application configuration as a list of instructions/actions to be executed, such as "place component on a node", "connect two components running on two nodes", "remove component from node", or "destroy connection between components". In this case, searching for the "best" application configuration becomes equivalent to choosing a subset of actions from a complete set of actions. This is a hard problem because the search space (i.e., the complete set of actions) is proportional to the size of the application and the size of the network.

In dynamic component-based frameworks, a good search algorithm should (1) efficiently find mappings between valid application configurations and the environment, (2) scale well in the presence of large amounts of irrelevant information, and (3) opti-

mize a given cost of the application deployment, given the complex expressions that define user and application requirements.

All these desired properties are influenced by the centralized or decentralized nature of the search algorithm. The main advantage of a centralized solution is that the algorithm can use information about the entire search space. However, it is difficult to aggregate all the information in one place, particularly if the environment is formed by multiple administrative domains. A decentralized solution does not have this problem; however, each instance of the algorithm works only with local information and a solution might not be always found. Thus, choosing the type of the algorithm directly impacts the framework usability and the application performance.

2.2.3 Automatic deployment of the application configuration

Once a valid application configuration is found, the components should be automatically deployed on the appropriate nodes and connected as required. The practicality and the efficiency of the deployment process are influenced by (1) the complexity of the run-time system responsible for automatically downloading, instantiating, starting, and connecting components, and (2) the overhead introduced by the automatic deployment process and its effects on the component-based application. The goal is to install in the network a minimal run-time system which supports an efficient deployment process; in addition, the performance of automatically deployed applications should be comparable to the performance of manually deployed ones.

Two of the major costs that affect the efficiency of automatically deployed applications are introduced by the application requirements for *security* and *data consistency* guarantees.

Satisfying security guarantees. In general, component deployments may span multiple administrative domains, necessitating cross-domain authentication and authorization among dynamically created principals. Providing security guarantees in such conditions is challenging because there is no centralized trusted third party to act as mediator between participants, and the security decisions must be taken when only partial knowledge about domains is exposed. In addition, the authorization process ideally should guarantee single sign-on, fine-grained, and customizable access control to resources.

Maintaining data consistency. In situations when several replicas sharing data are running in the network, component-based frameworks should ensure that the application consistency requirements are satisfied. What makes the data consistency problem interesting in this context is that the assumptions are different when compared to the same problem in distributed databases, distributed file-systems, or distributed-shared memory systems. In such systems, the consistency protocols improve their efficiency by making assumptions about the data structure or access patterns. Component-based frameworks deploy general applications and cannot make similar assumptions that are valid across all applications. In addition, applications deployed in component-based frameworks dynamically adapt to environment and client QoS changes, thus potentially modifying the application consistency requirements. For example, the airline reservation system described in Section 2.3.2 should allow users to browse flights (where weak consistency is appropriate), buy tickets (where strong consistency is required), and switch between the two modes of operation. Therefore, a dynamic component-based framework must provide a data consistency protocol that is flexi-

ble, application-neutral, but still capable of using application-specific information to improve its efficiency.

2.3 Motivating applications

In order to better illustrate the challenges of automatically deploying distributed applications in heterogeneous environments, this section introduces two examples of component-based applications: (1) a security sensitive web-based e-mail application and (2) an airline reservation system. These applications serve as running examples throughout the rest of this document.

2.3.1 Security sensitive web-based e-mail application

The e-mail service provides the expected functionality — user accounts, folders, contact lists, and the ability to send and receive e-mail. In addition, it allows a user to associate a trust level with each message depending on its sender or recipient. A message is encrypted and signed according to the sender's sensitivity and sent to the mail server, which transforms the ciphertext into a valid encryption corresponding to the receiver's sensitivity and saves the new ciphertext into the receiver's account. Such a transformation can be performed by using proxy functions [51]. The cryptographic keys are generated when the user first subscribes to the service.

The e-mail service is constructed by flexibly assembling the following components: (i) a *mail server* that manages e-mail accounts, (ii) *mail clients* of differing capabilities, (iii) *cache mail servers* that replicate the mail server as desired, and (iv) *ciphers* that ensure confidentiality of interactions between the other components by

encrypting and signing messages. These components allow the e-mail application to be deployed in different environments. If the environment is secure and has high available bandwidth, the mail clients can be directly linked to the mail server. The existence of insecure links and untrusted nodes requires the deployment of cipher pair to protect message privacy. Similarly, the cache mail server can be used to overcome links with low available bandwidth and high latencies.

Deployments of the e-mail application need to satisfy several quality requirements. First, clients could ask for minimum execution times for various operations (e.g., to send a message, or to receive messages). Second, given the sensitivity of most messages and the generally insecure environment, clients could require that their messages are protected against eavesdroppers. In order to satisfy the security requirements, components could be customized to implement cryptographic techniques (e.g., encryption, signatures, obfuscation) to protect against malevolent nodes and eavesdroppers. The efficiency requirements could be satisfied by replicating components close to clients to offset high network latencies, and implementing efficient and flexible data consistency protocols.

2.3.2 Airline reservation system

The airline reservation system allows users to browse and buy tickets for flights based on flight number, departing and arriving cities, or travel dates. The main components are *reservation clients* of different capabilities (*viewers* and *buyers*), a main *flight database* that contains all information about existing flights, and *travel agents* that can be replicated as necessary to assist the *reservation clients* when browsing the database or buying tickets.

The airline reservation system needs to ideally provide several levels of QoS for clients, where each level is defined by the transaction privacy, the latency in accessing the required information, and the type of operations to be performed (e.g., browsing the database or buying the tickets). The privacy of a transaction can be ensured by deploying cipher modules around insecure links. The speed of accessing the database can be increased by placing travel agents close to the clients.

For efficiency, in a distributed environment, the data consistency infrastructure can exploit the differences between the consistency requirements for various components. For example, a viewer does not require the most up-to-date information on flight seat availability. However, a buyer may need fresh information in order to make an educated decision. Thus, the travel agent assisting a viewer can have more relaxed consistency requirements than a travel agent assisting a buyer. In addition, a viewer can become at any point a buyer and the travel agent should be able to provide the requested information in a timely manner.

2.4 Summary

This chapter has discussed the three main challenges of automatically deploying component-based applications. Existing systems and techniques that partially address these challenges are described in Chapter 3, which also identifies their shortcomings. The solutions developed in this research to address these shortcomings are presented in the Chapters 4, 5, and 6, and evaluated in Chapters 8, 9, and 10.

Chapter 3

Related Work and Background

This section is divided into two parts. The first part describes previous work related to adaptation-capable frameworks. The second part introduces the necessary background to understand the solutions described in this document. The related work part is further split into two parts. The first part describes four representative adaptation-capable systems which address the challenges presented in Chapter 2: Globus Grid [33], Ninja [77], Conductor [94], and CANS [40]. The second part highlights how independent modules of other projects address subsets of those challenges.

3.1 Adaptation-capable frameworks

The goal of satisfying user QoS requirements when accessing distributed applications is closely related to the goals of other recently proposed approaches for QoS-aware deployment of applications in heterogeneous and dynamically changing distributed environments. These approaches can broadly be classified into two categories. The

first category, exemplified by the systems such as Globus Grid [45] and Darwin [13], focuses on identifying and reserving appropriate resources in the network to satisfy application requirements, in a sense *adapting the network to the application*.

The second category complements the first by examining how the application can itself be adapted to achieve desired QoS requirements, in situations where the network resources should be treated as a given. The techniques that have been proposed can be further broken down into two classes. The first class comprises systems such as EPIQ [29], ErDos [28], Active Harmony [44], and the Application Tunability framework [15], which assume that the application structure is more or less fixed and adaptation is achieved by altering the *internal behavior* of one or more of the components (e.g., changing an internal algorithm). The second class of approaches, exemplified by systems such as Active Frames [66], Active Streams [10], Eager Handlers [99], Ninja [77] Conductor [94], and CANS [40], has focused on *external behaviors* by looking at the adaptation of *data streams* flowing between static application components and using *application-specific filters* that can be dynamically introduced and placed at appropriate places in the network.

For a better understanding of these various approaches, the next sections describe in detail four representative adaptation-capable systems spanning these categories: Globus Grid, Ninja, Conductor, and CANS. Each framework description highlights the solutions provided by the framework to the challenges presented in Chapter 2: (1) designing an application model able to capture the application behavior (*application model*), (2) searching for a valid application configuration (*planning algorithm*), (3) creating an efficient and practical deployment process by efficiently providing security and data consistency guarantees (*security guarantees* and *consistency guarantees*).

3.1.1 Globus Grid

Grid technologies were initially developed to enable resource sharing between applications belonging to dynamically created virtual organizations. Representative applications include collaborative visualization of large scientific datasets, distributed computing for computational intensive applications, and coupling of scientific instruments with remote computers and databases. The same way the Web has evolved from a repository of information into a complex repository of services, the Grid is evolving into an Open Grid Services Architecture (OGSA) [33], capable of supporting scientific applications, enterprise applications, and B2B partnerships. The Grid architecture is defined as a combination of five layers: (1) the *fabric layer*, which provides the resources shared by Grid applications, (2) the *connectivity layer*, which defines the core communication and authentication protocols, (3) the *resource layer*, which provides protocols for secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operation on individual resources, (4) the *collective layer*, which captures interactions across collections of resources, and (5) the *application layer*, which comprises the user applications that use the Grid technologies for a successful execution. Of these layers, Globus solutions to the problems identified in Chapter 2 are provided by the connectivity layer, the resource layer, and the collective layer.

Application models. Grid applications are defined as sets of components that communicate with each other using files. This means that each component has as input a set of files, processes the data contained in them, and writes the results into new files. Whenever a user wants to execute a Grid application in the Grid infrastructure,

the user needs to provide complete information about how the components are connected, where the original input and final output files are located, and what are the application resource requirements (i.e., CPU, bandwidth, memory). An alternative application model allows users to specify only abstract workflows, where an abstract workflow is defined as a DAG whose the nodes are application components (i.e., logical transformations) and the links represent the files that connect two components.

The language used to create application specifications is the Resource Specification Language (RSL), which defines the syntax used to compose resource descriptions. In RSL, components are allowed to define specific <attribute, value> pairs, where each attribute is a resource description and serves as a parameter to control the behavior of one or more components in the resource management system

The common shortcoming of both the file-based and abstract workflow-based models is that they do not allow applications to define extensible sets of general properties. There are several causes for this: (1) the RSL language defines only a fixed set of attributes, (2) GARA, the Globus Grid planning algorithm, uses the semantics of each attribute during planning, and (3) most applications deployed in the Grid environment are scientific applications which define their constraints based on properties such as time, memory, and CPU.

Planning. The original design of the Globus toolkit [32] envisioned most planning functionality to be realized by the GARA (Globus Architecture for Reservation and Allocation) [34] module. In order to deploy applications with minimal resource consumption, GARA assumes a pre-established relationship between the application tasks. GARA supports resource discovery and selection based on attribute matches,

and allows advance reservation for resources like CPU, memory, and bandwidth. However, it does not consider application specific properties and there is no mechanism to specify component properties that are affected by the environment. Examples of such properties include the communication security.[1]

More recently, the Globus toolkit has included an AI-based planner called Pegasus [21], which maps workflows onto the Grid environment. Pegasus relies on the Globus Replica Location Service [17] to locate the replicas of the desired files and the Globus Monitoring and Discovery Service [18] to find the available network resources and their properties (e.g., load, memory, disk space). Based on the abstract flow and the information provided by the Grid services, Pegasus transforms the abstract workflow into a concrete workflow. This concrete workflow describes in detail the order of executing components on appropriate nodes and moving files between nodes.

Even though Pegasus selects at runtime the components that could achieve the given goal, Pegasus does not consider quantitative application requirements such as the time to execute an operation or the quality of the generated image.

Security guarantees. The Globus module responsible for deploying applications into the network is GRAM (Grid Resource Acquisition and Management), which can be invoked directly by users or by the GARA or Pegasus planning algorithms. GRAM provides support for a secure deployment process based on two major software components: the *Job Manager* and the *Gatekeeper*.

[1] Globus sets up secure connections between application components by default, thereby satisfying this particular constraint. However, more general properties, such as quality of data produced by a component as a function of available bandwidth, cannot be specified.

The *Job Manager* is responsible for parsing the user job request, interfacing with the resource's control system, and initiating the user's job. During the job execution, the Job Manager monitors its progress and handles any job management requests (e.g., suspend, stop, resume).

The *Gatekeeper* handles all the authentication, authorization, and secure communication problems, based on the Globus Grid Security Infrastructure (GSI) [33]. GSI assumes the existence of a Public Key Infrastructure (PKI) and a single shared namespace across domains. Recent work has looked into replacing the PK credentials with Kerberos tokens [1]. In GSI, all resource providers (P) have the necessary authentication/authorization information for all possible users (U), thus implying a storage space proportional to $P \times U$. Follow-on work to GSI, CAS (Community Authorization Service) [61] divides the users into communities such that all providers know about communities only. In this way, CAS improves the memory storage to $C \times (P + U)$, where C is the number of communities.

OGSA extends the Globus toolkit to support Web services; thus, the security infrastructure needs to provide solutions for the security issues of both classic Grid applications and Web services. Examples of security challenges specific to Web services include expressing Web Service security policies, creating format standards for token exchanges, and designing standard mechanisms for authentication and establishment of secure contexts and trust relationships. The solutions are implemented as part of the Globus Toolkit 3 (GT3) and represent a combination of the Web Services Security standards and the GSI protocols [91].

From the point of view of the challenges identified in Chapter 2, the security infrastructure proposed by OGSA/GSI falls short because it does not allow cross-

domain authentication and authorization when the communities are hiding part of the information.

Consistency guarantees. Given the file-based interaction model assumed by Globus applications, support for data consistency within the Globus toolkit is restricted to files. It is expected that should components require any consistency guarantees, these will be implemented as part of the application. For files, the Globus Grid support consists of the Replica Location Service (RLS) [17]. RLS is responsible for finding the physical files corresponding to one logical file name. As part of RLS, all hosts must keep a Local Replica Catalog (LRC) that maps physical files to unique logical file names. In addition, RLS keeps Replica Location Indexes scattered across a Grid environment; the RLIs are responsible for keeping lists of pairs (logical file name, RLC). The data consistency problem arises in Globus Grid when the RLS needs to keep consistent the state saved by RLIs. The solution proposed by the Globus RLS is to provide relaxed consistency guarantees by periodically refreshing the information in RLIs.

Thus, RLS is the only module that has data consistency requirements. Globus Grid does not provide any such guarantees to applications.

3.1.2 Ninja

Ninja [77] enables the construction of scalable, robust, distributed Internet services and permits clients to access such services using a large class of heterogeneous devices. Ninja achieves this goal by dynamically connecting the clients to the required services, and transforming the data sent between them as necessary. The main archi-

tectural components are: (1) *vSpace*, an infrastructure that provides facilities for service component replication, load-balancing, and fault-tolerance, (2) *DDS*, a cluster-based, scalable data storage platform that exposes a coherent image of persistent data, (3) *SDS*, a secure discovery service that provides all clients with a secure directory to services, and (4) *APC*, an automatic path configuration module that computes the best service configuration that can be accessed by clients.

Application model. Ninja regards applications as sets of *operators*, *connectors*, and *paths*. An operator is responsible for processing the input data and producing output data. A connector maintains the connection between two operators by implementing specific communication protocols. A path is a sequence of operators and connectors between a source (client) and a destination (service). The role of a data path is to transform the data stream from the source into a data stream acceptable to the destination. For each operator and connector, the application needs to specify the name, the URL where the sources are located, and sets of input and output data types. Each data type contains information regarding network and media properties. The network properties include reliability, minimum and maximum bandwidth, and port numbers. The media properties include the type of supported media (e.g., text, image, audio, speech, video, and slides) and any media-specific properties.

The shortcomings of the Ninja application model with respect to the challenges from Chapter 2 are (1) both network and media properties sets are fixed, and (2) the application specification does not capture any interactions between the application and the environment.

Planning. The goal of the planning module, APC [14], is to create a Ninja path given the endpoints of the required path, a partially ordered list of operators, and an acceptable range of costs (e.g., latency, bandwidth, cpu consumption). APC performs two steps: (1) logical path creation, and (2) physical path creation. The former selects the appropriate operators and combines them depending on the data type information. The output is a logical path that contains operators connected by connectors. The latter takes as input the logical path and the state of the network, attempts to find the nodes where the operators either exist or could be installed, and makes the necessary connections.

The current implementation of APC does not deploy new instances of operators. Instead, it takes into consideration only operators already running in the network. In addition, APC optimizes only for the length of the logical path, as opposed to the general quantitative and qualitative constraints that might be of interest to a component-based application.

Security guarantees. Once APC finds a valid plan, Ninja starts any required dynamic operators and sets up the appropriate connectors between operators. In addition, the operators register with SDS [76], a secure discovery service. SDS is a scalable, secure, and fault-tolerant data repository that provides clients with directory-style of access to all available services. SDS is secure, because it employs cryptographic techniques to authenticate both ends of a communication channel and ensure that all messages are private. SDS provides two types of security services: (1) clients can specify which services are trusted, and (2) services can specify what type of capabilities should be owned by a client allowed to access them.

SDS performs PK-based authentication and authorization, instead of supporting more dynamic and complex relationships between entities (e.g., role-based access control lists). In addition, once SDS matches clients and services, Ninja provides no security guarantees during application deployment.

Consistency guarantees. All persistent data is saved by Ninja services in a scalable, available, and consistent cluster-based storage layer called DDS [42]. DDS organizes data as a distributed hash table, split in partitions. In order to ensure that data is available, DDS replicates the partitions on several nodes in the cluster. Even though there exist several copies of the data, clients see a single logical copy of the data. This is achieved by executing a two-phase commit protocol to keep all replicas strictly consistent. In this way, any replica can serve read requests, but all replicas must be updated upon a write request.

One of the missing pieces of the coherence protocol is the lack of support in maintaining consistency between replicas spread across multiple clusters. Also, Ninja does not support application-specific weak consistency requirements essential for efficiency in multiple cluster environments.

3.1.3 Conductor

Conductor [94] is a middleware system that adapts applications by injecting functionality (i.e., adaptors) into the network path between clients and services. Conductor supports legacy applications by intercepting, examining, and eventually modifying the communication between a client and service, if the setup costs are offset by the adaptation benefits. Conductor provides the illusion of a normal TCP connection,

when in fact Conductor handles the reliable end-to-end delivery of data.

Application model. In the Conductor system, applications are defined as a set of adaptors. Each adaptor is a stand-alone Java-based module, with fixed functionality. The complete specification of an adaptor contains four parts: (1) the protocols supported by the adaptor, (2) the node resources required by the adaptor, (3) pre-conditions and post-conditions for deploying the adaptor expressed as functions of link characteristics, and (4) a set of interoperability operators that restrict the adaptor combinations.

Planning. Given the Conductor application model, the planning algorithm [73] finds the best adaptor deployment on a route between a client and a service. The planning algorithm has two phases: (1) gathering link information, and (2) computing the adaptor deployment. Conductor starts gathering information from the client toward the service. Each node adds its own properties to the total state information and forwards it to the next node. Once the total state information reaches the destination node (i.e., service), the second phase is executed on that node, in a centralized fashion. The second phase contains two steps. First, Conductor associates link problems such as low available bandwidth or lack of security with sets of adaptors that can solve those problems. Second, the adaptors are combined and mapped onto network nodes.

As of this writing, the planning algorithm does not provide any optimality guarantees. In addition, Conductor restricts itself to single input, single output components, focusing on satisfying resource constraints.

Security guarantees. Once the adaptor combination is found, Conductor deploys the adaptors on the appropriate nodes. Each Conductor-aware node relies on a security box [64] to provide security guarantees. The security box is responsible for node authentication, protection of the planning process, and data protection. Node authentication is very important because only trusted nodes should be allowed to provide information for the planning process and execute adaptors. Malevolent nodes can affect the planning process by either providing misleading data or modifying the correct data or the planning decisions that pass through the nodes. Conductor protects the planning algorithm by encrypting and signing the messages sent by trusted nodes. In addition, Conductor can protect data messages sent between clients and services through insecure links by deploying encryption/decryption adaptors.

The challenge in Conductor is to provide finer-grained control over the security process and allow the security requirements to influence the planning decisions.

Consistency guarantees. Conductor does not provide any data consistency guarantees to applications. The assumption is that adaptors would be designed to guarantee data consistency if an application requires it.

3.1.4 CANS - Composable, Adaptive Network Services Infrastructure

CANS [40] is an application-level infrastructure that customizes the data path between clients and services by injecting application-specific functionality into the network. CANS performs three types of application adaptation: (1) *intra-component adaptation*, where each service detects and adapts to minor resource variations on its own, (2) *data path reconfiguration and error recovery*, where the data path undergoes

localized changes, and (3) *re-planning*, where existing data paths are destroyed and rebuilt as a result of large-scale variations.

Application models. CANS regards applications as sets of components, where each component is a self-contained piece of code. There are two types of components: soft-state *drivers* and hard-state *services*. The former are standalone mobile code core modules that perform some operation on the data stream. The latter encapsulates more heavy-weight functionality and can save persistent data. Both types of components process data type streams and connect to each other depending on the compatibility of their input and output data types. The type system defined by CANS allows applications to define stacks of data types in order to capture the effects of multiple drivers on one data stream. In addition, each data type is augmented with a set of extensible link properties (e.g., security and reliability). This enables CANS to capture the interactions between a data stream and a link when the data stream crosses that link.

CANS supports a sophisticated application model, but only allows applications to specify their behavior using linear functions; the shortcoming is the lack of support for arbitrary functions.

Planning. CANS adapts applications by computing at run-time the best data path between applications and services. The planning algorithm [39] finds a data path that transforms a source data type into a destination data type, given a route and a type graph. The route contains a list of nodes connected by links into a chain and their resources. The vertices in the data type graph represent the application data types. The edges connecting two vertices represent components that transform one data type into

another data type. One of the goals of CANS is to minimize the resource consumption while satisfying the application and user requirements. However, finding the optimal solution is NP-hard. The novel idea introduced by CANS is discretizing the set of values for resources and using a dynamic programming strategy to find the optimal data path. CANS adopts similar component restrictions as Conductor, but can handle constraints imposed by the interactions between application components and network resources, and additionally can efficiently plan for a range of optimization criteria. For example, the CANS planner can ensure that node and link capacities along the path are not exceeded by deployed components, while simultaneously optimizing an application metric of interest (e.g., response time).

Security guarantees. Once the planning algorithm finds the best data path, CANS deploys the drivers and the services on the appropriate nodes. The infrastructure that supports automatic deployment consists of *communication adapters*. The adapters are responsible for creating the communication channels between (1) application and drivers, and (2) drivers and services. From a security point of view, CANS ensures privacy of the data streams only if there are drivers that implement encryption services. Otherwise, CANS does not provide any security guarantees.

Consistency guarantees. CANS provides consistency guarantees only during path reconfiguration, to ensure that there is no loss of data when a path is reconfigured. However, these mechanisms assume a specific filter model for components, which may not be applicable in general component-based applications.

3.2 Individual techniques

As described in Chapter 2, the main challenges of automatically deploying component-based applications in heterogeneous environments are (1) creating sophisticated *application models* that capture the application behavior, (2) efficiently use these models to *find a valid component deployment,* and (3) provide an *efficient and practical deployment process* by satisfying application security and data consistency guarantees. This section examines some other projects that have proposed isolated techniques to address these challenges.

3.2.1 Application description

Most relevant previous work on application description models has occurred in the context of component-based frameworks, including the four systems discussed earlier.

In general, the only information provided by the application to the framework specifies the component *functionality* and *required services*. Both the component functionality and the required services can be specified at the level of (1) *interfaces* (Java [83] and CORBA IDL [72]), (2) *methods* (.NET WSDL [89]), or (3) *data types* (Globus Grid [35]). This information is sufficient for frameworks to decide how to download components into the network and connect them.

More recently, models capture additional information pertaining the application deployment, such as deployment conditions and effects and interactions between applications and the environment. Examples of frameworks that use such sophisticated models were described earlier in the chapter (i.e., Globus Grid, Ninja, Conductor, CANS).

3.2.2 Planning algorithms

A growing number of projects are currently looking at building applications at run-time by dynamically selecting and/or mapping components onto the network. Such systems can be divided into two classes. The systems in the first class assume the existence of an external planner (e.g., Active Frames [66], Eager Handlers [99], Active Streams [10]). The systems in the second class implement their own planner, and can be further divided into two subclasses. The first subclass includes systems which assume a pre-established relationship between application tasks, in order to deploy them with minimal resource consumption (e.g., Globus Grid). The second subclass of planners both select and deploy a subset of components, while satisfying application and network constraints (e.g., Ninja, Conductor, and CANS). A detailed discussion of the planning algorithms implemented by these systems was given in the previous sections of this chapter.

3.2.3 Security guarantees

Previous efforts in providing security guarantees have looked at (1) cross-domain authorization, (2) support for fine-grained access control, and (3) modeling application and network resource properties to permit their use by automated planning modules. The first two are classic problems of providing security guarantees in a distributed environment. The last is more specific to this work. The next paragraphs discuss various approaches for each of them.

Cross-domain authorization. Component-based frameworks target application deployment in heterogeneous environments spanning multiple administrative domains;

thus, they raise new security issues. Several systems (e.g., DCE [75], DCOM [88], CORBA [71], Globus Grid [33]) aim to solve the cross-domain authentication and authorization problems that result in such systems.

DCE [75] provides authentication and authorization based on private-key cryptography with a trusted third party. CORBA [71] and the Web services infrastructure [46] provide a general interface for authentication and authorization, leaving it up to application programmers on how exactly they choose to implement it. SESAME [54] authenticates users and provides them with an authorization credential (Privilege Attribute Certificate) that can be used for all authorization decisions. The Legion system [43] controls heterogeneous, independent, and distributed resources presenting the user with the image of a single, coherent environment. From a security point of view, all resources are considered to be objects residing in a single shared namespace, and are uniquely identified by a Legion Object Identifier (LOID) [31] that contains a public key. Users are authenticated using shortlived Legion credentials [93] generated the first time the user logs on into the system. As described earlier, the Globus Grid system relies on the Globus Grid Security Infrastructure (GSI) [33] to handle all of its authentication, authorization, and secure communication problems.

Compared to the challenges described in Chapter 2, the shortcomings of these projects include (1) assuming the existence of a single policy root (hence namespace) for credentials, and (2) not providing support for translation between global system-wide and local credentials (i.e., the rights associated with a request are modulated to the credentials associated with it as opposed to the local credentials these translate to).

Granularity of access control. One of the challenges of providing security guarantees is supporting fine-grained access control to resources. For example, applications should permit users to access applications at the level of objects, interfaces, or methods, as required by the application policy.

The Java 2 environment [41] combined with the Java Authentication and Authorization Service (JAAS) [62] addresses this challenge by relying on security managers and policy files to define resources that in principle can support any granularity of access control. Unfortunately, the security manager only checks rights to access JVM resources, like files or sockets. In order to protect other resources, the applications need to implement their own access control mechanisms.

DCE [75] and CORBA [71] enable any level of granularity for access control by letting the applications to define their own notion of resource. DCOM [88] applications can control access to low level object only by taking advantage of the API's exposed by the DCOM programatic security. Legion [43] objects must implement a special function, *M*ayI that is called to check credentials every time a user invokes a method on the object.

Such mechanisms are not appropriate to address the challenges presented in Chapter 2, because they require applications to take the access control decisions at run-time (Legion, DCOM). Ideally, the process of satisfying the security guarantees should be completely transparent to the application.

Expressing component and network properties. Most dynamic component-based frameworks rely on an application registration step, where complete specifications of the application components are provided to the framework. As explained in Chap-

ter 2, the challenge is to give applications the necessary freedom in expressing their behavior and requirements, and still be able to efficiently use the information to find a valid application configuration.

Unfortunately, most component-based systems deal with this problem by restricting the applications to specifying only a small number of resource consumption properties (e.g., CPU usage, bandwidth consumption) determined a priori. Examples of such systems include Ninja and Conductor, which were described earlier.

3.2.4 Data consistency protocols

The data consistency problem has been extensively researched in *distributed databases*, *distributed file systems (DFS)*, and *distributed shared memory systems (DSM) for symmetric multi-processors (SMP)* and *wide-area environments*. A common theme throughout is the use of application-specific information to efficiently satisfy application consistency requirements.

Distributed shared memory - SMPs and cluster environments. Shared-memory multiprocessors have emerged as an efficient way of providing increased computing power and speed. A shared-memory multiprocessor node contains multiple low-cost processors connected with shared memory modules by an interconnection network. In order to reduce the memory access time, a cache memory module is attached to every processor. Often, several processes running on different processors are working on different copies the same data. The data consistency problem states that all these copies need to be consistent. Currently, there are hardware and software solutions.

Hardware solutions are efficient, but increasingly complex and implement only a

limited number of consistency protocols. Software-based data consistency protocols were developed to avoid this complexity while providing multiple levels of consistency. Munin [12], View Caching [55], Treadmarks [56], and HLRC [78] are examples of software DSM systems that parse application-specific information to choose the appropriate level of consistency at the granularity of pages or objects.

Treadmarks and HLRC provide release consistency to a shared memory organized as a linear array of bytes. Conflicts between memory accesses are prevented by marking the corresponding instructions as critical sections. The difference between these two systems is the way they propagate updates. Treadmarks is a distributed system, where updates are sent between peers. HLRC defines a home node for each page and all updates are sent and merged into that home node.

More recent work [26, 3] indicates that providing only one protocol does not work well for all parallel programs. Munin annotates variables with their expected access pattern, while the View Caching system defines a view as the data used by a user-defined method and uses view-specific knowledge (data access patterns) to choose the appropriate coherence protocol.

Distributed shared memory - Wide area environments. The natural extension to executing applications in SMPs is deploying the applications in wide-area environments such as the Internet. The problem is to ensure data consistency between replicas spread across a long latency network, while minimizing the synchronization traffic. Interweave [16] and ObjectViews [65] are representative of the solutions in this area, which solve this problem by using application-specific information on the data structure and/or access pattern. In InterWeave, applications define the consistency unit as a

data segment formed by data blocks and views as subsets of blocks. Views reduce the synchronization traffic, because sharers of the same segment can have different views. Object Views define views as restrictions of an original object such that all accesses to the object are performed through its views. Based on the view information, a combined run-time and compiler solution decides which object parts need to be updated for correct execution.

Distributed databases. The consistency problem in distributed databases is to maintain data correctness (e.g., mutual consistency) and availability when multiple read/write operations are simultaneously executed on several replicas. The provided consistency guarantees range from one-copy serializability [4] to weak consistency [19] and continuously weak consistency [95].

The general assumption is that the execution of a transaction transforms an initial correct database state into another correct state. A database state is considered to be correct if the execution of a set of transactions is serializable and the execution of any transaction is atomic (e.g., either all or none operations are executed). In this context, ensuring that the execution of simultaneous operations on replicated data is serializable is called *one-copy serializability* [4]. However, providing one-copy serializability can be very expensive and not necessarily required by all applications. The solution is to relax the consistency guarantees and allow the read/write operations to simultaneously execute on multiple replicas, even when the operations create conflicts between the database replicas. Bayou [19] is an example of a weakly connected replicated storage system that uses application specific functions to resolve conflicts. TACT [95] extends this work by recognizing that different applications have different

consistency requirements. TACT defines a continuous consistency space along three dimensions and allows the applications to specify their consistency requirements as a combination of three parameters: *numerical error*, *order error*, and *staleness* [96].

In the case of weak consistency, the challenge is how to allow the read/write operations to simultaneously execute on multiple replicas, even when the operations create conflicts between the database replicas. Two of the implemented solutions are (1) to continuously send updates between replicas based on application-specific information about the data structure and access pattern [95], and (2) to use application-provided functions to detect and resolve conflicts [19].

Distributed file systems (DFS). Similar to distributed databases, distributed file system spread information across wide-area networks. Their goal is to allow users to transparently access both their local and remote files [74]. Some of the differences between databases and file systems are that: (1) databases are usually much larger than file systems, (2) file systems allow operations on files and directories, and (3) file systems save the information in files and directories organized as a tree. This paragraph discusses only how some of the most successful DFS developed recently (Unix [74], Locus [90], Sprite [69], Network File System (NFS) [79], Andrew File System (AFS) [67], Ivy [68], Ficus [53], Coda [80]) use these differences to their advantage. From the point of view of the chosen consistency model, most distributed file systems can be divided into systems that provide *serializability* of simultaneous conflicting writes [79, 53], and systems that guarantee *close-to-open* consistency [68, 80]. In order to efficiently provide such guarantees, distributed file systems use application specific information. First, DFSs reduce the synchronization traffic by using

information on the data structure when choosing the appropriate granularity levels (pages [90, 69], files [79, 68], volumes [67, 53, 80]). Second, disconnected replicas are reconciled by using at least one of the following three methods: (1) executing automatic procedures [90, 68, 80], (2) using application-specific procedures [68, 60], and (3) triggering an error and allowing the user to fix conflicts manually [90, 80].

The common theme across all of these systems is that the underlying cache coherence protocols are able to make assumptions valid across applications and efficiently use this information to design appropriate consistency protocols and define granularity levels. However, none of these assumptions directly apply to component-based systems, where components can encode arbitrary behavior. Before these techniques can be applied, one needs mechanisms which address the challenges specific to component-based frameworks.

3.2.5 Summary

The analysis of the related work leads to the conclusion that no system addresses all challenges described in Chapter 2. Instead, the systems provide solutions to simplified subproblems. For example, Globus Grid restricts the application specifications, CANS supports only applications composable as a chain, Conductor does not consider any security requirements during planning, and Ninja does not deploy new components, but uses already deployed ones.

This work addresses the challenges in their general form through the solutions described in Chapters 4, 5, and 6. Chapter 4 introduces a flexible application specification model. Chapter 5 presents an AI-based planning algorithm which uses this

model to find valid application configurations. Chapter 6 describes how the application configuration found by the planning algorithm can be deployed in an efficient and practical manner, by satisfying the application security and data consistency guarantees.

3.3 Background

In general, the techniques described in Chapters 4, 5, and 6 can be applied to a variety of programming languages and component-based frameworks. To make the concepts clearer, this thesis uses examples based on the Java implementation of these techniques in the Partitionable Services Framework (PSF), as described in Chapter 7. The implementations rely on DisCo [38], a middleware infrastructure that allows users to automatically discover the required services, and securely download and install code from remote locations. This section describes three of the main tools provided by DisCo: Switchboard, dRBAC, and the DisCo Discovery Service. The communication between PSF entities is achieved using either Java RMI or Switchboard; Switchboard is a secure communication abstraction developed to continuously monitor the trust relationships between entities. The security guarantees are provided based on dRBAC, a distributed role based access control and trust management system. The discovery service is used to discover instances of PSF running in the system.

For completeness, this section briefly discusses these technologies. Readers already familiar with them may wish to skip to Chapter 4.

3.3.1 Java Language and Runtime Environment

The Java language and runtime environment [83] allows applications to run on a variety of machines, independent of the underlying software and hardware. Rather than executing on the native operating system, Java programs are transformed into bytecode and interpreted by a Java Virtual Machine (JVM).

This work takes advantage of several important features of the Java technology: *mobile code*, *sandboxes*, and *reflection*.

Mobile code. When executing a Java program, the JVM is responsible for locating and loading the appropriate bytecode. One of the features of Java is that JVMs can find and download remote bytecode based on URLs. PSF uses this feature to automatically deploy components into the network.

Sandboxes. One of the challenges of executing mobile code on a remote node is protecting the node from malevolent code. In Java, this challenge is addressed by first verifying the bytecode and then executing it. During execution, the JVM uses

Table 3.1: Example of a policy file.

```
grant signedBy "Advisor" {
  permission java.io.FilePermission "/tmp/*", "read,write";
};
grant {
  permission java.util.PropertyPermission "java.vendor";
};
grant signedBy "superuser", codeBase "file:/home/superuser/*" {
  permission java.security.SecurityPermission "Security.insertProvider.*";
  permission java.security.SecurityPermission "Security.removeProvider.*";
  permission java.security.SecurityPermission "Security.setProperty.*";
};
```

a *policy file* to check whether the code is allowed to perform operations that could damage the host (e.g., write a file). Such a policy file contains rules that specify how the code coming from given URL's and signed by recognized entities is granted or not the permissions to execute on the node. Table 3.1 contains a simple example of a policy file. PSF uses sandboxes to enforce that components deployed on remote nodes execute with the correct permissions.

Reflection. Reflection is a mechanism provided by Java to inspect the structure of a Java object without having access to its sources. With Java reflection, a Java object can find the class of another Java object; get information about the class's modifiers, fields, methods, constructors, and super-classes; create instances of classes whose names are not known until runtime; verify whether variables or methods are defined by a class or interface; and determine the value of a variable or invoke a method even if the name is unknown until runtime.

PSF uses reflection to complement the Javassist API's [85] when automatically generating components, and to instantiate components whose names are provided only at runtime.

3.3.2 Java Remote Method Invocation

Java RMI [84] integrates the distributed object model into the Java [83] programming language in a natural way while retaining most of the Java programming language's object semantics. The main features of Java RMI are: (1) supporting seamless remote invocation on objects in different virtual machines, (2) preserving the type-safety provided by the Java platform's runtime environment, and (3) maintaining the safe envi-

Table 3.2: RMI pseudo-code.

```
1  public class Client() {
2    public void run() {
3      String name = "//" + args[0] + "/Server";
4      ServerInterface server = (ServerInterface) Naming.lookup( name );
5  }}
```

```
6  public class Server extends UnicastRemoteObject
7                       implements ServerInterface {
8    public void run() {
9      name = "//"+InetAddress.getLocalHost().getHostAddress()+"/Server";
10     Server engine = new Server();
11     Naming.rebind( name, engine );
12 }}
```

```
13  public interface ServerInterface extends Remote {
14    void foo() throws RemoteException;
15  }
```

ronment of the Java platform provided by security managers and class loaders. Java RMI is one of the two communication abstractions used by PSF to create connections between components. The other communication abstraction is Switchboard, as described in Section 3.3.4.

Table 3.2 shows the pseudo-code for an RMI client and an RMI server. In this example, the `ServerInterface` describes the methods that could be accessed by clients through remote calls; thus, it needs to extend the `Serializable` interface (line 13). The `Server` implements the `ServerInterface` and extends the `UnicastRemoteObject` (lines 6 and 7). The `UnicastRemoteObject` provides the methods needed to create remote objects and export them (i.e., make them available to remote clients). In order to register with the RMI Naming service, the server must "bind" to the namespace (line 11). Once the server is registered with RMI, clients can look up a server that presents a set of desired attributes (line 4). If such a server exists,

Table 3.3: dRBAC delegation types.

Self-certifying	[Subject → Issuer.Role] Issuer with $Attr_1$=Val_1, $Attr_2$=Val_2, ...
Third-party	[Subject → Entity.Role] Issuer with $Attr_1$=Val_1, $Attr_2$=Val_2, ...
Assignment	[Subject → Entity.Role '] Issuer with $Attr_1$=Val_1, $Attr_2$=Val_2, ...

the clients receive a reference to the server. All method calls will be executed against that reference.

3.3.3 dRBAC

dRBAC [23] is a PKI-based trust management and role-based access control system originally developed for expressing and enforcing security policies in coalition environments spanning multiple administrative domains. Such environments are characterized by partial trust and the absence of central policy roots. dRBAC credentials, called *delegations*, express the mapping of an equivalence class of access rights in one trust domain to members of another equivalence class, possibly in another trust domain. Each of these equivalence classes is represented by a dRBAC *role*. These delegations potentially include attenuation of valued attributes.

PSF uses dRBAC to authenticate and authorize entities across multiple administrative domains, and translate properties between namespaces. A summary of relevant features of dRBAC follows; a more complete description appears in [23].

Each dRBAC delegation is cryptographically signed by its issuer. Additional credentials may be required as evidence of the issuer's authorization to administer the

rights proved by the delegation. As with other role-based access control systems, dRBAC delegations may be transitively chained to form proof graphs indirectly authorizing a required class of access rights. A dRBAC credential can be tagged with expiration dates and also may additionally require online validation monitoring from an authorized "home" which is aware of any revocation of the delegation. Similar to other distributed trust management engines (SPKI [27], KeyNote [6], PolicyMaker [7]), dRBAC supports third party delegations and linked namespaces.

Table 3.3 presents the three types of dRBAC credentials: self-certifying, third-party, and assignment delegations. The self-certifying and third-party delegations allow an Issuer entity to give the permissions associated with an Entity.Role role to a different entity or role (Subject). The difference between them is based on whether the owner of that role is also the Issuer. An Issuer entity uses the assignment delegation to give the *right of assignment* for Entity.Role to another entity (Subject) located outside the Issuer's space. The assignment delegations permit the usage of private roles outside the defining domain. The ($'$) mark indicates that the Subject is allowed to assign Entity.Role to other Subjects.

Using dRBAC, a trust-sensitive component C can determine if a set of dRBAC credentials X gives some subject S the set of access rights represented by a role R continuously over some duration. To do this, C presents the public identity of S, a set of required access rights R, and the credentials X to a dRBAC implementation. The dRBAC module first authenticates the signatures and establishes validity monitors for all the credentials in X. Authorization is granted if the dRBAC module can construct a graph (proof) from valid and authenticated credentials in X that "proves" that S possesses the rights required by R.

dRBAC credentials are stored in a distributed repository. To assist in collecting dRBAC credentials that authorize a particular role, dRBAC contains a mechanism that relies on *discovery tags* associated with credential subjects and objects. These tags identify an entity as "searchable from subject" or "searchable from object", permitting queries about credentials involving the entity to be directed as appropriate to its home node.

3.3.4 Switchboard

Switchboard [24] permits the establishment of secure, authenticated, and continuously authorized and monitored connections between a pair of components. The latter property distinguishes Switchboard from abstractions like SSL/TLS [37, 22]. As mentioned before, PSF uses Switchboard and Java RMI to create connections between the deployed components. In addition, Switchboard is used for the communication between the the PSF modules as explained in Chapter 7.

Prior to forming a Switchboard connection, the components at either end provide their *authorization suites*—PKI identities (including private keys for authentication), dRBAC credentials to be supplied to the partner, and `Authorizer` objects for evaluating the partner's credentials. Authorizers generate `AuthorizationMonitors`, which inform either partner when the trust relationship changes. When Switchboard connections span multiple hosts, a cipher is established using a key-exchange protocol, and connectivity is monitored using replay-resistant heartbeats that indicate liveness and round-trip latency. Switchboard connections provide a two-way procedure-call (RPC) interface appearing as a custom socket on top of which requests can be routed.

Table 3.4 shows the pseudo-code necessary to create Switchboard connections.

Table 3.4: Switchboard code.

```
1  public class Client {
2   public void run() {
3    InetSocketAddress serverAddress = new InetSocketAddress( address,
4                                                             port );
5    Authorizer authorizer = new DrbacAllAuthorizer();
6    SbEventHandler smockSbEventHandler = new DrbacSbEventHandler();
7    SbRpcStack stack = Switchboard.lookup( serverAddress, serverName,
8                                           clientKeys, authorizer,
9                                           clientCredentials,
10                                          smockSbEventHandler);
11   Object o = stack.getOutcallProxy();
12   ServerInterface server = (ServerInterface) o;
13 }}
```

```
14 public class Server implements ServerInterface {
15  public void run() {
16   Authorizer clientAuthorizer = new DrbacAllAuthorizer();
17   InetSocketAddress serverSocketAddr = new InetSocketAddress( address,
18                                                               port );
19   Switchboard.serve( serverSocketAddr, serverName, serverKeys,
20                      clientAuthorizer,
21                      serverCredentials,
22                      (Serializable) this );
23 }}
```

```
24 public interface ServerInterface extends Serializable {
25    public void foo();
26 }
```

The operations executed by the client and the server in order to communicate through Switchboard are similar to the operations required by Java RMI. Similar to Java RMI, the `ServerInterface` must extends the `Serializable` interface (line 24). However, the `Server` must implement only this interface (line 14). The `Server` must bind with the namespace (lines 19-22) before clients can search for it (lines 7-10). The difference is that Switchboard requires that the `Server` and the `Client` provide more parameters beside the name and the reference of the object. They also have to provide their credentials, keys, and authorizers, which will be used by Switchboard to

perform the mutual authorization of both the client and the server. Additional details about Switchboard can be found at `http://pdsg.cs.nyu.edu/switchboard`.

The continuous monitoring property of Switchboard connections is crucial for supporting single sign-on access control in dynamic environments, where client and/or network credentials can change in the middle of long-lived component interactions. Such a change in credentials invalidates the corresponding dRBAC proofs, and results in notification to the `AuthorizationMonitors` at either end of the connection. These monitors can then take appropriate action, including requiring a component to re-validate itself prior to approving future requests.

3.3.5 DisCo Discovery

DisCo Discovery is a mechanism that allows users and application components to locate nearby instances or providers of needed services. In DisCo, each distinct service is identified by a unique ServiceDescriptor. Providers must first register with DisCo by providing signed credentials which reference themselves as authorized providers for an enclosed ServiceDescriptor. To provide reliable fail-over in cases where the discovery mechanism does not locate a local authorized provider, ServiceDescriptors also identify a default service home.

Clients interested in connecting to a particular service, generate a discovery query specifying a specific service descriptor. DisCo is responsible for matching the query with an advertisement published by a provider. If such a match exists, DisCo returns the information about the service provider.

In PSF, entities (e.g., clients, nodes) use the DisCo Discovery service to discover and contact running instances of the PSF modules.

Chapter 4

Describing Application and Environment Characteristics

A central feature of a dynamic component-based framework is its ability to adapt to different network conditions by assembling distributed applications from an appropriate combination of components. This chapter describes the information required by the framework to achieve flexible application assemblies. Examples of such information include (1) a high-level *service specification model*, which declares constraints on linking one component to another instead of statically specifying linkages, and (2) an *environment specification model*, which captures the structure and the state of the environment. Both models use *properties* to capture aspects of the environment, the application, and the interactions between them. The next section defines the general notion of properties; the following two sections describe in detail the application and the network models.

4.1 Properties

Properties represent attributes associated with the network and the applications. A property is defined as a tuple $(name, value)$, where *name* is a unique name and *value* can be a Boolean, real, or interval value – i.e., $value \in D$, where $D = \mathbb{B} \cup \mathbb{R} \cup (\mathbb{R} \otimes \mathbb{R})$. [1] In general, a property can be defined as a function of other properties and its value can be computed at run-time.

Properties are used to capture the current status of the network, the application behavior, and how the network and the application affect each other. For example, the network might be divided into secure and insecure links. Similarly, an application might want to distinguish between sensitive and non-sensitive messages and require that the former be protected against eavesdroppers when transiting across the network links. In order to capture these requirements, two Boolean properties, `Secure` and `Privacy`, can be created and associated with the network links, respectively the application components. The insecure links will have the `Secure` property set to `False`, while the secure links will set this property to `True`. Similarly, messages can be associated with the `Privacy` property. If a message is private (`Privacy` is `True`) and crosses a secure link, the privacy of the message is intact after crossing. However, if the message crosses an insecure link, its privacy is compromised. In order to capture such a behavior, the application can specify that the `Privacy` property of the message after crossing a link is equal to the Boolean AND operation between the `Secure` property of the link and the `Privacy` property of the message before crossing the link.

[1] $\mathbb{B} = \{true, false\}$ represents the set of Boolean values. $\mathbb{R}$ is the set of real values. $\mathbb{R} \otimes \mathbb{R}$ is the set of intervals with real limits.

It is important to note that the framework does not assume any information about the semantics of a given property with respect to the application. In the previous example, the `Privacy` property refers to whether or not data produced by a component can be deemed confidential. However, the dynamic component-based framework does not "understand" the English word "Privacy"; its only concern is with the range of values that can be associated with the property. The above model is very flexible and allows the expression of a variety of application requirements and effects.

4.2 Environment specification model

As mentioned in Chapter 2, component-based applications are often deployed in environments which are *highly heterogeneous*, have a *considerable size*, and are divided into *multiple administrative domains*. Capturing the state of the environment is important because the application behavior is often influenced by the environment and dynamic component-based frameworks have to consider the current network state in order to make correct decisions.

Example. To illustrate the requirements of the environment specification model, this section starts with an example. Consider a network environment made up of hosts interconnected with links. In order to dynamically deploy applications, the framework needs various pieces of information about the characteristics of the environment. For example, the model may need to capture the fact that a node has an IP address of 216.165.111.134, is running Windows XP, has installed the JDK 1.4 Java library, and has 200 Mb memory and an Intel Pentium II processor with 100 units of normalized

CPU resource available. Similarly, for a link, the network model may need to capture both quantitative features (e.g., the end points of the link are $< N_1, N_2 >$, the link latency is 10ms, and the link maximum available bandwidth is 50Mbps), and qualitative features (e.g., information about the security and the stability of the link).

These issues are addressed by modeling the environment as a set of nodes N connected by a set of links $L \subseteq N \times N$. Each node and link has tuples of properties associated with it. If $D_e = \mathbb{R} \cup \mathbb{B} \cup (\mathbb{R} \otimes \mathbb{R})$ is the set of possible values for properties, we can define two functions, *node* (4.1) and *link* (4.2), to map nodes and links to their respective sets of properties. In general, properties are defined as non-reversible functions of other properties and computed at run-time based on the current values of their parameters.

$$node(n): \quad N \rightarrow D_e^{k_n} \quad , k_n \text{ is the number of properties of node } n \in N \tag{4.1}$$

$$link(l): \quad L \rightarrow D_e^{k_l} \quad , k_l \text{ is the number of properties of link } l \in L \tag{4.2}$$

For the example described above, the sets of properties are:

1. $node(n) = \{(IP = 216.165.111.134), (OS = WinXP), (memory = 200), ...\}$
2. $link(n1, n2) = \{(secure = T), (stable = T), (latency = 10), ...\}$

One of the important features of this model is that it permits the specification of two different classes of properties. In the first class, properties can be *dynamic* or *static*. In the second class, properties can be *classic* or *general* properties.

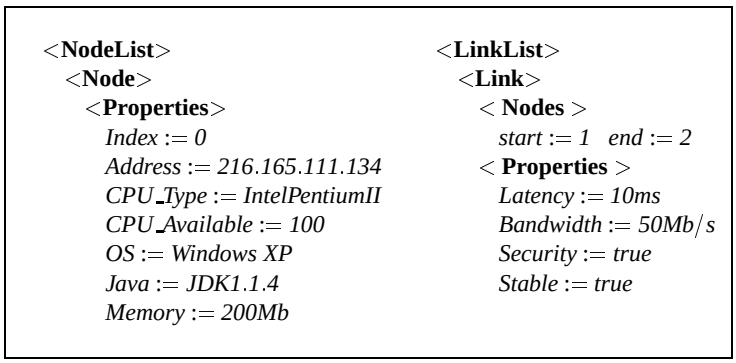
Dynamic properties are associated with network resources that can be consumed, e.g., node CPU, link bandwidth; thus, they are associated with non-negative real values that can be modified at run-time. *Static properties* represent network properties

assumed fixed during the life time of an application, e.g., the security of a link or the trust level of a node.

Most dynamic component-based frameworks take into consideration only *classic* network properties, such as the available CPU of a node and the available bandwidth of a link. However, these properties are not always sufficient to reason about the "best" application configuration given the current state of the network. For example, banking applications might require that links be *secure* and nodes be *trusted*. Thus, a dynamic component-based framework must capture *general* properties, such as security, trust, existing software, nodes owned by the right organization, etc..

Table 4.1 illustrates how the node and link properties described at the beginning of this section are captured by this model.

Table 4.1: Node and link descriptions.



4.3 Application specification model

Chapter 2 has described the challenges of capturing the application behavior and highlighted the need for a model able to describe the component functionality, the conditions and the effects of deploying a component into the network, and the conditions

and the effects of connecting two or more components.

Example As example, this section describes the information that should be captured by the application model for the security-sensitive web-based e-mail application introduced in Section 2.3.1. The e-mail application is defined by a set of six components (`MailClient`, `ViewMailClient`, `MailServer`, `ViewMailServer`, `Encryptor`, `Decryptor`) that implement three interfaces corresponding to the normal and encrypted server interfaces (`MailServerInterface`, `EncryptedMailInterface`), and the client interface respectively (`MailClientInterface`). For each component, the application model should indicate when the component can be deployed on a node and connected to another component. The only component discussed in detail below is the `ViewMailServer`.

The deployment conditions associated with `ViewMailServer` should specify that (1) the node should have enough capacity to serve incoming requests, (2) the number of incoming requests should not exceed a certain maximum, and (3) the component should be able to forward a certain portion of requests to other components.

The effects of deploying the `ViewMailServer` component should capture the interaction between the `ViewMailServer` and the network. For example, they should describe (1) how the `ViewMailServer` consumes the node's CPU capacity, (2) what is the capacity of the `ViewMailServer` to process incoming requests, (3) how the `ViewMailServer` consumes link bandwidth because of the communication with other components, and (4) how the communication is affected by the current state of the environment. Table 4.2 shows the partial specification of the `ViewMailServer` component, which will be used throughout this chapter to illustrate the application model.

Table 4.2: Component/Interface descriptions.

<**Component** name = VMS >	<**Conditions**>
<**Linkages**>	$Node.NodeCPU \geq (MSI^i.NumReq * MSI^i.ReqCPU)$
<**Implements**>	$MSI^r.NumReq \geq (MSI^i.NumReq * MSI^i.RRF)$
<**Interface** name = MSI^i >	$MSI^i.NumReq \leq MSI^i.MaxReq$
<**Properties**>	$MSI^r.Privacy = True$
$MSI^i.Trust - derived$	$MSI^r.Trust \geq 5$
$MSI^i.Privacy - derived$	
$MSI^i.NumReq - derived$	<**Effects**>
$MSI^i.ReqSize - derived$	$MSI^i.Privacy := True$
$MSI^i.RRF := 10$	$MSI^i.Trust := Node.Trust$
$MSI^i.ReqCPU := 2$	$MSI^i.ReqSize := 1000$
$MSI^i.MaxReq := 100$	$MSI^i.NumReq := MIN(MSI^r.NumReq/MSI^i.RRF,$
<**Requires**>	$MSI^i.MaxReq, Node.NodeCPU/MSI^i.ReqCPU)$
<**Interface** name = MSI^r >	$Node.NodeCPU := Node.NodeCPU -$
	$MSI^i.NumReq * MSI^i.ReqCPU$

<**Interface** name = MSI >

<**Crosslink**>

$MSI^d.Privacy := MSI^o.Privacy$ AND $Link.Secure$

$Link.BW := Link.BW - MIN(Link.BW, MSI^o.NumReq * MSI^o.ReqSize)$

$MSI^d.NumReq := MIN(MSI^o.NumReq, Link.BW/MSI^o.ReqSize)$

$MSI^d.ReqSize := MSI^o.ReqSize$

VMS = ViewMailServer, MSI = MailServerInterface Superscripts r and i indicate required and implemented interfaces, o and d correspond to interfaces at link origin and destination.

In order to capture such information as the one described above, the application specification needs to capture (1) the application *functionality*, (2) the *deployment requirements effects*, and (3) the *linkage conditions and effects*. This information should be given per component (i.e., local), and yet be sufficient to reason about the global behavior of the application.

The challenges that need to be addressed when providing the above information are: (1) allowing applications to specify their behavior using general, application-relevant quantitative (e.g., frame rate, operation time, number of messages) and qualitative (e.g., security) terms, (2) allowing application conditions and effects to be captured by sophisticated expressions, and (3) allowing the use of terms from different namespaces when describing the application.

This model addresses the first two challenges by defining applications as sets of *interface types* (I_t) and *component types* (C_t), similar to an object-oriented language such as Java. In addition, the applications can define sets of properties which are associated with interfaces and components, and use these properties to describe the application functionality, requirements, and effects. Each interface and component type can be regarded as a template. The dynamic component-based framework is responsible for instantiating the templates by determining at run-time the expected application behavior. The advantage of dynamically instantiating interfaces and components is the flexibility in creating only instances which are appropriate to the given network state and user requirements.

The solution to the last challenge above is deferred until Section 6.5. In this section, the application and the network properties are considered to belong to the same namespace.

4.3.1 Component and interface types

Each component type $c \in C_t$ is defined as the tuple $c = (I^{impl}, I^{req})$. $I^{impl} \subset I_t$ is the set of *implemented* interfaces and describes the component functionality. $I^{req} \subset I_t$ is the set of *required* interfaces and indicates the services needed by the component to execute correctly. The required interfaces are similar to the Java RMI remote references, where a Java object can specify what types of remote interfaces are needed for correct execution. In the case of the e-mail application, the component types are `ViewMailClient`, `MailClient`, `MailServer`, `ViewMailServer`, `Encryptor`, and `Decryptor`. The last two components represent two instantiations of the cipher module described in Chapter 2. The interface types are (1) `MailClientInterface` implemented by mail clients, (2) `MailServerInterface` implemented by the `MailServer`, `ViewMailServer`, and `Encryptor`, and (3) `MailServerInterface_Encrypted` implemented by `Decryptor`.

Using the information provided by the implemented and required interfaces, a dynamic component-based framework can dynamically determine how to connect the components and create possible application configurations. In general, a component *A* can be connected with a component *B* if component *B* implements some of the interfaces required by component *A*. Figure 4.1 shows how the e-mail application components can be combined in various ways based on their implemented and required interfaces. Every path from the mail clients to the mail server indicates a valid composition of the application. For example, the mail clients (`MailClient` and `ViewMailClient`) can be connected directly to the mail server (`MailServer`), through a cache mail server (`ViewMailServer`) (as might be required to offset high link laten-

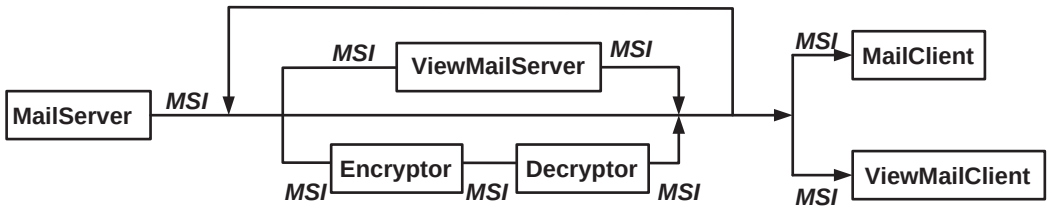


Figure 4.1: Valid application configurations of the e-mail application.

cies), through a pair of cipher modules (`Encryptor` and `Decryptor`) (as they might be required to ensure message privacy), or through any combination of these.

However, the framework cannot take decisions such as placing a cache only if there are high latency links, or placing encryptor/decryptor pairs only if there are insecure links, based on the implemented and required interfaces alone. The application specification must also associate properties with interfaces, as explained below. In such cases, the framework must also verify that the properties of the implemented interfaces satisfy the properties of the required interfaces, before connecting two components.

In addition to implemented and required interfaces, component and interface types are characterized by sets of application-specific *properties*. Let's assume that D_c and D_i represent the sets of possible values for component, respectively interface properties. The component and interface properties, which can be node specific, can be captured by the function *comp* (4.3) and *inter* (4.4), where k_c is the number of properties for component $c \in C_t$ and k_i is the number of properties for interface $i \in I_t$.

$$comp(c,n): \quad C_t \times N \rightarrow D_c^{k_c} \tag{4.3}$$

$$inter(i,n): \quad I_t \times N \rightarrow D_i^{k_i} \tag{4.4}$$

These properties are used as parameters in functions that specify conditions and effects of component instances being deployed on nodes and interface instances crossing links, as explained next.

In the e-mail application example, the application-specific properties include the trust level (`Trust`) and message security (`Privacy`), which indicate, respectively, the maximum message sensitivity level and whether or not the interface preserves message confidentiality. Other properties include the number of incoming requests (`NumReq`), the maximum response size for a request (`ReqSize`), the request reduction factor (`RRF`), the amount of CPU consumed to process each incoming request (`ReqCPU`), and the maximum number of requests that can be processed by the component (`MaxReq`). The `RRF` attribute gives the ratio of requests sent to required interfaces in response to requests on the implemented interfaces. Using these properties, the mail application is able to define the deployment and linkage conditions and effects, as explained below.

4.3.2 Deployment conditions and effects

Deployment conditions define the conditions that need to be true before a component can be deployed on a node. Similarly, the deployment effects capture the presence of the component functionality on a node and the consumption of node resources by the component.

The component model expresses these constraints and effects using in general non-reversible functions involving the associated component type, interface type, and network properties. Specifically, a component is deployed on a node only if the required interfaces are present on the node and there are sufficient node resources. Af-

ter deployment, the implemented interfaces become available on the node and the dynamic properties of the node are altered. Functions $Depl\_Cond$, Eff_{env}, Eff_{interf}, and Eff_{comp} capture the component deployment conditions, and the effects of deploying a component on a node (see Table 4.3). The input for all these functions contains the node properties, and the properties of the required interfaces. Using this notation, a component $c = (I^{impl}, I^{req}) \in C_t$ can be deployed on a node $n \in N$ if $Depl\_Cond(node(n), inter(i^r, n)_+) = true$, where $i^r \in I^{req}$. [2] Similarly, the effects of deploying a component on a node are captured by the functions shown below.

- $node(n) = Eff_{env}(node(n), inter(i^r, n)_+)$ models the network resource properties.
- $comp(c, n) = Eff_{comp}(node(n), inter(i^r, n)_+)$ computes the properties of the component instance running on node n.
- $inter(i^i, n) = Eff_{interf}(node(n), inter(i^r, n)_+))$ calculates the properties of each interface $i^i \in I^{impl}$ implemented by the component.

Returning to the example, the following function from Table 4.2 $Node.NodeCPU \geq (MSI^i.NumReq * MSI^i.ReqCPU)$ expresses the condition that a `ViewMailServer` can be deployed only on a node with sufficient available CPU to serve the required number of requests.

Similarly, $Node.NodeCPU := Node.NodeCPU - MSI^i.NumReq * MSI^i.ReqCPU$ expresses the fact that the `ViewMailServer` component consumes a portion of the CPU resource, once it is deployed on a node.

[2] The $inter(i, n)_+$ notation is shorthand for $inter(i_1, n), inter(i_2, n), \ldots, inter(i_k, n)$.

4.3.3 Linkage conditions and effects

Once a component is deployed on a node, all its implemented interfaces are available on that node. In order to model the fact the application and environment properties influence each other, the *link crossing* operation is introduced. The intuition behind this operation is that connecting two components over a link is equivalent to having the interfaces implemented by one component cross over the link until they reach the other component. Using the link crossing operation, one can compute the properties of an interface on a destination node as a function (in general, non-reversible) of the link properties and the properties of the interface on the source node. Similar functions can describe how the dynamic properties of the link are changed as a result of this operation. In general, the interface functions can be evaluated either (1) from required to implemented interfaces — *publish-subscribe* applications, or (2) from implemented to required interfaces — *request-reply* applications. In *publish-subscribe* applications, servers send data streams to clients. In *request-reply* appli-

Table 4.3: Application conditions and effects.

Component deployment	Link crossing
Conditions	Conditions
$Depl\_Cond: \quad D_e^{k_n} \times D_{i+}^{k_i} \to \mathbb{B}$	$Cross\_cond: \quad D_i{}^{k_i} \times D_e{}^{k_l} \to \mathbb{B}$
Effects	Effects
$Eff_{env}: \quad D_e^{k_n} \times D_{i+}^{k_i} \to D_e^{k_n}$ $Eff_{comp}: \quad D_e^{k_n} \times D_{i+}^{k_i} \to D_c^{k_c}$ $Eff_{interf}: \quad D_e^{k_n} \times D_{i+}^{k_i} \to D_{i+}^{k_c}$	$Eff_{env}: \quad D_i{}^{k_i} \times D_e{}^{k_l} \to D_e{}^{k_l}$ $Eff_{interf}: \quad D_i{}^{k_i} \times D_e{}^{k_l} \to D_i{}^{k_i}$

cations, clients make requests to servers and servers send back replies. Although a dynamic component-based framework could work with both types of applications, this thesis focuses on request-reply applications. If function *Cross_Cond* captures the link crossing conditions, an interface $i \in I_t$ can cross a link $l = (n_1, n_2) \in L$ if the function $Cross\_Cond(inter(i^i, n_1), link(l))$ evaluates to `True`. Functions Eff_{env} and Eff_{interf} describe the effects of the interface on the link and vice-versa.

- $link(n_1, n_2) = Eff_{env}(inter(i^i, n_1), link(n_1, n_2))$ models the effects on link properties.
- $inter(i^r, n_2) = Eff_{interf}(inter(i^i, n_1), link(n_1, n_2))$ models the effects on interface properties.

In the example illustrated by the Table 4.2, the fact that the `MailServerInterface` should cross only links with sufficient available bandwidth is captured by the implicit condition that the available bandwidth remaining after the interface crosses the link is greater than 0:

$$Link.BW := Link.BW - MIN(Link.BW, MSI^o.NumReq * MSI^o.ReqSize) \geq 0$$

4.4 Summary

This chapter has described the application and the network specification models that could be used by a dynamic component-based framework to automatically deploy components into a heterogeneous environment. The application model extends the basic model of component-based applications by allowing user-specified non-reversible expressions to capture the conditions and the effects of component deployments and

linkages, and the interactions between the deployed components and the network. These expressions are defined using component and interface properties as parameters. Unlike other models (CORBA, Web Services, OGSA) which use only standard pre-defined properties such as node CPU and link bandwidth, this model allows the specification of general, user-determined, qualitative (e.g., privacy) and quantitative (e.g., frame rate, trust level) properties. The expressivity of these models is evaluated in Chapter 8.

Chapter 5

Computing Application Configurations

Dynamic component-based frameworks resolve incoming user requests by trying to find application compositions that satisfy users' QoS requirements and application deployment conditions, given the current state of the network. As mentioned in Chapter 2, the problem is to both select the appropriate set of components and map those components onto the network, given that the application specification uses non-reversible functions to define deployment and linkage conditions and effects. One can think of this problem in terms of actions required to realize an application configuration: "place component of a node" and "connect component *A* to component *B*".

This problem can be considered an instance of the classic AI planning problem, where a planning algorithm searches for a set of actions that achieve a goal, given some initial state. However, in the AI space, a similar planning problem is compu-

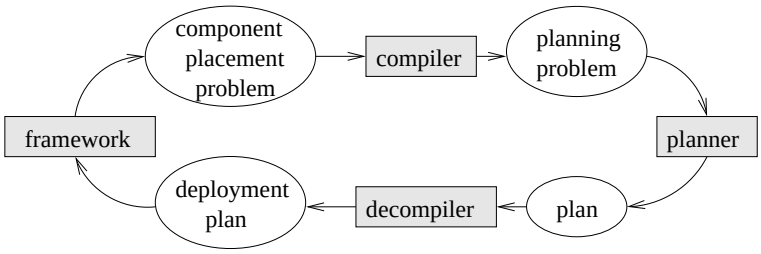


Figure 5.1: Process flow graph for solving ACP.

tationally hard (PSPACE-complete), and complete algorithms, i.e., those that always find a solution if one exists, usually do not scale well. Algorithms achieve good performance on practical problems by effectively pruning different parts of the search space, even though the worst case scenarios take exponential time. In component-based frameworks, scalability concerns stem from two sources: the size of the network, and the number of components. Thus, the planning algorithm needs to deal with two issues: (1) scalability, and (2) complexity of resource expressions.

The following sections describe in detail the planning algorithm developed to solve the problem of finding deployments for component-based applications.

5.1 Structure of the planning algorithm

The problem of finding component compositions which satisfy the user QoS requirements and the application conditions, given the current state of the network, is called the *Application Configuration Problem* (ACP). The module responsible for finding such valid deployments is the *planning module* with the structure shown in Figure 5.1. The compiler module transforms a framework-specific representation of the ACP into an AI-style planning problem, which can be solved by the planner. The

decompiler performs the reverse transformation, converting the AI-style solution into a framework-specific deployment plan.

The next sections explain in detail how each step is performed when searching for a valid application deployment.

5.2 Compiling the ACP into an AI-style planning problem

The traditional STRIPS representation of the AI planning problem is defined by the *initial state of the world*, a set of *operators*, and a *goal* [92]. The initial state contains a complete set of ground literals. The operators represents actions that are available to the planning algorithm and could change the state of the world. Each operator is described with a conjunctive precondition and a conjunctive effect, and defines the transition function from one world to another one. An operator can be executed in any world w that satisfies the precondition function. The result of executing the operator is a new world w' obtained by adding the literals from the effect to the starting world w. The goal is a propositional conjunction of literals. In AI, any world w is good as long as it satisfies the goal formula. This representation can be enriched with numeric measures which capture resources consumed or produced by operators. In this case, a goal is satisfied by a world w if both the propositional conjunction and the resource constraints are satisfied.

The description of ACP can be mapped to an AI planning problem in the following way. The initial state of the world is described by the network topology, the existence of interfaces on nodes (Boolean values), and the availability of resources (real values). The set of operators contains two kinds of elements: (1) <pl_component>(?n)

Table 5.1: Example of AI operator - place ViewMailServer on a node.

```
1 plVMS(?n: node)
2  PRE: avMSI(?n)
3       cpu(?n) > MSIMaxReq * MSIReqCPU
4       numReq(MSI,?n) > MSIMaxReq * MSIRRF
5       sec(MSI,?n) = True
6       trust(MSI,?n) > 5
7  EFF: avMSI(?n), plVMS(?n)
8       numReq(MSI,?n):=MIN( numReq(MSI,?n)/MSIRRF, MSIMaxReq,
9                            cpu(?n)/MSIReqCPU )
10      cpu(?n):=cpu(?n) - numReq(MSI,?n) * MSIRRF * MSIReqCPU
11      sec(MSI,?n):=True
12      trust(MSI,?n):=ntrust(?n)
13      reqSize(MSI,?n):=1000
```

places a component on a node, and (2) <cr_interface>(?n1,?n2) sends an interface across a link. An operator schema has the following sections (line numbers refer to the code fragment given in Table 5.1):

- *logical preconditions* of the operator, i.e., a set of Boolean variables (propositions) that need to be true for the operator to be applicable (line 2);
- *resource preconditions* described by arbitrary functions that return Boolean values (line 3-6);
- *logical effects*, i.e., a set of logical variables made true by an application of the operator (line 7);
- *resource effects* represented by a set of assignments to resource variables (lines 8-13).

The schema shown in Table 5.1 describes the placement of the ViewMailServer (VMS) component on a node. The preconditions result from the conditions in Table 4.2 and the fact that MailServerInterface (MSI) is a required interface. The

effects come from the effects section of Table 4.2, with `MaxReq` providing the upper bound on the `NumReq` parameter of the implemented interface.

Given the operator definition above, the compilation of the ACP into a planning problem is straightforward. For each component type and node, the compiler generates an operator schema for a placement operator. In addition, an operator for link crossing is generated for each interface type and link. One of the attractive features of the compiler is that it generates operators on demand, i.e., only if they are necessary during planning. The initial state is created based on the properties of the network. The goal of the ACP is translated into a Boolean goal of the planning problem.

5.3 The planning algorithm

In ACP, the size of the network and the number of components affect the number of operators in the compiled problem. Since often practical problems do not require the use of all possible operators, what distinguishes a good ACP solution is its ability to scale well in the presence of large amounts of irrelevant information. The solution combines multiple AI planning techniques and exploits the problem structure to drastically reduce the search space.

The intuition behind the planning algorithm proposed in this chapter is to start first from the goal and search for a subset of operators that achieve the goal; then, search in this subset for operators achievable from initial state that reach the goal. By breaking up the search into this two-step process, the search is focused only on the subset of operators that are relevant for achieving the desired goal. Because the resource constraints are in general non-reversible functions, the algorithm verifies whether the

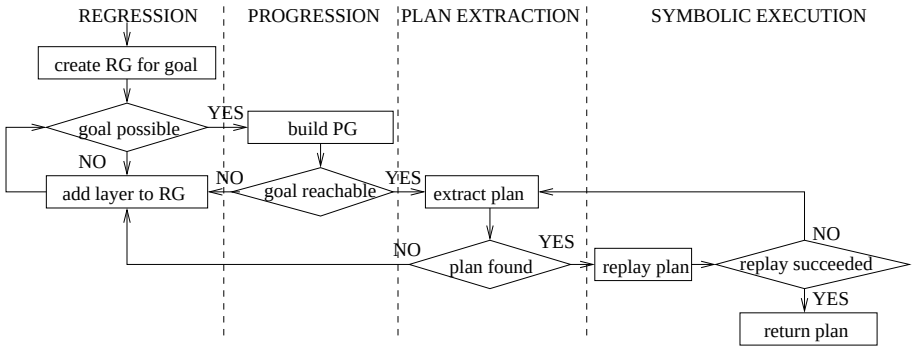


Figure 5.2: The algorithm. RG stands for “regression graph”, PG for “progression graph”

resource constraints are satisfied during this second search, instead of the first one. If no plan is found, the algorithm goes back to the first step and increases the subset of operators.

The algorithm uses two data structures: a *regression graph* (RG) and a *progression graph* (PG). RG contains operators relevant for the goal. An operator is *relevant* if it can participate in a sequence of actions reaching the goal, and is called *possible* if it belongs to a subgraph of RG rooted in the initial state. PG describes all world states *reachable* from the initial state in a given number of steps. Only possible operators of the RG are used in construction of the PG. The algorithm consists of four phases shown in Figure 5.2 and described below.

1. First, a regression phase determines what operators are relevant for the goal. An operator is *relevant* if it can participate in a sequence of actions reaching the goal. The set of relevant operators is further reduced to a set of *possible* operators. An operator is possible if it is relevant and belongs to a subgraph of the regression graph rooted in the initial state.

2. Second, a progression phase finds all states possibly reachable from the initial state given the set of possible operators, thus performing some additional pruning of the regression graph.

3. Third, an exhaustive search in the resulting graph is performed to achieve completeness. An algorithm is complete if it finds a solution when a solution exists.

4. Plans found in the third phase are symbolically executed to ensure soundness in the presence of non-reversible expressions in resource preconditions and effects. If no plans are found, the algorithm re-iterates from the first step by adding one more layer of operators.

Regression phase. The regression phase considers only logical preconditions and effects of operators in building the RG, an optimistic representation of all operators that might be useful for achieving the goal. RG contains interleaving facts and operator levels, starting and ending with a fact level, and is constructed as follows.

- Fact level 0 is filled in with the goal.
- Operator level i contains all operators that achieve some of the facts of level $i-1$.
- Fact level i contains all logical preconditions of the operators of the operator level i.

RG is initially constructed until the goal becomes possible, but may be extended if required.

Progression phase. RG provides a basis for the second phase of the algorithm, the construction of the progression graph. This graph describes all states reachable from the initial state in a given number of steps. PG also contains interleaving operator and fact levels, starting and ending in a fact level. In addition, this graph contains information about mutual exclusion (mutex) relations [59], e.g., that the placement of a component on a node might exclude placement of another component on the same node (because of CPU capacity restrictions). Because of this reason, the PG is less optimistic than the RG.

- Fact level 0 contains facts true in the initial state.
- For each of the propositions of level $i-1$ a copy operator is added to level i that has that fact as its precondition and effect, and consumes no resources (marked with square brackets in the figure).
- For each of the possible operators contained in the corresponding layer of the RG, an operator node is added to the PG if none of the operator's preconditions is mutex at the previous proposition level.
- The union of logical effects of the operators at level i forms the i^{th} fact level of the graph.
- Two operators of the same level are marked as mutex if (1) some of their preconditions are mutex, (2) one operator changes a resource variable used in an expression for preconditions or effects of the other operator, or (3) their total resource consumption exceeds the available value.

- Two facts of the same level are marked mutex if all operators that can produce these preconditions are pairwise mutex.

Since the progression graph is less optimistic than the regression graph, it is possible that the last level of the PG does not contain the goal, or some of the goal propositions are mutually exclusive. In this case a new step is added to the RG, and the PG is reconstructed.

Plan extraction phase. If the PG contains the goal and it is not mutex, then the plan extraction phase is started. This phase exhaustively searches the PG [8], using a memoization technique to prevent re-exploration of bad sets of facts in subsequent iterations.

Symbolic execution. Our work supports non-reversible functions in resource preconditions and effects. For this reason, symbolic execution is the only way to ensure soundness of a solution. It is implemented in a straightforward way: a copy of the initial state is made, and then all operators of the plan are applied in sequence, their preconditions evaluated at the current state, and the state modified according to the effect assignments. Note that correctness of the logical part of the plan is guaranteed by the previous phases; here, only resource conditions need to be checked.

5.4 Decompiling the AI-style solution into a PSF-specific solution

The plan is a sequence of operators: (1) place component on a node `pl<component><node>` and (2) cross link with an interface

(cr<interface><from><to>). For example, plMSn2 represents the action of placing the MailServer on node N_2, and crMSIn2n1 indicates that MailServerInterface crosses the link between nodes N_2 and N_1.

Given a set of such operators, it is straightforward to obtain a framework-specific deployment plan, which consists of (*component*, *node*) pairs and linkage directives. An example of the latter is (MS,n2,MSI,VMS,n1): send the MailServerInterface implemented by the MailServer component located on node n2 to the ViewMailServer component on node n1.

5.5 Example execution of the planning algorithm

To illustrate the planning algorithm, Figure 5.3 describes a simple scenario of deploying the e-mail application components into a heterogeneous network.

The initial state of the ACP problem contains the e-mail application components described in Chapter 2 and a network with three nodes (N_0, N_1, and N_2) and three links. Nodes N_0 and N_1 are connected by a fast and secure link. Nodes N_0 and N_2 are connected through a slow and insecure link, while the link between nodes N_1 and N_2 is secure, but slow. The goal of the ACP problem is to allow a user executing on node N_0 to efficiently and securely access the MailServer running on node N_2.

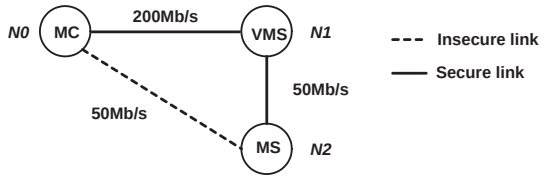


Figure 5.3: Component deployment.

The only possible solution is to deploy the `MailClient` on node N_0 and connect it to the `MailServer` through a `ViewMailServer` deployed on node N_1. Directly linking the `MailClient` to the `MailServer` is not possible because the link between them does not have enough available bandwidth to satisfy the `MailClient` requirements. In addition, the direct link is considered unsafe.

The component (`MailClient` and `ViewMailServer`) and interface (`MailServerInterface`) specifications were discussed in Chapter 4. Tables 5.2 and 5.3, together with Table 5.1, illustrate the way the planning algorithm compiles the specifications into operators.

Figure 5.4 shows the RG built by the planning algorithm in order to solve the problem described above, given the original set of operators. Bold, solid, and dashed lines correspond to possible subgraphs with 3, 4, and 5 steps respectively. Figure 5.5 shows the progression graph corresponding to the regression graph. Straight lines show relations between propositions and operators, and the dotted arc corresponds to a mutex relation. For example, there exists a mutex relationship between `crMSIn2n1` (i.e., crossing `MailServerInterface` from N_2 to N_1) and `crMSIn1n2` (i.e., crossing `MailServerInterface` from N_1 to N_2) because their preconditions are in conflict.

Table 5.2: AI operator - place `MailClient` on a node.

```
1 plMC(?n: node)
2  PRE: avMSI(?n)
3       cpu(?n) > MSIMaxReq * MSIReqCPU
4       numReq(MSI,?n) > 100
5       sec(MSI,?n) = True
6       trust(MSI,?n) > 5
7  EFF: plMC(?n)
8       cpu(?n):=cpu(?n) - numReq(MSI,?n) * MSIReqCPU
```

Table 5.3: AI operator - cross link with `MailServerInterface`.

```
1 crMSI(?n1: node,?n2: node)
2  PRE: avMSI(?n1)
3       bw(?n1,?n2) > numReq(MSI,?n1) * reqSize(MSI,?n1)
4  EFF: avMSI(?n2)
5       bw(?n1,?n2):=bw(?n1,?n2) - numReq(MSI,?n1) * reqSize(MSI,?n1)
6       sec(MSI,?n2):= sec(MSI,?n1) && sec(?n1,?n2)
7       reqSize(MSI,?n2):=reqSize(MSI,?n1)
```

The graphs are constructed as described below.

The RG starts with the goal `placedMCn0` and searches for all operators that achieve this goal (Level 0). The only eligible operator is to place `MailClient` on node N_0 (`plMCn0`), and it requires that `MailServerInterface` be available on node N_0 (`avMSIn0`) (Level 1). There are two possible ways of having `MailServerInterface` available on that node: cross the `MailServerInterface` from node N_2 or from node N_1; however, both ways require that `MailServerInterface` is available on those two

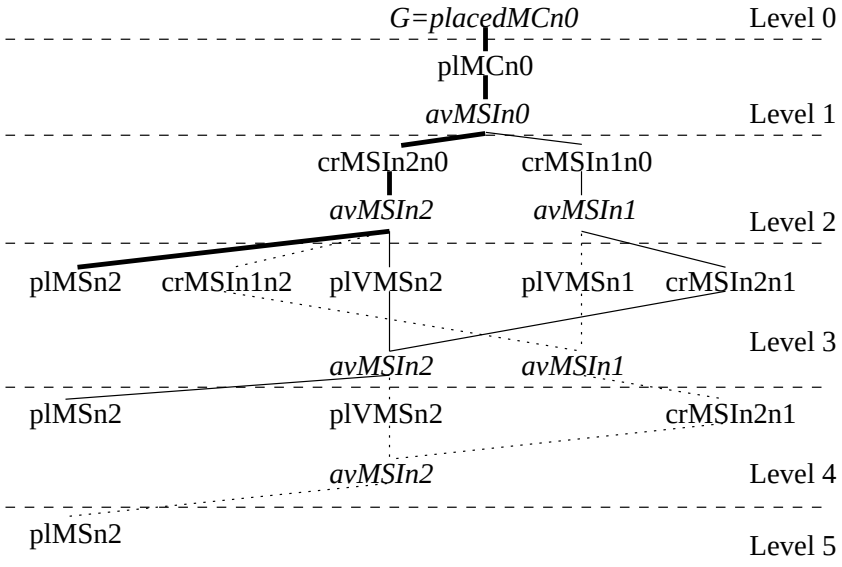


Figure 5.4: Regression graph.

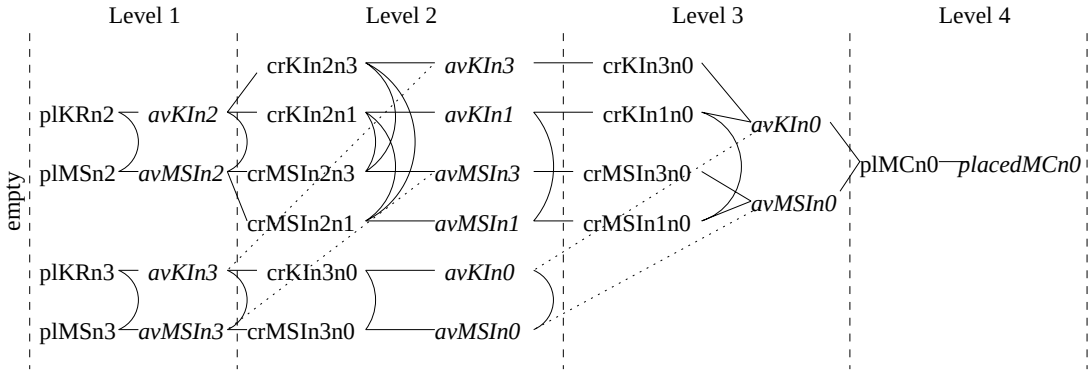


Figure 5.5: Progression graphs.

nodes (`avMSIn2` and `anMSIn1`) (Level 2). Interface `MailServerInterface` is available on node N_2 if (1) the `MailServer` is placed on node N_2, (2) the interface crosses from node N_1, or (3) the `ViewMailServer` component is placed on node N_2. Similarly, `MailServerInterface` is available on node N_1 if the `ViewMailServer` component is placed on node N_1 or the `MailServerInterface` crosses from node N_2 (Level 3). At this moment, the RG reached a point where the operator `avMSIn2` is achievable from the initial state, because `MailServer` is running on node N_2. Thus, a possible solution is to directly connect the `MailClient` on node N_0 to the `MailServer` on node N_2. However, the planning algorithm determines that this is not a valid solution because the security constraint is not satisfied.

In this case, the algorithm re-iterates by adding an extra step to the RG. In order to execute any of the operators from Level 3, the `MailServerInterface` interfaces needs to be available on nodes N_2 and N_1. Level 4 is similar to Level 3, and will not be described again. However, in Level 4, the RG reaches again a state achievable from the initial state, `plMSn2`. The new plan is to connect the `MailClient` on node N_0 to

the `MailServer` on node N_2, where the connection is routed through node N_1. Such a plan respects the security requirements of the `MailClient`, but violates the efficiency requirements. The efficiency of the connection is limited by the worst link, in this case (N_0, N_1). Thus, the plan is not valid and the algorithm needs to re-iterate.

In Level 5, RG reaches once more the initial state and the new plan is to place a `ViewMailServer` on node N_1, connect the `ViewMailServer` to the `MailServer`, and connect the `MailServer` on node N_0 to the `ViewMailServer`. This plan satisfies both the security and efficiency requirements of the `MailServer` and represents the output of the planning algorithm.

5.6 Limitations of the planning algorithm

The ACP planning problem is complex and, although the planning algorithm described in this chapter is sufficient for the example applications motivating this work, several additional challenges must be addressed in order to obtain an ideal solution.

- The planning algorithm does not support formulae involving parameters of implemented interfaces; instead, it generates a conservative solution by using upper bounds on values of such parameters. These upper bounds are computed during the progression step.

- The planning algorithm uses the level-based expansion of the RG/PG to optimize only the number of steps required to achieve a goal. Ideally, it should optimize more complex application-specific cost functions.

- The solution returned by the planning algorithm is greedy, meaning it consumes

the maximum of the resources available. Ideally, the algorithm should plan in such a way that the solution consumes only the minimum necessary resources.

- The algorithm's response is always Boolean: it returns a plan if one exists, or none otherwise. A more desirable approach would be for the algorithm to negotiate with the user in case the user requirements cannot be satisfied.

- The planning algorithm plans only one request at a time. This prevents the planner from providing generally optimal solutions. A possible solution would be to allow the planner to aggregate several incoming requests and plan accordingly.

Some of these shortcomings are starting to be addressed by other researchers (e.g., see [86]).

5.7 Summary

This chapter has presented a planning algorithm able to deploy component-based applications into heterogeneous environments, while dealing with scalability and arbitrary resource functions. The novel feature of this algorithm is that it combines and solves in one step both the component composition and the component mapping problems, instead of dealing with them separately. What differentiates this algorithm from similar algorithms is its ability to scale with the size of the network and its support for non-reversible resource functions and application structures. Chapter 9 presents a detailed evaluation of the planning algorithm.

Chapter 6

Deploying Component-Based Applications: Efficiency and Practicality

This chapter starts by briefly re-iterating the challenges faced by dynamic component-based frameworks when deploying distributed applications in heterogeneous environments, and then presents a solution based on the notion of *views*. Section 6.3 defines views as component customizations. The next three sections describe how views improve the automatic deployment process by increasing the chances of successful planning, and providing efficient support for satisfying security and data consistency guarantees.

6.1 Challenges of the deployment process

As explained in Section 2.2.3, the challenges of achieving a practical and efficient deployment process include the facts that: (1) the original set of components might not be always sufficient to find a valid configuration that satisfies the application requirements, and (2) the efficiency of automatically deployed applications might degrade because of application requirements for security and data consistency guarantees.

Providing security guarantees when deploying components across multiple administrative domains is challenging because there is no centralized trusted third party to act as mediator between participants, and the security decisions must be taken when only partial knowledge about domains is exposed. In addition, the authorization process ideally should guarantee single sign-on, fine-grained, and customizable access control to resources.

The data consistency problem arises when several replicas sharing data are deployed into the network. In general, component-based applications dynamically adapt to environment and client QoS changes, thus potentially modifying the application consistency requirements. Therefore, a dynamic component-based framework must implement a data consistency protocol that is flexible, application-neutral, but still capable of using application-specific information to improve its efficiency.

6.2 Overview of the solution

Views represent one possible solution to addressing all of these challenges. Views [65] were originally introduced in the context of parallel programming languages support-

ing a shared object space. In that context, views allowed the reduction of coherence traffic by defining a coherence granularity smaller than the object and encapsulating application-specific protocols. Views can be more generally thought of as *customizations* of components providing the needed flexibility to generate configurations that satisfy application requirements. They allow the application developer to specify the appropriate access control granularity and capture the application information necessary in efficiently maintaining data consistency. In order to support the dynamic deployment of views, this thesis proposes a run-time infrastructure built atop a *decentralized role-based access control and trust management system* and an application-neutral *data consistency protocol*.

The role-based access control and trust management system provides an integrated solution to cross-domain authentication, access control, and translation between local constraint specifications. The latter represents a novel use of a classic technique: properties are viewed as credentials belonging to a local domain and the translation between two properties is equivalent to finding a chain of credentials staring from the former and finishing with the latter.

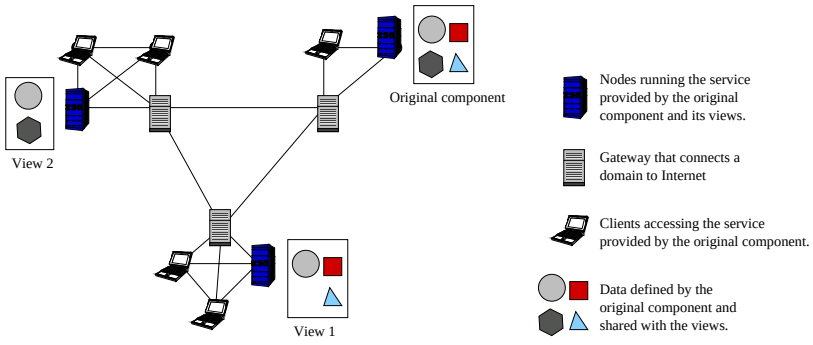


Figure 6.1: Using views during deployment.

The data consistency protocol maintains consistency between different views of the same component. The novel feature of the consistency protocol is that it satisfies the consistency requirements of component-based applications (application-neutral) deployed in various configurations (flexible), while using application-specific information embodied in the view specification. The data consistency traffic is minimized by allowing the application to specify (1) *data properties* to characterize the shared data, (2) *quality triggers* to indicate when updates need to be pushed or pulled between views, and (3) *merge/extract methods* to merge/extract updates from/into views and original components [50], all using an application-neutral protocol.

Figure 6.1 illustrates one example of using views. In this example, the application consists of only one component that can be replicated as desired, and several clients who want to access the functionality of this application from three domains. The main characteristics of this component are that: (1) the data, which is represented in the figure by the four geometric shapes, used by the component needs to be kept consistent among replicas, and (2) the component consumes a large amount of resources while running. In order to satisfy the user QoS requirements, a framework might decide to instantiate the component in each domain. However, such a decision might not always be realizable. First, the network resources might not be sufficient to satisfy the component resource constraints. Second, from a security point of view, a client would get access to the entire component, even if he might not have the appropriate rights. Third, the application efficiency might degrade because of the cost of maintaining consistency of the component state. A different approach is to automatically customize the original component based on the resource availability and/or client credentials, thereby creating new components with different properties and requirements,

i.e., views. In Figure 6.1, the views are using subsets of the original data. Using such views may prove beneficial in this example scenario because they can (1) consume fewer resources than their "parent", (2) enforce the appropriate access control to the component functionality and data, and (3) reduce the consistency traffic by defining a smaller coherence granularity than the entire object.

6.3 Views - Component customizations

6.3.1 Definition

Views provide a mechanism by which to define multiple physical realizations of the same logical component. A component is a *view* of another component, called *original component*, if it shares (1) functionality or (2) data with the original component. For a more formal definition, let's assume that C_t is the set of component types belonging to an application, and a component $c \in C_t$ implements a set of methods F_c and defines a set of public variables V_c. A new component $v \in C_t$ is a *view* of the original component c if the view has at least one of the following two properties: (1) $F_v \cap F_c \neq \varnothing$, and (2) $V_v \cap V_c \neq \varnothing$. We refer to the former as *object views*, and to latter as *data views*. In practice, most views are likely to exhibit characteristics of both object and data views.

In the context of dynamic component-based frameworks, views define a family of auxiliary components that embody different ways of realizing the component functionality. In general, the functionality of the original component can either be completely replicated in the auxiliary component, be completely present in the original component (with the auxiliary component just serving as a gateway to this function-

ality), or be somewhere between these two extremes.

Table 6.1: The original Java object.

```
public interface MessageI {
  void sendMessage(Message m)
  Set receiveMessages() }

public interface AddressI {
  String getPhone(String name)
  String getEmail(String name) }

public interface NotesI {
  void addNote(String note)
  boolean addMeeting(String name) }

public class MailClient implements
  MessageI, AddressI, NotesI {
  Account[] accounts;
  void sendMessage(Message mes){}
  Set receiveMessages(){}
  String getPhone(String name){
    findAccount(name).getPhone();
  }
  String getEmail(String name){
    findAccount(name).getEmail();
  }
  void addNote(String note){}
  String addMeeting(String name){}
  Account findAccount(String name){
  { return accounts.get(name); }
}
```

Table 6.2: The view specification.

```
<View>
  <Name = ViewMailClient_Partner>
  <Represents>
    <Name = MailClient>
  <Restricts>
    <Interface> name=MessageI
                type=local
    <Interface> name=NotesI
                type=rmi
    <Interface> name=AddressI
                type=switch
  <Adds_Fields>
    <Field name = accountCopy
  <Adds_Methods>
    <MSign> VMC_Partner()
    <MBody> /** constructor body **/
    /** Additional methods, including some
    required by the data consistency
    protocol (see Section 6.6)
    **/
  <Customizes_Methods>
    <MSign> addMeeting(String)
    <MBody> /** new code for method**/
```

6.3.2 Specifying views

There are several possible ways of obtaining the information required for constructing a specific view. The approach adopted in this work is based on annotations, where users can identify which parts of an original component need to be copied, added, or modified into the view.

Table 6.3: View source code.

```
public interface MessageI {
  void sendMessage(Message mes)
  Set receiveMessages()
}

public interface AddressI extends Serializable {
  String getPhone(String name)
  String getEmail(String name)
}

public interface NotesI extends Remote {
  void addNote(String note) throws RemoteException
  boolean addMeeting(String name) throws RemoteException
}

public class ViewMailClient_Partner implements
  MessageI, AddressI, NotesI {

  Account[] accounts; NotesI notesI_rmi;
  AddressI addrI_switch;

  public ViewMailClient_Partner (String[] args) {
    /** rmi code **/
    notesI_rmi = (NotesI) Naming.lookup(...);
    /** switchboard code **/
    addrI_switch = (AddressI) Switchboard.lookup(...);
    /** user supplied code **/
  }

  void sendMessage(Message mes){/** the original code **/}
  Set receiveMessages()        {/** the original code **/}
  String getPhone(String name){return addrI_switch.getPhone();}
  String getEmail(String name){return addrI_switch.getEmail();}
  void addNote(String note)   {notesI_rmi.addNote(); }
  boolean addMeeting(String name){/** user supplied code **/}
}
```

For a better illustration of the view concept, Table 6.2 shows a simple schema that can be used as a guideline, using as example a component from the mail application given in Table 6.1. The `ViewMailClient_Partner` is a restricted version of the `MailClient` component, able to send/receive messages, add notes into a remote diary, and query the address book in a secure fashion. Such a component is useful if clients use untrusted machines (e.g., an airport terminal) to check their e-mail. This example assumes that the components are written in Java. A minimal view is fully described by a name (`ViewMailClient_Partner`), and a represented component (`MailClient`). The minimal view can be enriched by providing a list of implemented interfaces, such as `MessageI`, `AddressI`, `NotesI`, defining new methods and fields, and copying or customizing existing methods. For each interface, the view description can specify a type (*local*, *rmi*, or *switch*) to indicate how the interface is available to clients. The methods defined by a *local* interface should be available only to clients running in the same JVM as the view. Interfaces can be also required to be only available on the original component. Access to such interfaces is permitted either via RMI (*rmi*) or a secure communication channel called Switchboard (*switch*), as described in Section 6.5. As part of the view definition, the user can also specify methods required for data consistency, as discussed in Section 6.6.

The actual generation of the code for a view is deferred to the time this view is first deployed. This ensures that despite their flexibility, views incur management costs proportional to their utility.

6.3.3 View instantiation

VIG (View Generator) is a tool developed to automatically generate Java-based views. VIG takes as input the class file of the original component and the definition of the view given as a set of rules. The output is a new classfile corresponding to the view.

Table 6.3 shows the Java code corresponding to the view bytecode, generated from the original component from Table 6.1 and the set of rules from Table 6.2.

VIG extends the API's provided by the Javassist [85] toolkit to manipulate Java objects at the bytecode level. The view generation process consists of two steps: (1) reading the view description and the represented component, and (2) modifying existing method/interfaces and adding new methods to the view. If the view already exists, VIG does not attempt to recreate the view. If VIG is unable to generate correct bytecode (e.g., a new method uses a variable that is not defined in the original component or the method), it triggers an error that indicates how the rules can be rectified. Therefore, VIG can be used to both generate views at runtime and guide the programmer's effort to correctly write the view rules. The grammar of the view definition rules (Table 6.4) is very simple, but expressive enough to comprise a large set of operations. In general, a view is specified by the *name* of the view and the *original class*. However, such a minimal view can be enriched with a set of *imported packages*, possible *extended classes/views*, and a list of *operations* (e.g., add/copy/customize fields/methods/interfaces). The following paragraphs enumerate and describe in detail each operation described by a rule. All line numbers refer to the Table 6.4. A more complete description of the VIG tool is given in Appendix A.

Table 6.4: Grammar for XML-based description of views.

```
1.    ViewDefinition := <views> ViewRules </views>
2.    ViewRules := ViewRule | ViewRules
3.    ViewRule := <view> ViewName RepresentedClass ImportedPackages
                  ExtendedClass ExtendedView RestrictedInterfaces
                  CopyList AddList CustomizeList </view>
4.    ViewName := <name> ViewClassName </name>
5.    ViewClassName := String
6.    RepresentedClass := <represents> RepresentedClassName </represents>
7.    RepresentedClassName := String
8.    ImportedPackages := ImportedPackages | ImportedPackage | empty
9.    ImportedPackage := <imports> ImportedPackageName </inports>
10.   ImportedPackageName := String
11    ExtendedClass := <extends> ExtendedClassName </extends> | empty
12.   ExtendedClassName := String
13.   ExtendedView := <vextends> ExtendedViewName </vextends> | empty
14.   ExtendedViewName := String
15.   RestrictedInterfaces := <restricts> InterfaceList </restricts> |
                              empty
16.   InterfaceList := InterfaceList | Interface
17.   CopyList := <copy> CopyFields CopyMethods </copy> | empty
18.   CopyFields := CopyFields | Field | empty
19.   CopyMethods := CopyMethods | Method | empty
20.   AddList := <add> AddFields AddMethods AddConstructors
                 AddInterfaces </add> |empty
21.   AddFields := AddFields | Field | empty
22.   AddMethods := AddMethods | Method | empty
23.   AddConstructors := AddConstructors | Constructor | empty
24.   AddInterfaces := AddInterfaces | Interface | empty
25.   CustomizeList := <modify> CustomizeMethods </modify> | empty
26.   CustomizedMethods := CustomizeMethods | Method
27.   Field := <field> FieldDeclaration </field>
28.   FieldDeclaration := String
29.   Method := <method> MethodDeclaration </method>
30.   MethodDeclaration := String
31.   Constructor := <constructor> ConstructorDeclaration </constructor>
32.   ConstructorDeclaration := String
33.   Interface := <interface name= InterfaceName type= InterfaceType > |
                    <interface name= InterfaceName >
34.   InterfaceName := String
35.   InterfaceType := String
```

View name (lines 4-5). The name of the view is a mandatory field and should contain the fully qualified name of the view. (e.g., `mail.client.ViewMailClient`).

Original class name (lines 6-7). The name of the original component is a mandatory field and should contain the fully qualified name of the original class (e.g., `mail.client.MailClient`).

Imported packages (lines 8-10). Similar to the Java language, VIG allows the use of import declarations to specify the location of used methods and fields. Whenever VIG encounters a method or a field that is not defined in the original class or its super classes, VIG searches the list of imported packages for the class that declares the field or the method.

Extended class (lines 11-12). Similar to the Java language, a view can extend one Java class. The name of the extended class should be a fully qualified name.

Extended view (lines 13-14). In addition, VIG allows a view v to extend another view v'. The only condition is that the original class of the extended view v' is either the same or is extended by the original component of the initial view v. The view extension consists of three steps: (1) generating the extended view v', if the extended view does not exist, (2) adding the descriptions of the two views $v'' = v + v'$, and (3) generating the newly formed view. The process iterates if the extended view extends another view. It is important to note that a view cannot extend both a class and another view.

Restricted interfaces (lines 15-16). The list of restricted interfaces indicates how the functionality of the view is a subset of the functionality of the original component. A restricted interface is specified by the fully qualified name and the type of the interface. The type can be *local*, *rmi*, or *switch*. The default type is *local*. *Local* interfaces represent that part of the functionality of the original component that could be executed on a remote node. Thus, local interfaces do not require any processing, and can be copied as is. *rmi* and *switch* interfaces represent the functionality of the original component that should be accessed only by remote invocation. Such interfaces are transformed by VIG by extending `java.rmi.Remote`, respectively `java.io.Serializable`. In addition, all methods defined by interfaces are annotated by VIG as *local*, *rmi*, or *switch*, depending on their corresponding interfaces. The processing of the actual method implementations is described below.

Add interfaces (line 24). VIG is responsible only for adding the given interfaces into the view declaration. The programmer needs to ensure that all the methods declared by the interfaces are also implemented in the view. The prototype VIG implementation does not check whether all methods have implementations.

Copy fields (line 18). VIG looks up the class that contains the field declaration and copies the field into the view. In general, fields are added either because they are used by a method, or because they are declared by the view description. Similar to the case of methods, the only problem is finding the correct field declaration. VIG solves this problem by following the inheritance chain.

Add fields (line 21). VIG adds the field to the view, if the field is not already defined in the view.

Copy methods (line 19). As in the case of interfaces, methods can be defined as *local*, *rmi*, or *switch*. Methods are defined as *local* when they can be copied and locally executed on remote nodes. This is useful when methods are compute-intensive and do not process sensitive data. Methods are annotated as *rmi* and *switch* when they should always be accessed through remote invocations. For example, methods that implement a secret algorithm or process secure data should not be executed on remote, untrusted nodes.

Methods defined as *local* or by *local* interfaces are copied from the represented component into the view. The bodies of the methods defined by *rmi* or *switch* interfaces are transformed into simple RMI, respectively Switchboard calls against the original component. The main problem when copying existing methods from the original class into the view is to find the correct implementation. This problem arises when there is an inheritance chain from the original component to a unique super class. VIG solves this problem by following the inheritance chain and looking for the right implementation. Once the right method implementation is found, VIG parses the method code and recursively copies the declarations of all used class fields and invoked local methods.

Add methods (line 22). In order to add a new method, VIG extracts the method signature and body from the view description. The actual addition is simplified by the fact that Javassist allows the correct insertion of pure Java code into the view

bytecode. [1]

Add constructors (line 23). The view should always have at least the default constructor. In addition, one can define other constructors. The process of adding a constructor is identical to adding a method.

Customize methods (line 26). Customizing a method is equivalent to a combination of copying and adding methods. First, VIG searches for the correct implementation of the given method. Next, it creates a new method by changing the body of the found method with the new body. Then, VIG adds the new method to the view.

In addition to generating the bytecode for views, VIG can be also used as a debugging tool when writing the rules. There are two common checks executed by VIG before each operation. The first one triggers a `BadFormatException` if the field or the method to be added/copied/customized already exists in the view. The second check is triggered as a consequence of VIG trying to find the correct definition of a given field or method. VIG searches, in order, the original component, the super classes of the original component, and the classes belonging to any of the imported packages. If the search is successful, VIG uses that field or method declaration to perform the operation. Otherwise, VIG triggers a `BadFormatException`. The programmer can use the feedback provided by VIG to correct the rules.

As mentioned before, views offer three advantages in the context of dynamic component-based frameworks: (1) improving the likelihood of successful component

[1] Javassist verifies the syntax of the added Java code.

deployment, (2) providing a finer granularity at which to authorize and enforce access control, and (3) minimizing the data consistency traffic by defining smaller granularities than the entire component. The following three sections explain in more detail each of these advantages.

6.4 Using views to improve chances of successful planning

Dynamic component-based frameworks work with a set of reusable components, selecting among and customizing these components as appropriate for the application and client QoS requirements. Whether or not the planning module is in fact able to come up with a deployment schedule is dependent on the set of available component types.

Views provide a convenient mechanism for enriching the set of components, without requiring onerous application developer input. By merely distributing component functionality between the original and auxiliary objects, views increase the likelihood of the planner finding a component deployment in constrained environments.

Additional flexibility arises from allowing view properties to be configured at creation time. For example, the `Privacy` property of a newly created view is set to `True`, if the view definition adds cryptographic mechanisms to protect the outgoing messages. Such properties can be specified in the same way as the original component properties (see Chapter 4).

6.5 Using views to satisfy security guarantees

Views help to satisfy security guarantees by providing a flexible mechanism to restrict access control to original components. In order to better understand how views can be used to provide security guarantees, Figure 6.2 illustrates the security requirements in a dynamic component-based framework by contrasting an insecure deployment process with its secure version. The shaded boxes represent the additional steps which transform an insecure deployment process into a secure one. Clients requesting access to an interface (which triggers new deployments or attachment of the clients to existing components) must first be authenticated and then authorized to receive the appropriate level of service. Based on the client credentials, the framework must enforce the appropriate level of access. Then, the planning module takes into consideration the client credentials, the component credentials, and network resource credentials to generate a valid deployment that achieves the desired level of service. If such a

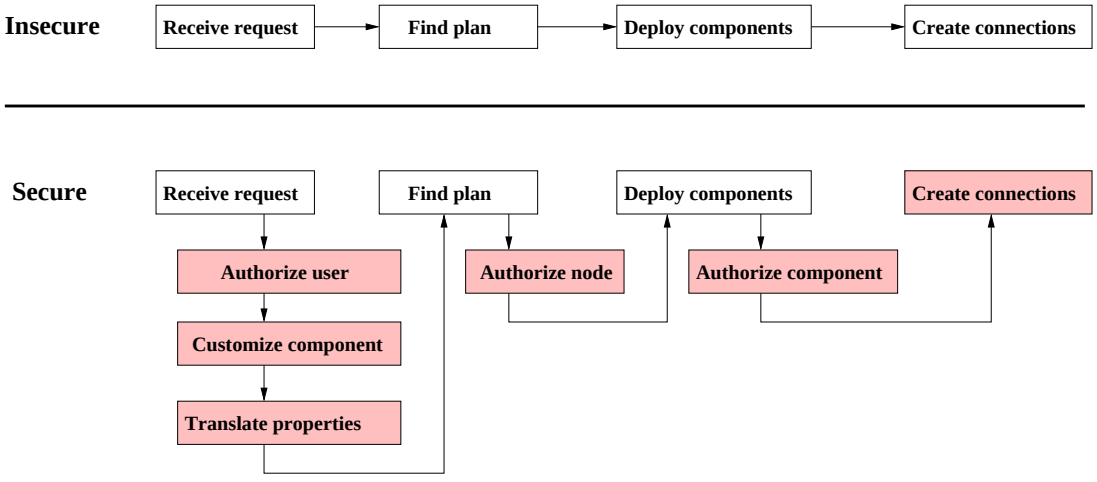


Figure 6.2: Distributed application deployment protocol - Secure vs. insecure.

deployment exists, the framework deploys the components on the appropriate nodes and makes the necessary connections between components. This requires that the components and the network resources authorize each other: a node is authorized to host a component, and a component is authorized to execute on a node. Additionally, deployed components may make their own requests for their required interfaces, which triggers this process recursively. In order to ensure that the communication between components is private, the framework creates connections on top of a secure communication abstraction (e.g., SSL or Switchboard).

The additional steps of the secure deployment process can be divided into four distinct classes: (1) *cross-domain authorization* of entities (i.e., users, components, nodes), (2) *translating environment properties* into properties meaningful for the application, (3) *enforcing the appropriate access control*, and (4) creating *secure channels* between components. First, the following sections introduce the mechanisms employed in order to achieve these goals. Then, their use is illustrated in the context of the e-mail application example in Section 6.5.5.

6.5.1 Authorizing entities across domains

As described in Chapter 2, dynamic component-based frameworks need to deal with the problem of authorizing entities across multiple administrative domains, when each domain is allowed to define its own space of credentials and only make public a subset of this space. One approach to this problem, advocated by systems such as Globus Grid [36, 61], is to translate between a system-wide "grid credential" (virtual organization-level credential in CAS) and local accounts to authorize and enforce security policy for client requests.

This section proposes a solution to this problem, which generalizes this approach. The solution is based on dRBAC, the distributed role-based access control system and trust management system described in Section 3.3.3. dRBAC offers advantages of scalability (multiple policy roots are permitted), easier configuration (local policy need not include translation between grid and local credentials, which is automatically inferred), and finer-grained control (the rights afforded a request can be modulated to the credentials associated with it as opposed to the account these translate to). A detailed discussion of the latter advantage is deferred to Section 6.5.3.

dRBAC provides mechanisms by which each administrative domain can issue independent credentials to its clients, components, and network resources, and yet these credentials can be combined to permit cross-domain authorization decisions. The latter is enabled using dRBAC delegations, which provide a mechanism for mapping roles in other domains to roles in the current one. This allows domains/resource owners to set their own security policy, independent of who is likely to access them. Clients belonging to other domains are authorized for a service as long as they present credentials that prove their possession of a role local to the service's domain. Instantiated components receive their own set of credentials permitting use of similar mechanisms for servicing their requests.

6.5.2 Translating properties across domains

Most component-based systems restrict the set of properties used by application specifications to a fixed set. One of the goals of this work is to allow applications to define general, quantitative, and qualitative properties belonging to different namespaces. One of the main challenges in achieving this goal is translating between properties

belonging to different namepaces.

The solution to this problem is motivated by the observation that dRBAC credentials are just *statements* about entities within and across administrative domains, whose authenticity can be cryptographically verified. Thus, a dRBAC credential that grants the permissions associated with an Object role to a Subject role can also be interpreted as the statement that "it is true that Subject **is an** Object".

This interpretation allows the use of dRBAC credentials to encode various application and network-level properties and constraints on these properties, which drive the deployment process. Properties associated with application components and network resources are encoded using dRBAC credentials. Constraints are specified in terms of dRBAC system queries: "is X a Y?" (more precisely, the constraint is that X must possess role Y). Note that by design, dRBAC permits properties and constraints to be defined in terms of local names, relying on *role mapping* delegations to define translations across domains.

6.5.3 Enforcing the appropriate level of access control

Once client credentials are verified and accepted, the framework is responsible for ensuring that the client cannot access application functionality beyond its rights. Restricting access at the level of methods or interfaces is easily achievable by defining appropriate views, because views can be built to implement a subset of the functionality of the original object. Also, views can be customized to have different internal implementations depending on their intended uses, say by selecting appropriate property values. Access control lists can be established, per component, which specify the level of service (the view) associated with a given dRBAC role. As described

Table 6.5: Access control rules associated with `MailClient`. These rules are also used to trigger automatic view creation.

Role	View name
`Comp.NY.Member`	`ViewMailClient_Member`
`Comp.NY.Partner`	`ViewMailClient_Partner`
others	`ViewMailClient_Anonymous`

earlier, such policy can be established using only roles within the local namespace: cross-domain requests are first translated by dRBAC into local roles before any access control decisions are made.

Table 6.5 depicts the description of some access control rules created for the e-mail application scenario. All members of a company (`Comp.NY.Member`) are allowed access to a view (`ViewMailClient_Member`) to send/receive messages, access the phone and email directories, and add notes and meetings to their calendar. Partners of that company (`Comp.NY.Partner`) can access the `ViewMailClient_Partner` and execute the same operations, with the exception that the functionality for setting up a meeting is reduced to only requesting the right to set up a meeting. All other clients have only the right to access the `ViewMailClient_Anonymous` and browse the email directory.

Enforcing such access control decisions comes naturally because views contain only the subset of object state required for their local methods, and must interact with the original component to realize the rest of their functionality (if any). Views permit single sign-on usage, because authentication and authorization decisions can be completed when the view is first instantiated. After that, clients are free to access the view

they receive, without additional access control. Moreover, by using a Switchboard secure communication channel between the view and the original component, as described below, requests that are deferred to the original component can also proceed without requiring additional checks.

6.5.4 Creating secure connections

Once the views are generated, the deployment infrastructure issues to the generated view its own set of credentials, downloads them onto their target nodes, and connects them to other components using secure channels. These channels ensure that component interactions possess the desired security properties, and avoid the need for additional access checks after the channel has been established.

6.5.5 Case study: Using views to securely deploy component-based applications

The following example uses the e-mail application described in Chapter 2 to illustrate the use of views for satisfying security requirements.

In this example, three sites (e.g., New York, San Diego, and Seattle) have users, nodes, and links, and are running a dynamic component-based framework in order to automatically provide access to the e-mail application owned by the New York site. The sites in New York and San Diego represent two offices of the *Comp* organization. The site in Seattle represents the office of the *Inc* organization, which is a partner of the *Comp* organization.

For each site, the framework has a security module (*Guard*) that generates certificates, defines roles, creates access control lists, authenticates, and authorizes entities.

The assumptions are that (1) *NY-Guard* is responsible for the correct use of the e-mail application and all clients located in New York, (2) *SD-Guard* manages the San Diego clients even though they should be considered as belonging to the same logical domain as the New York clients, and (3) *SE-Guard* manages all clients from Seattle.

In order to provide security guarantees, a dynamic component-based framework should:

1. Authorize clients before accessing a service. The three users are: Alice from New York, Bob from San Diego, and Charlie from Seattle. They should be allowed to access the e-mail application only if they have credentials accepted by the New York site.

2. Authorize nodes before choosing them for component deployment. In this example, the New York site has Dell machines running Linux, San Diego site has Dell machines running SuSe, and the Seattle site has IBM machines running Windows. Components should be deployed on a node only if the node is trusted by the New York site, the owner of the application. This step also includes mapping the network properties onto application specific properties.

3. Provide the necessary credentials, such that nodes can authorize components before executing them. In this example, all components owned by the New York site are trusted and can be executed on any node of the three sites.

Table 6.6 presents an example of dRBAC credentials that ensures correct authorization of clients, nodes, and components. As explained in Chapter 3, dRBAC correctly authorizes an entity to own a required role if there is a valid chain of dRBAC credentials that maps the entity's original role into the required role.

Table 6.6: The roles and certificates generated by the Guard modules

New York	
User Auth.	(1) [Alice → Comp.NY.Member] Comp.NY
	(2) [Comp.SD.Member → Comp.NY.Member] Comp.NY
	(3) [Comp.SD → Comp.NY.Partner '] Comp.NY
Node Auth.	(4) [Dell.Linux → Mail.Node with Secure={true,false} Trust=(0,10)] Mail
	(5) [Dell.SuSe → Mail.Node with Secure={true,false} Trust=(0,7)] Mail
	(6) [IBM.Windows → Mail.Node with Secure={false} Trust=(0,1)] Mail
	(7) [Comp.NY.PC → Dell.Linux] Dell
Comp. Auth.	(8) [Mail.MailClient → Comp.NY.Executable with CPU=100] Comp.NY
	(9) [Mail.Encryptor → Comp.NY.Executable with CPU=100] Comp.NY
	(10) [Mail.Decryptor → Comp.NY.Executable with CPU=100] Comp.NY
San Diego	
User Auth.	(11) [Bob → Comp.SD.Member] Comp.SD
	(12) [Inc.SE.Member → Comp.NY.Partner] Comp.SD
Node Auth.	(13) [Comp.SD.PC → Dell.SuSe] Dell
Comp. Auth.	(14) [Comp.NY.Executable → Comp.SD.Executable with CPU=80] Comp.SD
Seattle	
User Auth.	(15) [Charlie → Inc.SE.Member] Inc.SE
NodeAuth.	(16) [Inc.SE.PC → IBM.Windows] IBM
Comp. Auth.	(17) [Comp.NY.Executable → Inc.SE.Executable with CPU=40] Inc.SE

Client authorization. The goal is to allow clients to use credentials defined by their local *Guard* for both local and cross-domain authorization. The framework achieves this goal by using dRBAC to find a mapping from a role to another role, even if

the roles are created by different domains. For example, Bob works in San Diego and holds credential (11) created by *SD-Guard*, which associates the role `Comp.SD.Member` with Bob's identity. If Bob wants to access the mail service, he should be authorized as one of the roles defined by *NY-Guard* (e.g. `Comp.NY.Member` or `Comp.NY.Partner`). In this case, the run-time system is able to to conclude that Bob is entitled to the role of `Comp.NY.Member` because of the availability of credentials (2) and (11).

Node authorization. The node authorization process consists of two steps: (1) the actual authorization of the node, and (2) the transformation of the node properties into properties meaningful to the application. The second step decides whether the node is useful during planning or not. The first part of the node authorization can be easily achieved in a similar way to the client authorization.

A more interesting question is how to transform the node properties generated by one administrative domain into properties that are meaningful to the application. The challenge arises from the fact that the component developer has no a priori knowledge of the node(s) where the component may be deployed. For example, the policy of the e-mail application in Table 6.6 is expressed only in terms meaningful to the programmer's domain, and states that (1) Dell machines installed with Linux are secure and have a trust level between 0 and 10 (credential 4); (2) Dell machines installed with Suse are secure with a trust level between 0 and 7 (credential 5); and (3) IBM machines installed with Windows are not secure and have a trust level between 0 and 1 (credential 6). Similarly, all node credentials are defined in terms of properties local to their domains. For example, all machines from the San Diego site have a credential (13) generated by Dell, stating that they are running SuSe. The framework decides

to deploy a mail component on a node only if run-time infrastructure finds a possible chain of credentials that maps a node credential to a policy credential. In this example, the machines from San Diego can be mapped from credential (13) to credential (5).

A similar process can be used to map link properties onto application properties.

Component authorization. The second part of the mutual authorization between a node and a component requires that a node accept the component before allowing it to run. In this case, nodes should be able to authorize components, even if the components might belong to a different domain. In this example, *NY-Guard* creates local credentials for three components that need to be deployed in different domains: `MailClient` in New York, `Encryptor` in San Diego, and `Decryptor` in Seattle. *SD-Guard* defines a `Comp.SD.Executable` role to specify that all components having this role will be allowed to run on the San Diego machines with a limit of 80% in CPU consumption. In a similar way, *SE-Guard* defines a `Inc.SE.Executable` role that restricts the CPU consumption to 40%. Then, both *SD-Guard* and *SE-Guard* associate these roles to the `Comp.NY.Executable` role. This allows *NY-Guard* to generate only local credentials (`Comp.NY.Executable`) for the `MailClient`, `Encryptor`, and `Decryptor` components. Whenever a component is deployed on a node, it presents a chain of credentials. Using dRBAC, the guard associated with the site authenticates the signatures and establishes the validity of all the credentials in the chain. The component is accepted only if the guard recognizes the chain of credentials as valid.

6.6 Using views to efficiently satisfy data consistency requirements

The data consistency problem arises when a dynamic component-based framework deploys several data views of the same component. In such cases, the views and the original component might require that the shared data be synchronized. There are several challenges in providing consistency for component-based applications that differentiate this problem from ensuring consistency in distributed shared memory, distributed database or distributed file systems. First, the data consistency protocol must work with general component-based applications, without any knowledge on the application internals. Thus, the consistency protocol cannot make any assumptions (e.g., read/write patterns or the data structure) that are valid across all applications. The only application-specific information used by the protocol is the one exposed by the application through its interfaces, properties, and requirements. Second, different component-based applications require different consistency levels and the protocol should be able to accommodate all applications. Third, the data consistency protocol should be able to dynamically adapt to any dynamic changes in the application consistency requirements.

Therefore, the goal is to design a data consistency protocol that satisfies the consistency requirements of component-based applications (*application-neutral*) deployed in various configurations (*flexible*), while using *application-specific information* to improve efficiency.

This section describes such a protocol which uses views to satisfy the data consistency requirements of component-based applications. This protocol is *application-neutral* because it does not make any assumptions about the data structures defined

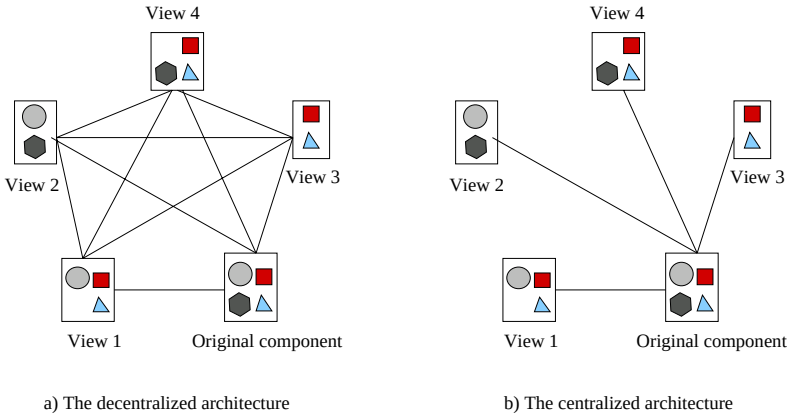


Figure 6.3: Decentralized vs. centralized architectures

by component-based applications, or their data access patterns. However, the protocol uses *application-specific* information in order to minimize the data consistency traffic. The *flexibility* of this protocol stems from two factors. First, the protocol implements two modes of operation – *strong* and *weak* – in order to support applications with different consistency requirements. The strong consistency mode ensures that there is only active view running in the system, providing essentially one-copy serializability semantics. The weak consistency mode allows multiple active views to simultaneously work on the shared data and specify more relaxed consistency levels. Second, the protocol allows views to either modify at run-time their weak consistency levels or switch between the strong and weak modes of operation.

6.6.1 Overview of the data consistency protocol

One way of understanding the data consistency protocol is to consider how one might enforce data consistency in a system where all views associated with an original component are identical. In such a system, the functionality of a centralized data con-

sistency protocol is realized by two entities: (1) a *directory manager* associated the original component and (2) a *cache manager* associated with each view. The responsibility of the directory manager is to keep track of which views are running in the system and control which views are allowed to be active (i.e., working on the shared data). The role of the cache managers is to forward to the directory manager any requests made by the views, and execute the commands sent back by the directory manager. In other words, the cache managers are responsible for controlling the flow of updates between the views and the directory manager.

One of the challenges in designing a data consistency protocol is deciding which information should be propagated as an update. Current systems propagate either *logs of modifications* to be replayed by replicas or *modified data* to be merged into replicas. The first solution does not work in the context of component-based application because views represent different layouts of the same component and might not implement the same methods. A log defined by one view might not be executable on a different view. Thus, the second solution seems to be more appropriate. The challenge in this case is how to extract and merge the data from/into views and the original component if the framework has no knowledge about the data structure and semantics. The solution is to use application-specific functions to extract and merge data.

The data consistency protocol described at the beginning of this section has a centralized architecture. The centralized version is more appropriate in the context of component-based applications because it takes advantage of the pre-established relationship between the involved entities. All views are logical representations of the same original component and the original component is regarded as a sink (primary-

copy) where all updates are propagated (Figure 6.3). Another advantage of using a centralized protocol is the reduced number of merge/extract methods necessary to communicate updates. A decentralized protocol, which considers all views as peers, requires merge/extract methods between each pair of views ($O(n^2)$); the centralized protocol requires only such methods between views and the original object ($O(n)$). The downside of the centralized protocol is its assumption that the original component is always running in the system. Fail-safe mechanisms can be implemented; however, they are not the focus of this thesis.

The interactions between the directory manager and the cache managers are controlled by application-specific information provided at view granularity. The next sections describe in detail the nature of the information provided by the application and its use by the the data consistency protocol to satisfy application consistency requirements. A detailed description of the messages passed between views, cache managers, and the directory manager can be found in Appendix B.

6.6.2 Application specific information

In general, a dynamic component-based framework has no knowledge of the application internals, except what is exposed by the application through its interfaces. In addition, the framework cannot make any assumptions (e.g., data structure or access patterns) that are valid across all applications. Without using any application-specific information about the shared data, the data consistency protocol can only execute based on worst-case assumptions, such as that all views conflict and that updates should be sent to all views.

The solution proposed in this section requires the application to expose additional

information, which is used to improve the efficiency of the consistency management. This information is specified at the level of views and falls into three categories: (1) *data properties* to characterize the shared data and indicate *which* views need to be synchronized, (2) *quality triggers* to indicate *when* updates need to be pushed or pulled between views, and (3) *merge/extract methods* which define *what* information should be synchronized.

Data properties. Both the original component and the views use *data properties* to inform the underlying infrastructure of the characteristics of the shared data. A property p is defined as a tuple $(name_p, D_p)$, where $name_p$ is a unique name and D_p represents the property values. D_p can be an interval $D_p = [d_{min}, d_{max}]$ or a set of discrete values $D_p = \{d_1, d_2, ..., d_n\}$.

The framework uses data properties to determine which views share the same data, if such sharing relationships cannot be statically described. Static relationships are specified using a static map. The map is created once, when the directory manager is initialized. The map contains a symmetric matrix, where the number of rows and columns equals the number of views. If two views v_i and v_j share data, than the elements (i, j) and (j, i) in the matrix are set to 1. Otherwise, the elements are set to 0.

Sometimes, it is difficult to statically specify the relationship between two views. The static matrix indicates such a possibility by setting the corresponding cell value to -1. In such cases, a *dynamic set of data properties* (see Definition 6.1) can be used to search for views that share data. If two views v_p and v_q are defined by two property sets P and Q, and the sets have a non-empty intersection, the framework considers that

the two views share data. Views can dynamically change their shared state, and the framework captures this behavior by allowing views to change their sets of properties at run-time. This method is very flexible and can reduce the coherence traffic by not triggering false conflicts.

$$\begin{aligned} \textit{dynamic\_conflict} &: C_t \times C_t \rightarrow \{0,1\} \\ \textit{dynamic\_conflict}(v_p, v_q) &= \begin{cases} 0 & \text{, if } P \cap Q = \varnothing \\ 1 & \text{, if } P \cap Q \neq \varnothing \end{cases} \end{aligned} \tag{6.1}$$

The intersection of two property sets $P = \{p_1, p_2, ...\}$ and $Q = \{q_1, q_2, ...\}$ is defined as the set of intersections between any two properties of P and Q (see Definition 6.2), with the assumption that a set of properties does not contain two properties with the same name (i.e., $name_i \neq name_j \quad \forall i, j$).

$$P \cap Q = \left\{ r \mid \exists p_i \in P \textit{ and } \exists q_j \in Q \; s.t. \; p_i \cap q_j = r \right\} \tag{6.2}$$

The intersection between two properties $p = (name_p, D_p)$ and $q = (name_q, D_q)$ is not empty if the properties have the same name and the intersection of the value sets is not empty (see Definition 6.3).

$$p \cap q = \begin{cases} \varnothing & \text{, if } name_p \neq name_q \\ (name_p, D_p \cap D_p) & \text{, if } name_p = name_q \end{cases} \tag{6.3}$$

Quality triggers. In general, replicas are responsible for deciding when updates should be either pushed or pulled. This is natural because the synchronization decisions closely depend on the state of the data and the consistency requirements defined by the application. This consistency protocol accommodates such behavior by allowing views to explicitly pull and push updates.

One of the goals of a dynamic component-based framework is to simplify applications by taking such decisions on behalf of the application. In addition to making explicit calls to push/pull data, views are also allowed to delegate to the system the right to make the synchronization decisions by defining *push/pull/validity triggers*.[2] *Push triggers* indicate when to send the current value of data from the view to the original component. *Pull triggers* indicate when the view needs to update the shared data with the value held by the original component. *Validity triggers* are executed by the directory manager upon receiving a pull request from a view, and indicate if the data currently held by the original component is "good enough" for the requesting view. If it is not "good", the directory manager is responsible for getting the most recent data from the other conflicting views and send it to the requesting view.

If $\mathscr{T}$ is a discrete representation of time, push/pull triggers defined by a view v specify the synchronization moments as a boolean expression of time ($t \in \mathscr{T}$) and view variables $(x_1, x_2, ...)$, where $x_i \in V_v, \forall i$ (see Definition 6.4). Similarly, a validity trigger defined by a view v of an original component c is defined as a function of time ($t \in \mathscr{T}$) and variables of the original component ($x_i \in V_c, \forall i$).

$$T_v(t, x_1, x_2, ...) : \mathscr{T} \times V_v^* \rightarrow \{true, false\} \tag{6.4}$$

There are two ways for the cache manager to evaluate the current values of the object variables: (1) the object provides the necessary methods to access the variables, or (2) the cache manager uses reflection to examine the variables (when the components are defined in languages that support this feature). The implemented prototype of the dynamic component-based framework described in Chapter 7 works with Java-based

[2] These triggers are used only when there are multiple active views running in the system (i.e., weak mode).

applications, so the latter mechanism is used.

Merge/Extract methods. In order to synchronize the state of all active entities (i.e., views and original component), the framework needs to propagate the updates from views to the original component and in reverse direction. The questions are (1) what information to propagate, and (2) how to reconcile any conflicts?

As discussed earlier, the consistency protocol creates updates by using merge/extract methods defined for each view. As in Coda [80] and Bayou [19], the data consistency protocol uses these functions to detect and resolve possible conflicts. Ideally, the code for the merge and extract methods should be automatically generated by the framework at run-time, as part of view generation process. An alternative solution, which admits the possibility of application specific customization and which is adopted in this thesis, is to add to the view definition the descriptions for the cache coherence-specific methods that *extract* the view state and *merge* updates into it. In this way, the framework is able to create views and associate the appropriate caching information at run-time.

The next three sections explain in detail how the data consistency protocol uses the data properties, triggers, and merge/extract methods to guarantee the required consistency between views. The first two sections describe the directory manager, the cache manager, and the interactions between them. The behavior of both the directory manager and the cache manager is captured using state machines that interact through messages. A complete description of the messages sent between views, cache managers, and the directory manager can be found in Appendix B. The last section

presents two examples that illustrate the data consistency protocol.

6.6.3 Directory manager

The directory manager is the central entity of the data consistency protocol. Each original component is associated with a directory manager, and its role is to keep track of all active views in the system, deal with conflicts, and assist the views in acquiring or releasing the most current image.

Figures 6.4 and 6.5 describe the behavior of the directory manager protocol when the consistency level is strong, respectively weak. Strong consistency ensures that there exists only one active view at any moment of time. Weak consistency allows several views to simultaneously execute on active images. Initially, the directory manager is in the *DM_VALID* state. This means that the only entity running in the system is the original component. Upon creation, cache managers register with the directory manager by sending a unique ID and a set of view properties (`registerCM`). When the directory manager receives a `getImage` request from a cache manager, it checks whether the the view conflicts with other views. If there are conflicting

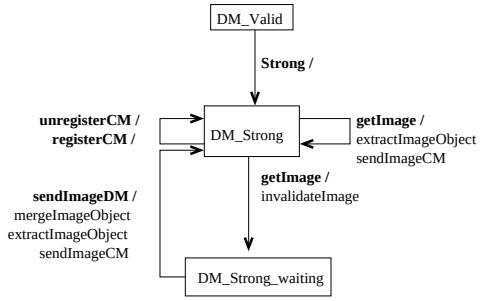


Figure 6.4: DM - Strong consistency

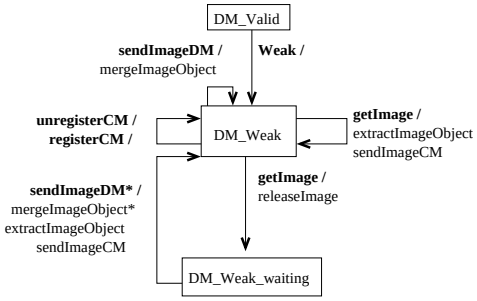


Figure 6.5: DM - Weak consistency

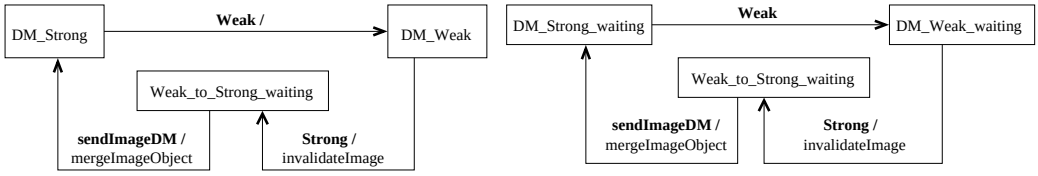


Figure 6.6: Directory manager - Transitions between levels of consistency

views, the directory manager asks them to send the current images, merges these into the original component, and sends the final image to the requesting cache manager (`sendImageCM`). A cache manager is removed from the list of active cache managers only if it sends an `unregisterCM` message.

The difference between **strong** and **weak** consistency lies in the way the directory manager behaves when there are multiple views running in the system compared with only one active view, and a `getImage` request arrives. In the strong consistency case, the directory manager asks the only view that has an active image to release the image (`invalidateImage`) and enter a *CM_INVALID* state. In the weak consistency case, the directory manager asks all conflicting views to send their updated image (`releaseImage`). However, the views remain in the *IM_ACTIVE* state and can keep working. The images extracted from views (`extractImageView`) are merged into the original component (`mergeImageObject`). Only after this, the directory manager extracts the image from the original component and sends it to the requesting view.

The directory manager can dynamically switch between the two levels of consistency. Switching from strong to weak consistency does not require additional steps, because the strong case is equivalent to a weak one with only one active view. The directory manager can simply change the mode of operation. Switching from weak to strong consistency is achieved by gathering all the active images and merging them

into the original component (`invalidateImage`). In this way, the directory manager ensures that there is only one active view in the system.

6.6.4 Cache manager

A cache manager is automatically initialized when the view is created. Figure 6.7 shows how the cache manager changes its state depending on external events. The initial state of the cache manager is *CM_INVALID*. First, the cache manager registers with the directory manager by presenting a list of properties that characterizes the data used by the view (`registerCM`). Once registered, the cache manager changes its state to *CM_REGISTER*. This means that the cache manager is ready to start. However, the data contained by the view is incorrect. The view initializes its state by sending an `initImage` request to the cache manager. The cache manager forwards the request to the directory manager (`getImage`), changes the state to *CM_WAITING* and waits for the correct image of the original object to arrive from the directory manager. When the directory manager has the correct image, it sends the image to the cache man-

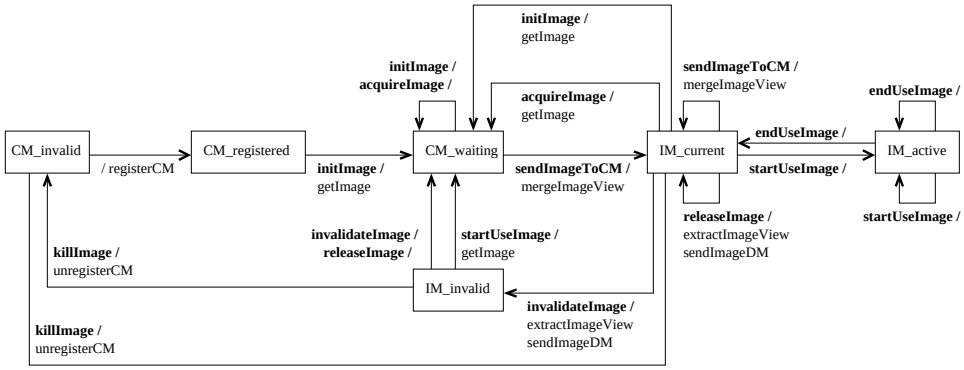


Figure 6.7: Cache manager

ager (`sendImageDM`). The cache manager merges the latest image of the data into the view by executing (`mergeImageView`) and changes the state to *IM_ACTIVE*. Upon finishing, the view releases the current image (`killImage`) to the directory manager and announces that the view is no longer active (`unregisterCM`).

During execution, the view uses the `startUseImage` and `endUseImage` messages to delimit the portions of the code which process shared data. In the strong consistency case, a view uses these two messages to (1) acquire and release the control over the shared data and (2) prevent the cache manager from extracting the image of the working data while it is being modified. In the weak consistency case, the usage of both triggers and the `startUseImage` and `endUseImage` methods reduces the traffic between the cache manager and directory manager. Without triggers, the cache manager would ask pull and push updates every time either of the two methods is executed. Without the methods, the cache manager would send updates every time the triggers would evaluate to true. The combination of the triggers and methods forces the cache manager to send and receive updates only when the methods are executed and the triggers evaluate to `True`.

There are two reasons for the cache manager to extract the current image and send it to the directory manager: (1) the directory manager asks for the image by sending `invalidateImage` or `releaseImage` messages, and (2) the synchronization policy of the object requires it (i.e. the caching trigger evaluates to true). In both cases, the cache manager extracts the current state of the image (`extractImageView`) and sends it to the directory manager (`sendImageCM`).

6.6.5 Examples of interactions in the data consistency protocol

Strong consistency

Figure 6.8 illustrates the interactions between the directory manager and the cache managers associated with two views running in the strong mode. In this example, the only active entities in the system are the original component C and its views V_1 and V_2. The property defined by all entities is P. The values associated to P by the three entities are different: $\{x,y\}$ for V_1, $\{x,z\}$ for V_2, and $\{x,y,z\}$ for the original component.

When the view V_1 is deployed, the view creates a cache manager (step 1) that registers with the directory manager (step 2) and asks for the current data (steps 3,4).

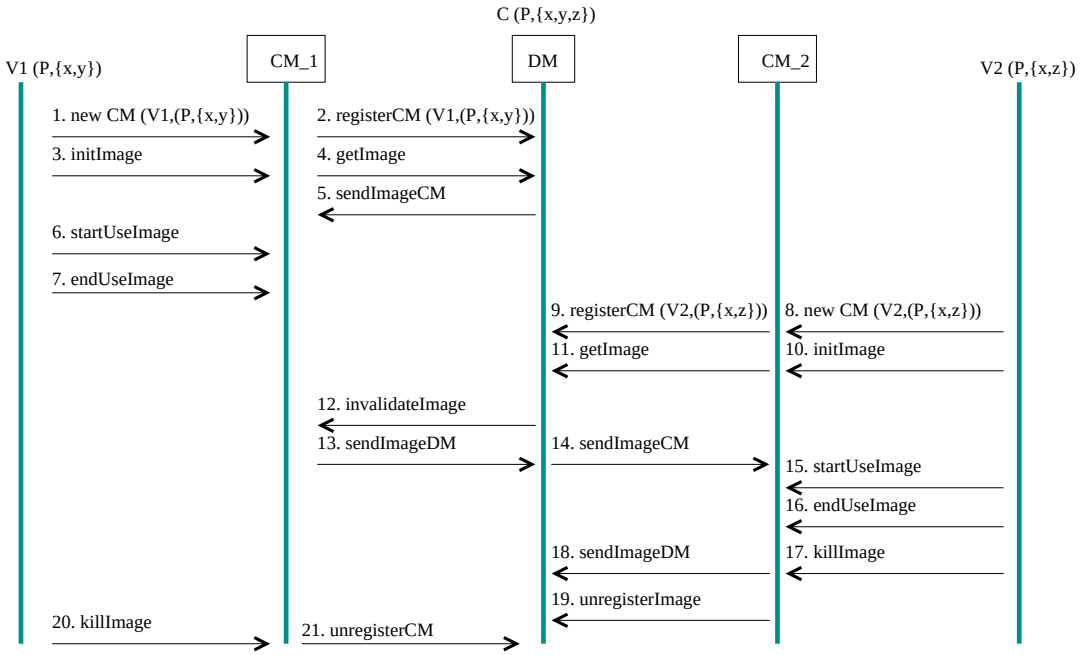


Figure 6.8: Strong consistency - Data consistency protocol

The directory manager looks for other views sharing data with the requesting view. Currently, there is none and the directory manager extracts the data from the original component and sends it to the view (step 5). When view V_2 asks for the current data, the directory manager finds that V_1 is active and conflicts with V_2. The directory manager sends an invalidation request to V_1, waits until V_1 executes `endUseImage`, stops V_1 from working, and gives the control to V_2 (steps 12,13,14). This ensures that there is only one active view in the system. In order to prevent the cache manager to merge or extract updates while working on it, the view needs to mark the code that processes the data as mutually exclusive (steps 6,7). At the end, the view announces to the cache manager and thus the directory manager about its intention to stop using the data (steps 20,21).

Weak consistency

Figure 6.9 illustrates the behavior of the two views V_1 and V_2 when their data consistency requirements are weak. In addition, the view V_1 defines time-based pull triggers that periodically evaluate to `True`. There are a few differences between the way the directory manager and the cache managers interact in the strong and weak scenarios. First, the directory manager no longer invalidates the data images used by views, when it needs to extract updates. In the weak case, the directory manager only asks the cache managers to release the data, and allows the views to continue running on stale data. Second, the cache manager associated with the view V_1 does not ask the directory manager for the current image of the data every time the view executes `startUseImage`. Because the view has also defined pull triggers, the cache managers send a `getImage` request to the directory manager only when the view execute

`startUseImage` and the pull trigger evaluates to `True`.

6.7 Summary

This chapter has presented several mechanisms based on *views* which could be employed by a dynamic component-based framework to achieve the goals of a secure, efficient, and practical deployment. Views are defined in Section 6.3 as customizations of original components, which could be dynamically instantiated at run-time using the VIG tool. Views help achieve a practical and efficient deployment process,

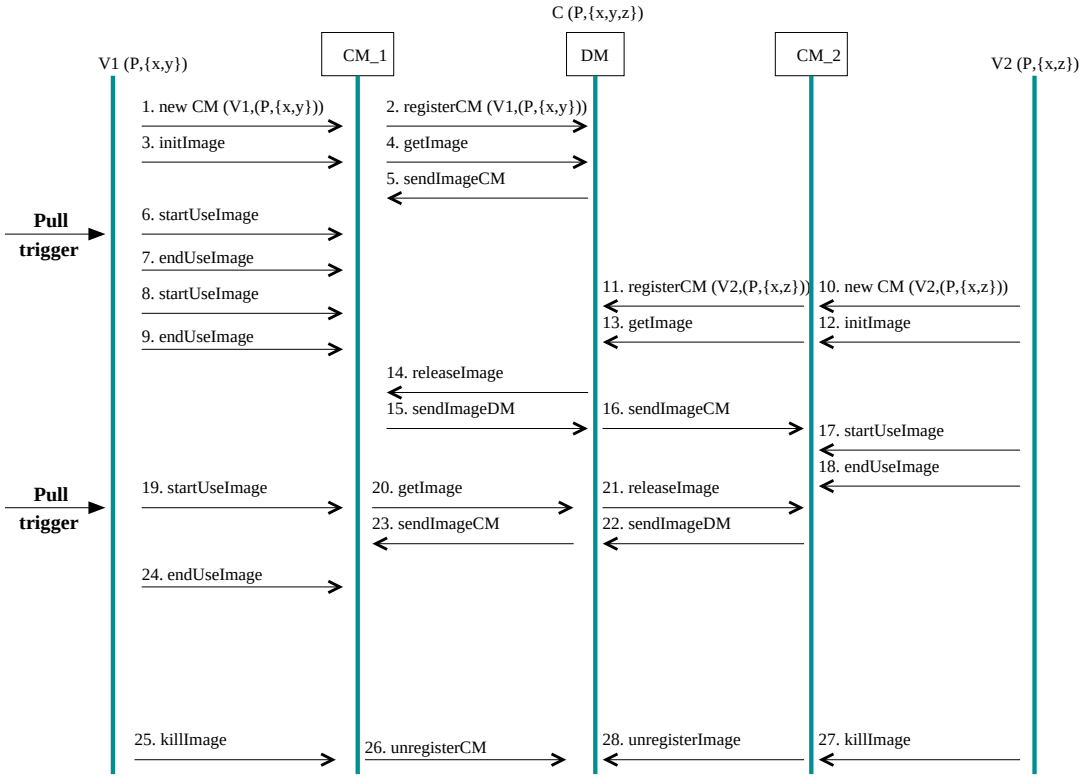


Figure 6.9: Weak consistency - Data consistency protocol

by permitting planning flexibility, by customizing components to satisfy application security requirements, and by serving as the granularity at which application consistency is achieved using application-specific weak consistency protocols.

Views, used together with dRBAC and Switchboard for a secure deployment, offer advantages of *scalability* (multiple policy roots are permitted), *easier configuration* (local policy need not include translation between global system-wide and local credentials, which is automatically inferred), and *finer-grained control* (the rights afforded a request can be modulated to the credentials associated with it as opposed to the local credentials these translate to).

Views, together with the data consistency protocol described in this chapter, help reduce the cost of maintaining consistency among replicated application components by allowing the application to provide application-specific information at the level of views and thus customize the consistency protocol as necessary.

Chapter 9 illustrates the costs of generating views, and the benefits of using views as the coherence granularity of a data consistency protocol.

Chapter 7

Partitionable Services Framework

This chapter describes the Partitionable Services Framework (PSF), a dynamic component-based framework built to test and evaluate the techniques presented in the previous chapters.

As described in Chapter 2, component-based frameworks consider applications as being dynamically built out of independent components that can be flexibly assembled to suit the properties of their environment. The frameworks facilitate on-demand transparent migration and replication of these components at locations closer to clients while still retaining the illusion of a monolithic service. PSF is an instance of a dynamic component-based framework which uses the modeling, planning, and deploying techniques to achieve an efficient and practical deployment process.

The PSF architecture contains two main components: the *PSF Runtime* and the *PSF Deployment infrastructure*. The former is responsible for taking the decisions that provide users with seamless access to distributed applications, while the latter actually executes those decisions.

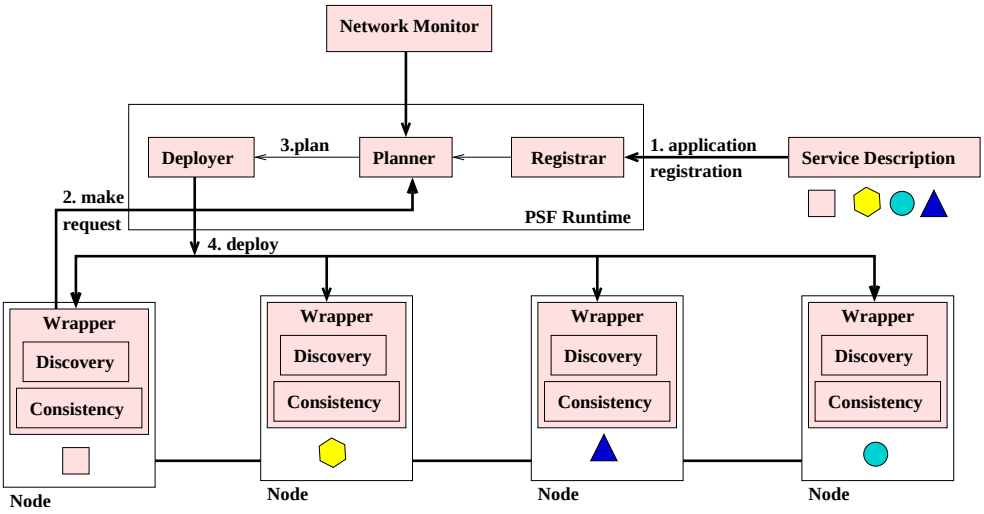


Figure 7.1: PSF architecture.

The PSF Runtime contains three modules: (1) the *registrar*, (2) the *planner*, and (3) the *deployer*. The PSF deployment infrastructure is formed by *wrapper daemons* assumed to be running on every node. Beside supporting component deployment, wrappers provide other services such as discovering the PSF Runtime upon starting, and implementing the data consistency protocol described in Section 6.6. All PSF modules communicate through Switchboard secure channels built on top of dRBAC. In Figure 7.1, the PSF modules are represented by shaded boxes, and the secure channels are marked on the figure with thick lines.

PSF is implemented in Java and benefits from the latter's support for dynamic class loading, verification, and installation. The prototype PSF implementation has focused only on run-time aspects that are novel to the framework. In a complete system, PSF needs to be integrated with other components that are responsible for fault handling and network monitoring [30].

7.1 PSF modules

Registrar. Before users can use PSF, the applications must perform a registration step. During this step, a component-based application provides full specifications for each of its components, including the component functionality, and their deployment requirements and effects (see Chapter 4). These specifications are described in XML-based files, similar to the ones shown in Chapter 8.

Planner. The main active component of the PSF framework is the planning module, which is responsible for determining how best to satisfy client requests by instantiating service components at appropriate locations in the network. The planner makes this decision based on three inputs: (1) the application specification, (2) the current state of the network, and (3) the user-specified QoS requirements, by implementing the algorithm described in Chapter 5. The output of the planner is a valid component composition described as a list of actions. PSF supports two actions: *start component* and *connect component*. The former specifies the name of the component, the web server where the source code of the component is located, and any initialization parameters. The latter contains a list of connections to other components that need to be created in order for this component to execute correctly. Depending on the application requirements, the planner may be invoked one or more times to realize an application deployment.

One of the choices which had to be made when building PSF was whether to design PSF as a centralized or a decentralized framework. A centralized version has the advantage of having access to all of the information and using it to make deci-

sions, but poses the challenge of how to gather and filter the required information. The decentralized version eliminates this challenge; however, it bases its decisions on incomplete information about the status of the entire system. In the end, PSF was designed as a centralized system because it is difficult to reason about the behavior of the planning algorithm were it to operate with only partial information. This issue represents an avenue of future work to further refine the techniques developed in this thesis.

Deployer. Once the planner finds a valid application composition that satisfies the user-specified QoS requirements, it communicates the solution to the deployment module. This module is responsible for securely deploying the components and connecting them as necessary, by contacting the wrapper daemons running on each node and sending them the appropriate instructions. Chapter 6 has discussed the steps involved in this process.

Wrappers. To ensure that the service deployment appears seamless to clients, the framework relies upon run-time functionality embodied in the wrapper modules; wrappers represent the minimal functionality that must be implemented by every node that is part of the PSF environment. Once a component is downloaded on a node, the wrapper running on that node is responsible for initializing and connecting the component to other components, according to the required interfaces specifications.

Beside deploying the component, the wrappers provide additional services, such as *discovery* and *data consistency*. Upon starting, wrappers need to contact the PSF Runtime and register the nodes as participants in the PSF infrastructure. The *discov-*

ery service searches for an active instance of the PSF Runtime and create the initial connection. The discovery service is provided using the DisCo infrastructure, as explained in Section 3.3.5. Once the components are deployed and running, the wrappers are responsible for satisfying the application data consistency requirements by implementing the data consistency protocol presented in Chapter 6.

Secure communication. All PSF entities (clients, wrappers, and the PSF Runtime) communicate through Switchboard secure channels; the authentication and the authorization services are built on top of dRBAC, as discussed in Chapter 6.

7.2 Module interactions

This section describes the time-line of actions enabled by the framework, as illustrated in Figure 7.1.

- The runtime system is responsible for registering applications with the framework and serving incoming client requests. The service registers itself with the framework, providing a meta-description of its constituent pieces and procedures for their assembly (Step 1).
- Whenever a client wants to access the service, the client sends the request to the run-time system (Step 2), along with supporting credentials.
- Once the runtime system receives a client request, the runtime system performs the necessary security checks (authentication and authorization), decides which level of service the client has the right to access, and calls the planner module

to compute a valid application configuration. In general, calls to the planner can be made (1) at run-time, when clients make requests, or (2) offline, when PSF receives aggregated requests from clients.

- The planning module considers both the service specification and the current network conditions to come up with a service partitioning that best satisfies the client request (Step 3).

- To achieve this partitioning, the run-time system of the framework may need to deploy additional components; *wrappers* running on each node facilitate remote installation and *coherence* modules enable application-specific consistency among replicas (Step 4).

Once the service components have been installed, the client can seamlessly access the service provided by the application.

Chapter 8

Expressivity Evaluation

This chapter evaluates the expressivity and ease-of-use of the three techniques presented in Chapters 4, 5, and 6. The e-mail application described in Chapter 2 is used as an example to illustrate both the type of application properties that can be captured, and the amount of code necessary to enable a component-based application to use the techniques. Section 8.1 describes the information provided by the application specification, given as XML files. Section 8.2 analyzes the PSF-related code added to the e-mail application.

8.1 Expressivity of the application specification

One of the main challenges in automatically deploying component-based applications into heterogeneous environments is capturing the relevant, yet sufficient information about application behavior. As described in Chapter 2, the application specification needs to describe: (1) the application *linkages* and the *properties* of interest for those

Table 8.1: ViewMailServer - XML description.

```
<Name>ViewMailServer</Name>
<Linkages>
  <Implements>MailServerInterface
  <Requires>MailServerInterface
<Init>
  ViewMailServer.RRF:=50
<Conditions>
  Node.NodeCPU > ( Node.Sin*(0.39*Node.Sin-0.4) +
                   Node.Sout*2*(0.39*Node.Sout-0.4) +
                   Node.TotalRR*(Node.Rout+Node.Rin)*((0.5*Node.TotalDF)+
                   (0.39*Node.TotalRecvRate-0.4)) +
                   Node.TotalRR*(2+0.39*Node.TotalRR-0.4)
  Node.RRF > ViewMailServer.RRF
  MailServerInterface.Privacy = True
<Effects>
  MailServerInterface.Privacy:=True
  Node.NodeCPU:=Node.NodeCPU -  Node.Sin*(0.39*Node.Sin-0.4) +
                Node.Sout*2*(0.39*Node.Sout-0.4) +
                Node.TotalRR*(Node.Rout+Node.Rin)*((0.5*Node.TotalDF)+
                (0.39*Node.TotalRecvRate-0.4)) +
                Node.TotalRR*(2+0.39*Node.TotalRR-0.4)
  ViewMailServer.RRF:=Node.RRF
  MailServerInterface.MessageSize:=1
  MailServerInterface.SendRate:=MailServerInterface.SendRate * Node.RRF
  MailServerInterface.LinkBandwidth:=MailServerInterface.SendRate*Node.RRF*
                                     MailServerInterface.MessageSize
  MailServerInterface.TFMS:=3
  MailServerInterface.TLMS:=3+MailServerInterface.TLMS
```

linkages, (2) the component deployment *conditions* and *effects*, (3) the component customization rules (*views*), and (4) if relevant, the data consistency *properties*, *quality triggers*, and *merge and extract methods*.

Table 8.1 presents the XML-like specification for the `ViewMailServer` component of the mail application. The expressions shown in this example are computed based on the profiling results obtained for the e-mail application components and presented in Appendix C. The complete description of how such expressions were obtained can be found in Appendix D.

The following sections illustrate how the e-mail application takes advantage of the expressivity of the application model. For clarity, the examples presented in this chapter are simplified versions of the one given in Table 8.1.

8.1.1 Setting

The goal of the application specification is to provide to PSF all information necessary to automatically deploy the components in heterogeneous environments, such that the user QoS requirements are satisfied. Ideally, the application specification should capture the application behavior, requirements, and effects.

Behavior. Given the e-mail application, clients can request access to its services by contacting and informing PSF of their QoS requirements. For example, clients may want to connect from insecure sites (e.g., airport terminals) and ask for *privacy* (i.e., all sensitive messages should be protected against eavesdroppers) and *efficiency* (i.e., the time spent to send and receive messages should be below a given threshold) when accessing their e-mail.

Once a client makes a request, the request is handled directly by the mail server, or by a cache mail server. Once a client connects to the cache mail server, the cache asks the original mail server for the most up-to-date information related to that client. Then, the cache mail server executes client requests on the local copy of the accounts, only if the account exists on the cache. Otherwise, the request is forwarded to the original mail server.

Requirements. The application specification should provide all the information necessary to satisfy user QoS requirements. Examples of user requirements include (1) *message protection* (e.g., sensitive messages should not be seen on untrusted nodes such as airport terminals, or sensitive messages should not be transmitted in the clear on insecure links), and (2) *efficiency* (e.g., the server should sustain the required request rate or the time to reach the first/last mail server should be under a threshold). The network resources that could affect the user QoS requirements are available node CPU, available link bandwidth, link latency, node trust, and link security. The first three resources may influence the efficiency of the e-mail application. The last two resources may affect the user's security requirements.

Effects. In order to satisfy the user QoS requirements, the framework needs to use the application-provided information to take certain decision. In case the connection between a client and the mail server crosses a slow link, the efficiency requirements can be satisfied by placing a cache mail server close to the client. Similarly, `Encryptor/Decryptor` pairs can be placed to protect messages transmitted over insecure links.

In order to take such decisions, the application needs to provide information about: (1) when to create *linkages* between components (Section 8.1.2), (2) what are the *application properties* (Sections 8.1.3 and 8.1.4), and the *deployment conditions* (Section 8.1.5) and *effects* (Sections 8.1.6), and (3) what is the *resource consumption* (Section 8.1.6). As discussed in Chapter 6, the application can improve the automatic deployment process by also providing information about (1) *view configurations* (Section 8.1.7), (2) *security requirements* (Section 8.1.8), and (3) *data consistency requirements* (Section 8.1.9).

8.1.2 Linkages

In order to create valid component deployments, a dynamic component-based framework needs to determine how components should be connected to each other.

The e-mail application indicates how components should be *logically* connected by specifying the interfaces implemented and required by each component. Figure 8.1, which first appeared in Chapter 2, is included here to illustrate all possible compositions for the e-mail application. The `MailServer` component implements the `MailServerInterface` interface. The `ViewMailServer` component both imple-

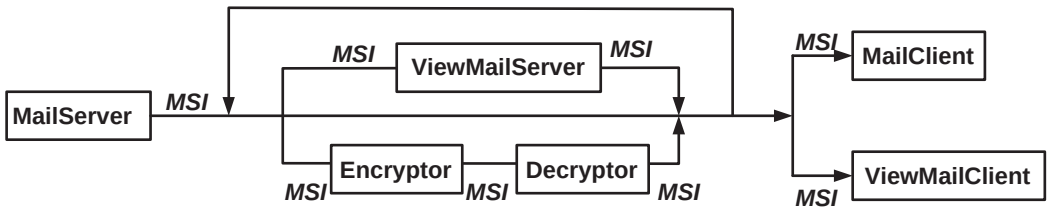


Figure 8.1: Valid component compositions in the e-mail application.

ments and requires the `MailServerInterface` interface. The `Encryptor` component requires the `MailServerInterface` interface, but implements a new interface `MailServerInterface_Encrypted`, which is required by the `Decryptor` component. The two mail clients require the `MailServerInterface` interface. Based on this local information, PSF can create different component linkages. For example, the two types of mail clients can be connected to the mail server (1) directly, (2) through the `ViewMailServer`, (3) through the `Encryptor` / `Decryptor`, or (4) through any combination of the above.

However, more information is required before the framework can use the components as intended. For example, the `ViewMailServer` should be used as a cache mail server that offsets the high link latencies, and the encryptor/decryptor pair should be deployed only when the privacy of messages needs to preserved against eavesdroppers. The e-mail application can capture such information about the components by associating properties with the components and their interfaces. For example, in order to protect the client messages, the mail application can define and associate a Boolean `Privacy` property with the `MailServerInterface` interface; PSF can use this property to deploy the `Encryptor`/`Decryptor` pair and ensure that the client messages are protected. The properties and their usage are discussed in more detail in the following section.

8.1.3 Interface properties

The properties of interest for the `MailServerInterface` interface are as below:

- `Privacy` takes the value `True` or `False` depending on whether the messages should be protected or not. This property can be used to ensure that messages are protected when sent over insecure links. For example, if the `MailServer` saves sensitive messages, the `Privacy` property associated with its implemented `MailServerInterface` is set to `True`. If the link between the `MailClient` and the `MailServer` is insecure, i.e., the `Secure` property is set to `False`, the application specification can capture the effects of a sensitive message crossing an insecure link by setting the `Privacy` property of the `MailServerInterface` after crossing the link to `False`. This means that a direct connection between the `MailClient` and the `MailServer` is not valid if the client wants his messages to be protected, i.e., the required `MailServerInterface` to have the `Privacy` property set to `True`.

- `Trust` is equivalent to the notion of clearance level in a military organization, where one person is allowed to read a document only if his clearance level is higher than the one of the document. In the e-mail application, this property is very useful to differentiate untrusted nodes, such as airport terminals, where a user can read only the messages with a required trust level less than the trust level of the node.

- `ReqCPU` is the amount of CPU consumed by processing one request from the client (e.g., send a message). This property is used to compute the maximum amount of CPU required by the `ViewMailServer`, and ensure that the `ViewMailServer` is deployed only on nodes that have more CPU resources available than the amount required.

- `NumReq` is the average number of requests made by a component in one second and is used to characterize the flow of messages between components. For example, a client which needs to interact with the mail server at a certain rate, should not be connected to a `MailServer` whose request processing rate cannot sustain the incoming requests (see the experiment in Section 10.5 for a discussion of what happens in such cases).

- `ReqSize` is the average size of a message and it is used by the application to estimate the CPU consumption and the request processing time.

- `BW` is the amount of bandwidth consumed when two components communicate with each other. Correctly estimating the value of this property allows the framework to accurately keep record of the network resources consumed by the application, and thus ensure that application has the expected performance.

- `TFMS` and `TLMS` represent the time it takes for a message to reach the first instance of a mail server, respectively last mail server. In general, clients might be connected through a series of `ViewMailServer` components and a final mail server. `TFMS` is important because it measures the application efficiency, as seen by the client. `TLMS` captures the time necessary for a message to reach the last mail server, thus represents the time to propagate a message to all users.

8.1.4 Component properties

Beside the properties associated with interfaces, the application specification can also define properties associated with the components. In the `ViewMailServer` example, there are two such properties: `NodeCPU` and `RRF`.

NodeCPU indicates how much CPU resource is required by the ViewMailServer component in order to execute in an efficient way. A simple expression that defines it is $\texttt{NodeCPU} = \texttt{ReqCPU} * \texttt{NumReq}$. More complex and realistic expressions obtained by profiling the e-mail application are given in Appendix D.

RRF captures the benefits of placing a ViewMailServer component close to clients. RRF is defined as the fraction of the incoming requests that cannot be processed by the ViewMailServer component and therefore needs to be forwarded to a MailServer component. RRF is important because it quantifies the caching benefits offered by the ViewMailServer component when placed close to clients. In Table 8.1, the RRF is given as a constant. A more detailed explanation of how the RRF value can be computed at run-time is provided in Appendix D.

Given these properties, the deployment and linkage conditions and effects can be expressed as follows.

8.1.5 Deployment conditions and effects

The deployment conditions illustrated in Table 8.1 can be divided into qualitative and quantitative conditions. A qualitative condition evaluates to True or False, depending on the current state of the environment. For example, the ViewMailServer should be deployed on a node only if the connection between the ViewMailServer and the MailServer is secure. In general, the quantitative conditions can be further divided into resource requirements and component-specific requirements. An example of the former is that a node should have sufficient available CPU resource, before deploying a component on that node. Examples of the latter include the ViewMailServer requirements to be deployed on a trusted node, or inside a domain where caching

benefits exceed a certain threshold.

The effects of deploying the `ViewMailServer` on a node, as captured by the application model described in Chapter 4, include: (1) consumption of resources by the component, and (2) availability of the interfaces implemented by the component on that node. The properties of the newly deployed component are computed given the properties of the environment and the properties of the required interfaces.

Table 8.2: Linkage conditions and effects for the mail application.

```
<Property type = Boolean> Secure
  <Crosslink>Interface2.Secure:=Interface.Secure & Link.Secure

<Property type = Float>LinkBandwidth
  <Crosslink>Interface2.BW:=min( Link.BW, Interface.BW )
  <Crosslink>Link.BW:=Link.BW - min( Link.BW, Interface.BW )

<Property type = Interval>Trust
  <Crosslink>Interface2.Trust:=Interface.Trust

<Property type = Float>SendRate
  <Crosslink>Interface2.SendRate:=min((Link.BW/Interface.MessageSize),
                                      Interface.SendRate )
<Property type = Float>Latency
  <Crosslink>Interface2.Latency:=2*Link.Latency + Interface.Latency

<Property type = Float>TLMS
  <Crosslink>Interface2.TLMS:=Interface.TLMS +
                              Interface.MessageSize*Link.Latency/Link.BW
```

8.1.6 Linkage conditions and effects

Table 8.2 illustrates the linkage conditions and effects for some of the properties associated with interfaces. For the e-mail application, they should capture how the privacy of e-mail messages and the time to execute an operation are affected by the state of the network.

The linkage conditions determine what are the properties and resources required by an interface before crossing a link. For example, the `MailServerInterface` requires that the available bandwidth of the link supports the traffic created by users.

The linkage effects describe the relationships between the interface and the link, once an interface has crossed the link. For example, the `Privacy` property of the `MailServerInterface` interface is affected when crossing insecure links. In addition, the traffic created by users consumes part of the available bandwidth of the link.

8.1.7 View specification

In order to improve the deployment process as discussed in Chapter 6, the e-mail application defines two views: `ViewMailClient_Partner` and `ViewMailServer`. The first one implements only a subset of the `MailClient` functionality, and represents a secure alternative of the original `MailClient`, designed to be executed by untrusted users. The latter implements the same functionality as its original component, `MailServer`, but caches only a subset of the original user accounts. Such a view is useful to offset high link latencies, when placed close to clients.

Table 8.5 illustrates the view generation process described in Section 6.3, by presenting the code for the `ViewMailClient_Partner` view, as defined in Table 8.4. Table 8.3 contains the Java code for the `MailClient`. Table 8.4 indicates how the `ViewMailClient_Partner` bytecode should be generated from the `MailClient` bytecode. The output of VIG is presented in Table 8.5. The `ViewMailClient_Partner` is a restricted version of the `MailClient` component, able to send/receive messages, add notes into a remote diary, and query the address book in a secure fashion. Such a component is useful if clients use untrusted machines (e.g., the airport terminal in our

Table 8.3: The original Java object

```
public interface MessageI {
  void sendMessage(Message m)
  Set receiveMessages() }

public interface AddressI {
  String getPhone(String name)
  String getEmail(String name) }

public interface NotesI {
  void addNote(String note)
  boolean addMeeting(String name) }

public class MailClient implements
  MessageI, AddressI, NotesI {
  Account[] accounts;
  void sendMessage(Message mes){}
  Set receiveMessages(){}
  String getPhone(String name){
    findAccount(name).getPhone();
  }
  String getEmail(String name){
    findAccount(name).getEmail();
  }
  void addNote(String note){}
  String addMeeting(String name){}
  Account findAccount(String name){
  { return accounts.get(name); }
}
```

Table 8.4: The rules to define a view

```
<View>
  <Name = ViewMailClient_Partner>
  <Represents>
    <Name = MailClient>
  <Restricts>
    <Interface> name=MessageI
                type=local
    <Interface> name=NotesI
                type=rmi
    <Interface> name=AddressI
                type=switch
  <Adds_Fields>
    <Field name = accountCopy
  <Adds_Methods>
    <MSign> VMC_Partner()
    <MBody> /** constructor body **/
    /** Additional methods, including some
    required by the data consistency
    protocol (see Section 6.6)
    **/
  <Customizes_Methods>
    <MSign> addMeeting(String)
    <MBody> /** new code for method**/
```

setting) to check e-mail.

The minimal view is fully described by a name (`ViewMailClient_Partner`), and a represented object (`MailClient`). In addition, the view implements a list of restricted interfaces (`MessageI`, `AddressI`, `NotesI`), defines new fields (`accCopy`) and methods, and customizes existing methods (`addNote`). The `AddressI` and the `NotesI` interfaces are defined as *rmi*, respectively *switch*. According to the Java RMI and Switchboard schemas presented in Chapter 3, the interfaces and the methods defined

Table 8.5: View source code.

```
public interface MessageI {
  void sendMessage(Message mes)
  Set receiveMessages()
}

public interface AddressI extends Serializable {
  String getPhone(String name)
  String getEmail(String name)
}

public interface NotesI extends Remote {
  void addNote(String note) throws RemoteException
  boolean addMeeting(String name) throws RemoteException
}

public class ViewMailClient_Partner implements
  MessageI, AddressI, NotesI {

  Account[] accounts; NotesI notesI_rmi;
  AddressI addrI_switch;

  public ViewMailClient_Partner (String[] args) {
    /** rmi code **/
    notesI_rmi = (NotesI) Naming.lookup(...);
    /** switchboard code **/
    addrI_swi = (AddressI) Switchboard.lookup(...);
    /** user supplied code **/  }

  void sendMessage(Message mes){/** the original code **/}
  Set receiveMessages()        {/** the original code **/}
  String getPhone(String name){return addrI_swi.getPhone();}
  String getEmail(String name){return addrI_swi.getEmail();}
  void addNote(String note)   {notesI_rmi.addNote(); }
  boolean addMeeting(String name){/** user supplied code **/}
}
```

by the AddressI and NotesI interfaces are modified to become RMI, respectively Switchboard. In general, a view definition must also contain descriptions of several special methods: at least one constructor declaration, and complete implementations for data consistency-specific methods, if data consistency is needed.

8.1.8 Security requirements

The security requirements of the e-mail application are captured by the Privacy and Trust properties. These two properties are influenced by the environment, and reflect how the actual messages would be affected if they would traverse that environment. For example, a sensitive message has the property Privacy set to True, and the Trust levels very high. By setting high levels of Trust, a user specifies that those messages are very sensitive and should not be transmitted to components running on untrusted nodes (i.e., the Trust level of the nodes is less than the Trust level of the message). If this message crosses an insecure link with the Secure property set to False, its own Privacy property becomes False, meaning that the message is no longer protected. Similarly, if the message is saved on a machine with Trust level very low, the message could be read by users who should not have that right.

8.1.9 Data consistency requirements

The only components that have data consistency requirements are the ViewMailServer and the MailServer components. A ViewMailServer is defined as a data view of the MailServer, because it serves a subset of the users who have accounts on the MailServer. As described in Chapter 6, a component-based application can effi-

ciently use the data consistency protocol implemented by PSF, if the application specifies: (1) data properties, (2) quality triggers, and (3) extract and merge methods.

Table 8.6: `ViewMailServer` code to create data properties.

```
1  ViewPropertyList createPropertyList() {
2    ViewProperty v1 = new ViewProperty( property_User,
                                         ViewProperty.SET,users );
3    Vector value = new Vector(); value.add( new String( "null" ) );
4    ViewProperty v2 = new ViewProperty( property_Folder,
                                         ViewProperty.SET,value );
5    value = new Vector();  value.add( minLevel );
6    ViewProperty v3 = new ViewProperty( property_MinLevel,
                                        ViewProperty.SET,value );
7     value = new Vector();  value.add( maxLevel );
8     ViewProperty v4 = new ViewProperty( property_MaxLevel,
                                        ViewProperty.SET,value );
9     value = new Vector();  value.add( strength );
10    ViewProperty v5 = new ViewProperty( property_Strength,
                                        ViewProperty.SET,value );
11    value = new Vector();  value.add( new String( "null" ) );
12    ViewProperty v6 = new ViewProperty( property_StartingDate,
                                        ViewProperty.SET,value );
13    ViewPropertyList vpl = new ViewPropertyList();
14    vpl.addProperty( v1 );  vpl.addProperty( v2 );
15    vpl.addProperty( v3 );  vpl.addProperty( v4 );
16    vpl.addProperty( v5 );  vpl.addProperty( v6 );
17    return vpl;
}
```

Data properties. The data properties capture the most important features of the shared data that needs to be kept consistent. In the case of the e-mail application (see Appendix: E), these properties are:

- *User*: The name of the user who has an account on the mail server;
- *Folder*: The name of the folder that needs to be kept consistent;

- *Minimum* and *maximum levels*: Only the messages with `Trust` levels between these two levels should be saved on the `ViewMailServer`;
- *Strength*: Depending on the value of this property, the entire message is saved on the `ViewMailServer` (1), or only its header (0);
- *Starting date*: Only the messages received after this date should be sent to the `ViewMailServer`.

These properties are sufficient to both determine which replicas need to see which updates, and what data should be sent. Table 8.6 illustrates how the `ViewMailServer` component creates the list of data properties.

Quality triggers. In general, triggers can be defined as a Boolean expression with any number of parameters. In the realization of the `ViewMailServer` component, the only triggers of interest are time-based. The lines of code from Appendix E, which define the triggers are given below. As part of constructing an instance of the cache manager, the `ViewMailServer` provides the following parameters: the name of the view (line 2), the list of data properties (line 4), the initial mode of operation (weak consistency in this example – line 5), and the time-based quality triggers (line 6).

Table 8.7: ViewMailServer code to create the cache manager.

```
1  cm = new CacheManagerImpl( cmArgs,
2                             "ViewMailServer'',
3                              this,
4                              createPropertyList(),
5                              CacheManagerImpl.CM_WEAK,
6                              ``t < 10000'', ``t < 10000'', ``t < 0'',
7                              null, 1 );
```

Merge and extract methods. A view needs to provide implementations for the methods that extract updates from- and merge updates into- the view and the original component. The following table illustrates the merge and extract methods provided by the `ViewMailServer` view definition. The complete code can be found in Appendix E. Each extract method is responsible for using the property list to extract the relevant data from each object (i.e., view and original component). Similarly, the merge methods take as input the most up-to-date data and copies it into the object.

Table 8.8: `ViewMailServer` pseudo-code to create data properties.

```
1  ObjectImage extractFromObject(Object object,ViewPropertyList propList){
2    MailServerInterface parentMailServer = (MailServerInterface) object;
3    Vector accounts = parentMailServer.getAccounts( propList )
4    return new ObjectImage( accounts);
5  }

6  ObjectImage extractFromView( ViewPropertyList propList ) {
7    Vector accounts = parentMailServer.getAccounts( propList )
8    return new ObjectImage(aAccounts);
9  }

10 void mergeIntoObject( Object object, ObjectImage image,
11                       ViewPropertyList propList ) {
12   MailServerInterface parentMailServer = (MailServerInterface) object;
13   Vector accounts = image.getImage( propList );
14   parentMailServer.mergeAccount(accounts);
15 }

16 void mergeIntoView( ObjectImage image, ViewPropertyList propList ) {
17   Vector accounts = image.getImage( propList );
18   this.mergeAccount(accounts);
19 }
```

Conclusion. In order to use PSF, users and application developers must specify QoS requirements and the application specification, and reason about what properties and expressions make sense. While this process may be non-trivial and time-consuming

for some applications, this process is (1) *natural*, because properties and expressions are realistic representations of the actual user requirements and application behaviors, (2) and *local*, because all specifications are provided at the level of components. Hence, it is our hope that the benefits offered by this process outweighs its costs.

8.2 Analysis of the PSF-related code

In order to evaluate whether the process of using PSF is burdensome, this section presents the PSF-specific code added to the `ViewMailServer` component of the e-mail application. The added code represents the second type of information provided by the application, beside the application specification discussed in the previous section. Appendix E lists the complete Java code of the `ViewMailServer` component.

8.2.1 PSF

Table 8.9 uses the `ViewMailServer` component to illustrate how a component can become PSF-aware. A component is enabled to use the PSF framework if it extends the `PSFObject` class. This class contains only one function which is used by PSF to automatically connect components: `setInterface(String interf, Object obj)`. When the PSF deployment module receives from the planning module a list of components that need to be deployed and instantiated, the deployment module is responsible for sending the appropriate commands to the wrappers. Each wrapper is then responsible for instantiating the components on nodes, advertising their functionality, creating the necessary connections, and starting the components. In order to connect two components, the wrapper associated with one component needs to discover the inter-

faces advertised by the other component (using DisCo, as explained in Section 3.3.5), obtain a remote reference to the other component, and assign the remote reference to the appropriate variable of the first component. In Table 8.9, the `setInterface` method receives as input the remote reference to a component that implements the `MailServerInterface` interface, and assigns this reference to a variable defined by `ViewMailServer` as of type `MailServerInterface`.

Table 8.9: `ViewMailServer` pseudo-code.

```
1  public class ViewMailServerDiscoMultiThreaded extends PSFObject
2    implements MailServerInterface, ViewInterface, Runnable {

3      CacheManagerImpl_Disco cm = null;
4      ViewPropertyList propertyList;

5      public void run() {
6        parseArguments( arguments );
7        server = new Server_MultiThreaded( arguments );
8        new Thread( server ).start();
9        cm = new CacheManagerImpl_Disco( cmArgs,
10         "ViewMailServerDiscoMultiThreaded,
11         this,createPropertyList(),CacheManagerImpl.CM_WEAK,
12         ``t<10000'', ``t<10000'', ``t<0'', null, 1 );
13       cm.switchToWeak();
14       cm.initImage();
15     }

16     public void sendMessage( Message message ) {
17       cm.startUseImage( "send message" );
18       server.sendMessage( message );
19       cm.endUseImage( "send message" );
20     }

21     public void setInterface( String interf, Object obj ) {
22       if( interf.equals( "mailSW.server.MailServerInterface" ) )
23         remoteServer = (MailServerInterface) obj;
24     }
25}
```

8.2.2 Data consistency

As described in Section 6.6, the API's exposed by the cache manager to the view are very simple and easy to use. Table 8.9 highlights the consistency-related lines of code.

The flow of operations is as follows: (1) `ViewMailServer` creates at start-up the cache manager (lines 9-12), (2) initializes the data (line 13,14), (3) works with data (lines 16-19), and (4) stops the cache manager. Beside the functionality of a mail server, the `ViewMailServer` is also responsible for extending the `ViewInterface` interface (lines 21-26), which requires the implementation of the extract/merge methods. Table 8.8 shown earlier in this chapter illustrates the merge and extract methods. Note that this information just communicates what state is extracted/merged and is not concerned with when exactly this functionality is invoked at run-time by the coherence system.

8.3 Summary

This chapter has evaluated the expressivity of the models described in Chapter 4 by showing how the models are capable of capturing the behavior of the e-mail application.

As expected, the e-mail application specification identifies the appropriate properties that could be used to define the component deployment and linkage conditions and effects, and describes the security and data consistency requirements. The first advantage of PSF is the fact that the process of providing this information is *natural* (i.e., properties and expressions are realistic representations of the actual user requirements and application behaviors), and *local* (i.e., all specifications are provided at the

level of components).

Beside the application specifications, application developers must also modify the original components in order to use PSF. The second advantage of PSF is the small number of additional lines required in order for a component to become PSF-aware.

Chapter 9

Performance Evaluation of Individual Techniques

This chapter evaluates the performance, in isolation, of each of the techniques discussed in the previous chapters: (1) the planning algorithm, (2) the view generation tool, (3) the Switchboard communication abstraction, and (4) the data consistency protocol. The techniques are evaluated as integral parts of PSF in Chapter 10.

This evaluation helps assess the run-time overheads of PSF. Once a client makes a request to access the functionality of a component-based application that is registered with PSF, PSF is responsible for constructing a valid application configuration (technique 1) using customized components (technique 2), and installing and connecting the components into the network (technique 3). In addition, PSF provides a data consistency service (technique 4) to satisfy application data consistency requirements.

The following four sections present a thorough analysis of each technique.

9.1 Analysis of the planning algorithm

This section characterizes the run time and the nature of the deployments produced by a Java-based implementation of the planning algorithm, for different application behaviors and network conditions. Ideally, the planning algorithm should be efficient, scale well with the size of the network and the complexity of the application, and find solutions if they exist.

The measurements were taken on an AMD Athlon XP 1800+ machine, running Red Hat 7.1 and the J2RE 1.3.1 IBM Virtual Machine.

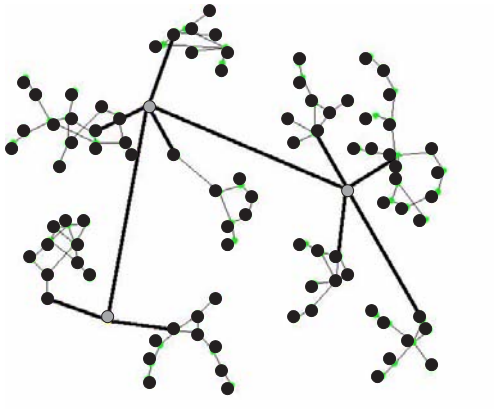


Figure 9.1: Network with 99 nodes.

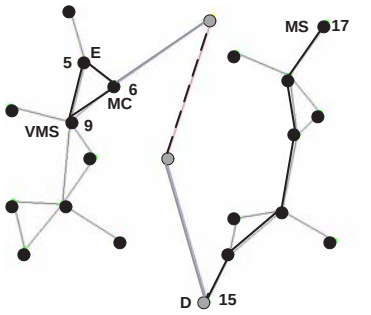


Figure 9.2: Network with 22 nodes.

The testbed for all experiments consists of eight different wide-area network topologies (N_k) generated by the GT-ITM tool [11], where $k \in \{22, 33, \ldots, 99\}$ represents the number of nodes in the network. Figures 9.1 and 9.2 illustrate the N_{99} and the N_{22} networks. Each topology simulates a WAN formed by high speed and secure stubs connected by slow and insecure links. The initial topology configuration files (.alt) were augmented with link and network properties using the Network EDitor

tool [57]. Examples of such properties include latency, available bandwidth, and security. The links inside a stub were developed to have 100Mbps available bandwidth and the `Secure` property set to `True`. The links connecting the stubs were labeled to have 10Mbps available bandwidth and their `Secure` property set to `False`.

The performance of the planner is evaluated using two applications — the **e-mail service** described in Section 2.3.1, and a **webcast** service described later in this section. The goal in both applications is to place the client components on specific nodes. In both cases, the "best" deployment is defined as the one with the fewest number of components.

The planner was tested by running six different experiments. The next paragraphs present in more detail the goal, the description, and the results of each experiment.

Experiment 1: Planning under various conditions. The purpose of the first experiment is to show that the planner finds a valid component deployment plan even in

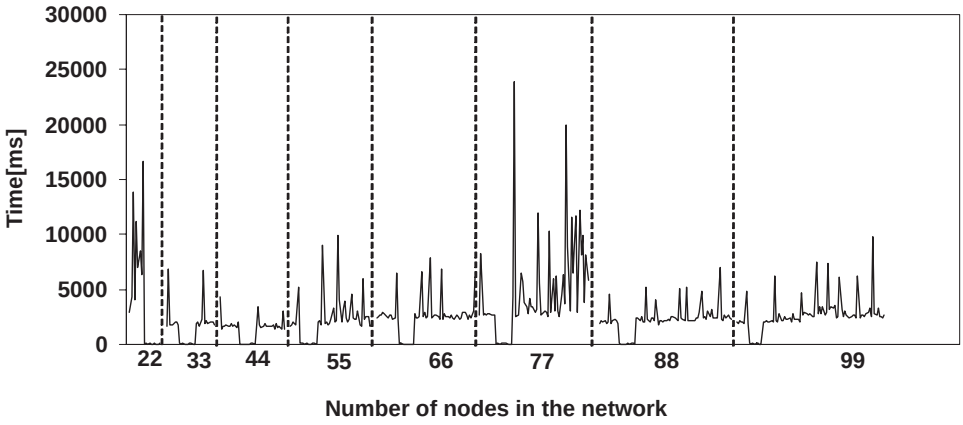


Figure 9.3: Planning under various conditions.

hard cases, and usually does so in a small amount of time. The experiment, involving the mail service application, is conducted as follows. For each network topology N_k, where $k \in 22, 33, \ldots, 99$, and for each node n in the network N_k, the goal is to deploy a `MailClient` component on the node n, given that the `MailServer` is running on some node. The algorithm indeed finds a solution when one exists.

Figure 9.3 shows the time needed to find a valid plan; each N_k network is marked by dashed lines. The data points in the figure correspond to the following cases. When the client and the server are located in the same stub, the algorithm essentially finds the shortest path between two nodes. This case corresponds to the data points almost equal to 0 in every region of the graph. Placement of a client in a different stub requires inserting the `ViewMailServer`, `Encryptor`, and `Decryptor` components into the path, and therefore takes longer. The large run-times for the N_{22} and N_{77} networks can be explained by the fact that many resource conflicts are identified only during the last phase of the algorithm. The two networks above have a bigger number of low-bandwidth insecure links between stubs as compared to the others. Because of this, the algorithm constructs and checks many logically correct plans that fail during symbolic execution due to resource restrictions.

Experiment 2: Scalability w.r.t. network size. This experiment shows how the performance of the algorithm is affected by the size of the network. Taking the N_{99} network topology as reference, this experiment starts with a small network with only two stubs, and then adds one stub at a time until the original 99-node configuration is achieved. For each of the obtained networks, the planner is asked to place `MailClient` on a fixed node.

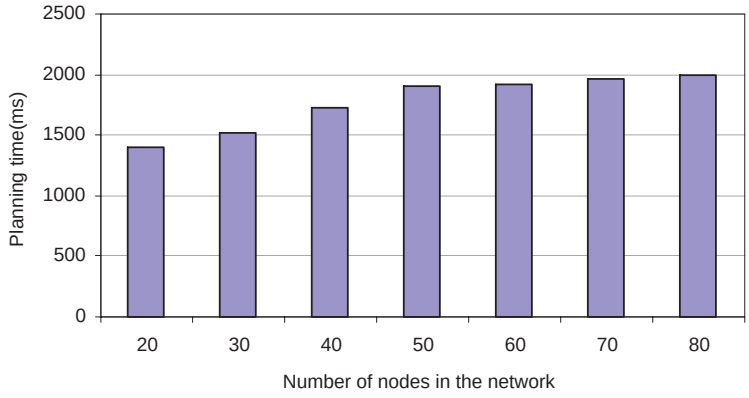


Figure 9.4: Scalability w.r.t. network size for the e-mail application.

As shown in Figure 9.4, the running time of the planner increases very little with the size of the network. Moreover, the graph tends to flatten. Such behavior can be explained by the fact that the regression phase of the algorithm considers only stubs reachable in the number of steps bounded by the length of the final plan. Even this set is further pruned at the progression stage. Therefore, the planning algorithm is capable of identifying the part of the network relevant for the solution, without

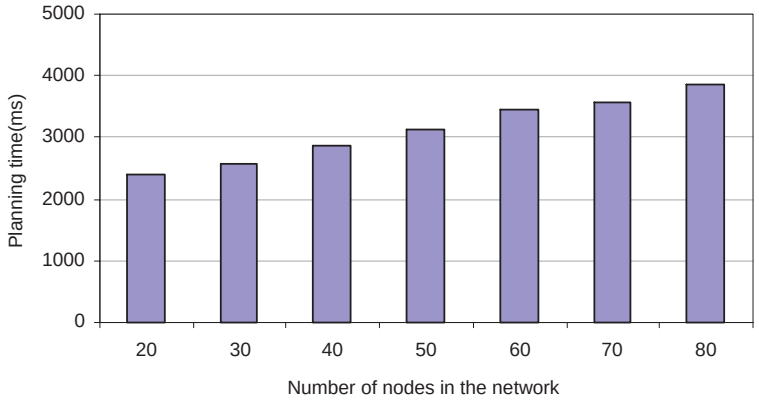


Figure 9.5: Scalability w.r.t. network size for the webcast application.

additional preprocessing.

Experiment 3: Complex application structure. The e-mail application used in the above experiments requires only a chain of components. An important feature of the planning algorithm is that it can support more complicated application structures, i.e., DAGs and even loops. To verify that the planner behavior is not negatively affected by DAG-like structures, this experiment generates deployments for a webcast service. The webcast application (Figure 9.6) consists of a `Server` that produces images and text, a `Client` that consumes both, and additional `Splitter`, `Zip`/`Unzip`, and `Filter` components for splitting the stream and reducing the bandwidth requirements for the text and image data respectively. The DAG structure arises because of splitting and merging the image and text streams.

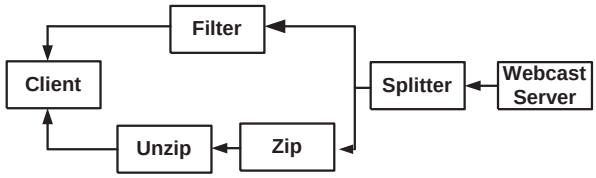


Figure 9.6: Logical component deployment for the webcast application.

In this case, the goal for the planner is deployment of the `Client` component on a specific node, given that the `Server` was separated from it by links with low available bandwidth. Figure 9.5 illustrates the running time of the algorithm as a function of the network size and validates our assertion.

Experiment 4: Scalability w.r.t irrelevant components. The scalability of the planning algorithm depends on whether components are (1) absolutely useless components that

can never be used in any application configuration, (2) components useless given availability of interfaces in the network, or (3) useful components, i.e., those that implement an interface relevant for achieving the goal and whose required interfaces are either present or can be provided by other useful components.

Figure 9.7 shows the performance of the planner in the presence of irrelevant components. The two plots correspond to two situations: the mail service application augmented first with ten useless components which implement interfaces not used by the e-mail application, and then with ten components that implement interfaces meaningful to the application, but require interfaces that cannot be provided. The useless components are rejected by the regression phase of the algorithm and do not affect its performance at all. Slight fluctuations are a result of artifacts such as garbage collection. Components whose implemented interfaces are useful, but required interfaces cannot be provided can be pruned out only during the second phase, which also takes into account the initial state of the network (the required interfaces might be available somewhere from the very beginning). The running time increases as a result of processing these components in the first phase (polynomial in the number of compo-

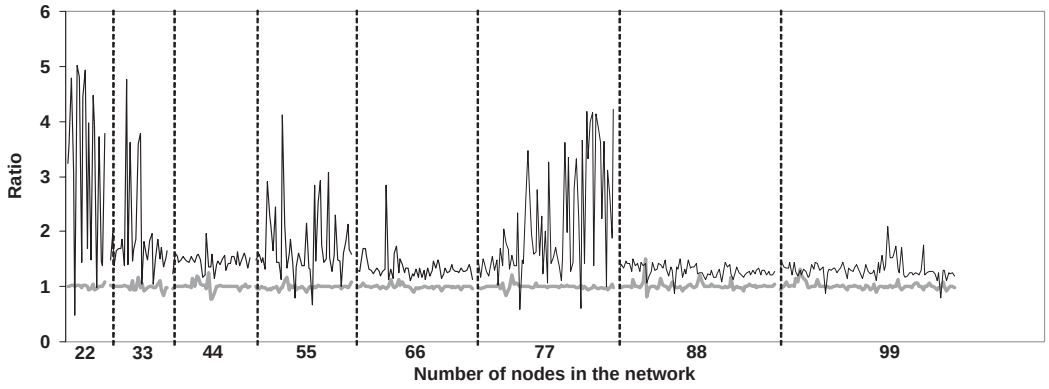


Figure 9.7: Scalability w.r.t. increasing number of irrelevant components.

nents). This phase takes time polynomial in the size of the relevant part of the network and the number of components, so, even though the ratio of total running times shown on Figure 9.7 can reach 5, the actual difference is small, and becomes less significant as the size of the problem increases.

Experiment 5: Scalability w.r.t. relevant components. Figure 9.8 shows that the planner performance scales with increasing number of useful components. The plots correspond to four cases: `5 comp` represents the first experiment on the original N_{99} topology, where all five components of the mail service may need to be deployed; `4 comp`, `3 comp`, and `2 comp` cases represent situations where the network properties are modified such that all links become fast (i.e., `ViewMailServer` is not needed), secure (i.e., the `Encryptor`/`Decryptor` pair is not needed), or both secure and fast (i.e., only `MailClient` and `MailServer` need to be deployed) respectively.

The choice of whether a useful component is actually used in the final plan is made during the third phase of the algorithm, which in the worst case takes time exponential in the length of the plan. Larger numbers of useful components increase the branching

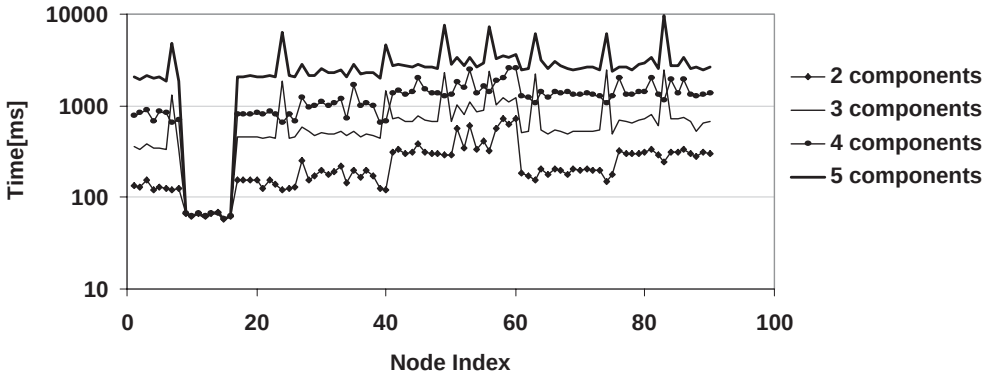


Figure 9.8: Scalability w.r.t. increasing number of relevant components.

factor of the progression graph described in Chapter 5, and therefore the base of the exponent. This means that in hard cases (very strict resource constraints, multiple component types implementing the same interface, highly connected networks) the initial planning can take long. However, as shown below, new components can be added quickly to extend existing plans.

Experiment 6: Reusability of existing deployments. In practical scenarios, by the time a new client requests a service, the network may already contain some of the required components. In order to see how the planning time is affected by reuse of existing deployments, this experiments starts with the webcast application and the N_{99} topology where the `Server` was present on a fixed node, and analyzes the planning costs for the goal of putting the `Client` on each of the network nodes in turn. The x-axis in Figure 9.9 represents the order in which the nodes were chosen. The network state is saved between the runs, so that clients can join existing paths. The assumptions are that clients are using exactly the same datastream, and there is no overhead for adding

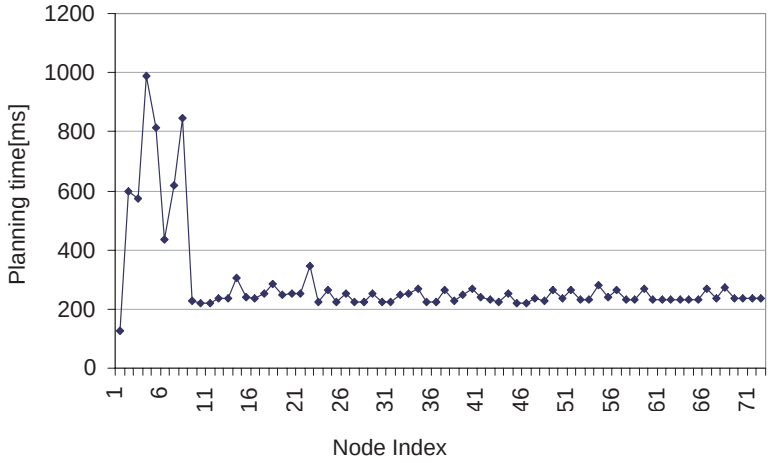


Figure 9.9: Reuse of existing deployments.

a new client to a server.

As expected, it is very cheap to add a new client to a stub that already has a client of the same type deployed (this corresponds to the majority of the points in Figure 9.5), because most of the path can be reused. The problem in this case is effectively reduced to finding the closest node where the required interfaces are available.

Other points on the graph correspond to the following cases. The initial deployment of a client in a stub other than that of the server is the most expensive. Deployment of a client in the stub where the server is running is almost free. Deployment of the first component at each stub given that there is a component deployed somewhere in the network is much cheaper than the very first deployment: at least the path inside the server's stub and, possibly, a part of the path through transit nodes are reused.

Conclusion. These experiments show that the planning algorithm is efficient and scales well with the size of the network and the number of components and interfaces defined by the application. However, there are cases when the algorithm degenerates into an exhaustive search algorithm. In order to deal with such cases, the algorithm can take advantage of various memoization techniques and remember partial solutions. In addition, the planning algorithm returns solutions that consumed resources in a greedy way. Follow-on work by other researchers has started studying possible solutions to reduce the amounts of consumed resources by discretizing the resource space [86].

9.2 View generator

This section evaluates whether the view generation process described in Section 6.3.3 is suitable for use by PSF to generate views at run-time. Ideally, the view generation process should take a small amount of time compared with the rest of the deployment process.

In general, VIG can be considered efficient if its performance is comparable to the performance of Javac when compiling a similar Java program. The experiments described below measure the running time and the size of the generated classes for both VIG and Javac [83]. The experiments were run on a 800MHz machine running Windows XP and using the JDK 1.4 environment.

For a correct comparison between VIG and Javac, views are defined by the original bytecode and the VIG-specific XML files. The views are contrasted with their corresponding Java files. VIG uses the XML files to generate bytecode, while JAVAC compiles the Java files. The Javac execution time and the size of the generated classes depend on the flags set for Javac. For a thorough examination, Javac is executed with the following flags: -g (full debugging information), -g:none (no debugging information), -O (optimization turned on), and no flags (some debugging information).

The two views used in this evaluation are (1) *vms*, the `ViewMailServer` described in the section 2.3.1, and (2) *Vtest*, a complex example developed for debugging reasons. The former is simple but large (the original object has 427 lines of code, the Java file corresponding to the generated view has 974 lines of codes, and the XML description file has 149 lines). The latter example is very complex, but small (the original code has almost 100 lines of code, the Java file corresponding to the generated

view has 50 lines of code, and the XML file has 87 lines).

The running time and the sizes of the generated classes are measured and compared in Figures 9.10 and 9.11. The time to generate the complex example (*Vtest*) with VIG is smaller than the time to generate it with Javac, while the opposite is valid for *vms*. The explanation is that the example is very small and full of code to be optimized (unused variables, unused methods, and unreachable code), while *vms* is large and barely optimizable. VIG does not perform any type of optimization, thus taking less time for *Vtest* and more for *vms*. Javac tries to optimize the Java code, thus executing more for *Vtest* and less *vms*.

Figure 9.11 shows that VIG generates classes almost equal in size with the ones generated by Javac -g:none, if there are no optimizations to be performed. Otherwise, VIG generates smaller files.

Conclusion. From the point of view of running time and sizes of the generated bytecode files, VIG is comparable with Javac. In fact, the size of the generated files can

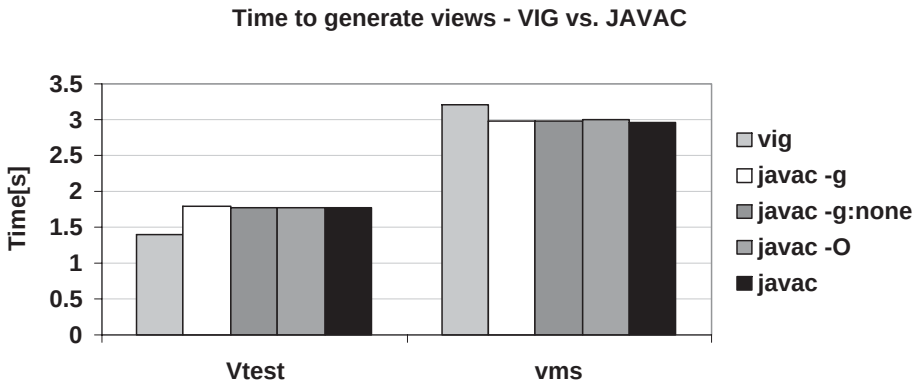


Figure 9.10: Time to generate views

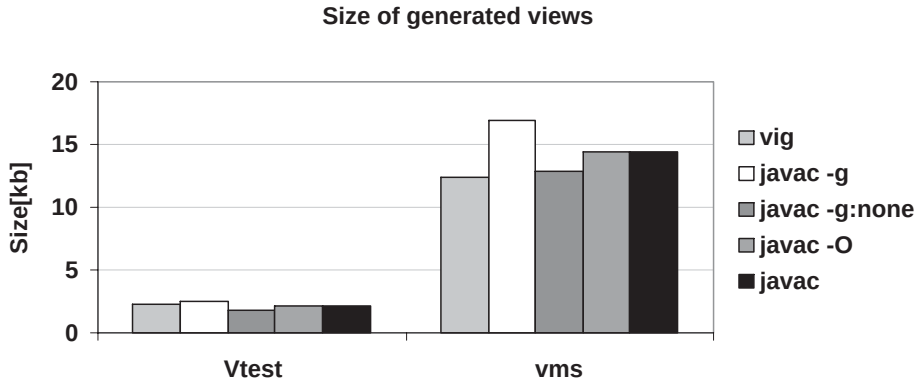


Figure 9.11: Size of generated views

be improved by carefully adding compiler optimization techniques to VIG without paying major running time penalties. The efficiency of VIG makes it suitable for use in dynamic environments, where views need to be generated at run-time, on demand.

9.3 Data consistency protocol

This section evaluates the costs and the benefits of the data consistency protocol by observing its behavior when PSF deploys the airline reservation application system described in Section 2.3.2. The goal of this evaluation is to show that the data consistency protocol augmented by application specific information, is *efficient* (it reduces the number of messages sent between cache managers and the directory manager), *adaptable* (it switches between various consistency levels), and *flexible* (allows the application to control the consistency levels by defining quality triggers).

Experiment 1: Eficiency. The efficiency of the data consistency protocol is evaluated by measuring the number of messages generated by the consistency protocol and

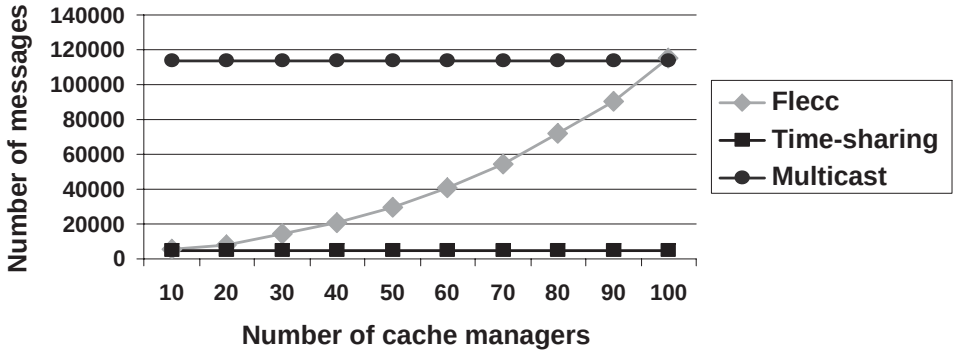


Figure 9.12: Number of messages sent between the cache manager and the directory manager.

comparing it with the number of messages generated by a time-sharing protocol and a multicast-based protocol. The time-sharing protocol schedules the travel agents one at a time and allows them to execute until completion. In this way, the number of control messages between the directory manager and the cache managers is minimum, since no conflicting operations are present. The multicast-based protocol runs all agents concurrently and does not discriminate between cache managers. Because it receives updates between all of them, the number of messages between the directory manager and the cache manager is maximum.

The testbed for this experiment is a cluster with 30 Linux-based nodes simulating a LAN environment. The experiment executes 100 travel agents connected to a main database. All travel agents execute the same sequence of operations: (1) create the cache manager, (2) set the mode of operation to weak, (3) initialize the data, (4) execute in a loop `pullImage`, `startUseImage`, reserve tickets for a flight, `endUseImage`, (5) stop the cache manager. Each travel agent defines a property ("Flights") that contains a list of all the served flights. The number of travel agents that serve similar flights is initially 10, and increases in increments of 10 up to 100. The consistency

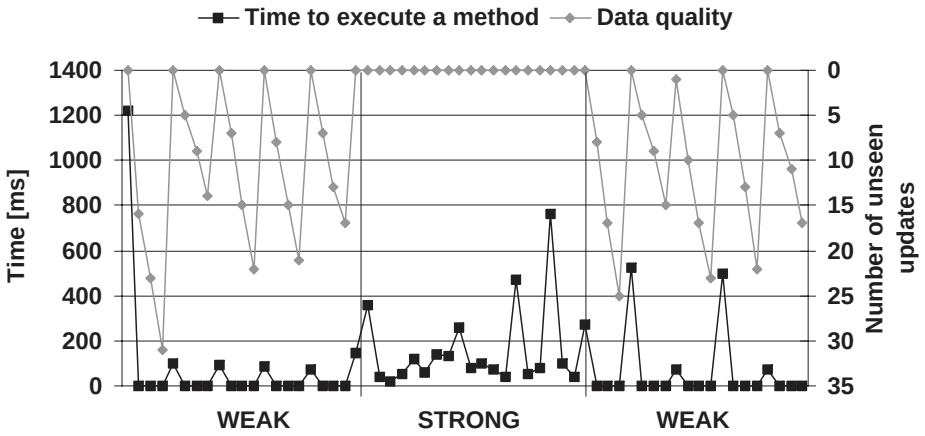


Figure 9.13: Time to execute a method vs. that quality of the used data, when the cache manager switches from WEAK mode to STRONG mode, and back.

requirements of each travel agent is to always execute on the most current data; thus updates need to be sent between travel agents. Figure 9.12 shows how the data consistency protocol adapts the number of control messages by computing the conflicting travel agents based on their properties. The data consistency protocol reduces the number of messages sent between the directory manager and the cache managers, by sending messages only to interested parties.

Experiment 2: Adaptability. Often, component-based applications dynamically change their data consistency requirements at run-time. For example, a travel agent might serve a set of users browsing for tickets, and thus start with weak consistency requirements. If a user changes from a browser to a buyer, the travel agent must immediately change its consistency requirements to strong consistency. Ideally, component-based frameworks should support such applications by changing the data consistency protocol accordingly.

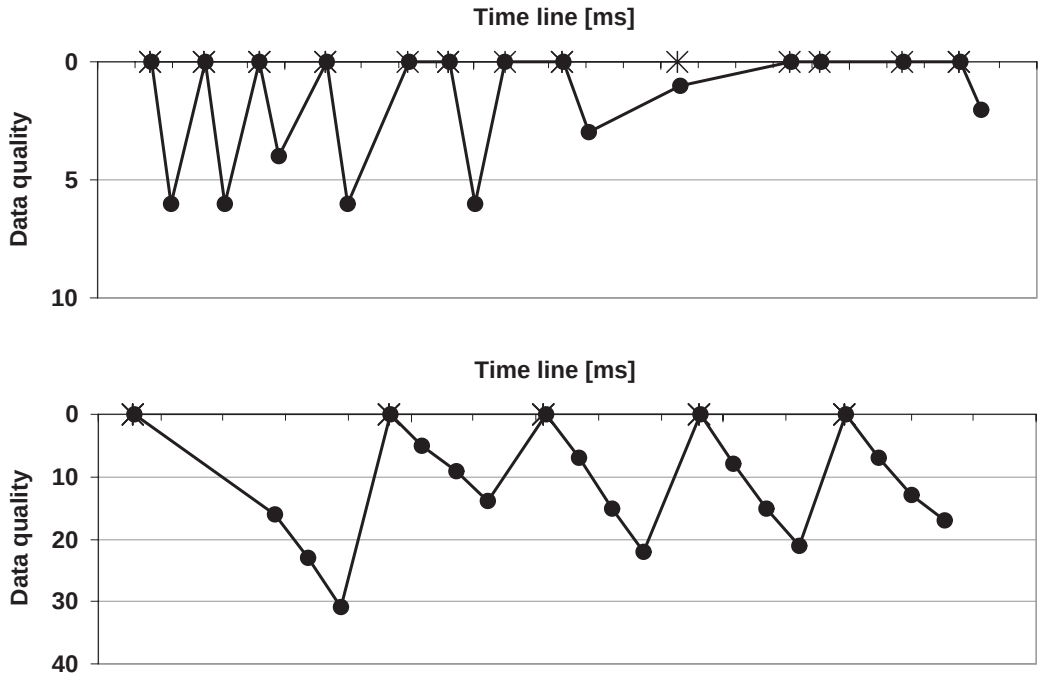


Figure 9.14: Number of remote updates not seen by a cache manager running in WEAK mode, when views define pull/push trigger or not.

In order to measure the adaptability of the data consistency protocol, this experiment deploys ten conflicting travel agents connected to the main database, all running in the same LAN. Initially, the travel agents start in weak mode and execute in a loop `pullImage`, followed by a sequence of several $<$ `startUseImage`, reserve tickets, `endUseImage` $>$ operations. After that, the travel agents switch to strong mode, and execute the same set of operations. In the last phase, the travel agents switch back to weak and execute the same operations. This experiment measures the time to execute a method (including `startUseImage` and `endUseImage`) and the quality of the data seen during the execution. The quality of the data is computed as the

number of remote unseen updates to the shared data. Figure 9.13 shows the trade-off between the time to execute a method and the quality of the data used during the execution. On the y-axis, the graph is split into two parts. The upper part represents the time to execute the methods, while the lower part shows the quality of the data. On the x-axis, the graph shows the travel agent execution time line: WEAK, STRONG, WEAK. The conclusion is that the execution time is small when the travel agent is willing to execute on stale data (WEAK mode of operation, where the data quality decreases in time) and increases if the data needs to be most recent (STRONG mode of execution, where the data quality is always the best). More importantly, this trade-off is simply communicated by the application to the underlying system as consistency requirements.

Experiment 3: Flexibility. The impact of quality triggers on the number of messages and the quality of the data is evaluated in this experiment by running ten conflicting travel agents in weak mode, with and without triggers. The triggers are responsible for pulling and pushing updates between the cache managers and the directory manager. The experiment measures the quality of the data and the number of messages generated between the cache managers and the directory managers. The graph from Figure 9.14 illustrates the usefulness of controlling data update requests by explicit `pullImage` requests and pull triggers. The x-axis shows the moments in time when the actual updates are received by a representative travel agent (the updates are marked by stars). The y-axis shows the measured data quality for every method executed by the travel agent. The lower plot represents a travel agent which explicitly pulls the image before executing every fourth method. The upper plot represents the same travel

agent that defined a time-based pull trigger. Figure 9.14 shows how the quality of data is improved when the travel agents define triggers compared with the case when the travel agent do not specify triggers. However, the cost of the improved data quality is an increased number of messages (116 – without triggers versus 182 – with triggers).

Conclusion. As explained in Chapter 2, a practical and efficient deployment process depends on several factors, including having an efficient and flexible data consistency protocol. The experiments described above assert the claim that the data consistency protocol described in Chapter 6 is efficient by reducing the number of messages sent between replicas, and is able to flexibly adapt to changes in the application consistency requirements.

9.4 Switchboard

Switchboard represents the secure communication abstraction used by PSF to create secure channels between all entities. Because of this, the efficiency of Switchboard directly influences the efficiency of the entire deployment process.

First, this section evaluates the Switchboard performance by measuring the time and the CPU resource consumed by a client and a server when communicating through Switchboard. The first two experiments described below were run on a cluster of 30 Linux-based nodes divided into five domains; the available bandwidth and latency of each link were emulated by the Click modular router [25]. In both experiments, the Switchboard server was running in one domain and clients were located in each of the five domains. Also, different rates of client requests were obtained by injecting

controlled delays between client Switchboard calls. The goal of these experiments is to verify whether Switchboard has a predictable behavior and scales well with the number of clients simultaneously making calls to the same Switchboard server.

The second part of this evaluation measures the performances of both Switchboard and RMI-SSL. RMI-SSL is a communication abstraction that emulates the Java RMI protocol on top of SSL [37]. Ideally, the performance of Switchboard should be comparable to the one of RMI-SSL.

Experiment 1. The first experiment assumes the following network configuration. Intra-domain links have 100Mbps available bandwidth and 0ms latency. The links connecting the four domains to the one where the server is located have the following characteristics:

- 50Mbps available bandwidth and 20ms latency;
- 70Mbps available bandwidth and 10ms latency;
- 20Mbps available bandwidth and 70ms latency;
- 56Kbps available bandwidth and 100ms latency.

Given such a network, one client connects to the server from each domain and makes Switchboard RPC calls with messages of different sizes (1K, 5K, 10K) and various delays (0ms, 50ms, 100ms, and 150ms). This experiment measures (1) the average time for a message to reach the server (*one-way*), (2) the average time to make the call and receive the answer from the server (*two-way*), and (3) the average CPU consumed by a client to make a Switchboard call. The goal is to see how the time and the CPU consumption are influenced by the network characteristics.

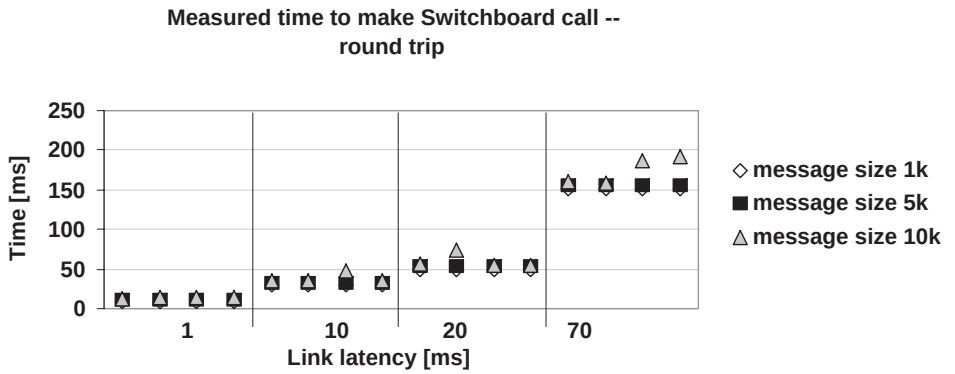


Figure 9.15: Average time to make a Switchboard call.

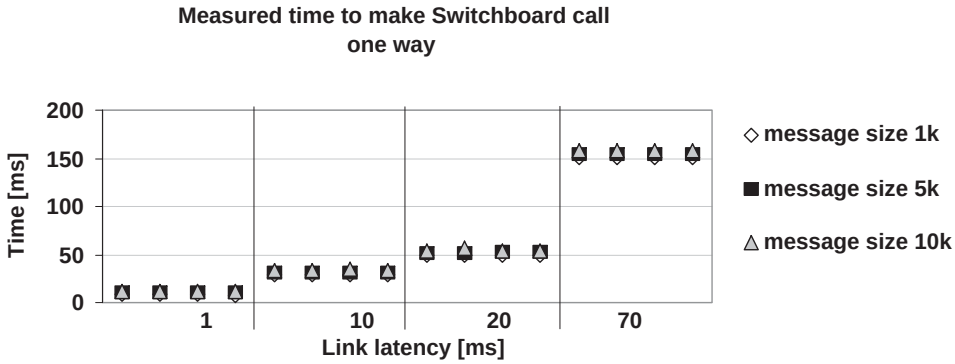


Figure 9.16: Average time necessary for a message to reach the server.

Latency: Figures 9.15 and 9.16 illustrate the time consumed by the Switchboard client to make calls against a remote server. The first figure shows the time spent until a message reaches the server; the second figure also contains the time until the client receives an answer back from the server. One notices that the times are proportional with the latency between links, which is the expected behavior.

CPU usage: The CPU resource consumed by a client when making Switchboard requests is affected mainly by the available bandwidth and the delay between requests. Figure 9.17 illustrates the measured values of the CPU consumption. As expected, the

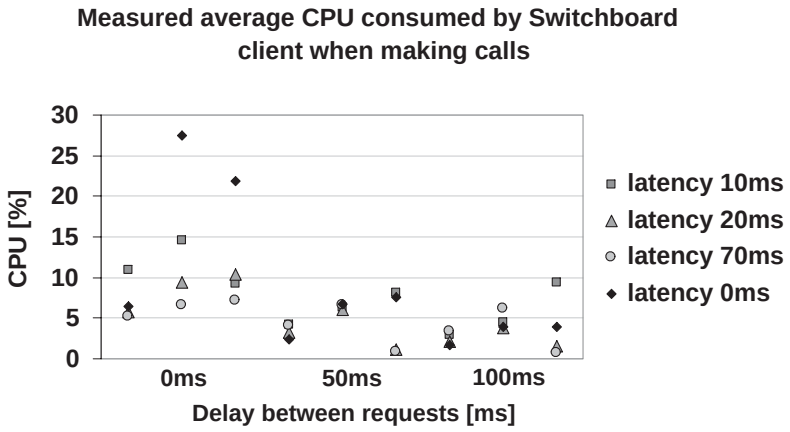


Figure 9.17: Average CPU consumed by client when making Switchboard calls.

average amount of the CPU resource consumed by a client when making Switchboard calls is not significant.

Experiment 2. In the second experiment, the network is defined as one domain containing all 30 nodes, where all the links have 100Mbps available bandwidth and 0ms latency. The number of clients connected to the server increases from 5 to 20 in increments of 5, the message sizes vary from 1k, 5k, to 10k, and the delays between client calls are 50ms, 100ms, and 150ms. The average time to make a Switchboard call is measured on the client side, while the average CPU consumed is measured on the server side. The goal of this experiment is to determine how the server behavior is influenced by the rate of incoming Switchboard calls, and ascertain any scaling bottlenecks .

Latency: Figures 9.18 and 9.19 show that the average time consumed by a client to make a Switchboard request, when multiple clients connect to the server, linearly

depends on the number of clients simultaneously connected to the server.

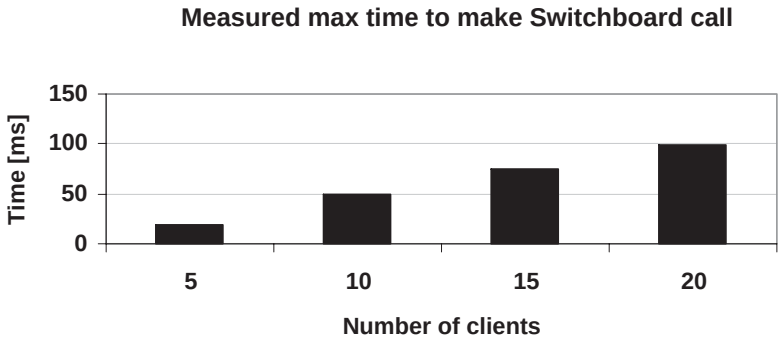


Figure 9.18: Average time necessary for a message to reach the server, when there are multiple clients.

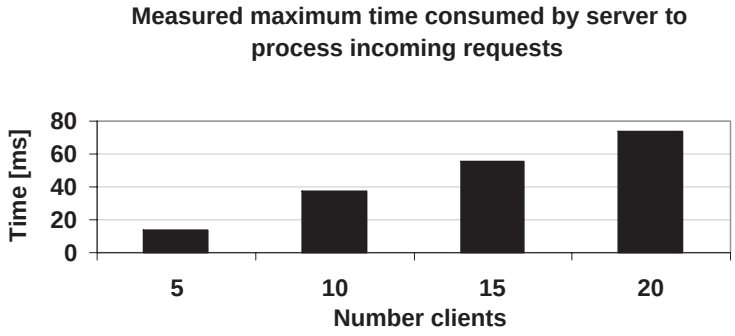


Figure 9.19: Average time necessary to make Switchboard call, when there are multiple clients making requests.

CPU usage: The CPU resources consumed by the server when accepting Switchboard requests from multiple clients depends on the number of incoming requests and the size of the messages sent in those requests. Figure 9.20 illustrates the actual CPU share consumed by the server when accepting and processing requests. The results show that the CPU consumption grows linearly with the number of client requests,

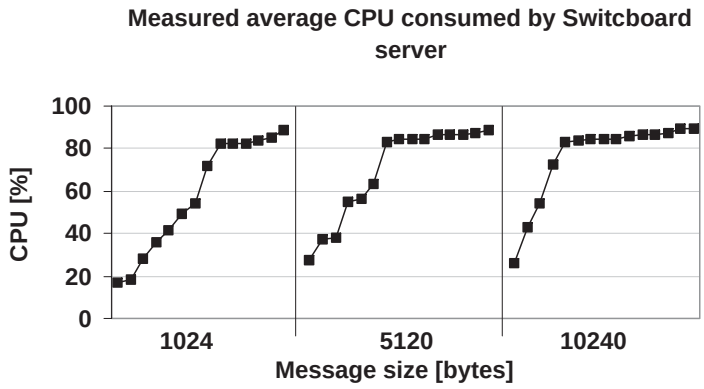


Figure 9.20: Average cpu consumed by client to send a message.

until it reaches a maximum.

The two experiments described above show that Switchboard has a predictable behavior and scales well with the number of clients simultaneously making Switchboard calls.

Experiment 3. The third experiment was run between two Linux machines running within the same domain. This means that the latency between them is practically 0ms and the available bandwidth is 100 Mbps. Both the RMI-SSL and the Switchboard communication abstractions were tested by sending messages of various sizes between a client and a server. In the case of RMI-SSL, the experiment varies also the frequency of authenticating the client and the server. Usually, SSL authenticates the two entities only once, upon creating the connection. However, one of the advantages of Switchboard is its continuous monitoring of the network. This means that, in the extreme case, the effect of running Switchboard is equivalent to authenticating entities for every method call in RMI-SSL. In this experiment, the authentication frequency

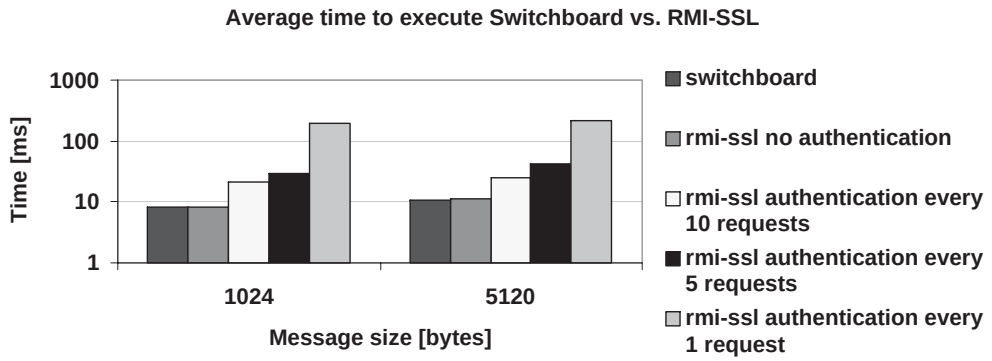


Figure 9.21: Switchboard compared with RMI-SSL.

changes from 0 (no verification), 1 (verification for every method call), 5 (verification every 5 method calls), to 10 (verification every 10 method calls).

The results are presented in Figure 9.21 show that the performance of Switchboard is comparable with the one of RMI-SSL. In average, Switchboard outperforms RMI-SSL because Switchboard provides continuous monitoring of the trust relationship more efficiently that RMI-SSL. In Switchboard, one entity is immediately notified if the other entity lost its credential and the secure channel is terminated. RMI-SSL obtains a similar effect by verifying the identities of both entities for every method, which results in the worst performance of all cases.

9.5 Summary

This chapter has evaluated the techniques used to automatically deploy component-based applications, as independent modules. The results show that: (1) the planning algorithm is efficient, scales well with the size of the network and the application

complexity, and finds a solution if one exists, (2) the efficiency of the view generation process enables component-based frameworks to generate views at runtime and on-demand, (3) the data consistency protocol is efficient by minimizing the update messages exchanged between replicas, and support flexible data consistency requirements, and (4) the Switchboard communication abstraction is efficient and behaves in a predictable manner. The next chapter evaluates how these mechanisms come together in the context of a complete application.

Chapter 10

Performance Evaluation of PSF

This chapter evaluates the benefits and the costs of using the Partitionable Services Framework (PSF) to automatically deploy component-based applications in heterogeneous environments by deploying a complete application and measuring its performance when accessed by multiple users. The main goal of this evaluation is to answer the following questions:

1. Does the dynamic, automatic configuration selection feature of PSF help? Is it sufficient to select some small subset of component deployments, or does choosing between different application configurations help?

2. What is the cost incurred by component-based applications when they are automatically deployed by PSF?

3. How does the efficiency of the automatically deployed applications compare with the efficiency of manual deployments of similar configurations?

4. What is the nature of the deployments chosen by PSF? In particular, can PSF

automatically reach deployments that may be non-obvious because PSF takes into consideration the user QoS requirements, the application specification, and the current state of the network?

The next section describes the testbed and the application used to run the experiments. The following four sections evaluate each of the questions listed above.

10.1 Experimental platform

Testbed. The testbed used to run the experiments is presented in Figure 10.1, and consists of 30 nodes, organized in 3 domains: *Domain 0*, *Domain 1*, and *Domain 2*. Each domain is represented by a fully-connected graph. The intra-domain links are considered to be *secure* and *fast* (i.e., 100Mbps available bandwidth and 0ms latency), while the links connecting the three domains are *insecure* and *slow* (i.e., the

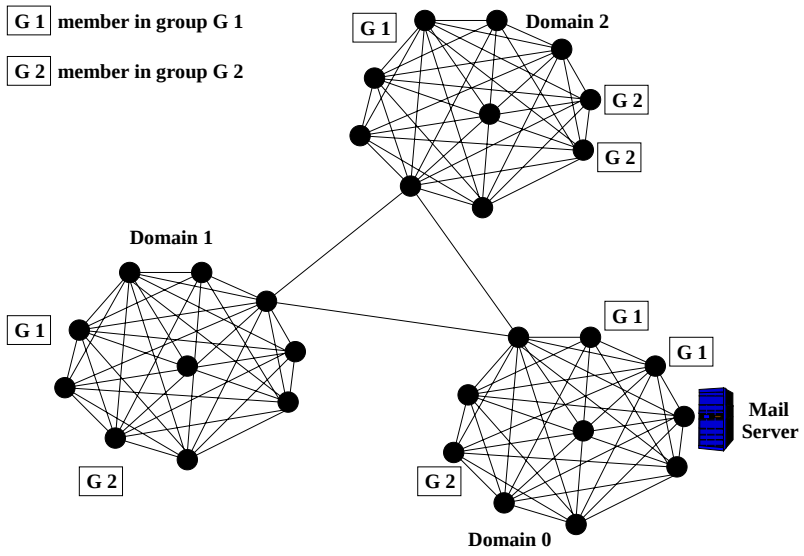


Figure 10.1: Test bed

link between domains 0 and 1 has 70Mbps available bandwidth and 10ms latency, the link between domains 0 and 2 has 20Mbps available bandwidth and 70ms latency, and the link between domains 1 and 2 has 56Kbps available bandwidth and 100ms latency). Nodes are additionally classified as *internal* or *gateway* nodes. The only difference between them is that gateway nodes are resource constrained. The testbed is simulated on a cluster with 30 nodes running Linux 2.4.19, with 512M of memory, and AMD Duron 1.2GHz processor. The link available bandwidth and latency were emulated using the Click modular router.

Application. The experiments measure the performance of the e-mail application introduced in Section 2.

The application specification model described in Chapter 4 requires that the e-mail application provide the following information for each of its components: the deployment conditions and effects, the linkage conditions and effects, and the interactions between the components and the environment. The actual application specification presented in Chapter 8 follows this model, and lists the set of properties relevant to the e-mail application and the functions that capture the application behavior. These functions were obtained by using the profiling results shown in Appendix C, as explained in Appendix D.

The results given in Appendix C indicate the costs of executing different operations for each component of the e-mail application. Appendix D computes the CPU consumption and the response time when sending messages for the `MailClient`, `MailServer`, and `ViewMailServer` components; in addition, the appendix computes the `RRF` value for the `ViewMailServer` component. The average CPU share utilized

by a component in the unit of time is calculated as the sum of the CPU shares consumed for each operation, as described in Appendix D. Similar reasoning is done for computing the response time and the `RRF`.

Beside the application specification, the application also provides information to be used by the data consistency protocol. In the e-mail applications, all updates are pushed and pulled based on triggers defined using time (e.g., in this scenario, every 10 seconds).

These characteristics are very important because they affect the performance of the clients connected to the `ViewMailServer`, and the benefits of placing a cache close to clients (see the explanation in Section 10.2).

User behavior. Clients are divided into two groups $Group_1$ and $Group_2$, where each group has four clients. For each group, clients are characterized by the following parameters:

S The number of messages sent in the unit of time;

R The number of "receive messages" operations executed in the unit of time;

D The fraction of new received messages deleted by the client for every "receive message" operation;

M The percentage of messages sent by the client from one group to the clients from another group.

The actual values for each parameter are given in Table 10.1. Beside the QoS requirements, users were also required to provide the information about their behavior

Table 10.1: Characteristics for user behavior.

	$Group_1$	$Group_2$
S [messages/sec]	1	2
R [receive operations/sec]	0.2	0.1
D [%]	50	70
$M\ Group_1$ [%]	80	10
$M\ Group_2$ [%]	20	90

to PSF. Appendix D explains in more detail how the application models incorporate this information to derive component conditions and effects.

Once connected to a mail server, clients start sending and receiving messages; in addition, clients delete a portion of the incoming messages every time they receive new messages.

Settings. The experiments were run in the context of two different settings. The common factor in both settings is the goal – all users are trying to simultaneously access their e-mail accounts by connecting to the `MailServer` running in Domain 0. The difference between the two settings is the way the eight users are divided between the three domains, as shown in Table 10.2. The columns show how many users from each group are located in each domain. Figure 10.1 illustrates the first setting, where the users are identified by their group number.

Table 10.2: The two user settings for the experiments.

Setting 1			Setting 2		
	$Group_1$	$Group_2$		$Group_1$	$Group_2$
Domain 0	2	1	Domain 0	4	0
Domain 1	1	1	Domain 1	0	0
Domain 2	1	2	Domain 2	0	4

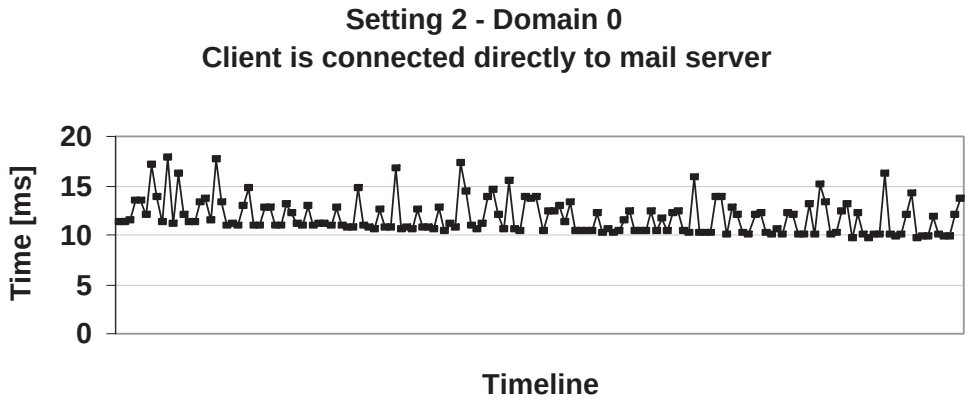


Figure 10.2: Performance of client in Domain 0 and Setting 2.

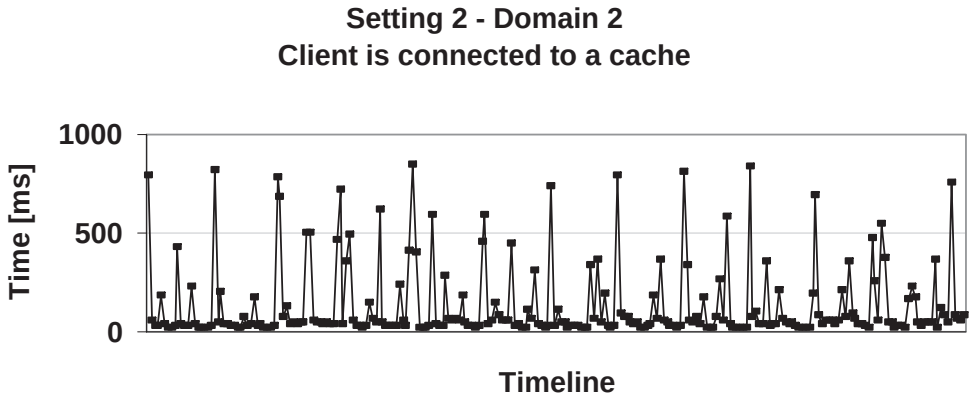


Figure 10.3: Performance of client in Domain 2 and Setting 2.

Figures 10.2 and 10.3 illustrate the behavior of representative clients in Setting 2, if the users are manually connected to the mail server. The manual configuration chosen as the reference point for the following experiments, connects the clients from Domain 0 directly to the `MailServer`, and the clients in Domain 2 to the `MailServer` through a `ViewMailServer`. The reason for placing the `ViewMailServer` is to offset the high latency of the links between domains 0 and 2. For each client, the graphs show the measured time to send a message. The results show that clients in Domain 0

take between 10ms and 15ms to send a message, while the time seen by clients in Domain 2 is affected by the periodic caching-related activities of the `ViewMailServer`. However, the time seen by clients in Domain 2 between caching events is equal to the time seen by clients in Domain 0.

The goal of observing the client behavior (Figures 10.2 and 10.3) is to observe the performance of the e-mail application and see how it compares with the performance of the e-mail application when automatically deployed by PSF. Ideally, PSF should find application configurations which improve or are comparable to this manual configuration. All results shown in the following experiments are compared against averages taken over the results shown in Figures 10.2 and 10.3.

10.2 Deploying various application configurations

For each setting, this first set of experiments measures the average time to send a message if the mail components are deployed under various configurations. For Setting 1, PSF found the following application configuration:

- `MailClient` components are deployed in Domain 0 and directly connected to the `MailServer`;
- One `ViewMailServer` component is deployed in Domain 1 and connected to the `MailServer`; multiple `MailClient` components are deployed in Domain 1 and connected to the `ViewMailServer`;
- Similarly to Domain 1, one `ViewMailServer` and multiple `MailClient` components are deployed in Domain 2; the `ViewMailServer` is connected to the

`MailServer`, and the `MailClient` components are connected to the `ViewMailServer`.

The automatic deployment found by PSF for Setting 2 is identical with the automatic deployment found for Setting 1 with the exception that no components are deployed in Domain 1.

For clarity, the graphs show only the measurements for representative clients. The x-axis represents one client in each of the three domains. The y-axis is the average time to send a message.

For Setting 1, the graph shown in Figure 10.4 illustrates how the average time to send a message is influenced by the state of the environment and by the components deployed in the network. The two deployments compared are (1) connecting the clients directly to the `MailServer` ("manual deployment-direct connection") and (2) connecting the clients from Domain 0 directly to the `MailServer` and the clients from Domain 1 and 2 through caches located inside the respective domains ("automatic deployment"). For the clients in Domain 0, the direct connection has the best performance because the links have low latency and high bandwidth. However, the

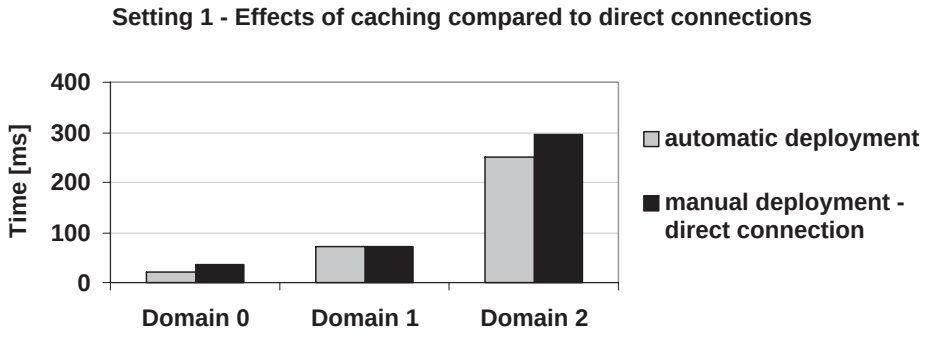


Figure 10.4: Various application configuration for Setting 1.

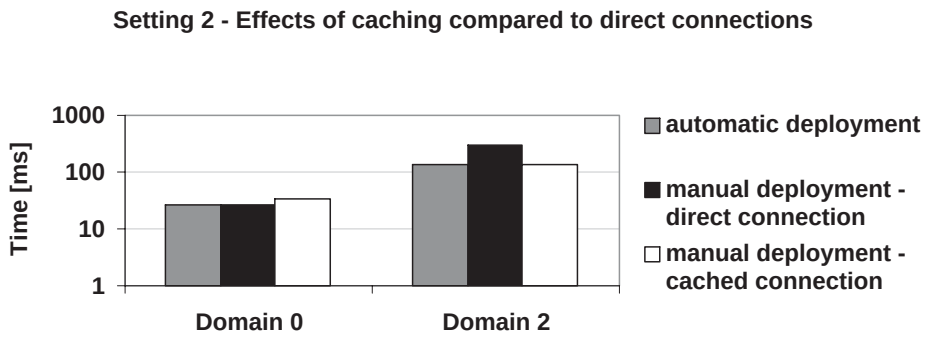


Figure 10.5: Various application configuration for Setting 2.

same application configuration does not work as well for the clients from Domains 1 and 2, because the inter-domain links have high latencies. The performance is improved by placing a cache close to the client. However, the improvement is not drastic because the clients are sending messages to all other clients, which are spread almost equally between domains. Thus, the average is dominated by that cost.

For Setting 2, where there is more locality of traffic, the results are as shown in Figure 10.5. In this case, the two application configurations compared are (1) connecting all clients directly to the `MailServer` ("manual connection-direct connection"), (2) connecting all clients through caches deployed in each domain ("manual connection-cached connection"), and (3) connecting the clients from Domain 0 directly to the `MailServer` and the clients from Domain 2 to a `ViewMailServer` placed in their domain ("automatic deployment"). Obviously, the first configuration is not good for the clients from Domain 2 because they are connected through a slow link, while the second configuration is not good for the clients from Domain 0 because it introduces unnecessary costs. The only solution that provides good performance for all clients is the third one, which is exactly what PSF finds as a valid solution.

This first set of experiments is indeed validating the claim that there is no single application configuration that satisfies general user and application requirements.

10.3 Costs of automatically deploying applications

This section evaluates the cost of deploying component-based applications into heterogeneous environments. In this experiment, four clients from Domain 0 and four clients from Domain 2 asked PSF to connect to the `MailServer` running in Domain 0 (i.e., Setting 2). The automatic deployment found by PSF consisted of (1) `MailClient` components deployed in Domain 0 and connected to the `MailServer`, (2) a `ViewMailServer` component deployed in Domain 2 and connected to the `MailServer`, and (3) `MailClient` components deployed in Domain 2 and connected to the `ViewMailServer`.

Ideally, the cost of automatically deploying the application should be negligible

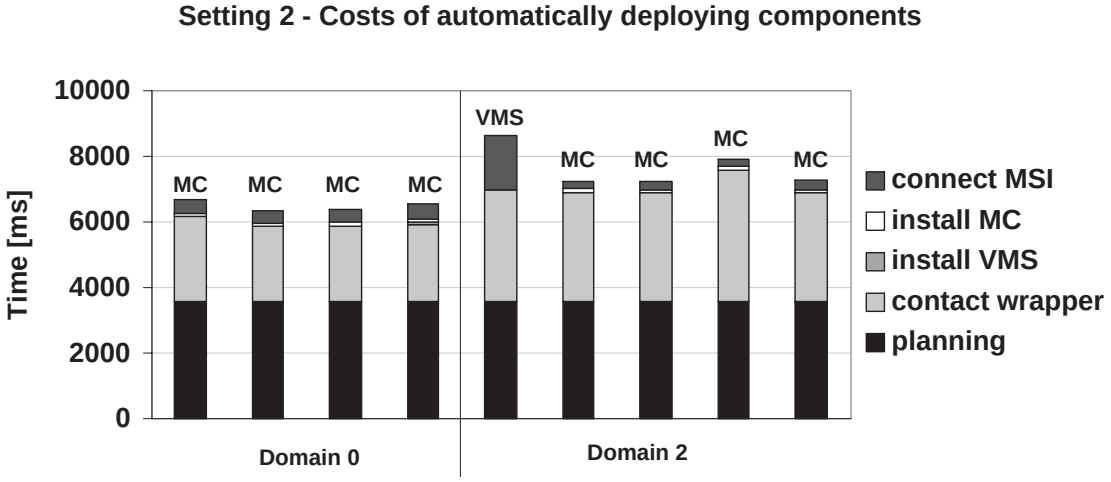


Figure 10.6: Costs of automatically deploying components.

when compared to the application life-time. In PSF, the deployment process is composed out of the following actions: (1) planning for a valid application configuration, (2) contacting the wrappers and sending the installation commands, (3) downloading and instantiating the components, and (4) creating the necessary connections between components. The actual costs are shown in Figure 10.6. The x-axis represents the two domains where components are deployed in Setting 2. The y-axis represents the time to execute each of four steps listed above. As discussed in Chapter 9, the planning cost depends on the size of the network and the complexity of the application. In these experiments, the planning takes less than 4 seconds. The other important costs are the time to contact the wrapper and connect the components with the required components, which together take less than 10 seconds.

In conclusion, the total cost to satisfy a client request depends on how many components need to be deployed and connected. In this experiment, the cost to deploy and connect the `MailClient` varies from 6-8 seconds for most clients to 15 seconds for the first client in Domain 2 (this client also pays for the deployment of the `ViewMailServer`). Even the 15 seconds can be reduced by parallelizing the deployment process. For many long lived applications, such a cost is acceptable.

10.4 Effects of automatic deployment on the application performance

This set of experiments measures the average time to send a message for two representative clients, in two cases. In the first case, the clients from Setting 2 are simultaneously asking PSF to access the `MailServer`; the application configurations

are automatically computed and dynamically deployed by PSF. In the second case, the components are manually deployed in exactly the same configuration as the one found by PSF. Figure 10.7 shows that the differences between the application performance in the two cases is negligible, most probably due to the variable behavior of the JVM (e.g., because of the garbage collection). Therefore, this set of experiments shows that the application performance is not affected in any way by the fact that the application is manually or dynamically deployed.

10.5 Finding non-obvious application configurations

One of the goals of PSF is to find valid application configurations that respect both the client QoS requirements and the application resource requirements. This problem is even more challenging when the environments have resource constraints.

The following four experiments use Setting 2 to measure the average time to send a message when the `MailServer` is running on a node where the available CPU share

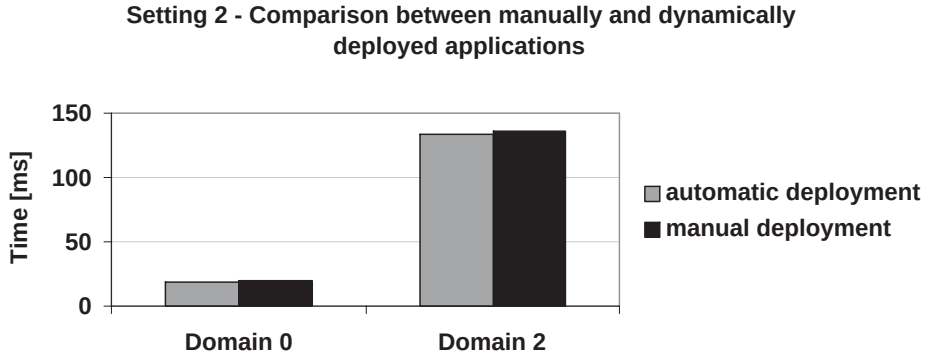


Figure 10.7: Effects of automatic deployment on the application performance.

drops from 100% to 50%, and clients are deployed in the following configurations: (1) manually connecting all clients directly to the `MailServer`, (2) manually connecting the clients from Domain 0 directly to the `MailServer` and the clients from Domain 2 through a `ViewMailServer` placed in Domain 2, (3) manually connecting all clients through `ViewMailServer` components placed in Domain 0 and 2, and (4) automatically deployed by PSF. The automatic configuration found by PSF consisted of two `ViewMailServer` components deployed in Domains 0 and 2 and connected to the `MailServer`, and multiple `MailClient` components deployed in Domains 0 and 2 and connected to the respective `ViewMailServer` components.

The results are presented in Figure 10.8. The x-axis shows only two representative clients from Domains 0 and 2. The y-axis is the average time to send a message, measured in milliseconds. The worst performance is given by the direct connection because all clients are sending their requests to the `MailServer` which is working on a CPU-bound node. The performance is improved for the clients in Domain 2 by placing a `ViewMailServer` in their domain. However, the clients from Domain 0 are

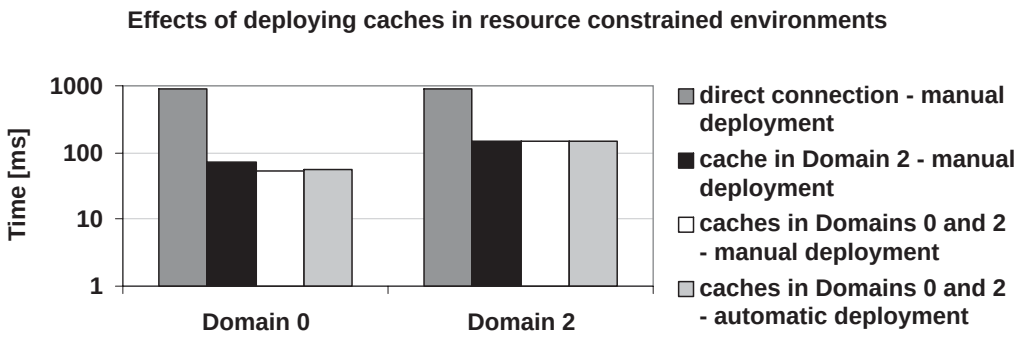


Figure 10.8: Deploying components in resource constrained environments.

still paying an increased cost. The solution found by PSF is to connect all clients through cache components, and the graph shows that this solution yields the best performance.

10.6 Summary

This chapter has evaluated the benefits and the costs of PSF when used to deploy a complete component-based application in heterogeneous environments. The results obtained show: (1) it is beneficial to design applications as sets of components because their flexibility enables dynamic component-based frameworks to find valid solutions under a variety of network conditions, (2) PSF is able to find solutions that satisfy both the user QoS requirements and the application deployment conditions, (3) PSF deploys applications in an efficient and practical fashion, and (4) the application performance is not affected once the application is deployed into the network.

Chapter 11

Summary and Future Work

This chapter summarizes the work presented in this dissertation and identifies some avenues for future work.

11.1 Summary

The key problem in contemporary distributed systems is automatically deploying distributed applications in dynamically changing heterogeneous networks, while ensuring that the user's QoS requirements are satisfied.

One attractive solution is to write distributed applications as sets of components that express at a high level their efficiency, security requirements, and their malleability to environment changes and continuous user input, and create adaptable component-based infrastructures that automatically and transparently deploy the applications with minimal user input.

The challenges in this approach are: (1) *modeling* the applications and the environ-

ment to capture complex application and environment properties and the relationships between them, (2) finding in a timely manner the best application deployment that satisfies both application and network constraints, and the user's QoS requirements (*planning*), and (3) *deploying* the applications in an efficient and practical manner.

This dissertation work has explored the thesis that by exposing qualitative and quantitative properties and relationships between component-based applications and heterogeneous environments, automatic deployment in resource constrained and dynamically changing environments becomes feasible. Feasibility implies that a valid application deployment is found if it exists, the automatic deployment process does not incur significant overhead compared to manually deployed solutions, and the QoS requirements specified by users are satisfied.

In order to validate this idea, this dissertation has described three techniques which: (1) model component-based applications, (2) search for valid deployments in resource constrained environments, and (3) deploy the applications in heterogeneous environments. In addition, this work has described how the three techniques work together as part of a dynamic component-based framework (PSF), in order to achieve the goal described above.

Application and network models. The models presented in Chapter 4 describe the application as a set of components, and the network as a set of nodes connected by links. A component is defined by sets of implemented and required interfaces, where each interface is associated with application-specific properties. These properties are used to express component deployment conditions and effects. Similarly, nodes and links are associated with network-specific properties that specify the effects of the environ-

ment on the deployed application and vice-versa. These models are richer than other models (CORBA, Web Services, OGSA) because they allow the specification of general qualitative properties (e.g., privacy, trust level) in addition to standard quantitative properties (e.g., CPU, bandwidth, latency) [48].

Planning. Given such a model, it now becomes possible to automatically decide which components make up the application and where they are located. The solution presented in Chapter 5 develops an efficient planning algorithm based on classical AI techniques, that takes advantage of the application and network description models. What differentiates this planning algorithm from similar algorithms is its ability to scale with the size of the network and the complexity of the application [58]. Recognizing that the chances of finding a valid plan increase with the diversity of the component set, the set of application components can be enriched by creating at runtime new custom components (*view*s) that have different properties than the original components.

Efficient and practical deployment process. Even after finding a suitable plan, several challenges need to be addressed before the process of deploying the components into network could be considered as efficient and practical. Two of the main problems are providing security and data consistency guarantees and they are discussed in Chapter 6.

Achieving a secure deployment process is difficult because all entities (i.e. users, applications, nodes, links) belong to different administrative domains. The assumption is that each domain defines its own space of properties and credentials, and no

domain has complete knowledge of all spaces. Thus, the challenges of efficiently providing security guarantees include: (1) authenticating and authorizing entities across domains, (2) translating network properties into application properties, (3) enforcing the appropriate access control rules, and (4) creating secure connections between all entities.

The solution presented in this thesis is based on the *dRBAC* trust management system, *views*,and a secure communication abstraction called *Switchboard*. *dRBAC* is a PKI-based trust management and role-based access control system originally developed for expressing and enforcing security policies in coalition environments spanning multiple administrative domains. With dRBAC, domains can define roles and properties as credentials belonging to private spaces, and associate those credentials with each entity. Both problems of entity authorization and property transformation can be reduced to the problem of mapping a role/property local to one domain to a role/property local to another domain. Thus, the framework authorizes entities or transforms properties by building dRBAC credentials graphs that solve the mapping problem [49]. *Views* are defined as customizations of original components. They represents a natural solution for providing appropriate granularity for access control, because one can restrict access to a component by simply removing its functionality. *Switchboard* addresses the last challenge by creating secure channels and continuously monitoring the trust relationships between communicating entities.

Th second problem of creating an efficient deployment process is maintaining consistency among several instances of the same component. The solution consists of a flexible data consistency protocol that uses *views* as the appropriate consistency granularity. The data consistency protocol described in Chapter 6 satisfies the consistency

requirements of general component-based applications (application-neutral) deployed in a variety of configurations (flexible), while using application-specific information. The performance of the data coherence protocol is improved by allowing the application to specify (1) data properties to characterize the shared data, (2) triggers to indicate when updates need to be pushed or pulled between views, and (3) merge/extract methods to merge/extract updates from/into views and original components [50].

PSF. The three techniques were implemented and tested as integral parts of the Partitionable Services Framework (PSF), as described in Chapter 7. PSF is a dynamic-component-based framework which views applications as being dynamically built out of independent components that can be flexibly assembled to suit the properties of their environment, and facilitates on-demand transparent migration and replication of these components at locations closer to clients while still retaining the illusion of a monolithic service. A run-time system is responsible for registering applications with the framework and serving incoming client requests. Whenever a client wants to access a service, the run-time system performs the necessary security checks (authentication and authorization), decides which level of service the client has the right to access, and asks a planning module to compute a valid component deployment.

The benefits of PSF have been evaluated by deploying representative component-based applications in an environment simulating fast and secure domains connected by slow and insecure links. Analysis of the programming and the deployment processes has shown that: (1) the code modifications required by PSF are minimal, (2) PSF appropriately adapts the deployments based on the network state and user QoS requirements, (3) the run-time deployment overheads incurred by PSF are negligible

compared to the application lifetime, and (4) the efficiency of PSF-deployed applications matches that of hand-crafted solutions.

11.2 Conclusions

This work has presented a set of techniques that could improve the process of dynamically deploying component-based applications into heterogeneous environments when integrated into a dynamic component-based framework. The main contributions of this work are:

- Defining suitable component and network *models* that capture the application behavior, deployment conditions and effects, and the network properties. These models allow applications to use general, quantitative and qualitative properties, and black-box functions to create local specifications (i.e., per component).
- Building a scalable *planner* which exploits the expressivity of the component model to reason about the global behavior of the application and efficiently find a valid plan.
- Building an efficient and practical application *deployment* process, which uses *views* to efficiently provide security and data consistency guarantees.

In conclusion, PSF manifests the desired behavior of a dynamic component-based framework, which automatically adapts distributed applications to their environment so as to satisfy user and application QoS requirements.

11.3 Future work

PSF addresses a subset of the issues raised by the process of automatically deploying component-based applications: modeling of applications and environments, efficient planning in resource constrained environments, authorizing entities in a loosely coupled federation of domains, and ensuring continuous and flexible application-level consistency for component-based applications. However, there are many other challenges that remain.

Extending the application model. The PSF application model should be extended to automatically infer the properties of component compositions based on the properties of the individual component. Similarly, the PSF linkage model should be enriched with semantics information, in the same spirit as emerging web services specifications such as BPEL4WS and OWL-S.

Extending the planning algorithm. The application configuration planning problem is complex and, although the planning algorithm described in Chapter 5 is sufficient for the example applications motivating this work, several additional challenges must be addressed in order to obtain an ideal solution. These challenges include: (1) supporting formulae involving parameters of implemented interfaces, (2) optimizing more complex application-specific cost functions than just the length of the plan, (3) allowing users to negotiate when their QoS requirements cannot be satisfied, and (4) allowing the planner to aggregate several incoming requests and plan accordingly.

Closer integration of networking monitoring systems. While the current implementation of PSF assumes a traditional network monitoring system like NWS or Remos, several interesting research problems need to be addressed to build a monitoring system that is secure (i.e. do not reveal state information on one domain to other domains), non-intrusive (i.e. some domains might not accept intrusive monitoring systems) and able to extract both qualitative (e.g. trust, privacy) and quantitative information (e.g. CPU, bandwidth) on the state of the environment.

Decentralization of PSF. Several modules of PSF which rely on a centralized data store should be decentralized. For example, a decentralized planning algorithm would eliminate the assumption of complete knowledge on both the application and the network, and would allow each domain to compute and deploy partial plans on its nodes.

Appendix A

VIG Algorithm

Chapter 6 defines the notion of views and presents a high-level view of the tool that generates views. For completeness, this Appendix describes in details the steps required to generate views.

A.1 VIG generation tool

Let's assume that $\mathscr{P} = \{p_1, p_2, \ldots\}$ is the set of available packages and $\mathscr{C} = \{c_i^{p_j}, c_m^{p_n}, \ldots\}$ is the set of all classes, where $c_i^{p_j}$ denotes a classic Java class c_i defined in a package p_j. The notation c will be used if the package it belongs to is not important. A class $c \in \mathscr{C}$ is defined by a set of imported packages $IP_c \subseteq \mathscr{P}$, a set of implemented interfaces $II_c \subseteq \mathscr{C}$, a set of declared fields $F_c = \{f_1, f_2, ...\}$, and a set of implemented methods $M_c = m_1, m_2, ...$ Each method $m_i(f_i, f_j, ..m_k, m_l, ...)$ is fully described by the used fields $\{f_i, f_j, ...\}$ and the called methods $\{m_k, m_l, ...\}$. We define the following relations between classes:

- $i \vdash c$: class c implements interface i;
- $v < c$: view v represents class c;
- $c_1 \rightarrow c_2$: class c_2 extends class c_1.

```
class ViewDef {
  String viewName;
  String representedClassName;
  Vector importedPackages;
  String extendedClass;
  ViewDef extendedView;
  Vector restrictedInterfaces;
  Vector copyFields;
  Vector copyMethods;
  Vector addFields;
  Vector addMethods;
  Vector addInterfaces;
  Vector customizeMethods;
}

generate_view( String xmlFile ) {

   /** look for the represented object **/
  ViewDef viewDef = parse_XML_file( xmlFile );
  c^q = look_for_class( viewDef.representedClassName );
  if (c ∉ 𝒞) or (q ∉ 𝒫) trigger ClassNotFoundException;

   /** create the view **/
  create v^p, where  v ∈ 𝒞,  p ∈ 𝒫
  set_view_name( v^p, viewDef.viewName )
```

```
/** set the imported packages **/
define IP_v = {p_1, p_2, ... }, where p_i ∈ 𝒫

/** set the extended class **/
excl = look_for_class( viewDef.extendedClass )
if (excl ∉ 𝒞) trigger ClassNotFoundException;
set excl such that ec → v^p

/** set the extended view **/
excl_exv = viewDef.extendedView.representedClass;
if !(excl_exv → c) trigger ExtendedViewNotCorrectException;
v = add_XML_descriptions( v, exv );

/** copy fields **/
for each field f in viewDef.copyFields do
  if (f ∈ F_v) trigger FieldExistsInViewException;
  fd = find_field( f );
  F_v = F_v ∪ f
done

/** add fields **/
for each field f in viewDef.addFields do
  if (f ∈ F_v) trigger FieldExistsInViewException;
  F_v = F_v ∪ f
done

/** add constructors - is similar to add methods **/

/** customize methods **/
```

```
  for each method m in viewDef.customizeMethods do
    if (m ∈ M_v) trigger MethodsExistsInViewException;
    md = find_method( m );
    change_body( md );
    copy_method( md, v );
  done

  /** add methods **/
  for each method m in viewDef.addMethods do
    if (m ∈ M_v) trigger MethodsExistsInViewException;
    copy_method( m, v );
  done

  /** copy methods **/
  for each method m in viewDef.copyMethods do
    if (m ∈ M_v) trigger MethodsExistsInViewException;
    md = find_method( m );
    copy_method( md, v );
  done

  /** add local interfaces **/
  for each interface i in viewDef.addInterfaces do
    for each method m defined by interface i do
       copy_method( m, v );
  done
}

find_field( f ) {
  if (f ∉ F_c) or
     (f ∉ F_b, ∀b s.t. b → c) or
```

```
      (f ∉ F_b,  ∀b, p  s.t.  b^p ∈ 𝒞  and  p ∈ IP_v)
    trigger FieldNotDefinedException;
  return field definition;
}

find_method( m ) {
    if (m ∉ M_c) or
       (m ∉ M_b,  ∀b  s.t.  b → c) or
       (m ∉ M_b,  ∀b, p  s.t.  b^p ∈ 𝒞  and  p ∈ IP_v)
      trigger MethodNotDefinedException;
    return implementation of m
}

copy_method( m, v ) {
  for each used field f by method m do
    if (f ∉ F_v) do
      fd = find_field( f );
      F_v = F_v ∪ f
    done
  done
  for each called method mm by method m do
    if (mm ∉ M_v) do
      mmd = find_method( mm );
      copy_method( mmd, v );
    done
  done
  M_v = M_v ∪ m
}
```

Appendix B

Interactions Between the Entities of the Data Consistency Protocol

Chapter 6 describes the data consistency protocol implemented by PSF to provide consistency guarantees to component-based application. For both the cache manager and the directory manager, the chapter shows the state machines that capture the interactions between views and cache managers, and cache managers and the directory manager. For completeness, this Appendix describes the messages exchanged between all entities (i.e., views, cache manager, and directory manager).

The data consistency interactions are divided into two classes: (i) view - cache manager interactions, and (ii) cache manager - directory manager interactions. The two classes contain the following messages, as described next.

View - Cache manager interactions.

`new CM()` The view creates a new cache manager and provides all the information

necessary: mode of operation, data properties, quality triggers, and merge/extract methods;

`initImage` The view asks the cache manager to initialize the shared data;

`startUseImage`, `endUseImage` The view marks the code that uses the shared data. This is necessary because the cache manager should never extract or merge updates from/into the shared data, when the view is working with it;

`pullImage` The view explicitly asks the cache manager for the current data;

`releaseImage` The view explicitly asks the cache manager to send the current data to the directory manager;

`killImage` The view announces the cache manager that it will stop running.

Cache manager - directory manager interaction.

`registerCM` The cache manager registers the view with the directory manager and provides the necessary application-specific information: mode of operation, data properties, validity trigger, and merge/extract methods;

`getImage` The cache manager asks the directory manager for the most recent data. This message is sent to the directory manager if the either the view explicitly requested it (`pullImage`) or the pull trigger associated with view was evaluated to true;

`invalidateImage` The directory manager asks a view to invalidate its data, if there was a `getImage` request from another view running in strong mode that conflicts

with the former view. Invalidation implies that the former view is no longer allowed to continue its work (e.g. is no longer active);

`releaseImage` The directory manager asks a view to release its data, if there was a `getImage` request from another view running in weak mode and if the two view conflict. After releasing the data, the view is allowed to keep working (e.g. is active);

`sendImageDM` The cache manager sends the current value of the shared data from the view to the directory manager. The possible causes of such a message are: (i) the view explicitly asked for it (`releaseImage`), (ii) the push trigger was evaluated to true, or (iii) as an answer to `invalidateImage` or `releaseImage`. Once it receives the data, the directory manager merges the data into the original component;

`sendImageCM` The directory manager sends the value of the data to the cache manager as an answer to a `getImage` request. First, the directory manager verifies if the current data is "good enough" for the requesting view by evaluating the validity trigger associated with view. If the result is true, the directory manager extracts the data from the original component and sends it to the cache manager. If the trigger evaluates to false, the directory manager uses the static conflict map and the dynamic sets or properties to find out which other views conflict with th requesting view. If the requesting view is running in strong mode, the directory manager sends `invalidateImage` to the conflicting view. Otherwise, the directory manager sends `releaseImage`. Upon receiving data from the conflicting views, the directory manager merges the data into the original component,

extracts the new value, and sends it to the cache manager;

`unregisterCM` The cache manager announces to the directory manager that the view has finished executing.

Appendix C

Gathering Profiling Information

Chapters 4 and 8 have described and evaluated the application specification model introduced by this thesis. However, the expressions used to illustrate the expressivity of this model capture the behavior of the e-mail application in a simplistic way. Appendix D presents a more complete model that accurately captures the resources utilized by the e-mail application components.

In order to create such a complete model, the following assumptions are made: all interactions between components at either end are the result of executing a limited set of operations, and all interactions pass through the Switchboard communication abstraction. For each component of the e-mail application, the experiments described in Section C.1 separately measure the resources consumed by each operation implemented by the component. In addition, Section C.2 performs similar measurements for Switchboard. They are put together to obtain expressions for overall application behavior in Appendix D. These expressions are then fed to the planner presented in Chapter 5 and used to generate the deployments characterized in Chapter 10.

C.1 E-mail application

This section describes two resources — *response time* and *CPU utilization* — consumed by two of the e-mail application components: `MailClient` and `MailServer`. The reason for measuring only these two is that the `ViewMailClient` and the `ViewMailServer` are views of the first two components, as defined in Chapter 6, and their resource consumption can be deduced from these results.

In order to profile the average CPU share utilized by a component in the unit of time, the experiments measure the average CPU shared consumed by the component when independently executing different operations. This information is then used to calculate the total CPU share consumed by the component as the sum of the CPU shares consumed for each operation, as described in Appendix D. Similar reasoning is done for measuring the response time.

The following sections describe in detail what are the resources consumed by components when executing various operations.

C.1.1 Mail server

The main operations performed by the `MailServer` component are: (1) allowing clients to send messages and merging the incoming messages in the appropriate accounts, (2) sending to clients meta-information about their accounts (e.g., number of folders, names of folders, number of messages present in each folder), and (3) sending to clients the messages present in their accounts, in response to a “receive messages” operation.

Each operation is discussed in detail in the following sections.

Delete message operation

This experiment measures the response time and the CPU utilization when the `MailServer` deletes a message from a user account. The experiment starts with one user account with one folder having different numbers of messages (100, 300, 500), of various sizes (1k, 5k, 10k), and measures the response time and the CPU utilization when deleting one message every 0.5s, 1s, or 1.5s. The server performs a linear search in order to find the right message.

Response time. The response time when deleting a message depends on how many messages are in the folder, as shown in Figure C.1. The measured data is fit very well by the expression $response\_time_{server\_delete}(n) = 0.052561 + 0.000906n$, where n is the number of messages in one folder.

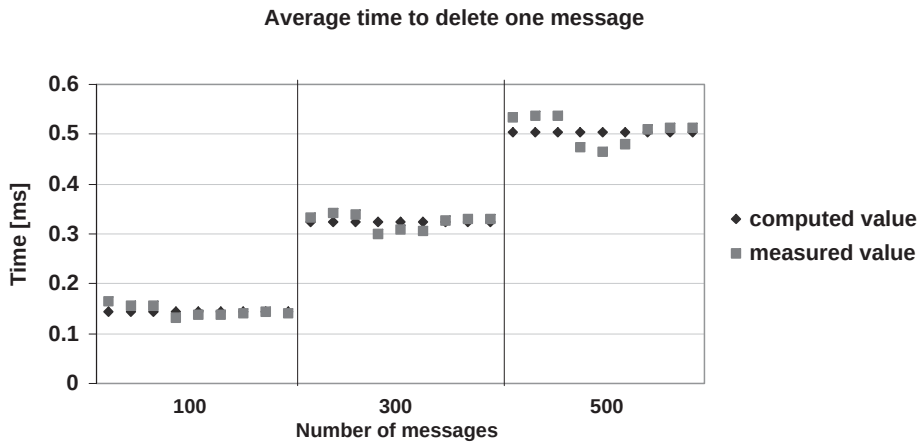


Figure C.1: Average response time when the server deletes one message.

CPU utilization. The average CPU utilization when the server deletes a message depends mainly on the size of the message; one function that approximates the measured values is: $CPU_{server\_delete}(s) = 0.325585 + 0.0000585s$, where s is the size of the message. Figure C.2 shows that the computed values approximate well the measured values.

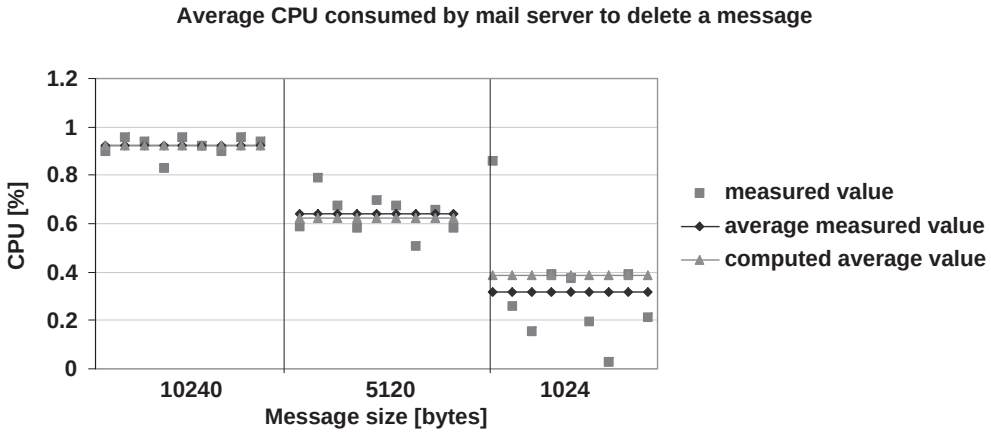


Figure C.2: Average CPU utilization when the server deletes one message.

Get account metadata operation

In order to evaluate the response time and the CPU utilization when extracting the meta-information of a user account, this experiment executes the `getAccountMetadata` command while varying the number of folders in the user accounts, and the delay between requests (0.5s, 1s, and 1.5s). The observation for both the response time and the CPU utilization is that neither is significant.

Response time. The average response time when the server gets the account metadata is insignificant and can be approximated by its upper bound. In this experiment, the upper bound is $response\_time_{server\_getAccountMetadata} = 0.14$, as shown in Figure C.3.

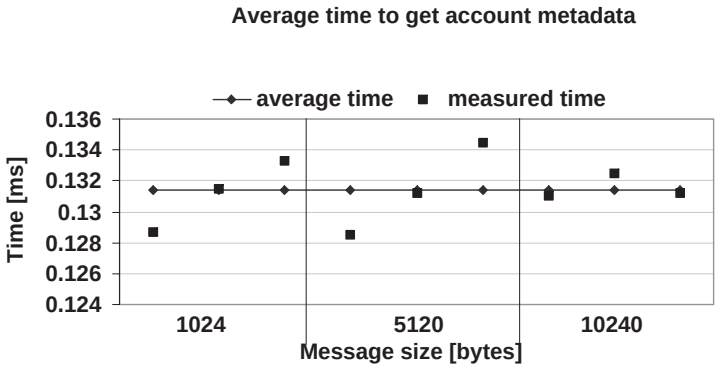


Figure C.3: Average response time when the server gets account metadata.

CPU utilization. The CPU utilization when the server extracts the account metadata belonging to a user depends on the number of folders in the account. In this experiment, it is almost 0: $CPU_{server\_getAccountMetadata} = 0$.

Get messages operation

This experiment initializes the mail server with one account and one folder. However, the number of messages in the folder varies from 100, 300, to 500, and the message size increases from 1k, 5k, to 10k. In addition, the requests are made with delays of 0.5s, 1s, and 1.5s between them.

Response time. The average response time when extracting messages from an account depends on the number of messages in the account. The measured response

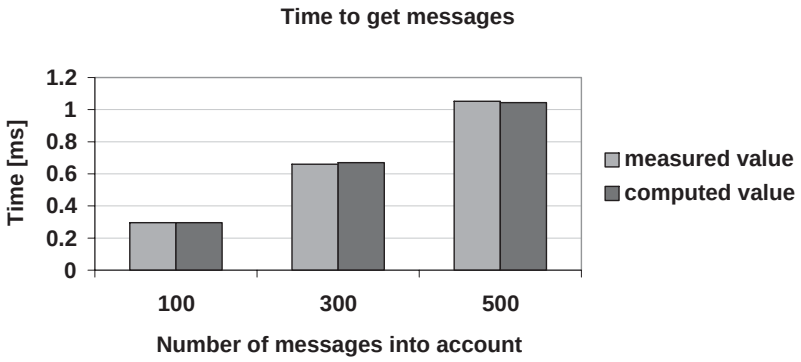


Figure C.4: Average response time when the server extracts messages.

times are shown in Figure C.4. The function that fits well these measurements is $response\_time_{server\_getMessages}(n) = 0.110456 + 0.001863n$, where n is the number of messages.

CPU utilization. From the measured data shown in Figure C.5, one can notice that the CPU utilization depends on the number of messages and the size of the messages. However, the value is small enough to be considered a constant:

$$CPU_{server\_getMessages}(n,s) = 0.865982114 - 0.001435318n + 0.0000841s \leq 2$$

, where n is the number of messages in the user account, and s is the average size of those messages.

Send message operation

During the "send message" operation, the server is responsible for adding the sent message into the user's account. This experiment measures the response time and the

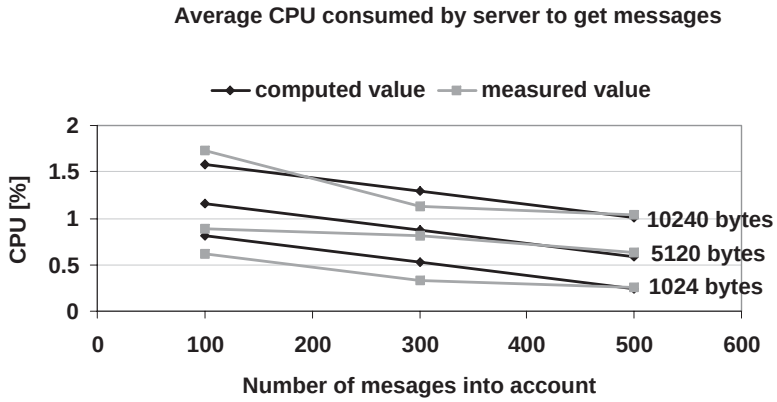


Figure C.5: Average CPU utilization when the server extracts messages.

average CPU utilization when varying several parameters: the number of messages in the destination folder (100, 300, 500), the delay between send message requests (0.5s, 1s, 1.5s), and the message size (1k, 5k, 10k).

Response time. The response time varies only a bit; thus, the measured values presented in Figure C.6 can be approximated by the maximum measured response time

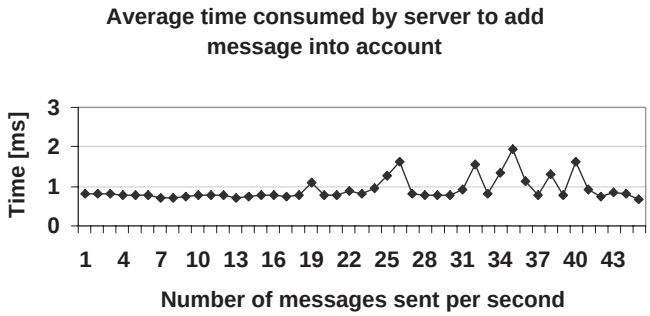


Figure C.6: Average response time when the server adds a message into account.

necessary: $response\_time_{server\_send} = 2$.

CPU utilization. The CPU utilization can be fit well by the function:

$$CPU_{server\_send}(s,n) = 0.590181003 + 0.0000774s - 0.000787544n \leq 2$$

, where n is the number of messages and s is the size of the message. The max value is so small, that it can also be considered as constant. Figure C.7 shows the measured CPU utilization.

C.1.2 Mail client

This section evaluates the performance of the `MailClient` component in a similar fashion as the previous section. The experiments measure the response time and the average CPU utilization when the client sends and receives messages.

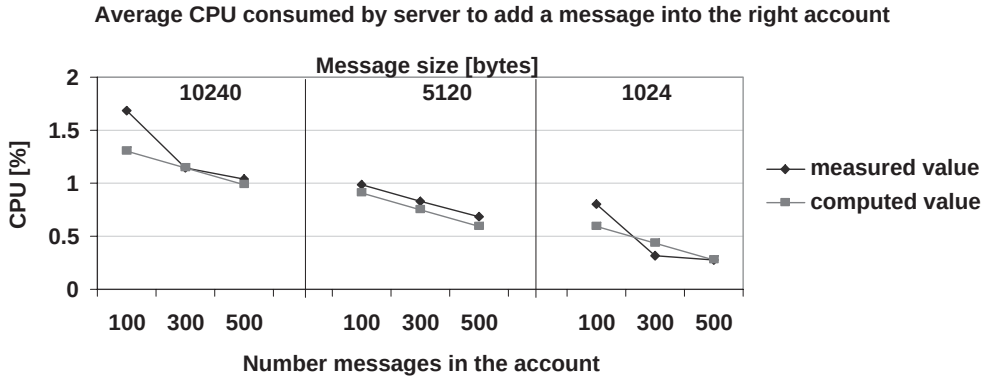


Figure C.7: Average CPU utilization when the server adds a message into account.

Send message operation

This experiment measures the resources consumed by the client when sending messages. Without the actual communication with the server, the experiments evaluate only the response time and the CPU utilization when the client creates the messages to be sent. The parameters which are varied are: the message size (512 bytes, 1k, 2k, 5k, and 10k), and the delay between requests (50ms, 100ms, 150ms).

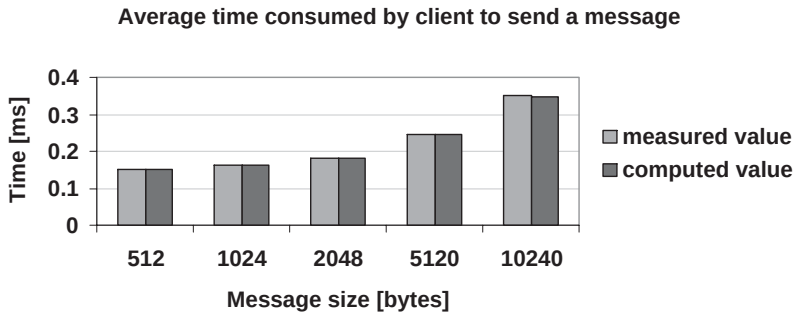


Figure C.8: Average response time when the client sends a message.

Response time. The response time when the client sends a message depends on the size of the message. Figure C.8 shows the measured response time, which can be fit by the following function: $response\_time_{client\_send}(s) = 0.139453 + 0.0000205s$. In this expression, s represents the size of the message.

CPU utilization. The CPU utilization depends on the size of the message. However, the measured data shown in Figure C.9 can be approximated by the upper bound. $CPU_{client\_send}() = 4$.

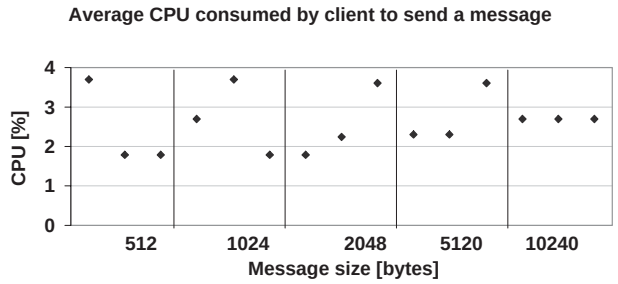


Figure C.9: Average CPU utilization when the client sends a message.

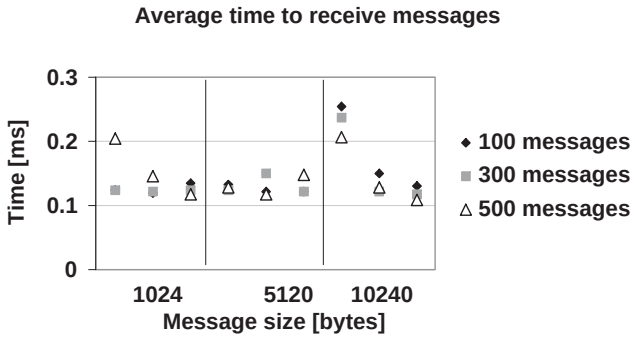


Figure C.10: Average response time when the client receives messages.

Receive messages operation

In order to measure the response time and the CPU utilization when the client receives messages from the mail server, the following experiment was conducted: the mail client executes the "receive messages" operation, while varying the size of the message (1k, 5k, and 10k), the delay between requests (0.5s, 1s, and 1.5s), and the number of messages received (100, 300, and 500).

Response time. The observed response time is sufficiently small to be considered constant: $response\_time_{client\_recv} = 0.2$. Figure C.10 shows that the measured average

response time when the client receives messages and merges them into the account is less than 0.2ms.

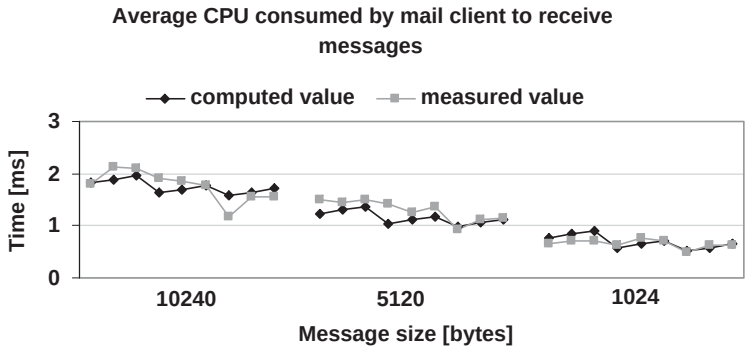


Figure C.11: Average CPU utilization when the client receives messages.

CPU utilization. Figure C.11 shows the measured CPU utilization when the client receives messages. The function that fits well the measured data is

$$cpu_{client\_recv}(s,r,n) = 0.23 + 0.000115s + 0.000355n + 0.1896r$$

, where s the size of the messages received from the mail server, r the rate of making requests to receive messages, and n the number of messages received from the mail server.

C.2 Switchboard

Switchboard represents the secure communication abstraction used by PSF to create secure channels between all entities. Because of this, the efficiency of Switchboard directly influences the efficiency of the entire deployment process. This section evaluates the Switchboard performance by measuring the round-trip and one-way latency,

and the CPU utilization when a client and a server communicate through Switchboard. The two experiments described below were run on a cluster of 30 Linux-based nodes divided into five domains; the Switchboard server is running in one domain and clients are located in each of the five domains. In both experiments, the rate of client requests are obtained by injecting various delays between client Switchboard calls.

C.2.1 Experiment 1.

The first experiment assumes the following network configuration. Intra-domain links have 100Mbps available bandwidth and 0ms latency. The links connecting the domains to the one where the server is located have the following characteristics:

- 50Mbps available bandwidth and 20ms latency;
- 70Mbps available bandwidth and 10ms latency;
- 20Mbps available bandwidth and 70ms latency;
- 56Kbps available bandwidth and 100ms latency.

Given such a network, one client connects to the server from each domain and makes Switchboard RPC calls with parameters of different sizes (1K, 5K, 10K) and delays of 0ms, 50ms, 100ms, and 150ms. This experiment measures (1) the average time for a message to reach the server (*one-way latency*), (2) the average time to make the call and receive the answer from the server (*two-way latency*), and (3) the average CPU utilization of a client when making a Switchboard call. The goal is to see how the latencies and the CPU utilization are influenced by the network characteristics.

Latency. The following two functions approximate the average time to make a call. $r = s/d$ is the rate of bits sent by the client, s is the message size, d is the delay between messages, l is the link latency, and b is the amount of link available bandwidth. The actual values for a_{ow}, b_{ow}, c_{ow}, a_{tw}, b_{tw}, and c_{tw} are shown in Tables C.1 and C.2. Figures C.12 and C.13 compare how accurate the approximations are when compared to measured times.

$$latency_{oneway}(s,r,l,b) = a_{ow} + b_{ow}s + c_{ow}lr/b$$

$$latency_{twoway}(s,r,l,b) = a_{tw} + b_{tw}s + c_{tw}lr/b$$

Table C.1: Parameters for one-way latency function.

Latency[ms]	Bandwidth[bits/ms]	a_{ow}	b_{ow}	c_{ow}
1	100000	3.639302	0.0000552	1.003602
20	50000	23.92174	-0.000375	1.126531
10	70000	14.33129	-0.000077	1.031183
70	20000	74.60529	-0.011782	3.397377
100	56	244.4245	-5.194152	2.9192

CPU utilization. The CPU utilization when a client makes Switchboard requests is affected mainly by the link available bandwidth and the delay between requests. The function that approximates well the measured CPU utilization is given by the following expression, where b is the amount of available bandwidth and d is the delay

Table C.2: Parameters for two-way latency function.

Latency[ms]	Bandwidth[bits/ms]	a_{tw}	b_{tw}	c_{tw}
1	100000	7.86287802	0.000054	1.003397911
20	50000	49.18318197	-0.00037079	1.116180956
10	70000	28.99475639	-0.0000749	1.028514878
70	20000	150.2371395	-0.011577819	3.337880997
100	56	476.9670374	-6.036063322	3.390659259

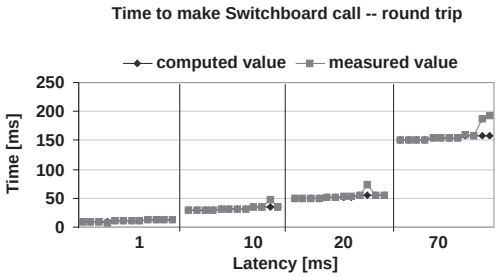


Figure C.12: Average round-trip latency when one client makes Switchboard calls.

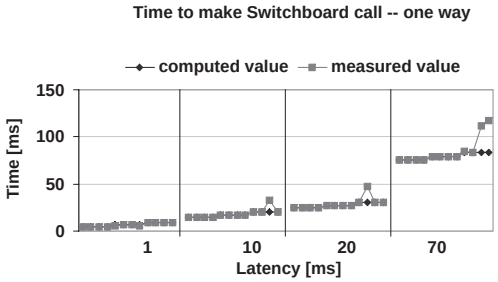


Figure C.13: Average one-way latency when one client makes Switchboard calls.

between requests. Figure C.14 illustrates the accuracy of the function.

$$CPU_{SW\_client}(b,d) = 8 - 33.84466116b + 0.00936185d - 0.091357796b/d$$

Experiment 2. In the second experiment, the network is defined as one domain containing all 30 nodes, where all the links have 100Mbps available bandwidth and 0ms latency. The number of clients connected to the server increases from 5 to 20 in increments of 5, the message sizes vary from 1k, 5k, to 10k, and the delays between

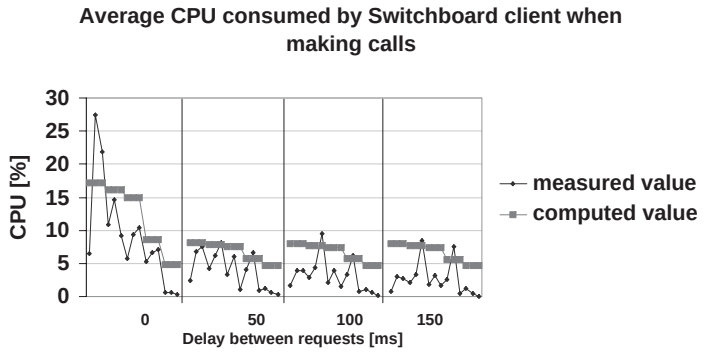


Figure C.14: Average CPU utilization when one client makes Switchboard calls.

calls are 50ms, 100ms, and 150ms. The average round-trip latency is measured on the client side, while the average CPU consumed is measured on the server side. The goal of this experiment is to determine how the server behavior is influenced by the rate if incoming Switchboard calls.

CPU utilization. The CPU utilization when the server accepts Switchboard requests from multiple clients, depends on the number of incoming requests and the size of the messages sent in those requests. $r = req * 1000/d$ represents the rate of incoming requests per second, where *req* is the number of clients making requests and d is the delay between the requests, measured in milliseconds. $CPU_{SW\_server}(r,s) = -0.442988 + 0.002377s + 0.395118r$, if $r \leq limit$. The table C.3 gives the limits, which depend on the message size. Figure C.15 illustrates the accuracy of the function by comparing the computed values with the measured ones.

Latency. The measured round-trip and one-way latencies, when multiple clients simultaneously make Switchboard requests, are shown in Figures C.16 and C.17. The

Table C.3: Limits of request rates

Message size [bytes]	Limit
1024	200
5120	133
10240	100

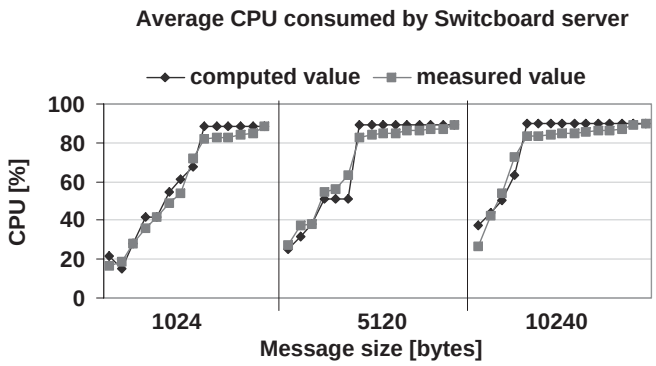


Figure C.15: Average CPU utilization when multiple clients are simultaneously making Switchboard calls.

functions that fit well the measured data are given by the following expressions, where n is the number of clients.

$$latency_{multiclients\_oneway} = -6.503749228 + 4.140272273n$$

$$latency_{multiclients\_twoway} = -8.501716537 + 5.516912531n$$

The two experiments described above assert the claim that Switchboard is an efficient communication abstraction which provides inexpensive security guarantees.

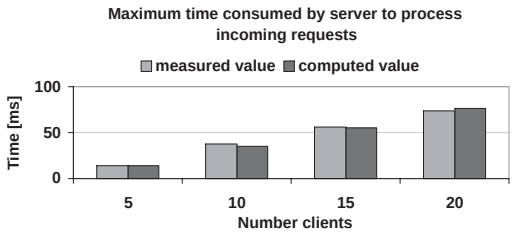


Figure C.16: Average one-way latency, when multiple clients are simultaneously making Switchboard calls.

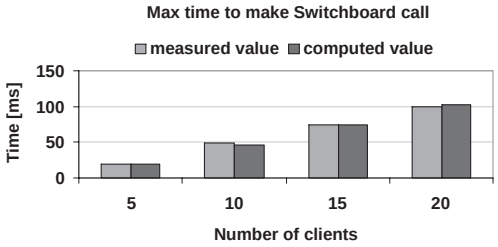


Figure C.17: Average round-trip latency when multiple clients are simultaneously making Switchboard calls.

C.3 Conclusions

This appendix has evaluated the resources consumed by two components of the e-mail application components when executing individual operations. The results presented here are used in Appendix D to accurately model the behavior of each component. This model is provided as input to the planning module described in Chapter 5.

The functions that fit well the measured resource consumption are (1) *constant*, (2) *polynomial*, and (3) *non-polynomial* functions. Hence, the planner's support for non-reversible functions is necessary.

Appendix D

Computing Properties for the Mail Application

Chapter 8 evaluates the expressivity of the application model introduced in Chapter 4, by illustrating how the model captures the behavior of the e-mail application described in Chapter 2. The expressions used in that example are only simplified versions. This Appendix explains how more complete expressions can be created to capture the application behavior in a more realistic way. This model takes into consideration the real load created by clients, given the behavior of each client.

D.1 Modeling the communication between clients

In order to capture the behavior of the e-mail application, the following assumptions are made: the environment is divided into domains, the `MailServer` is running in one domain, and clients are spread among all domains. Figure D.1, which was initially

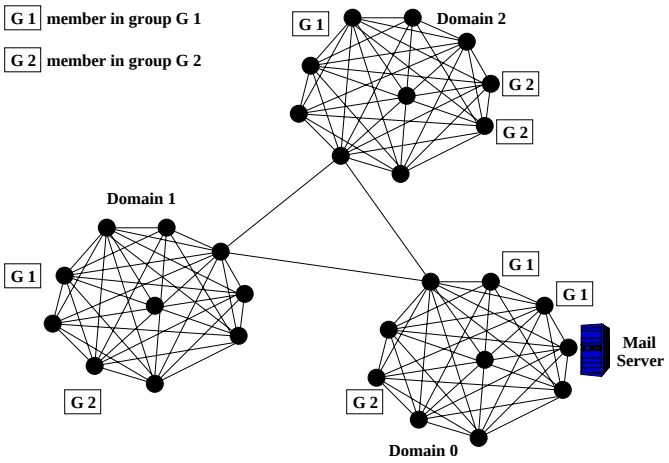


Figure D.1: Test bed

presented in Chapter 10, is used in this Appendix to illustrate the assumed setting. In this figure, the environment has three domains, and there are only two groups with four clients in each group. The clients are spread between the three domains as follows: (1) 2 clients from $Group_1$ and 1 client from $Group_2$ are in the same domain as the `MailServer`, (2) 2 clients from $Group_2$ and 1 client from $Group_1$ are in Domain 2, and (3) one client from each group is in Domain 1.

Client behavior. In order to better capture the application behavior, this section describes a generalization of the scenario presented above. Clients are divided into K groups, where each group g_i has C_i clients, $1 \leq i \leq K$. Clients in each group g_i are characterized by the following properties:

S_i The rate of sending messages in the unit of time;

R_i The rate of executing the "receive messages" operation in the unit of time;

D_i The percentage of new received messages deleted by the client;

M_i^j The percentage of messages sent by the client from group g_i to the clients from group g_j.

In addition, each client is assumed to execute the same operations in a loop: (1) sending messages, (2) executing one "receive messages" operation, and then (3) deleting part of the incoming messages. This is important because it permits reasoning about the flow of messages in the system.

If the network has D domains, each domain d_p serves a number of clients from each group, i.e., $\{n_1^p, n_2^p, ..., n_K^p\}$.

E-mail application. Creating a realistic model for the e-mail application requires two pieces of information: the client behavior and the application behavior. The main components discussed in this Appendix are `MailClient`, `ViewMailServer`, and `MailServer`.

Clients use the `MailClient` component to connect to a `ViewMailServer` or a `MailClient`, and access their e-mail accounts. The `ViewMailServer` is a cache mail server which can be placed close to clients to offset high latencies. In general, a `ViewMailServer` serves only a subset of the clients who have accounts on the `MailServer`. The `ViewMailServer` and the `MailServer` synchronize their information according to the following rules: (1) if a client connected to a `ViewMailServer` sends a message to a user who does not have an account on the `ViewMailServer`, the `ViewMailServer` forwards the message to the `MailServer`; (2) all new messages saved on the `ViewMailServer` are sent to the `MailServer` when the quality triggers discussed in Chapter 6 evaluate to `True`, and (3) for every message deleted on the `ViewMailServer`, the `ViewMailServer` sends an update notification to the

`MailServer`; all notifications are packed and sent as one batch when the quality triggers evaluate to `True`. All communication between components is done through Switchboard.

Figure D.2 illustrates the message flow in and out the `ViewMailServer`, if the `ViewMailServer` is located in domain d_p.

Modeling both the client and application behavior permits the creation of a more complete application specification for each component. For example, the `RRF` property of the `ViewMailServer` can be defined as the ratio between the traffic forwarded to the `MailServer` and the total traffic seen by the `ViewMailServer`.

For clarity, this section starts by defining intermediary properties and then uses them to build expressions for the component properties.

D.2 Intermediary properties

$S_{in}^{p}(i,j)$ represents the rate of messages sent by clients in group g_i in domain d_p to clients in group g_j in domain d_p.

$$S_{in}^{p}(i,j) = n_i^p S_i M_j^i n_j^p / C_j$$

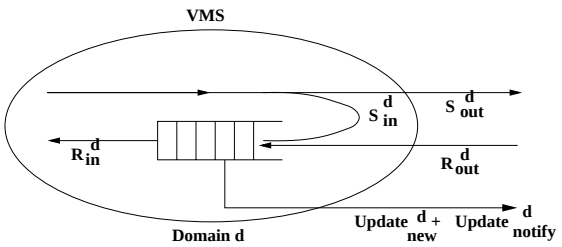


Figure D.2: Traffic going through the ViewMailServer associated with a domain d_p.

S_{in}^{p} is the rate of messages sent by clients from domain d_p to clients from domain d_p is:

$$S_{in}^{p} = \sum_{i=1}^{K}\sum_{j=1}^{K} S_{in}^{p}(i,j) = \sum_{i=1}^{K}\sum_{j=1}^{K} n_i^p S_i M_j^i n_j^p / C_j$$

$S_{out}^{p}(i,j)$ is the rate of message sent by clients from group g_i in domain d_p to clients in group g_j outside the domain d_p is:

$$S_{out}^{p}(i,j) = n_i^p S_i M_j^i \left(C_j - n_j^p\right) / C_j$$

S_{out}^{p} is the rate of messages sent by clients in domain d_p to clients outside the domain d_p is:

$$S_{out}^{p} = \sum_{i=1}^{K}\sum_{j=1}^{K} S_{out}^{p}(i,j) = \sum_{i=1}^{K}\sum_{j=1}^{K} n_i^p S_i M_j^i \left(C_j - n_j^p\right) / C_j$$

$R_{out}^{p}(j)$ is the rate of messages received by clients from group g_j in domain d_p, which were sent by clients from outside domain d_p is:

$$R_{out}^{p}(j) = \sum_{q=1,q\neq p}^{D}\sum_{i=1}^{K} S_{out}^{q}(i,j) n_j^p / C_j$$

R_{out}^{p} is the rate of messages received by clients from domain d_p, which were sent by clients from outside domain d_p is:

$$R_{out}^{p} = \sum_{j=1}^{K} Rout^{p}(j) = \sum_{q=1,q\neq p}^{D}\sum_{j=1}^{K}\sum_{i=1}^{K} Sout^{q}(i,j) n_j^p / C_j$$

$Update_{new}^{p}$ is the rate of messages received by clients in domain d_p that need to be forwarded outside the domain is:

$$Update_{new}^{p} = \sum_{j=1}^{K}\left[\left(\sum_{i=1}^{K} Sin^{p}(i,j)\right)(1 - D_j)\right]$$

$Update^{p}_{notify}$ is the rate of notifications (to indicate which messages received from outside the domain were deleted) forwarded from domain d_p outside is:

$$Update^{p}_{notify} = \sum_{j=1}^{K} [Rout^{p}(j)\,(1 - D_j)]$$

Given these measurements, one can create a model that completely captures the properties of the e-mail application components.

D.3 Computing e-mail application component properties

Chapter 8 listed the properties of the e-mail application and explained why the properties are relevant to the application. The only properties discussed in this Appendix are `RRF` for `ViewMailServer`, CPU utilization, `TFMS`, and `TLMS`.

RRF. By definition, the `RRF` factor for the `ViewMailServer` represents the fraction of the requests created by the clients located in one domain *d* that cannot be processed locally by the `ViewMailServer`, but need to be forwarded to the required interfaces. Given this definition, the `RRF` can be captured by the formula:

$$\texttt{RRF}^{d} = \left(Sout^{d} + Update^{d}_{new} + Update^{d}_{del}\right) / \left(Sin^{d} + Sout^{d} + Update^{d}_{new} + Update^{d}_{del}\right)$$

CPU utilization. For each component of the e-mail application, the total CPU utilization is defined as the sum of the CPU utilizations when the component executes various operations. For example, the `ViewMailServer` consumes CPU when deleting messages, processing messages sent by clients, sending messages to clients, and sending and receiving updates to and from the `MailServer`. The expression D.2 represents the total CPU utilization for the `ViewMailServer` component. In this expression, N^{d}

is the number of clients in domain d, R^d is the total receive rate, R_{pull} is the rate to pull messages from the `MailServer`, and R_{push} is the rate to such messages. The CPU utilization for the `MailServer` has the same expression.

The CPU utilization for the `MailClient` can be deduced in a similar fashion. Expression D.4 capture the CPU utilization for `MailClient`, where $RM(d', i)$ represents the total number of messages received by clients from domain d' and group i. The intuition is to add the CPU consumed by the `MailClient` when sending messages with rate S_i, receiving messages with rate R_i and deleting a fraction D_i of the received messages.

Response time. For a better understanding of how the response time is computed, let's consider the following scenario: the `MailClient` is connected to a `ViewMailServer`, which is connected to the `MailServer`. In this case, the time to reach the first server (`TFMS`) is equal to the sum of (1) the client response time when creating the message, (2) the round-trip latency to reach the `ViewMailServer` using Switchboard, and (3) the response time of the `ViewMailServer` when adding the message to the account. In order to compute the round-trip latency in step (2), the formula needs to sum over all the links crossed by the message. Formula D.5 shows exactly how the response time is computed.

Similarly, `TLMS` is the sum of the `TFMS` and the round-trip latency of the communication between `ViewMailServer` and `MailServer`, as shown in Formula D.6. The first sum represents the time spent crossing all the links between the `MailClient` and the `MailServer`; the second sum generalizes the first formula by considering that between the `MailClient` and the `MailServer` there might be any number of

`ViewMailServer` components.

$$
\begin{aligned}
CPU_{VMS} &= Sin^d\left(CPU_{server\_send} + CPU_{SW\_server}\right) + 2Sout^d CPU_{SW\_server} + \\
&\quad R * N^d\left(Rout^d + Rin^d\right)\left(CPU_{server\_delete} DF + CPU_{SW\_server} DF\right) + \\
&\quad R * N^d\left(CPU_{server\_recv} + CPU_{SW\_server}\right) + \\
&\quad R_{pull}\left(CPU_{pull} + CPU_{SW\_server}\right) + \\
&\quad R_{push}\left(CPU_{push} + CPU_{SW\_server}\right) \qquad \text{(D.1)} \\
R^d &= \sum_{i=1}^{K} R_i n_i^d \qquad \text{(D.2)} \\
CPU_{MC} &= S_i\left(CPU_{client\_send} + CPU_{SW\_client}\right) + \\
&\quad R_i\left(CPU_{client\_recv} + CPU_{SW\_client}\right) + \\
&\quad R_i RM(d', i) D_i\left(CPU_{client\_delete} + CPU_{SW\_client}\right) \qquad \text{(D.3)} \\
RM(d', i) &= \sum_{d=1}^{D}\sum_{i=1}^{K} n_i^d S_i M_i^{d'} / C_i \qquad \text{(D.4)} \\
\texttt{TFMS}_{send} &= response\_time_{client\_send} + \\
&\quad \sum\left(latency_{multiclients\_twoway} + latency_{client\_twoway}\right) + \\
&\quad response\_time_{server\_send} \qquad \text{(D.5)} \\
\texttt{TLMS}_{send} &= response\_time_{client\_send} + \\
&\quad \sum\left(latency_{multiclients\_twoway} + latency_{client\_twoway}\right) + \\
&\quad \sum\left(response\_time_{server\_send}\right) \qquad \text{(D.6)}
\end{aligned}
$$

Appendix E

ViewMailServer Code

Chapter 8 evaluates the usability of the PSF framework by describing the lines of code to be added to a component in oder to become PSF-aware. For completeness, this Appendix lists the complete code for the `ViewMailServer` component. The lines of interest are:

- **lines 1-2**: The `ViewMailServer` becomes PSF-aware by extending the PSFObject class; in addition, the `ViewMailServer` can use the consistency protocol provided by PSF if it extends the `ViewInterface` interface;
- **lines 8-16**: These lines contain the declarations for the cache manager, the list of properties, and each of the properties;
- **lines 21-24**: The cache manager is created; as part of the creation, the `ViewMailServer` provides the cache manager with information about the properties, and the quality triggers;
- **line 25**: The cache manager is started in the WEAK mood;

- **line 26**: The cache manager is asked for the most up-to-date data;
- **lines 28-50**: The functionality of the `ViewMailServer` is modified to interact with the cache manager; each method calls `startUseImage` and `endUseImage` around the code processing shared data;
- **lines 51-56**: This method is inherited from the `PSFObject` and allows the wrappers to create the connection with other components;
- **lines 57-85**: The `ViewMailServer` initializes the list of properties;
- **lines 86-164**: The `ViewMailServer` implements the extract and merge methods.

```
1   public class ViewMailServer extends PSFObject
2     implements MailServerInterface, ViewInterface, Runnable {
3     Server_MultiThreaded server = null;
4     Integer minLevel, maxLevel;
5     Integer securityClearance;
6     Integer strength;
7     String accounts_file;
8     CacheManagerImpl_Disco cm = null;
9     ViewPropertyList propertyList;
10    String property_User = new String ("user");
11    String property_Folder = new String ("folder");
12    String property_MinLevel = new String ("minLevel");
13    String property_MaxLevel = new String ("maxLevel");
14    String property_Strength = new String ("strength");
15    String property_SecurityClearance = new String ("secClearance");
16    String property_StartingDate = new String ("startingDate");
```

```
17    public void run() {
18      parseArguments( arguments );
19      server = new Server_MultiThreaded( arguments );
20      new Thread( server ).start();
21      cm = new CacheManagerImpl_Disco( cmArgs,
22          "ViewMailServerDiscoMultiThreaded,
23          this,createPropertyList(),CacheManagerImpl.CM_WEAK,
24          ``t<10000'', ``t<10000'', ``t<0)'', null, 1 );
25      cm.switchToWeak();
26      cm.initImage();
27    }
28    public void sendMessage( Message message ) {
29      cm.startUseImage( "send message" );
30      server.sendMessage( message );
31      cm.endUseImage( "send message" );
32    }
33    public void deleteMessage( String _user, String _folder,
                                 int _index ) {
34      server.deleteMessage( _user, _folder, _index );
35    }
36    public Account getAccountMetadata( String name,
                                         Integer secClearance ) {
37      cm.startUseImage( "get account metadata" );
38      Account acc = server.getAccountMetadata( name, secClearance );
39      cm.endUseImage( "get account metadata" );
40      return acc;
41    }
```

```
42    public synchronized Account getMessages(
43      Account acc, Folder folder, Integer min, Integer max,
44      Integer secClearance, Integer strength, Date startingTime) {
45      cm.startUseImage( "get messages" );
46      Account ac = server.getMessages( acc, folder, min, max,
47                                       secClearance, strength,
                                         startingTime );
48      cm.endUseImage( "get messages" );
49      return ac;
50    }
51    public void setInterface( String _interf, Object _obj ) {
52      if( _interf.equals( "mailSW.server.MailServernterface" ) )
53        remoteServer = (MailServerInterface)_obj;
54      if( _interf.equals( "flecc.dm.DirectoryManagerInterface" ) )
55        cm.connectToDM( (DirectoryManagerInterface) _obj );
56    }
57    ViewPropertyList createPropertyList() {
58      ViewProperty v1 = new ViewProperty( property_User,
59                                          ViewProperty.SET,users );
60      Vector value = new Vector();
61      value.add( new String( "null" ) );
62      ViewProperty v2 = new ViewProperty( property_Folder,
63                                          ViewProperty.SET,value );
64      value = new Vector();  value.add( securityClearance );
65      ViewProperty v3 = new ViewProperty( property_SecurityClearance,
66                                          ViewProperty.SET,value );
67      value = new Vector();  value.add( minLevel );
68      ViewProperty v4 = new ViewProperty( property_MinLevel,
69                                          ViewProperty.SET,value );
70      value = new Vector();  value.add( maxLevel );
```

```
71      ViewProperty v5 = new ViewProperty( property_MaxLevel,
72                                          ViewProperty.SET,value );
73      value = new Vector();  value.add( strength );
74      ViewProperty v6 = new ViewProperty( property_Strength,
75                                          ViewProperty.SET,value );
76      value = new Vector();  value.add( new String( "null" ) );
77      ViewProperty v7 = new ViewProperty( property_StartingDate,
78                                          ViewProperty.SET,value );
79      ViewPropertyList vpl = new ViewPropertyList();
80      vpl.addProperty( v1 );  vpl.addProperty( v2 );
81      vpl.addProperty( v3 );  vpl.addProperty( v4 );
82      vpl.addProperty( v5 );  vpl.addProperty( v6 );
83      vpl.addProperty( v7 );
84      return vpl;
85    }
86    ObjectImage extractFromObject(Object object,
                                    ViewPropertyList propList){
87      Date lastDate = null; Folder folder = null;
88      MailServerInterface parentMailServer =
          (MailServerInterface) object;
89      Vector value =
          (Vector) propList.getValue( property_SecurityClearance);
90      Integer securityClearance = (Integer) value.get( 0 ) ;
91      Integer minLevel =
          ((Vector)propList.getValue(property_MinLevel)).get(0);
92      Integer maxLevel =
          ((Vector)propList.getValue(property_MaxLevel)).get(0);
93      Integer strength =
          (Vector) propList.getValue (property_Strength).get(0);
94      String fold = (Vector) propList.getValue (property_Folder).get(0);
```

```
95      if (! fold.equals ("null")) folder = new Folder (fold);
96      String d =
          (Vector) propList.getValue (property_StartingDate).get(0);
97      Date startingDate = null;
98      if (! d.equals ("null")) startingDate = new Date (d);
99      int numberUsers =
          propList.getProperty (property_User).getValues().size();
100     Vector accounts = new Vector (20, 10);
101     for (int i=0; i<numberUsers; i++) {
102       Account acc = parentMailServer.getAccountMetadata(
103                     ((User) values.get (i)).getName (),
                      securityClearance );
104       Account acc1 = parentMailServer.getMessages (acc, folder,
                       minLevel, maxLevel,
105                    securityClearance, strength, startingDate);
106       Date dd = acc1.getLastDate();
107       if( ( lastDate == null ) || ( dd.after( lastDate ) ) )
            lastDate = dd;
108       accounts.add (acc1);
109     }
110     Vector v = new Vector();
111     if( lastDate == null ) v.add( "null" );
112     else v.add( lastDate.toString() );
113     propList.setValues( property_StartingDate, v );
114     ObjectImage image = new ObjectImage (accounts);
115     return image;
116   }
```

```
117   ObjectImage extractFromView( ViewPropertyList _vpl ) {
118     ObjectImage image = null; Date lastDate = null;
        Folder folder = null;
119     Vector value =
          (Vector) _vpl.getValue( property_SecurityClearance );
120     Integer secClearance = (Integer) value.get( 0 ) ;
121     Integer minLevel =
          ((Vector)propList.getValue(property_MinLevel)).get(0);
122     Integer maxLevel =
          ((Vector)propList.getValue(property_MaxLevel)).get(0);
123     Integer strength =
          (Vector) propList.getValue (property_Strength).get(0);
124     String fold =
          (Vector) propList.getValue (property_Folder).get(0);
125     if (! fold.equals ("null")) folder = new Folder (fold);
126     String d =
          ((Vector) _vpl.getValue (property_StartingDate)).get(0);
127     Date startingDate = null;
128     if (! d.equals ("null")) startingDate = new Date (d);
129     Vector values =
          _vpl.getProperty (property_User).getValues ();
130     int numberUsers = values.size ();
131     Vector accounts = new Vector (20, 10);
132     for (int i=0; i<numberUsers; i++) {
133       User user = (User) values.get(i);
134       Account acc = server.getAccountMetadata( user.getName(),
                                                   secClearance );
135       Account acc1 = server.getMessages (acc, folder, minLevel,
136            maxLevel, securityClearance, strength, startingDate);
137       accounts.add (acc1);
```

```
138       Date dd = acc1.getLastDate();
140       if( ( lastDate == null ) || ( dd.after( lastDate ) ) )
            lastDate = dd;
141     }
142     Vector v = new Vector();
143     if( lastDate == null ) v.add( "null" );
144     else v.add( lastDate.toString() );
145     _vpl.setValues( property_StartingDate, v );
146     image = new ObjectImage (accounts);
147     return image;
148   }
149   void mergeIntoObject( Object object, ObjectImage _image,
150                         ViewPropertyList propertyList ) {
151     MailServerInterface parentMailServer =
          (MailServerInterface) object;
152     Vector accounts = _image.getImage ();
153     for( int i = 0; i < accounts.size(); i ++ ) {
154       Account acc = (Account) accounts.get( i );
155       parentMailServer.mergeAccount (acc);
156     }
157   }
158   void mergeIntoView( ObjectImage _image,
                        ViewPropertyList _vpl ) {
159     Vector accounts = _image.getImage ();
160     for( int i = 0; i < accounts.size(); i ++ ) {
161       Account acc = (Account) accounts.get( i );
162       server.mergeAccount (acc);
163     }
164   }
```

Bibliography

[1] William Adamson and O. Kornievskaia. A Practical Distributed Authorization System for GARA. Technical Report 01-14, Center for Information Technology Integration, University of Michigan, 2001.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *18th ACM SOSP*, 2001.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[4] Phil Bernstein and Nathan Goodman. The Failure and Replicated Distributed Databases. In *ACM Transactions on Database Systems*, 1984.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W.Weiss. RFC 2475 - An Architecture for Differentiated Services. http://www.faqs.org/rfcs/rfc2475.html, December 198.

[6] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust

Management for Public-Key Infrastructures. In *Security Protocols International Workshop*, volume 1550, pages 59–63. Springer LNCS, 1998.

[7] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. In *Financial Cryptography*, pages 254–274, 1998.

[8] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[9] R. Braden, D. Clark, and S. Shenker. RFC 1633 - Integrated Services in the Internet Architecture: an Overview. http://www.faqs.org/rfcs/rfc1633.html, June 1994.

[10] F. Bustamante and K. Schwan. Active Streams: An Approach to Adaptive Distributed Systems. In *HotOS*, 2001.

[11] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[12] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[13] P. Chandra. Darwin: Customizable Resource Management for Value-Added Network Services. In *ICNP*, 1998.

[14] Sirish Chandrasekaran, Samuel Madden, and Mihut Ionescu. Ninja paths: An architecture for composing services over wide area networks.

[15] F. Chang and V. Karamcheti. A Framework for Automatic Adaptation of Tunable Distributed Applications. *Cluster Computing*, 4:49–62, 2001.

[16] DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. InterWeave: A Middleware System for Distributed Shared State. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, 2000.

[17] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Proceedings of Supercomputing 2002 (SC2002)*, 2002.

[18] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, 2001.

[19] D. B. Terry et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[20] D.C.Feldmeier, A.J. McAuley, J.M. Smith, D. Bakin, W.S. Marcus, and T. Raleigh. Protocol Boosters. *IEEE JSAC, Special Issue on Protocol Architectures for 21st Century*, 16(3):437–444, 1998.

[21] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta,

Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus : Mapping Scientific Workflows onto the Grid. In *Proceeding of the Across Grids Conference*, 2004.

[22] Tim Dierks and Eric Rescorla. The TLS Protocol, Version 1.1. In *Internet Draft*, 2004.

[23] E. Freudenthal et al. dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments. In *ICDCS*, 2001.

[24] E. Freudenthal et al. Switchboard: Secure, Monitored Connections for Client-Server Communication. In *RESH*, 2002.

[25] E. Kohler et al. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[26] Susan Eggers and Randy Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *15th Annual International Symposium on Computer Architecture*, pages 373–383, 1988.

[27] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Spki certificate theory, rfc 2693. In *Network Working Group, The Internet Society*, 1999.

[28] J. Sydir et al. QoS Middleware for the Next-Generation Internet. *NASA/NREN Quality of Service Workshop*, 1998.

[29] Mallikarjun Shankar et al. An End-To-End QoS Management Architecture. *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, 1999.

[30] Peter Dinda et al. The Architecture of the Remos System. *HPDC*, 2001.

[31] Adam Ferrari, Frederik Knabe, Marty Humphrey, Steve Chapin, and Andrew Grimshaw. A Flexible Security System for Metacomputing Environments. In *HPCN*, pages 370–380, 1999.

[32] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[33] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *http://www.globus.org/research/papers.html*, 2002.

[34] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *IWQOS*, 2000.

[35] Ian Foster. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[36] Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *ACM CCS*, pages 83–92, 1998.

[37] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol, Version 3.0. In *Internet Draft*, 1996.

[38] Eric Freudenthal and Vijay Karamcheti. DisCo: Middleware for Securely Deploying Decomposable Services in Partly Trusted Environments. In *ICDCS*, pages 494–503, 2004.

[39] Xiaodong Fu and Vijay Karamcheti. Planning for Network-Aware Paths. In *DAIS*, pages 187–199, 2003.

[40] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *USITS*, 2001.

[41] Li Gong. Java security: present and near future. *IEEE Micro*, 17(3):14–19, 1997.

[42] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *OSDI*, 2000.

[43] A. Grimshaw, Adam Ferrari, Frederik Knabe, and Mart Humphrey. Wide-Area Computing: Resource Sharing on a Large Scale. *Computer*, 32(5):29–37, 1999.

[44] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2(3):195–205, 1999.

[45] I. Foster et al. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Intl. Workshop on Quality of Service*, 1999.

[46] IBM Corporation and Microsoft Corporation. Security in a Web Services World: A Proposed Architecture and Roadmap. *htpp://msdn.microsoft.com/*, 2002.

[47] Information Sciences Institute, University of Southern California. RFC 793 - Transmission Control Protocol. `http://www.faqs.org/rfcs/rfc793.html`, September 1981.

[48] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogenous Environments. In *The 11th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[49] A. Ivan and V. Karamcheti. Using Views for Customizing Reusable Components in Component-Based Frameworks. In *The 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[50] A. Ivan and V. Karamcheti. Flecc: A Flexible Cache Coherence Protocol for Dynamic Component-Based Systems. In *Submitted to the International Parallel and Distributed Processing Symposium*, 2004.

[51] Anca Ivan and Yevgeniy Dodis. Proxy Cryptography Revisited. In *NDSS*, 2003.

[52] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 156–171. ACM Press, 1995.

[53] Thomas W. Page Jr., Richard G Guy., Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus Scalable Replicated File System. Technical Report UCLA-CSD 910005, Los Angeles, CA (USA), 1991.

[54] P. Kaijser, J. Parker, and D. Pinkas. SESAME: The Solution to Security for Open Distributed Systems. In *Computer Communications*, 1994.

[55] V. Karamcheti and A. A. Chien. View caching: Efficient software shared mem-

ory for dynamic computations. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, pages 483–489, 1997.

[56] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[57] T. Kichkaylo and A. Ivan. Network EDitor. `http://www.cs.nyu.edu/pdsg/projects/partitionable-services/ned/ned.htm`, 2002.

[58] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. In *The International Parallel and Distributed Processing Symposium*, 2003.

[59] J. Koehler. Planning under resource constraints. In *ECAI*, 1998.

[60] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter*, pages 95–106, 1995.

[61] L. Pearlman at el. A Community Authorization Service for Group Collaboration. In *IEEE Workshop on Policies for Distributed Systems and Networks*, 2002.

[62] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java platform. In *15th Annual Computer Security Applications Conference*, pages 285–290. IEEE Computer Society Press, 1999.

[63] Ulana Legedza, David J. Wetherall, and John Guttag. Improving the Perfor-

mance of Distributed Applications Using Active Networks. In *IEEE INFOCOM*, 1998.

[64] Jun Li, Mark Yarvis, and Peter Reiher. Securing Distributed Adaptation. In *OpenArch*, 2001.

[65] Ilya Lipkind, Igor Pechtchanski, and Vijay Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *OOPSLA*, pages 447 – 460, 1999.

[66] Julio Lopez and David O'Hallaron. Support for Interactive Heavyweight Services. In *HPDC*, 2001.

[67] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184–201, 1986.

[68] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. In *5th Symposium on Operating Systems Design and Implementation*, 2002.

[69] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[70] B.D. Noble and M. Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. *Mobile Networks and Applications*, 4, 1999.

[71] Object Management Group. CORBA Security Services, Ver. 1.8. *http://www.omg.org/*, 2002.

[72] Object Management Group. CORBA Component Model. *http://www.omg.org/*, 2003.

[73] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated planning for open architectures. *OPENARCH*, 2000.

[74] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[75] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.

[76] S. Czerwinski et al. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[77] S. Gribble at el. The ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[78] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM Protocols for SMP Clusters: Design and Performance. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, 1998.

[79] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[80] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), 1996.

[81] Sharman Networks. Kazaa. `http://www.kazaa.com`.

[82] Pradeep Sudame and B. R. Badrinath. Transformer Tunnels: A Framework for Providing Route-Specific Adaptations. In *USENIX Annual Technical Conference*, pages 191–200, 1998.

[83] Sun Microsystems. Java 2 Enterprise Edition. http://java.sun.com/j2ee.

[84] Sun Microsystems. Java Remote Method Invocation. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmitoc.html.

[85] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A Bytecode Translator for Distributed Execution of Legacy Java Software. In *ECCOP*, pages 236–255, 2001.

[86] T.Kichkaylo and V.Karamcheti. Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications. In *HPDC*, 2004.

[87] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, and David E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. In *Middleware'98,the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.

[88] W. Rubin et al. *Understanding DCOM*. Prentice Hall, 1999.

[89] W3C. Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl*, 2003.

[90] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *9th Symposium on Operating System Principles*, pages 49–70, 1983.

[91] Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman, and Steven Tuecke. Security for Grid Services. In *HPDC*, 2003.

[92] Daniel S Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.

[93] W. Wulf, C. Wang, and D. Kienzle. A New Model of Security for Distributed Systems. Technical Report CS-95-34, CS Department, University of Virginia, 1995.

[94] Mark Yarvis, Peter L. Reiher, and Gerald J. Popek. Conductor: A framework for distributed adaptation. In *HotOS-6*, 1999.

[95] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[96] Haifeng Yu and Amin Vahdat. Combining Generality and Practicality in a Conit-Based Continuous Consistency Model for Wide-Area Replication. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[97] Marcia Zangrilli and Bruce B. Lowekamp. Comparing passive network monitoring of grid application traffic with active probes. In *Proceedings of the 4th International Workshop on Grid Computing (GRID2003)*, pages 84–91, Phoenix, AZ, November 2003. IEEE.

[98] Lixia Zhang, Stephen Deering, and Deborah Estrin. RSVP: A new resource ReSerVation protocol. *IEEE network*, 7(5):8–?, September 1993.

[99] Dong Zhou and Karsten Schwan. Eager Handlers - Communication Optimization in Java-based Distributed Applications with Reconfigurable Fine-grained Code Migration. *3rd Intl. Workshop on Java for Parallel and Distributed Computing*, 2001.