PROVIDING FLEXIBILITY IN DISTRIBUTED APPLICATIONS USING A MOBILE COMPONENT FRAMEWORK

by

Radu Cristian Litiu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2001

Doctoral Committee:

Associate Professor Atul Prakash, Chair
Associate Professor Farnam Jahanian
Assistant Professor Brian Noble
Professor Daniel Teichroew
Guruduth Banavar, Research Scientist, IBM T.J. Watson

To Mom and Dad
My success is their success.

ACKNOWLEDGMENTS

Over time, many people brought their contribution to me becoming who I am right now professionally. To all of them, I bring my humble thanks. It would be hard for me to name them all here. I have to mention at least a few of them though. Long before I knew computers do exist, Ms. Livescu Sabina, my elementary school teacher in Rm. Vilcea, taught me to read and write, and basic arithmetic operations. She taught me that learning can be fun, and knowing more can be gratifying. During secondary and high school, Ms. Gibescu Victoria, my math teacher, helped me master mathematics, develop a logical thinking, and enjoy wrestling with logical problems end eventually solve them. She tried hard, but with little success, to teach me discipline.

I would like to thank Professor Nicolae Tapus and Professor Marian Dobre, my mentors during my undergraduate years at the Politehnica University Bucharest. They taught me a few little things about computers, bits, bytes, and kernel processes, and stimulated me to learn more.

I want to thank Professor Dionysios Kountanis from Western Michigan University for his support on and off the academic field. He helped me figure out what grad school is about and guided my first shy attempts to do research. He pushed me to be ambitious and to want more, and to make efforts to obtain more. At a time when I had few friends, he was more than my professor; he was my friend.

Among the people who helped me complete this dissertation work, I would like to thank especially my advisor, Professor Atul Prakash. I learned many things from him. He helped me become a better researcher and improve my paper writing skills. The idea for my dissertation topic came out of many discussions with him, during which we were bouncing ideas back and forth, trying to find problems and their solutions. He contributed a lot to polishing this idea and advancing the work. I always secretly admired him for both the

breadth and depth of his technical knowledge, and wished to become at least as good as him at least in one or two topics. The match between our personalities has been far from great and communication between us sometimes suffered, but the important facts remain: without him, I wouldn't have made it to the point of completing this dissertation. I will always owe him part of my success.

I also want to thank Professor Farnam Jahanian, Professor Brian Noble, Professor Daniel Teichroew, and Dr. Guru Banavar from IBM T.J. Watson for serving on my dissertation committee, and for their advice. I would like to thank Guru for his help in crystallizing my ideas and refining my work.

My graduate work was partially supported by the National Science Foundation. I thank them for their generous support.

I would like to thank Yizhou Cao, Vilasini Niranjan, Jimmy Chu, and Bobby Green for their work on the DACIA project.

I want to thank Amgad Zeitoun for bringing an informed friend's opinion and criticism to my research work, and for being a funny and sunny presence in the other corner of my office.

There are numerous people outside of my professional circle who have made my time in Ann Arbor exciting, enjoyable, and entertaining. I would like to thank all my friends for giving a hand in spending my time off-duty in a pleasant way.

I want to give special thanks to a special person: my girlfriend Alina. Over the past few years, she helped me find balance in my life, and filled the other half of my life, the non-professional one, in a meaningful and fulfilling way.

If I am who I am right now, I owe it in the first place to my parents, Anisia and Ioan Litiu. They are at the end of this list, but they are the first ones. They raised and educated me, and cried when I went far away from them. Together with my sister Dana, they always supported and encouraged me, and never stopped believing in me. Thank you!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Need for Adaptability and Flexibility in Distributed Applications

Managing distributed applications is becoming more challenging due to the increased reliance of both individuals and corporations on computer systems, and due to the various requirements placed on these systems. Many applications are geographically distributed, and must often reason in the absence of complete or consistent information. They may be subject to real-time constraints or resource limitations, or they may be safety critical in nature. *Continuous operation* and *adaptability* are two important requirements for tomorrow's systems. In many situations system administrators need to be able to perform maintenance, troubleshooting, and upgrading of applications without causing interruptions in the service provided. Applications need to adapt to changes in their operating environment and to bursts in user activity.

Current computing environments are characterized by continuous changes. In a distributed environment, a long running application is likely, at some point during its execution, to encounter changes to the environment within which it is executing. These environmental changes could include machine and network related failures, services being introduced, withdrawn, moved, or replicated, or the application's functional requirements being changed. For an isolated system it may not be a major problem to shut down the application, install a new software package or upgrade the existing ones, and then restart the application. There are many situations (e.g., systems used in telecommunications, banking,

and air traffic control) though in which it is not acceptable to shut down a complete system even for a short period of time, in order to fix faulty behavior, upgrade the existing system, or introduce new functionality or new hardware. There is a need to *introduce new software components or replace the existing ones while the system is running.*

Dynamic reconfiguration mechanisms that will allow applications to change their internal structure to ensure forward progress are therefore required.

The explosive growth of the World Wide Web and the proliferation of Internet applications and services pose significant challenges to application developers and system administrators, namely dealing with the heterogeneity encountered within the end systems and across the network infrastructure, as well as handling the variety of user and application demands. Heterogeneity and variability are even more stringent problems in mobile computing environments, which are becoming increasingly ubiquitous. Variability occurs along several dimensions:

- **User and Application Requirements**

  Different applications and users place different demands on the computing resources and communication infrastructure. These demands change over time, as systems need to scale up and to accommodate new applications. For best performance and functionality, different system architectures may be required as we go, for instance, from two-party to multi-party communication. The architecture may need to evolve from peer-to-peer to client-server, and from centralized to distributed. New business requirements, new technical developments and standards require the constant change and adaptation of applications.

  There may be varying material costs associated with the access to computing resources and the quality of connectivity, bandwidth, and reliability of network access. In some cases, the users are willing to make tradeoffs between the quality of the service they receive and the amount they are paying for the service. A service provider needs to be able to adapt its applications to individual user's needs.

- **Hardware and Network Variability**

  The computing devices used range from high-end machines, with significant computing power, memory, and graphic display capabilities, to simple devices such as personal

digital assistants (PDAs) or cell phones, that have low processing power and can only display text or primitive graphics. The same application should be able to run on various computing devices and to adapt to the capabilities of the device. Regardless of the type of device on which it executes, the application should provide the same services to the user, and a uniform interface for communicating with other applications. *Context-aware applications* [37] should be able to adapt to the characteristics of the environment where they execute, such as location and the input/output resources available, and to changes in the environment.

Network links characteristics in terms of delay, capacity, and error rate can vary significantly. Often the performance of an application can be improved by selecting the location where certain functions are executed based not only on the availability of computing resources, but also on the network topology and the characteristics of the interaction between various parts of the application.

The ideal architecture of the system depends on the available computing and network resources. The presence of resources and the demand for these resources may not be known *a priori*, when an application is designed.

- **User Mobility and Intermittent Connectivity**

  Users are increasingly mobile. They connect from various points, using a variety of devices. They run applications that are part of a distributed computing environment, being connected to services, data sources, collaborative partners, etc. There are numerous situations in which a user would not want to shutdown all the applications he is running, cut all the connections with other parties, and quit all the login sessions he has established, in order to move to a different place short time later, restart the very same applications, and manually re-establish the same connections. It would be desirable to provide support to users so that they are able to move applications from one computing device to another while maintaining seamless communication connectivity with other applications (Figure 1.1).

For example, consider a stock trading application that a user accesses through a web browser running on a desktop computer. The user logs on a server providing real-time stock quotes, then sets some preferences related to the stocks to monitor and the information to

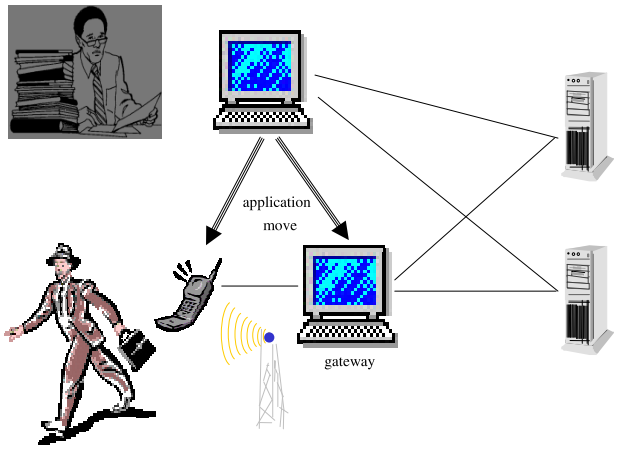

Figure 1.1: Applications need to support mobility of users, heterogeneity, and variations in available resources. They should be able to move from one computing device to another.

be displayed by the browser. He also sets some alarms to be triggered when a certain stock price reaches a desired value. The web client can be connected simultaneously to multiple data sources, providing stock quotes, news information, personal account information, etc. Assume that the user leaves his office and then, while on the move, wants to access the same information on a wireless device, such as a PDA or a cell phone, without having to login, set the preferences, and connect to the data sources again. It would be much easier if he could just move the application still running on the computer to the wireless device. The presentation of the data will be appropriate to the screen of the new device, and the price alarms will change, for example, from some flashing icons, into audio signals.

Operating in a mobile environment raises the problem of dealing with the inherent unreliability of mobile network connections and variations in connection quality. For many applications, it would be desirable that the underlying middleware layers mask transient network and communication failures. Using a combination of techniques such as data prefetching, message buffering and retransmission, applications and implicitly their users can be given the illusion of a persistent end-to-end logical connection, even over an unreliable network connection. At the same time, the communication middleware should be able to deal with persistent disconnections and to inform the applications accordingly.

For many applications and services, besides transferring data between endpoints using various transport protocols, other operations can be applied along the data path, such as

compression, encryption, authentication, buffering, aggregation, etc. The decision to apply these operations is currently made statically, but potentially it can be made on the fly, based on runtime factors and the execution environment. Some of these operations have to be applied in a particular order and at a particular location, but in some circumstances, the order of applying certain operations may be changed without affecting the end result. For example, if the source machine where the data originated has enough processing power, then compression can be done immediately at the source. If it does not, the data can be sent uncompressed, provided that the source is connected through a high-bandwidth link, and compression can be applied at an intermediate node. If compression is not possible, and the link cannot support the high rate of sending uncompressed data, the quality of the data transmitted will need to be degraded.

It is difficult to design a one-size-fits-all architecture that works well under all potential usage situations. Systems often end up making significant assumptions about the environment and must be redesigned for effective use if the assumptions no longer hold. For best performance and functionality, different system architectures need to evolve as we go, for instance, from two-party to multi-party communication, from peer-to-peer to client-server, and from centralized to distributed. We believe that there is a need to develop techniques for designing *flexible distributed applications and services that adapt better to variations in resource availability and application requirements, and to user mobility.*

There are several examples of distributed applications and systems that can be built through the composition of multiple units of computation, that implement various functions of the system [11, 38, 47, 74, 87, 101]. These systems allow the construction of customized modular configurations from a set of components chosen according to the applications needs. The configuration of a system has to be done statically though. After appropriately setting the desired configuration, the system has to be compiled and the modules linked. The application will run as it is, without the possibility to further modifying it at runtime.

In most of today's modular distributed applications, the operation of building an application through the combination and configuration of existing software components is carried out only during the initial phases of the application life-cycle, namely the design and construction time. Future distributed systems will have to extend this capability over the whole software life-cycle. Components in an application and the structure of the application should be able to change not only during the design and build phases, but also while

the application is executing (Figure 1.2). While the application is running, existing components should be upgraded and new components should be introduced, the bindings between components should be able to change, and the application should be reconfigured without impacting (or, at least, with minimal disruption of) the execution of the application.

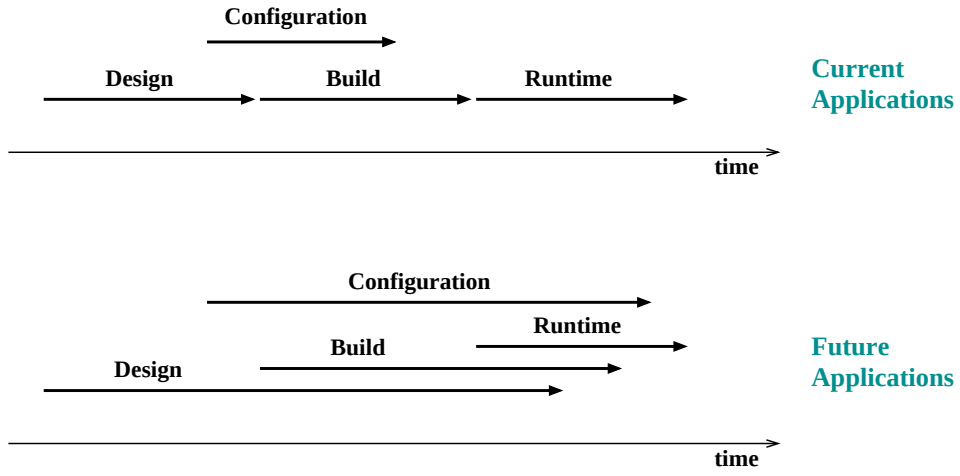

Figure 1.2: Software configuration in current distributed applications takes place only during the application design and build time. In order to be more flexible and more adaptable, future applications need to extend their reconfiguration capabilities during the runtime phase of the software life-cycle.

1.2 Focus of the Dissertation

This dissertation addresses some of the challenges of *building adaptive distributed applications that can change their structure at runtime.* We define *dynamic reconfiguration* as the process of modifying the structure of an application while the application is running. An application with a modular architecture, in which various components implement individual functions, can dynamically load new components, change the way various components interact and exchange data, move some of the functions from one host to another, and replicate some functions across multiple hosts.

The need for dynamic application reconfiguration has often been motivated by the *evolutionary change* of the application [56, 103] in response to changing requirements, or to correct faulty behavior. Several researchers have directed their attention to the problem of upgrading running systems and replacing individual components with newer versions [16]

or adding replicas of running components [53].

Support for runtime extension has become available in many operating systems (e.g., dynamic link libraries in UNIX and Microsoft Windows) and as part of component object systems (e.g., runtime component loading facilities in CORBA and COM). These facilities enable the evolution of systems without the need for recompilation, by allowing new libraries or components to be located, loaded, and executed during runtime. In most cases, the identity of the libraries and components to be loaded will only be known at runtime.

The dynamic loading facilities mentioned above allow only the loading or replacement of individual components or libraries. They do not take into consideration the semantics and functionality of a composition of components, and the interactions between these components. Also, they do not address the performance implications of being able to dynamically switch among multiple configurations of an application.

In addition to application evolution and component replacement, another motivating factor for our work on dynamic reconfiguration is the improvement of application performance. Not only can a distributed application evolve through reconfiguration, but the execution of the application can be made more efficient by changing the order and the location where various functions in the application are executed, according to resource availability and the patterns of interaction between components implementing these functions.

For instance, consider an application (Figure 1.3 a.) in which a component (P) applies some processing function to the data received from various data sources (S). At a further point in the data path, the data is filtered (F) according to certain criteria, before being delivered to the destination (D). In some situations, the execution of the application can be improved by applying the filter before the processing function. This eliminates the time needed to process data that will be subsequently discarded by the filter, and potentially reduces the network traffic. If the data processing is replicated across multiple hosts, the filter may also need to be replicated. Executing the filter right on the host where the data source is located can bring further improvement (Figure 1.3 b.).

In many situations, the decision over where to execute various components to obtain a more efficient execution can not be made statically, since it depends on runtime conditions. Therefore, certain components may have to be moved from one host to another, possibly after they have started execution.

Moving components of a running application between hosts raises several technical chal-

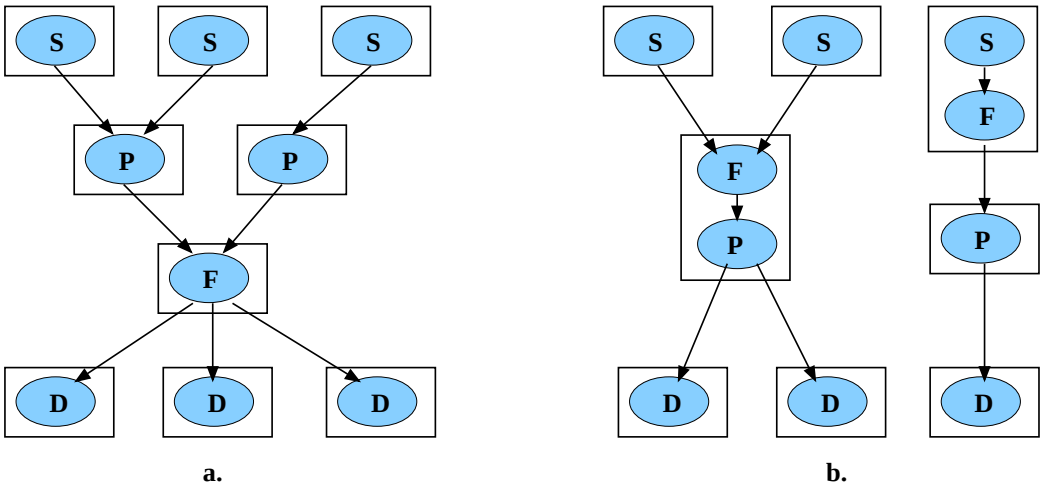

Figure 1.3: A modular application can be reconfigured at runtime to obtain a more efficient execution. Components can move from one host to another. The structure of the application can change through reordering or replicating some components. The components in this example are: S - data source, P - processing function, F - filter, D - destination. Rectangles represent hosts.

lenges. The state of the moving component and its connections has to be captured and transferred to the new location. The position and the relocation of a component should not affect the way it interacts with other components. The mobility of a component can be made transparent to communicating components and to the rest of the application. Based on our results, components in an application can continue to interact and to exchange data while they are moving. A combination of techniques such as message buffering and message forwarding ensures that data is not lost and all the messages addressed to a moving component are ultimately delivered.

The work presented in this dissertation explores the challenges and the benefits of applying dynamic reconfiguration to support the specific needs of mobile users. In our view, a mobile user is not only a user of a laptop computer that is carried around and connected at various points in a (wired or wireless) network. A mobile user utilizes various computing devices, sporting a wide range of connectivity, processing and display capabilities. In some cases, the user may be interested in transferring the state of a running application to a different device, and continuing the work on the new device from where it left off, while maintaining its previous interactions with communication parties.

When moving an application from one device to another, the application needs to adapt

to the capabilities of the device. For instance, it may not be possible to move a computing intensive application to a small device such as a PDA. Using our results, a client side application can be split into a part that does data processing and a part responsible for the interaction with the user. The computational part can permanently reside on a well connected desktop computer, with sufficient processing power. Meanwhile, the user interface component can move from one device to another, adjusting its display capabilities according to the specifics of the device.

One of the application domains that this dissertation is targeting is that of adaptive distributed collaborative applications or computer-supported cooperative work (CSCW). Several researchers have pointed out the importance of flexibility and adaptability in CSCW systems [10, 35, 59, 89]. The key point of many of these researchers is that there are significant tradeoffs in CSCW system design along many dimensions, and many of these tradeoffs in fact cannot be made *a priori*. They depend significantly on the context in which the system is going to be used.

Our work is complementary to the work mentioned above. It focuses on providing support for adapting the architecture of CSCW systems and location of system components and services to the context in which they are being used, scale of use, location of users, and to available resources. One of the goals of our work is to enable groupware applications to participate, on a limited basis, to collaborations on behalf of their users, while the users are disconnected or they are not active.

One of the difficulties encountered in the dynamic reconfiguration of distributed applications lies in *maintaining the consistency* of the application during and at the end of the reconfiguration process. Component consistency requires that the state of a component is not altered during reconfiguration. When a component is replaced or upgraded to a newer version, its state has to be transferred to the new component. End-to-end application consistency requires that the components in the application are in a safe state when reconfiguration is performed, so that the transactions in progress between components will ultimately complete, and the integrity of the data in traffic along a data path is not compromised.

This dissertation proposes several algorithms for maintaining application consistency during reconfiguration, based on blocking the activity of components and ordering the operations involved in the reconfiguration. These algorithms ensure the correct execution

of the application and maintain the integrity of the data exchanged while the application undergoes reconfiguration. They isolate the part of the application that is affected by the reconfiguration, and make sure that the rest of the application is not impacted by the structural changes.

A major component of the dissertation work is the design and implementation of a component-based framework, called DACIA[1], that provides support for building adaptive distributed applications [64]. One of our goals has been to provide a simple and easy to use, yet powerful architecture, that supports the construction and execution of reconfigurable applications. This dissertation explores the use of DACIA to model and develop reconfigurable distributed applications. It provides several examples of using code mobility and dynamic reconfiguration to improve the performance and usability of some test applications.

DACIA comes with a tool that allows the graphic visualization of a distributed application's structure. It also allows the manual administration of the application. It provides a collection of user-friendly mechanisms for reconfiguring an application, that support component instantiation, removal, and relocation, and establishing and removing connections between components.

A flexible infrastructure needs to provide mechanisms not only for changing the application structure at runtime (either manually or using an automated routine), but also for dynamically altering the rules and algorithms used for application reconfiguration. DACIA provides support for dynamically loading adaptive routines that perform automated application monitoring and reconfiguration.

The current version of DACIA is implemented using Java and runs on standard desktop computers as well as on PDAs that support the Java 2 Platform, Micro Edition (J2ME) [97]. It has been used to implement several prototypes of groupware applications that illustrate support for mobility and reconfigurability [63, 65].

1.3 Contributions

The work described in this dissertation addresses the issues regarding the dynamic reconfiguration of adaptive distributed applications. It particularly focuses on the applicability of a component-based framework to supporting mobile users and mobile environments. The

[1] Dynamic Adjustment of Component InterActions

specific contributions are:

1. **Dynamic application reconfiguration**

   We have designed and implemented a mobile component framework, DACIA, for building and executing adaptive distributed applications. An application with a modular architecture, in which various components implement individual functions, can change its structure at runtime. It can dynamically load new components, change the way various components interact and exchange data, move some of the functions from one host to another, and replicate some functions across multiple hosts.

2. **Support for application and user mobility and persistent connectivity**

   DACIA enables persistent connectivity between moving components. It allows a mobile user to simply "pull" an application or application component from one computing device and drop it on another computing device. The application maintains its state, no manual restart is necessary, and all connections are transparently re-established.

3. **Application parking**

   A mobile application can be parked while its user is disconnected or idle. A parked application is able to continue, with some limitations, to interact with other parties on behalf of the user. It can reside on the same computing device the user had been connected from, or it can move to a fixed host if the user's device is disconnected. When the user reconnects, eventually from a different place, she can take over the control from the parked application.

1.4 Organization of This Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 reviews the relevant work in the areas of component-based architectures, dynamic reconfiguration, and mobile applications. Chapter 3 presents the architecture of DACIA. Chapter 4 describes the support provided by DACIA for component mobility and mobile applications. Chapter 5 presents our approach to dynamic application reconfiguration. Chapter 6 illustrates some of the details of the DACIA framework implementation, as well as some performance results, and presents several applications we implemented using DACIA. Chapter 7 concludes with a summary of our contributions and directions for future work.

CHAPTER 2

A SURVEY OF RELATED WORK

The work completed in the context of this dissertation spans over several research areas, such as distributed systems, mobile computing, groupware systems, and software architecture. We view our contribution as an extension of the existing research results and their application in a novel way, sometimes in a completely different context than the one in which they have been previously investigated. This dissertation provides a unifying framework that addresses the following problems: a) *improving the performance of distributed applications*; b) *providing support for application and user mobility*; and c) *reducing the cost of application maintenance and upgrade*. We use a modular approach to build adaptive distributed applications. Applications developed using our mobile component framework can adapt to variations in load and resource availability through application reconfiguration and component mobility.

In this chapter we discuss some of the work previously done in areas that we consider strongly related to our research efforts: modular systems and component-based architectures, adaptive applications, dynamic application reconfiguration, code mobility and mobile agents, ubiquitous computing, and groupware systems.

2.1 Modularity and Application Decomposition

The idea of building computing systems through the composition of individual modules is not new; it has been used extensively in the design and implementation of systems ranging from layered operating systems [87] and network architectures [112] to more advanced distributed systems. The objective of a modular approach is to manage the complexity of

software systems, isolate failures, and facilitate the construction of customized configurations. A modular architecture offers unquestionable advantages, such as clear semantics, flexibility, extensibility, and easy reconfiguration. However, it can also introduce serious performance inefficiencies due to the overhead for communicating between components, crossing protection domains, and data encapsulation.

An approach frequently used to develop modular applications is to view an application as a *protocol stack*. Horus [101] and its ML implementation, Ensemble [38], treat protocols as abstract data types that can be stacked on top of each other in a variety of ways at runtime. Protocol modules have standardized top and bottom interfaces and they communicate with each other through message passing. Horus provides an object-oriented protocol composition framework; it supports objects for communication endpoints, groups of communicating endpoints, and messages.

Bast [30] applies the *Strategy design pattern* [29] to achieve flexible protocol composition. With Bast, a distributed system is composed of *protocol objects*, which are instances of *protocol classes* and have the ability to remotely invoke each other and to participate in various protocols. The Strategy pattern consists of objectifying an algorithm executed by a protocol object, i.e., encapsulating it into a *strategy object*, which is used by the *context object* represented by the protocol object. A strategy and its context are strongly coupled and the application layer only deals with instances of the protocol class.

The x-kernel [47] is an operating system kernel that provides an object-oriented framework designed to support the rapid implementation of efficient network protocols. It provides a uniform protocol interface and support library that allows the programmer to configure individual *protocol objects* into a *protocol graph* that realizes the required functionality. The x-kernel views a *protocol* as a specification of a *communication abstraction* through which a collection of *participants* exchange a set of *messages*.

Consul [72] is a communication substrate that offers support for building fault-tolerant distributed applications. It is based on the x-kernel and Psync, a group-oriented atomic multicast protocol that explicitly preserves the partial or causal order of messages. It consists of a suite of fault-tolerant protocols that together provide various services such as *broadcast, membership* and *recovery*. Consul implements the *replicated state machine* approach [91] and it provides a configurable architecture in which an application designer can build a system around a given collection of protocols with minimal effort.

The architecture implemented by Coyote [11] extends the notion of hierarchical composition used in the x-kernel with support for finer-grain *micro-protocol objects* and a non-hierarchical composition scheme for use within a single layer of a protocol stack. A two-level composition model is proposed. To construct a customized service, appropriate micro-protocols are configured together with a standard runtime system to form a composite protocol implementing the service. This composite protocol is then composed hierarchically with other protocols to form a complete network subsystem.

Scout [74] is a communication-oriented operating system built on the foundation provided by the *path* abstraction. A path represents the flow of data from an I/O source, through the system, to an I/O sink. The units of program development in Scout are called *routers*. Routers are organized in a *router graph*, defined at build time.

Some of the lessons learned from the design and implementation of Consul are presented in [73]. These can be applied to the design of any modular architecture. A main step in building a modular system is determining how to divide the required functionality into separate modules and then defining the appropriate interfaces. The goal of this process is to isolate each fundamental function, where such a function can be defined informally as one that is needed by multiple other modules. The modularization process is greatly affected by dependencies between modules. These include both direct dependencies caused by one module explicitly using another's operation and indirect dependencies where one module is affected by another without direct interaction. A closely related problem is the one of defining module interfaces, i.e., determining what the direct interactions should be and how they should be realized. The difficulty comes from the fact that the nature of these interactions evolves and changes as the system is constructed and the modules are used in new ways.

In a 1972 paper, Parnas [83] explores the criteria used in decomposing a system into modules and the efficiency of various decompositions. Two alternatives are proposed. The first one makes every major step in the processing a module, similar to a flowchart architecture. The second one is based on information hiding. A data structure, its accessing procedures and modifying procedures are part of a single module. Modules don't share data, but they communicate through method invocations. The author favors the latter approach, which has the potential to be more efficient due to reduced costs for exchanging data between modules, and also hides design decisions within a module. He acknowledges though

that achieving this efficiency requires a very careful decomposition, as well as assembler tools to aggregate several subroutines into a module.

We consider that it is very hard to achieve such a decomposition, the decomposition is often complicated, and the component writer has to know precisely the semantics of the whole application and how its functionality is achieved. Moreover, such an approach can easily become inefficient if there are frequent alternate invocations of methods implemented by different modules, which translate to transfers of control between modules. Therefore, in DACIA we support the functional decomposition of an application. Components are often loosely coupled and autonomous, and they communicate through message exchange. The application structure is clear and the interfaces between components are straightforward. An application can be changed or extended at runtime. New functionality can be added dynamically, without prior knowledge at the moment when it was first designed and implemented. In most cases, component developers only have to be aware of the specification of a particular component, without worrying about the way other components are implemented.

2.2 Optimizing Modular Architectures

The main disadvantage of modular systems consists of their performance inefficiencies, due to the overhead for crossing protection domains, traversing component stacks, and processing headers. Several techniques have been proposed in [39] to address this issue, based on identifying common execution sequences in protocol stacks, called *event traces*, and substituting them at runtime with optimized versions. The speed of computation can be increased by eliminating, whenever possible, the use of events between layers, using instead local variables, and by inlining the functions in a trace, thus eliminating multiple function calls. Other optimizations are compression of headers, and delaying the execution of some operations in certain situations.

Some compiler-based techniques for improving component processing latency are presented in [75]. The metric used is the *memory cycles per instruction* (mCPI), representing the average number of cycles that an instruction stalls waiting for a memory access to complete. The mCPI is reduced by (a) increasing the dynamic instruction stream density, (b) reducing the number of cache conflicts, and (c) reducing the critical-path code size. Three techniques are proposed: *outlining, path inlining*, and *cloning*. Outlining represents

the reduction of the length of a common execution path by moving error handling code out of the main line of execution, for example to the end of the function or to the end of the program. Path inlining consists of replacing an entire path with a single function. Although this leads to the growth of code size, it is efficient if the path is frequently executed. Cloning involves creating a copy of a function. The cloned copy can be relocated to a more appropriate address, specialized and optimized for a particular use. By choosing the point where cloning is performed, it is possible to tradeoff locality of reference with the amount of specialization that can be applied. Cloning at connection creation time will lead to one cloned copy per connection, while cloning at component stack creation time will require only one copy per stack.

2.3 Component-Based Architectures and Systems

There exist several distributed component architectures that share some of the goals and design principles with our work, and also exhibit a range of differences. CORBA [80] provides a distributed object model that supports location transparency, platform interoperability, and portability. Using an ORB (Object Request Broker), a client object can transparently invoke a method on a server object, without the need of being aware of where the object is located, its programming language, and its operating system. CORBA does not provide support for component mobility, communication connectivity for mobile components, and adaptability, features that are offered by DACIA. Our work focuses on providing support for adapting at runtime the structure of distributed systems and location of system components and services to the context in which they are being used, scale of use, location of users, and to available resources (e.g., CPU, network bandwidth, display).

The Distributed Component Object Model (DCOM) [92] is an application-level protocol for object-oriented remote procedure calls. DCOM supports remote objects through a protocol called the Object Remote Procedure Call (ORPC). This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server object can support multiple interfaces, each representing a different behavior of the object. A DCOM client calls the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces.

Coign [46] is an automatic distributed partitioning system for COM applications. Given

a binary application built from distributable COM components, Coign constructs a graph model of the application's inter-component communication through scenario-based profiling. Next, it applies a graph-cutting algorithm to partition the application across a multiple hosts in a network. Coign operates on binary applications; it can optimize applications without access to source code. Coign does not increase the parallelism in application code, nor does it perform horizontal load-balancing between peer servers.

ABACUS [4] is a run-time system that dynamically changes the function placement for data-intensive applications. The system monitors run-time resource usage and availability, without knowledge about the application internals. It learns about the most important inter-object communication patterns, and per-object resource requirements. Based on these observations, it decides on moving various functions between servers and clients in response to dynamic conditions.

The Rover Toolkit [51] implements a distributed object model that provides a uniform view of objects at the OS level and a *queued RPC* mechanism for disconnected operation and object migration. For instance, simple GUI code can be migrated to a mobile client, where it uses queued RPC to communicate with the rest of the application running on the server. In Rover, object migration is restricted to be between a user's desktop (server) machine and the user's mobile device (client). The goal of our work is to allow components to move freely across all the machines in the system.

Our approach of separating the mechanisms used to reconfigure an application from the reconfiguration policies has some similarities with the solution proposed by FarGo [43]. FarGo provides support for moving the components of a distributed application among multiple hosts during the execution of the application. The programming model proposed, called *dynamic application layout*, separates the programming of the layout of the application from the application logic. FarGo uses Java RMI to implement a reflective inter-component referencing model that allows the attachment of relocation semantics to inter-component references. An event-based monitoring service provides support for making runtime relocations decisions. The changes of an application layout in FarGo consist of finding the right place to execute components and migrating components at runtime. The model proposed by FarGo can potentially be extended with support for dynamically changing the connections between components and introducing new components.

Component-based distributed platforms have also been used in other contexts, such

as resource management in large-scale distributed environments. The Globus project [25] focuses on the infrastructure needed to build *computational grids*, execution environments that enable an application to integrate geographically-distributed instruments, displays, and computational and information resources. The Darwin project [13] develops a set of customizable resource management mechanisms, based on hierarchical scheduling, a signaling protocol, and service brokers.

One of the problems encountered in the specification of component-based applications is the need for a component interface description. For example, CORBA [80] uses the *Interface Definition Language* (IDL) to describe the method calls accepted by a component. In Darwin [67], a component is described in terms of the communication objects it provides (corresponding to output channels), and the communication objects it requires (corresponding to input channels).

Most existing component-based systems do not offer satisfying solutions to a whole range of new application requirements, such as the need to support Internet applications, quality of service, mobile devices and mobile applications, and ubiquitous computing. Web-based applications need to cope with issues such as: a) unpredictable number of users and load, b) stateless user sessions, and c) no reliability or QoS guarantees from the communication infrastructure. Mobile computing requires applications to deal with widely variable connectivity and resource availability, and to transparently handle temporary disconnections. There is a growing need to build customizable and flexible component-based architectures for inherently heterogeneous environments.

To the best of our knowledge, there are no toolkits that provide support for building dynamically reconfigurable modular applications. The goal of our work is to enable the runtime reconfiguration of distributed applications, by changing the connections between existing components, changing the location of execution of various components, or replacing a set of components with a different set of components. We specifically focus on minimizing communication costs when components are (re)located on the same host.

2.4 Adaptive Applications

Several adaptive solutions have been proposed to address the variability and resource constraints encountered both in wired and mobile computing systems. Mobile environments

represent an ideal field for the design and implementation of adaptive services, being characterized by unpredictable variation in network quality, wide differences in the availability of remote services, limitations on the resources available on the mobile hosts due to weight and size constraints, restricted battery power consumption, and lowered trust and robustness derived from exposure and motion [20].

Most of the existing work targets the improvement of network communication and bandwidth usage. Some of the solutions proposed (e.g., the *adaptive service model* in [66]) are based on admission control and resource reservation. Other research projects targeting mobile environments, such as Daedalus/BARWAN [41, 86] achieve adaptation primarily through the use of proxies that perform application-specific transformations of the data streams or *on-demand dynamic distillation* (data type-specific lossy compression) [26, 111].

In Odyssey [78], the responsibility for coping with changes in resource levels belongs to applications. Adaptation is regarded as a collaborative partnership between the system and individual applications. The system monitors resource levels and notifies applications of relevant changes. Each application independently decides how to adapt best when notified.

Conductor [110] provides a general mechanism to select and dynamically deploy combinations of *adaptive agents* to multiple points in a network. Similar in some of its goals (e.g., dynamic component composition, allowing applications to adapt to environmental changes) both to Conductor and to our work, CANS (Composable, Adaptive Network Services Infrastructure) [27] provides an application-level infrastructure for customizing the data paths between applications and services through the injection of application-specific components into the network.

Our work does not attempt to rival with the adaptive solutions proposed by other researchers. Instead, it provides an infrastructure that can be used to implement some of these solutions. It provides the mechanisms that enable the easy deployment of customized adaptive applications, and allow to dynamically change the adaptive algorithms implemented by a particular application. DACIA can be used to carry out some of the adaptive decisions made by the systems mentioned above, such as deploying adaptive agents or data filtering proxies [111].

Our solution to the problem of adapting to variability and heterogeneity is based on considerations over the functionality of an application and the way it is achieved. It looks at the structure of an application and strives to achieve a more efficient execution by changing

the way of interconnecting components and their location of execution. Adaptive solutions are application-specific and they are implemented in the application code. Our framework provides the mechanisms used to reconfigure an applications. General-purpose performance monitoring routines (e.g., network and inter-component communication load) can also be implemented in the framework.

One of the problems the Portolano project [22] strives to address is the high-cost and lack of flexibility of the current, *vertically integrated* system architectures. These architectures attempt to provide distinct solutions to specific problems. Thus it is difficult or even impossible for a user to obtain exactly the subset of services she requires. By contrast, a *horizontally integrated* architecture allows the deployment of various services on an as-needed basis. DACIA provides an example of such an architecture. It allows services to be built, composed, and configured dynamically, and to migrate services or service components based on proximity and resource availability. Persistent logical connectivity between DACIA components and message queuing and forwarding techniques address another issue considered by the Portolano group, namely the need to build mechanisms for managing intermittent connectivity into the infrastructure protocols.

2.5 Dynamic Application Reconfiguration

Dynamic reconfiguration is the process of modifying the structure of an application while the application is running. There is a need to modify long running applications through *evolutionary change* [103]. Evolutionary change can be defined as the accommodation of requirements changes, bug fixes, and environment changes. Developers and system administrators initiate evolutionary changes in response to evolving system requirements. Adaptive systems may themselves trigger modifications to their structure through *reactive change*. This occurs in response to application or system events. While application events are specific to individual applications, system events may be common occurrences such as processor or network failures. For applications where adaptive behavior and continuous service are required, reactive changes have to be expressed in a way that is robust to evolution.

The dynamic reconfiguration of an application should not compromise the application consistency. Maintaining application consistency leads to two problems: a) synchronizing the reconfiguration with the application execution, and b) managing the persistent

state of individual components and that of the whole application. The former requires the specification of a sequence of states that must be attained prior, during, and after the reconfiguration, and the use of algorithms for safe transition between these states. The latter demands that the application state maintained by components and their interconnections should be preserved throughout the reconfiguration process.

Warren and Sommerville [103] have identified two approaches for synchronizing the reconfiguration with the application execution. In the *active change management (ACM)* approach [56], the application components explicitly participate to the reconfiguration process. This requires components to be aware of and to respect the semantics of the change protocol. The *passive change management (PCM)* solution [23] uses a configuration manager to perform transitions between reconfiguration states, which are expressed in terms of asynchronous port interactions. The components do not directly contribute to the reconfiguration.

The management of persistent state introduces problems such as identifying the critical state and transferring the state from one component to another, where the data structures used by the two components may not be identical. Furthermore, the initialization of components which are dynamically introduced may require accessing the state of some of the existing components. The work of Hofmeister and Purtilo [42] addresses the issues regarding capturing the state of an executing task during the dynamic reconfiguration of an application. In order to support component replacement, their approach requires each component to provide two interface methods: one for capturing state information, and the other one for performing component initialization after the replacement.

Our work extends the previous work on dynamic application reconfiguration to the mobile environment. It provides support for maintaining persistent connectivity between moving components. We are also concerned with reducing the number of components affected by reconfiguration, and minimizing the impact of reconfiguration on applications and their users.

2.5.1 Configuration Programming

The *configuration programming* approach to software development [55] separates the functional specification and the implementation of individual components from the design and construction of a whole system, regarded as a set of components and their interconnections.

Components are functional entities which encapsulate *state*. They communicate through *connectors*, which can implement various policies for message exchange. For instance, they can queue data for effective data and control exchange, or they can specify whether the communication is synchronous or asynchronous.

The configuration programming paradigm suggests the use of two languages in the application development: a programming language for implementing individual components[1], and a configuration language for structuring an application. Configuration languages provide a set of commands for component creation, removal, binding, and execution. They provide support for expressing the entire configuration of a distributed application, specifying constraints in configuring applications, and specifying the policies and restrictions that affect configuration changes.

2.5.2 Configuration Languages and Formal Specifications

Configuration languages primarily serve the goals of specifying configurations and constructing the initial configuration of an application. Some languages also provide limited support for runtime application reconfiguration. The focus of our work is not the configuration specification, but the actual execution of the configuration changes. Our work provides mechanisms for connecting and disconnecting components, dynamically loading components, or moving components across hosts. This dissertation presents some examples of actually using our mobile component framework for constructing and executing various adaptive distributed applications. It quantifies the improvements in resource usage and the performance of these applications.

There exist several formalisms that express the dynamics as well as the mobile behavior of distributed systems. The $\pi - calculus$ [71] is an elementary calculus for describing and analyzing concurrent systems with evolving communication structure. It expresses process mobility through sending and receiving channels. Darwin [67] is a declarative binding language that can be used to define hierarchic compositions of interconnected components. The operational semantics of Darwin is described in terms of the π-calculus. Darwin supports the specification of both static structures fixed during system initialization and dynamic structures that can evolve during execution. In contrast to its predecessors, Conic [68] and

[1] multiple distinct languages can be used for component implementation

Rex [57], which had centralized sequential interpretations, Darwin has a distributed and concurrent interpretation that allows the construction of large distributed systems in an efficient manner.

The module interconnection language PCL [95] describes evolving systems by modeling the differences between multiple versions of a system and the relationships between different parts of the model. PCL was primarily designed to support static configuration management. The abstract model proposed by Warren and Sommerville [103] to manage automatic reconfiguration, based on PCL, uses a repository to store application *version descriptions*. A version description is defined as a set of attribute values which, when applied to an application specification, generates a unique system instance.

One of the formalisms proposed to express dynamic changes in software architecture is the *chemical abstract machine* (CHAM) [106, 107]. A CHAM is a set of *rewrite rules* that are applied to a *multiset* of data elements, whose syntax is specified by the designer. It can be used to express both programmed and ad-hoc (triggered by the user) reconfiguration.

Graph rewriting rules have been proposed to model dynamic architectures for applications represented as graphs of connected components [70]. An architectural style is viewed as a class of graphs, which can be generated using a context-independent grammar.

Information Flow Graphs (IFGs) have been used to specify logical flows of data in event distribution middleware systems [7]. An IFG allows the specification of *stateless* event transformations such as *select, transform, merge and split*, as well as *stateful* operations such as *collapse and expand*. Optimization algorithms enable the rewriting of an IFG (e.g., reordering select and transform operations) in order to improve the performance of event distribution.

Dynamic reconfiguration also occurs in mobile computing, where the architecture of the whole system is modified more frequently. One of the formal models for mobility is Mobile UNITY [69], an extension of the parallel program design language UNITY. Mobile UNITY defines coordination within a system of components in a separate, global interaction section.

2.5.3 Software Connectors

The *software architecture* research area [84, 93] highlights the importance of *software connectors* [3, 17] in the development of architecture-based applications. Connectors are explicit architectural entities that bind components together and act as mediators between them

[93]. They define the interactions between components. They free components from the responsibility of knowing how they are interconnected, and allow the specification of functional behavior independent of communication mechanisms. They also introduce a layer of indirection between components. Connectors implement various communication semantics (e.g., synchronous or asynchronous), and they can provide additional services such as buffering, fault-tolerance, etc.

Connectors can provide significant support for dynamic application reconfiguration. For example, to support component replacement, connectors can regulate the data flow between the component to be replaced and other components connected to it. They can redirect all communication from and to the component to be replaced to the new component. To support a replacement policy based on replication, incoming messages can be redirected to all or any one of the members of a set of component replicas. Connectors can also provide support for fault tolerance during reconfiguration. If a component becomes unavailable or a network connection is temporary interrupted, connectors can buffer or retransmit messages exchanged between components. Connectors allow changing the mechanisms and policies used to reconfigure an application without affecting the semantics of reconfiguration.

Software architecture regards connectors as top-level constructs. By contrast, in conventional programming languages, connectors are primitive and they implicitly exist in procedure calls or global variables. In software architectures, connectors are explicitly implemented as specialized data structures, shared variables, buffers, pipes, linker instructions, message exchange and routing mechanisms, local or remote procedure calls, or client-server protocols. Several research, as well as commercial middleware systems, implement various interconnection mechanisms, such as: Polylith [85], Field [88], DCE [90], CORBA [81], COM/DCOM [92], ILU [109], and Java RMI [98].

Similarly to the type of connectors at the architectural level, various middleware technologies use different methods of communication, such as remote procedure calls (RPC), message passing, passing object references, or shared memory. Software connectors provide a uniform interface to other connectors and components within an architecture. Internally, however, connectors based on different middleware technologies have different capabilities. Encapsulating the middleware functionality within software connectors maintains the integrity of an architectural style, by separating the logical inter-component connectivity from implementation-dependent factors. It is possible to implement connectors that cre-

ate bridges between different communication methods, or make translations from one to another, to fit within a specific architectural style [17].

One of the goals of using connectors is to conceal platform- and language-specific features, as well as physical characteristics of the communication medium and to provide a standard interface for communication across various platforms. For instance, asynchronous communication is more suitable between components located on remote hosts, especially in mobile environments, characterized by high latencies and reduced reliability. Conversely, synchronous communication can be desirable for local communication, due to efficiency considerations. Middleware mechanisms can hide these differences and provide a uniform syntax for inter-component communication.

2.6 Mobile Code and Mobile Agents

Code mobility has emerged as a promising solution for the design and implementation of large scale distributed applications. It attempts to complement existing approaches to distributed computing (e.g., client-server) and overcome some of their drawbacks and limitations, such as the reduced degree of configurability, scalability, and customizability.

Existing mobile code systems offer two forms of code mobility [28]. *Strong mobility* allows migration of both the code and the execution state to a different computing environment. *Weak mobility* allows only the transfer of code across different computing environments. There are several tradeoffs between these two approaches. Strong mobility is often implemented at the operating system level, e.g., in the form of *process migration* [79]. In many cases, the goal of process migration is to balance the load across network nodes. Process migration can be implemented in a transparent fashion, so that the application programmer has neither control, nor visibility of the migration. The amount of state involved in the migration can be high, due to the need to capture the execution state (e.g., stack and registers content, open file descriptors). On the other hand, the implementation inside the kernel can increase the efficiency of state capture at the initial location and restore at the destination.

Weak mobility requires the moving entity (e.g., process, object, or component) to reach a safe state before it is being moved, therefore it may introduce additional delays. At the same time, the amount of state that is being captured and transferred is smaller. *Object*

migration [52] implements a finer-grained mobility with respect to process-level migration. The programmer controls the migration, can determine object locations and the moment when migration occurs.

Code mobility has received a great deal of attention in the research community. Several languages have been developed to support code mobility, including Telescript [108], Obliq [12], and Sumatra [2]. The Java Virtual Machine (JVM) and Java's class loading model, coupled with features such as serialization, remote method invocation (RMI), the sandbox security model, and reflection, have lead to a wide adoption of Java in the development of mobile agent systems [54].

A *mobile agent* is a program which represents a user in a distributed environment and is capable of autonomously migrating from node to node, performing computations on behalf of the user. The main advantages of the mobile agent paradigm lie in its ability to move executing code and make use of remote computing resources, and in permitting increased asynchrony in client-server or multi-party interactions.

More recently, several Java-based mobile agent systems have been proposed, including Aglets [58], Odyssey from General Magic, Voyager [82]), Mole [8], and Ajanta [100]). These systems support the notion of mobile agents, sometimes with different interpretations. For example, in Telescript [108], an agent is represented by a thread that can migrate among different nodes carrying its execution state. However, in TACOMA [50] agents are just code fragments associated with initialization data that can be shipped to a remote host.

A key difficulty in deploying mobile agent systems is that of handling security [14], since agents are assumed to be capable of moving to arbitrary hosts (e.g., a database querying agent moving from host to host on the network). The following problems need to be addressed: a) protecting an execution environment against potentially malicious mobile code, and b) protecting the mobile code against potentially malicious hosts and execution environments. Some of the techniques proposed to address the inherent security risks of mobile code are: sandboxes [102], code signing (e.g., signed applets), firewalls, and proof-carrying code [77].

2.7 Ubiquitous Computing

The goal of *ubiquitous computing* (or *ubicomp*, for short) [104] is to make computational devices so pervasive throughout an environment that they become transparent to the human user. The ubicomp vision pushes computational devices out of conventional desktop interfaces and into the environment in increasingly transparent forms. In the ubiquitous computing world personal organizers talk to cell phones to cars to network computers, tailoring information to needs as they arise. Phones and hand-held devices, for example, know when users are actively working or note and give personal and business information as needed. *Context-aware applications* [37] are able to follow their users as they move around a building. They can adapt to the characteristics of the environment where they execute, such as location and the input/output resources available, and to changes in the environment.

Novel software engineering solutions are needed to provide the functional features required by ubiquitous and pervasive computing [1, 6]. These applications strive to remove the physical barriers between users and the work they accomplish via a computer and to produce transparent interaction techniques. They need to adapt their behavior in accordance to changes to the context of their use. They need to support user mobility by adapting their behavior based on knowledges of the user's current location, using, for instance, active badges [37].

To increase their flexibility, future distributed services should be developed through horizontal composition, as opposed to existing vertically integrated systems, that attempt to provide entire solutions to specific problems [22]. They should provide support for resource discovery and they should gracefully handle the intermittent connectivity characteristic to mobile environments.

2.8 Groupware Systems

One of the application domains that our work on DACIA is targeting is that of adaptive distributed collaborative applications or computer-supported cooperative work (CSCW). Several researchers have pointed out the importance of flexibility and adaptability in CSCW systems [10, 89]. The need to provide support for building flexible architectures for computer-supported collaboration in a heterogeneous and dynamic environment has also received a considerable amount of attention [19, 35, 104]. We believe that in fact there are many

dimensions of flexibility and adaptability in CSCW systems. Some of these dimensions include:

- access control [21, 94];
- concurrency control [19, 36];
- coupling of views [18];
- extensible architectures [24, 59].

The key point of many of these papers is that there are significant tradeoffs in CSCW system design along many dimensions, and many of these tradeoffs in fact cannot be made *a priori*. They depend significantly on the context in which the system is going to be used. Our work is complementary to the above work and focuses on providing support for adapting the architecture of CSCW systems and location of system components and services to the context in which they are being used, scale of use, location of users, and to available resources.

Other researchers have emphasized the importance of considering available resources in system design. Hudson and Smith point that CSCW systems may need to be designed to allow tradeoffs between context awareness and available resources (CPU, display, network) [44]. There is a cost to providing more awareness information in terms of information overload, screen real-estate, network resources, privacy, etc. There have also been debates over the merits of centralized architectures, peer-to-peer architectures, and replicated services in building groupware systems. Our goal is to provide mechanisms to CSCW system designers so that the systems and their architecture can be more easily reconfigured, at run-time if desired.

Although it targets a different application domain (centralized shared window systems), the work of Chung and Dewan [15] has many goals similar to ours, such as: migrate applications to make better use of the available resources, accommodate heterogeneous environments, and offset the cost of application migration through the benefits of more efficient computation. Their approach is based on the migration of a X Window client that receives inputs generated by multiple users, and the migration of the events logged at a particular site. In DACIA, a component can move from one host to another, and the messages re-

ceived at the previous location while the component is moving are forwarded to the new destination.

The importance of supporting mobility of users has also been argued recently. The work in the cooperative buildings area assumes that the users are mobile inside buildings and the work should be possible anywhere the users are (coffee table, walls, desktops, etc.) rather than users having to work on a standard desktop [96]. In other mobility work, Belloti and Bly argue that CSCW systems must be designed to support mobility because mobility can be critical to many work settings [9]. They conclude that CSCW systems must accommodate mobility rather than seek to eradicate it via desktop collaboration tools. In their study, they found that particular support is needed for "local mobility" where people walk between rooms or buildings at a local site. DACIA simplifies building groupware applications in which clients are mobile.

CHAPTER 3

DACIA ARCHITECTURE

DACIA is a framework for building adaptive distributed applications in a modular fashion, through the flexible composition of software modules implementing individual functions. A DACIA application is constructed by connecting in a particular configuration several components implementing various functions or parts of the application. The application can be seen as a directed graph of connected components. The links between components indicate the direction of the data flow within the application. The graph may have cycles and multiple paths may exist in the graph between two components.

Components in an application graph can be distributed over multiple hosts. Application graphs in systems such as x-kernel [47], Coyote [11], and Scout [74] are defined at build time, and they reside on a single host. A DACIA application graph is constructed at build time, and it is distributed over multiple hosts. The graph can be modified at runtime by changing the connections between components, introducing new components, or removing components. Some components in the application graph can move between hosts. The systems mentioned above do not support component mobility.

The novelty of our approach lies in the flexibility of building an application and the ability to change the application structure at runtime. The same application can be built in multiple ways, either by configuring differently the same set of components or by using different sets of components. For the same DACIA application, multiple semantically equivalent application structures can be defined (e.g., the application in Figure 1.3).

3.1 PROCs

In DACIA, a component is a *PROC* (Processing and ROuting Component)[1]. A PROC can apply some transformations to one or multiple input data streams. It can synchronize input data streams; it can split the items in an input data stream and send them alternately to multiple destinations. PROCs represent the basic building blocks for an application. They can be interconnected in multiple ways, according to certain rules and restrictions. A PROC is identified system-wide using a unique identifier obtained by combining the ID of the host where the PROC originated and a counter maintained by the host.

There are certain differences between PROCs and objects in other component software architectures. PROCs are not just encapsulated objects. They are relocatable data objects. They are executable entities that may hold state, may be interrupted and restarted, and they are involved in communications with other entities.

PROCs are loosely-coupled software entities. Except from certain periods of time when they interact with one another, their state is self-contained. They do not depend on the functionality provided by other PROCs. This is unlike component-based models such as Darwin [67], where a component requires some services provided by other components. A PROC is not required to be aware about the existence of other PROCs or about the structure of the application. For a PROC, the fact that the connected PROCs are local or remote is transparent. If context-awareness information is needed, PROCs can deliberately exchange information about each other and about other PROCs in the system.

3.1.1 Inter-component Communication

PROCs communicate by exchanging *messages* through *input and output ports*. A *connection* is established between one input port and one output port. This contrasts to the object reference model used by other component architectures, like CORBA [80], where components communicate through method invocation. Our choice is motivated by the fact that many of the applications we are considering exchange streamed data, for which an RPC-like invocation model is not appropriate. Ports provide a clean way of specifying the one-to-one connections corresponding to a graph structure. They allow PROCs to distinguish among messages received from different sources, to synchronize and to combine

[1] in the remainder of this document, we will use the terms PROC and component interchangeably

various data streams.

In DACIA, we avoid the use of connectors [3] as explicit entities. Instead, a connector (logical connection) is realized by pairing two ports belonging to two different PROCs. This simplifies the architecture of the system and improves the communication performance, by eliminating unnecessary indirections and the communication overheads associated. The implementation of the underlying mechanisms used for communication (both local and remote), managed by our DACIA framework, is transparent to individual components and to the application.

Existing component-based architectures usually support two types of communication: *synchronous* and *asynchronous*. Synchronous communication corresponds to the traditional blocking RPC semantics. In the asynchronous case, a caller component invokes a method on or sends a message to another component and then continues its execution without waiting for the completion of the request. Various architectures support either one or both the above methods. For instance, communication in Darwin is primarily synchronous[2]. A component that initiates a *transaction* on another component blocks waiting for the completion of the transaction. Since multiple dependencies exist between transactions, the synchronous approach can lead to long periods of inactivity for the blocked components. Besides the synchronous and asynchronous (one-way) communication, CORBA supports the *deferred synchronous* communication mode, which allows a client to continue its execution immediately after invoking a method call on a server, but to later poll the server for a result. This possibility is available only through the *Dynamic Invocation Interface (DII)*.

We designed and implemented the communication mechanisms in DACIA with the goal of minimizing the overhead of message exchange. Another objective was to provide uniform invocation methods regardless whether the communicating PROCs are co-located or they are located on different hosts. In a typical implementation for such a uniform solution (e.g., the use of a protocol such as TCP for both local and remote communication), the two components belong to two different processes (Figure 3.1.a). In the case of local communication, the cost of crossing address spaces and user-kernel boundaries is added.

DACIA provides a lightweight solution to local communication, by co-locating local PROCs within the same address space. The message exchange can be either synchronous

[2]Darwin also provides a *send* primitive for asynchronous interaction

or asynchronous. The basic primitives used to communicate between PROCs are:

- *output(portNo, message, isSynchronous)* - send a message to the specified output port, either synchronously or asynchronously.
- *input(portNo, message, isSynchronous)* - receive a message on the specified input port.

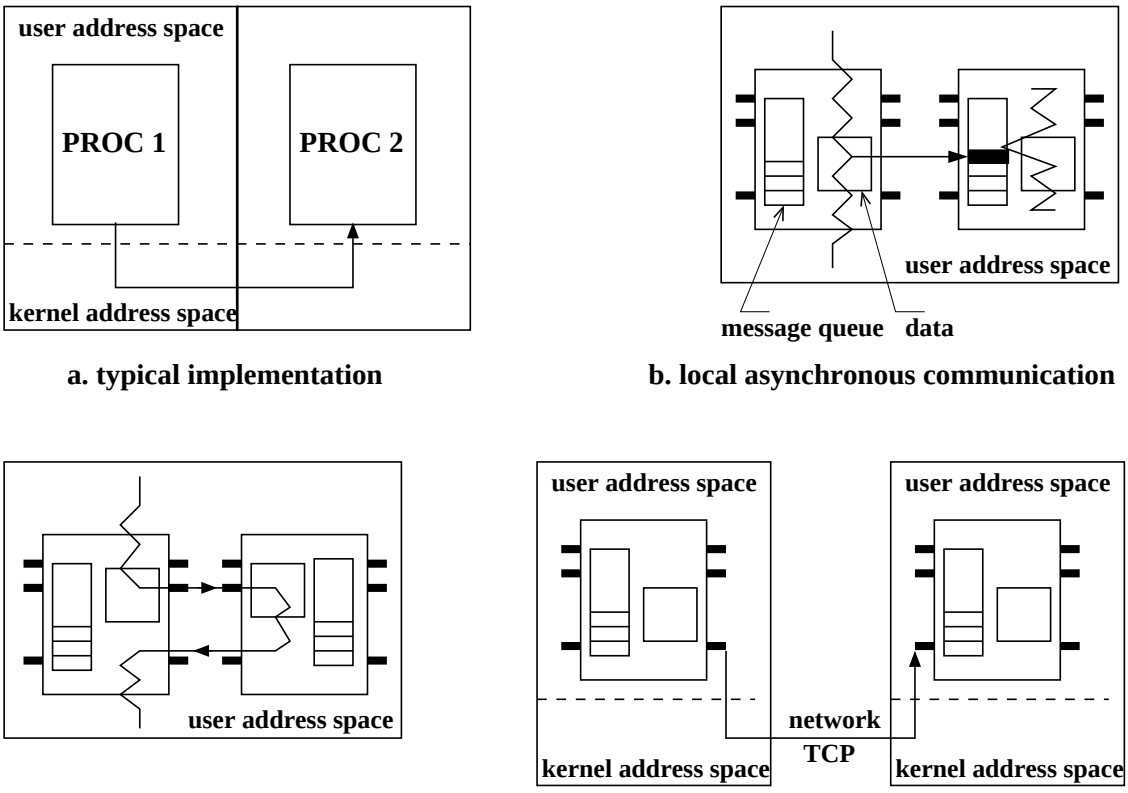

Figure 3.1: Inter-PROC communication. **a.** In a typical implementation, the two components belong to two different processes, and message exchange requires crossing address spaces. **b., c.** In DACIA, when PROCs are located on the same host, they are in the same address space and message exchange translates into simple procedure calls. If they communicate asynchronously, the cost of thread scheduling and message queue management is added. **d.** When PROCs are located on different hosts, the communication overhead increases due to the cost of network communication and crossing user-kernel boundaries.

In the case of asynchronous communication (Figure 3.1.b), the messages received by a PROC are inserted into the PROC's *message queue*. Every PROC has a thread (*asynchronous thread*) that handles the messages in the queue, usually in FIFO order. Alternatively, different queue policies can be implemented, based on the ports where particular messages are received, or on message priorities. A locking mechanism ensures the mutually

exclusive access of multiple threads to the queue. The asynchronous thread is blocked when the message queue is empty. A call to the *input()* method causes the message to be added to the message queue, and the corresponding asynchronous thread is notified. This thread is responsible for executing the message handling routine on the input message.

Using one asynchronous thread for each PROC may not be acceptable if a large number of PROCs exist on one host. In such a case, a thread pool could be used instead. Using this approach, all messages received asynchronously by a PROC are inserted into the PROC's message queue, from where they are executed in order by the threads in the pool, as they become available. Additionally, priorities can be attached to messages, and the queue becomes a priority queue.

Synchronous communication (Figure 3.1.c) further reduces the cost of exchanging messages between PROCs located on the same host. The PROCs share the same address space, and message exchange translates into simple procedure calls. To reduce overheads, the thread that executes the *output()* method on the source PROC also executes the *input()* method and the message handling routine of the receiving PROC. This eliminates the costs associated to thread context switch and message queue management incurred in the asynchronous case.

Remote communication (Figure 3.1.d) is done through the engines running on the source and destination hosts, respectively. The source engine is responsible for routing a message to the appropriate host. The destination engine dispatches the message to the receiving PROC, identified based on the PROC ID contained in the message. Multiple logical connections between pairs of PROCs are multiplexed over a single network connection between two engines.

Local communication can be either synchronous or asynchronous, while remote communication is always asynchronous. The same connection allows two co-located PROCs to exchange data both synchronously and asynchronously. Synchronous communication can be used to reduce the communication time and ultimately the end-to-end processing time along a data path involving multiple components. Asynchronous communication can be preferred in certain situations to reduce the length of a reactive invocation path, in the case of a sequence of messages generated as a result of handling previously received messages. The processing done by a thread can thus be reduced to executing only one (or a few) message handling routines, as opposed to executing the whole chain of message exchanges

and handling routines.

The *output()* and *input()* primitives resemble the primitives used by Hoare's *communicating sequential processes* (CSP) [40]. The work on CSP, carried out in the context of parallel programming, proposes that these primitives be part of the programming language. In CSP, communication occurs when one process names another as destination for output, and the second process names the first as source for input. The input-output matching has to be done for every individual message.

In DACIA, a connection between two communication ports allows multiple messages to be sent between the corresponding PROCs. The existence of multiple ports per PROC allows a PROC to be involved quasi-simultaneously in communication with several other PROCs. The communicating processes in CSP are blocked until communication occurs. DACIA supports both blocking (synchronous) communication semantics and non-blocking semantics, in which messages are accessed at the destination through the message queue. It also supports communication across multiple hosts.

A connection can be either directed or direction-less, depending on the particular needs of an application. All ports of a DACIA component are identical. However, the same port can act either as input or as output port, depending on how the connection is established. A connection is oriented from an output port to an input port. This represents the main direction of exchanging data between participant PROCs, and contributes to the directed graph structure of a DACIA application. Nevertheless, two connected PROCs can exchange data in both directions, using the same connection. The reverse direction can be used, for instance, to send replies for previously received messages, or to send notifications about message processing or PROC failures. Our experience with implementing DACIA components and applications has been that in many cases a pair of components needs to exchange data in both directions. Using a single connection instead of two reduces the effort of setting up the application, simplifies the structure of the application, and reduces the number of ports used.

Ports are not typed, and the messages exchanged can contain any type of object[3]. This eliminates syntactic checks at application build time, but can potentially introduce runtime

[3]Objects contained in messages need to be *serializable* for communication across hosts. Alternatively, default Java serialization routines can be overwritten by providing customized data marshalling/unmarshalling routines.

errors due to some PROCs not being able to interpret certain messages.

One reason for avoiding the use of typed ports and syntactic checks in DACIA is that syntactic correctness of an application, while necessary, does not always imply semantic correctness. When building a DACIA application through component composition, the focus is on the functional characteristics of various components and the semantics of their composition. If an application is semantically correct, then a component is able to process the data received from another component, assuming that the implementation of the components respects their specification. We can make an analogy with a Lego game, where various pieces of different types and colors can be connected in any configuration and they can stick together. However, only certain configurations make sense and are therefore usable.

There are situations in which a PROC does not need to completely understand a message it has received. Instead, the message handling routine operates only on parts of the message. In other cases, a PROC does not interpret a message received at all. The PROC can simply have the role of routing messages to other components, or synchronizing multiple input and output data streams.

In the absence of port types, a PROC provides multiple interfaces through the existence of multiple ports, which in many cases have different functionality. Different message handling routines are executed for messages received on different ports. For instance, a PROC performing the synchronization or merging of two data streams can use *port1* and *port2* as inputs for the data streams, *port3* as output port, and *port4* as a control channel.

DACIA allows applications to be temporarily inconsistent (some ports are disconnected) and it provides mechanisms for handling faulty component behavior or incorrect structure of the data exchanged between components. If a PROC receives a message of the wrong type, the PROC either gracefully handles the error, and eventually bounces the message back to the sender, or it throws an exception.

When an application is configured, not all the ports corresponding to a PROC have to be connected. This contrasts with the approach adopted by Darwin [67], where all the *provided* and *required* interface objects have to be matched. In DACIA, by connecting only some of a PROC's ports in a particular application configuration, only parts of the PROC's functionality may be exposed. In a different configuration, it is possible that the PROC acts differently.

For example, consider a PROC that merges several input data streams, applies some

filters to the resulted data, and then distributes the resulted stream to various subscribers to the data. If only one input port is connected, the merging function is not executed. Merging is done differently for two input streams and for four input streams. Different filters can be applied, depending on the ports where input is received. Similarly, if only one output port is connected, the data is just sent without any restrictions. If multiple output ports are connected, the PROC acts as a broadcast point. It can synchronize the rate at which it sends data to various subscribers, or it can assign priorities to various output ports. If no output port is connected, the PROC can drop all the input data without performing any operation, or it can do the merging and filtering and store the resulted data for subsequent delivery.

3.1.2 Hierarchical Component Composition

Primitive components are implemented by various programmers to perform specific functions. A primitive component is characterized by its functional description and the interfaces it provides to other components and to applications. It has a behavioral specification, as opposed to a structural description. Various implementations can exist for the same primitive component.

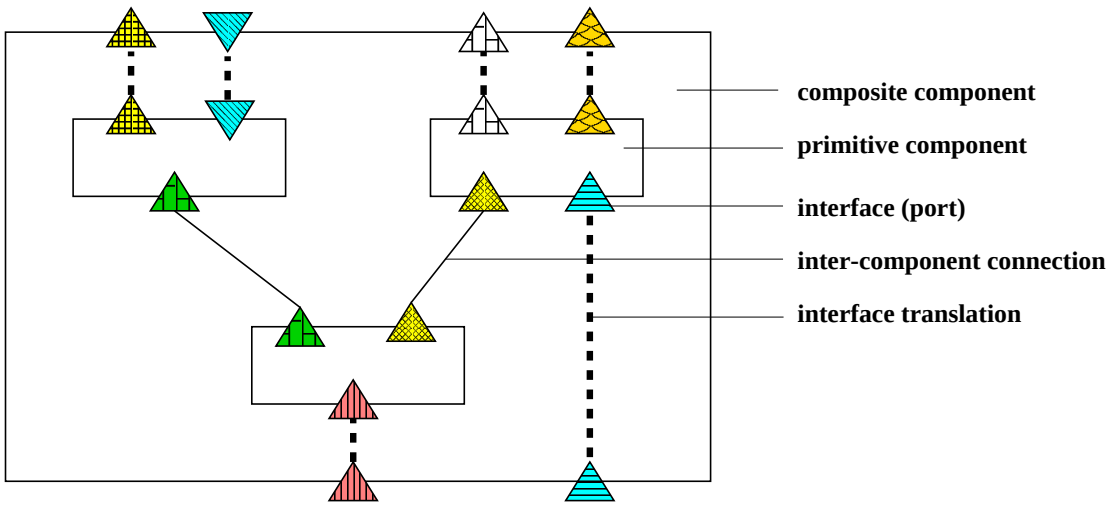

Figure 3.2: A composite component consists of a set of inter-connected primitive components. The interfaces of the composite component are a subset of all the interfaces provided by its sub-components.

A *composite component* (Figure 3.2) is created by connecting in a particular way several

primitive components. Some of the primitive components provide certain services to other components, through their respective interfaces (ports). The interfaces of a composite component usually represent a subset of all interfaces provided by its sub-components. A functional description is attached to a composite component. The component can be used as a single stand-alone unit that can participate to further compositions.

Composite components can be subsequently connected with other primitive and composite components to construct higher level composite components. In this way, the overall architecture of a system can be defined as a hierarchical composition of primitive and composite components, which at execution time may be located on multiple computers.

3.2 Engines and Applications

The *engine* is the most important part of the DACIA framework. The engine decouples an application and component-specific code and functionality from the general administrative tasks such as maintaining the list of PROCs and their connections, migrating PROCs, establishing and maintaining connections between hosts, and communicating between hosts. A DACIA distributed application (Figure 3.3) uses an engine on every host it runs on. We chose to use an engine per application per host, as opposed to sharing an engine running on a host between multiple applications, in order to minimize the cost of communication between PROCs and between PROCs and the engine. The engine and the PROCs run within the same address space, therefore the (synchronous) local communication translates into simple procedure calls.

The novelty of our approach lies in the flexibility of building an application and the ability to change the application structure at runtime. The same application can be built in multiple ways, either by configuring differently the same set of components or by using different sets of components.

A distributed application is created by first connecting engines running on multiple hosts, followed by connecting PROCs running on various hosts. An engine has only partial (and sometimes inconsistent) knowledge about PROCs running on other hosts and the global configuration of the application. It can accept incoming connections from remote engines. It can connect to another engine using the call:

- *connect(hostName, IPPortNo, update)* - connects to an engine running on host *host-*

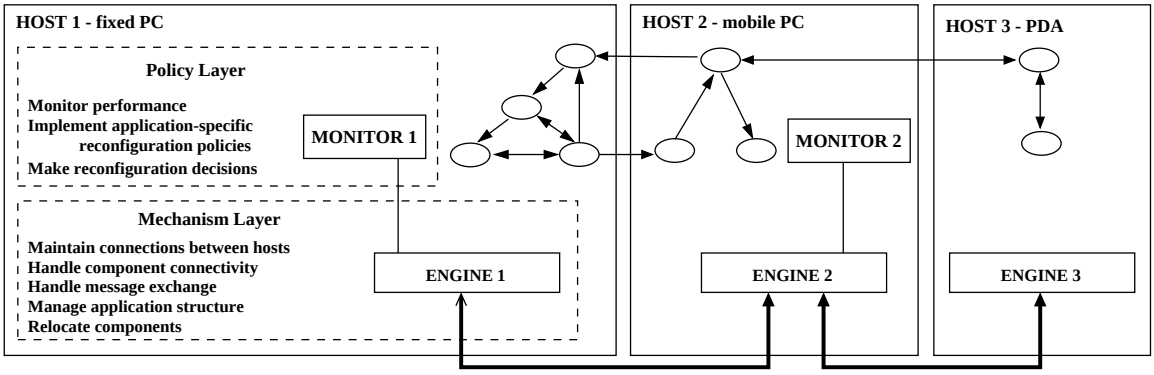

Figure 3.3: A DACIA distributed application is a directed graph of connected components (ovals represent components). An engine runs on every host. It manages the local components and the connections between components, both local and across different hosts. The monitor gathers performance data and implements application-specific relocation and reconfiguration policies.

Name, listening on port *IPPortNo*. If *update* is true, upon connecting, the engines exchange information about each other's local views of the PROCs in the system. This information can refer only to PROCs local to the remote engine, or to all the PROCs known to the engine.

The engine maps virtual connections between PROCs to either local or remote physical connections, and handles data transfers accordingly. Multiple virtual remote connections between pairs of PROCs are multiplexed over a single network connection between two engines. The connectivity between remote PROCs is maintained as long as the corresponding engines are connected. Sharing physical connections reduces the cost of establishing network connections in a highly dynamic application, where PROCs often connect to each other or they are disconnected. The frequency of connecting and disconnecting PROCs is much higher than the one with which engines connect to each other or network connections are broken. The network connection between two engines is used both for exchanging messages between pairs of PROCs situated on the two hosts, respectively, and for exchanging control messages between engines.

Similarly to the paths in Scout [74], the way DACIA components are connected dictates the flow of data in the system. Although the graph structure of a DACIA application is similar to the router graph in Scout, there are significant differences in terms of goals and underlying mechanisms. Scout is a communication-oriented operating system. A router

graph resides within a single address space. Our system is used to implement distributed applications that span over multiple hosts, and where components are often loosely coupled. A router graph is defined at application build time, while in our case an application graph can change at runtime.

3.3 Monitors

The engine of an adaptive application works in conjunction with a *monitor.* In DACIA, we separate administrative tasks such as maintaining the connections between components and migrating components (done by the engine) from the process of interpreting the semantics of an application and making reconfiguration decisions (done by the monitor). While the engine provides the *mechanisms* used to build and reconfigure applications, the monitor implements the *policy* layer in a DACIA application. The engine is not aware about the semantics of the application. The monitor *monitors the application performance, makes reconfiguration decisions*, and instructs the engine accordingly. The engine is responsible for establishing and removing connections between components and for moving components to other hosts.

Our approach of separating the mechanisms used to reconfigure an application, provided by the engine, from the reconfiguration policies, implemented by the monitor, has some similarities with the solution proposed by FarGo [43]. Dynamic application layout in Fargo separates the programming of the layout of the application from the application logic. The changes of an application layout consist of finding the right place to execute components and migrating components at runtime. We go further, allowing an application to dynamically change the connections between components, to introduce new components, and to change its structure.

The monitor is not the only entity that can do performance monitoring. The PROCs and the engine may also collect some performance data, which is interpreted by the monitor. For instance, the engine performs general-purpose measurements such as the latency of communication between hosts, the bandwidth available between them, or the amount of data exchanged between two PROCs. PROCs handle some application-specific metrics. Specialized API functions are used to communicate to the monitor the values of these latter measurements.

The engine and the PROCs are general-purpose and they can be reused to build multiple applications. The monitor is usually specific to the application and it incorporates relocation and reconfiguration policies applicable only to a particular application. A distributed application can use simultaneously multiple monitors, implementing different adaptations, sometimes based only on information locally available.

The existence of a monitor is not necessary on every host an application runs on. In some cases, for example in the case of a DACIA application running on a PDA (HOST 3 in Figure 3.3), it is desirable that the computation is as lightweight as possible. In this situation, a monitor running on a different host can make reconfiguration decisions for this computation. Through the interface exposed by the engine and the communication between engines, the monitor can find information about applications running on remote hosts and issue commands for reconfiguring remote applications. Alternatively, there may be no automated monitoring, and the reconfiguration can be done manually by a system administrator or a user, using a command-line interface[4].

3.4 Building and Executing DACIA Applications

Using DACIA, the application development and maintenance can be split among several categories of programmers and users. There is a clear distinction between the programming of components and the programming of applications. On one hand, component programmers write PROCs that have to respect particular component specifications and to implement some required interfaces. On the other hand, application programmers develop applications by joining components that may have been written by different programmers. The functionality of a particular application is achieved by interpreting the semantics of individual components, as well as the end-to-end semantics of groups of interacting components. Application programmers also write customized monitors, which implement application-specific adaptive policies and issue commands for runtime reconfiguration of the application.

An application developer constructs the initial configuration of an application by selecting the necessary components from a component repository and connecting them in an appropriate manner, in order to achieve the desired functionality. The physical distribution of components in an application can be specified completely orthogonally to the logical

[4]this interface is presented in Section 6.1.3

structure of the application. Once the initial configuration has been constructed, the application can be executed. At runtime, new components can be added to the application, existing components can be moved across hosts, and connections between components can be added or removed. These actions can be initiated either automatically by the application, or through an application system administrator's explicit intervention.

Dynamic reconfiguration is primarily achieved through the execution of adaptive functions implemented by monitors. It is not necessary that every DACIA application uses a monitor though. Application developers are responsible for writing application-specific monitors. An application is initially written according to some well specified needs and corresponding to a target execution environment. Nevertheless, during the lifetime of the application, the execution environment may change. Consequently, the application needs to evolve to adapt to the new conditions. At some point, a system administrator can decide that the original adaptive algorithms need to be changed or new algorithms are necessary. New monitors can be developed and tested off-line. Then they are loaded dynamically, thus changing the application.

The execution of adaptive monitors is complemented by the intervention of system administrators and, in some cases, even regular users of an application. Using the command-line interface provided, they can manually issue commands to reconfigure the application. Monitors are usually responsible for adapting to anticipated changes that happen on a regular basis. Users and administrators intervene to perform isolated changes, or to change the application so that it better suits their evolving needs.

The specification and configuration of an application can be done using the same programming language used for coding the components, or a different language. The potential use of different languages for these two programming levels reflects the separation of concerns in programming an application. Our current implementation of DACIA employs the first approach. We use a simple programming API to "glue" together components and to write monitoring routines. Additionally, the command-line interface of DACIA provides a set of primitives for manually reconfiguring an application.

Several formalisms or languages have been proposed by other researchers (e.g., Conic [68], Darwin [67], PCL [95], Lua [49]) for specifying and configuring component-based distributed applications. Currently, DACIA does not use a formalism for component and application specification. One of the potential directions for future research for the DA-

CIA project concerns the use of a language for specifying application configurations and for performing dynamic reconfiguration.

A configuration language supports a small set of commands that perform component instantiation and deallocation, connection establishment and removal, and stops and resumes the execution of a component. DACIA offers this functionality through the programming API and the command-line interface, which can be used to implement adaptive monitoring functions and to perform manual reconfiguration, respectively. The programming API can be accessed from Java programs.

3.5 Structuring Distributed Applications

Our preliminary experience with using DACIA to build distributed applications indicates that certain types of PROCs are likely to be useful. The PROCs used to develop interactive applications (Figure 3.4) can be classified according to their characteristics and functionality as follows:

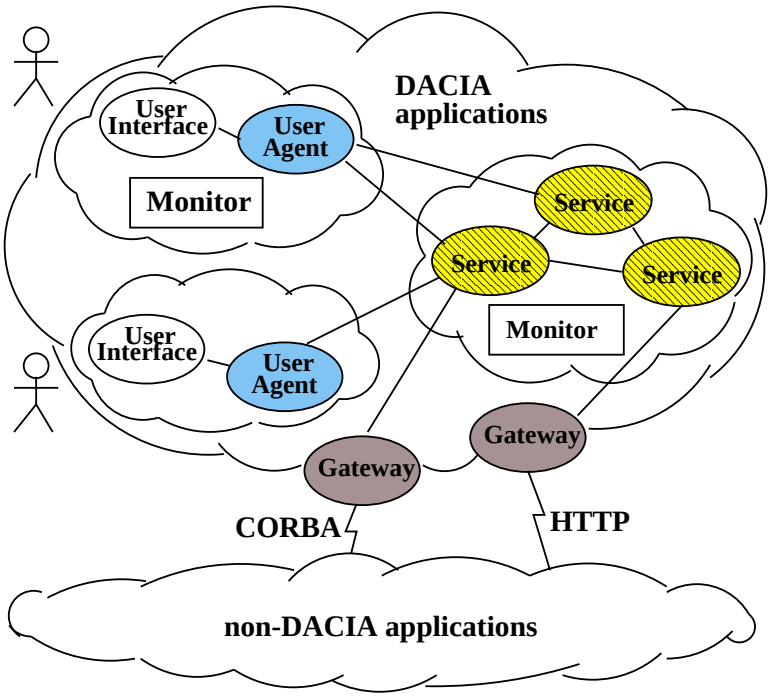

Figure 3.4: A DACIA collaborative application may contain PROCs of various types: User Interface, User Agent, Service, and Gateway. Via gateways, DACIA applications can interact with non-DACIA components.

- *User Interface PROCs* represent interfaces between human users and applications.
- *User Agent PROCs* are persistent representations of users of a distributed application. They represent the non-interface part of a client-side application. The important state of the client should be part of the User Agent.
- *Server/Service PROCs* can be used to implement various services, such as data processing and distribution, caching and storage, client and group management, etc. A particular service can be implemented in multiple ways, using different sets of components, connected in various configurations.
- *Gateway PROCs* enable a DACIA application to interact with the external world. A Gateway PROC implements both the communication protocols used by DACIA and other protocols used to communicate with other systems. Each Gateway PROC maps messages between an external protocol, e.g. HTTP, and PROC-to-PROC messages.

A client-side application consists of User Interface and User Agent PROCs. If desired, they can be combined into a single PROC. An application can be potentially enhanced by adding sensors that detect a user's presence next to a host, as in [37]. Based on sensor data, the User Agent can be moved by a monitor to the new host and an Interface can be instantiated accordingly.

Separating the client code into a User Interface PROC and a User Agent PROC is useful if the client is expected to run with different interfaces on various devices. It simplifies development – similar to the separation of Models and Views in the Model-View-Controller development paradigm [29]. The separation also simplifies client parking (Section 4.6). While a user is disconnected, her corresponding agent can still interact with communicating parties on behalf of the user. The User Interface is not needed in such a case. After the user re-joins the application, potentially from a different host, she locates and reconnects to her agent using its unique PROC identifier.

Services can be built through either the horizontal or vertical composition of various Service PROCs. Through *horizontal composition*, PROCs or groups of PROCs can be replicated and distributed over multiple hosts. Thus a service can scale up to support a large number of clients. In the case of *vertical composition*, PROCs of multiple types are composed to build a service. These PROCs may provide services to each other and may

depend on each other.

Different parts of a large-scale distributed application may fall under different administrative domains. They employ different coordination policies, resource management routines, and reconfiguration algorithms. Multiple application-specific monitors can be used to provide distributed coordination for various parts of a groupware system. For instance, one monitor can manage the interactions between PROCs implementing a service, while another monitor controls a client-side application.

3.6 Consistency of Application Structure

An important characteristic of DACIA applications is the relaxed consistency model with regard to multiple hosts' views of an application's structure. At any time, an engine has accurate information about the local PROCs and their connections. The engine has only partial (and sometimes inconsistent) knowledge about PROCs running on other hosts and the global configuration of a distributed application. We chose to tolerate partially inconsistent views because for a large scale distributed application it would have been very inefficient to propagate all configuration changes to all hosts in the system. Moreover, complete information about parts of a distributed application running on some remote hosts is usually not necessary. When this information is needed (e.g., at a host where a monitor executes, in order to make a reconfiguration decision), an engine can query other engines about their local views using the call:

- *update(hostName, allProcs)* - updates the information about PROCs known by other engines and their connections. *hostName* specifies the engine to request the information from. If *hostName* is null, a request is sent to all engines connected to the local engine. If *allProcs* is true, the update contains information regarding all PROCs known to the remote engine. Otherwise, the update refers only to PROCs local to the remote engine.

The PROC information received from a remote engine in response to an update request may conflict with the information the requesting engine already has. The change sequence numbers (*changeSeqNo*) of each PROC allows one to distinguish between newer an older PROC information. The *changeSeqNo* of a PROC is used to keep track of the order of applying changes to the PROC. It is incremented every time a change is applied to the PROC (connect, disconnect, or move). It is also incremented when an operation failure

notification is received by the engine where the PROC resides. The *changeSeqNo* is always incremented only on the host where the PROC is located. It is passed together with the PROC information when an update is requested.

If an engine E1 obtains from another engine E2 conflicting information about a PROC, then the information with higher change sequence number for the PROC is considered more accurate. If E1 has PROC information with higher or equal sequence number, E1 retains that information. Otherwise, E2's version is accepted. The *changeSeqNo* of a PROC is always incremented on the engine where the PROC resides, and it is propagated to other engines together with PROC state information. Assuming no byzantine failures, this implies that the information about a PROC at two different sites is the same if they have the same *changeSeqNo*.

The information an engine has about PROCs located on that engine is always either the same or more accurate than the information a remote engine has about these PROCs. Due to the asynchronous nature of communication in distributed systems, it is possible that stale, inaccurate information about a PROC is received from the host where the PROC resides. The use of change sequence numbers eliminates the possibility of old updates being applied over more recent PROC information.

When a connection is established between two engines, the engines exchange information about each other's local views of the PROCs in the system. In this way, an engine can obtain information about PROCs running on remote hosts with which it is not connected. When a configuration change occurs (e.g., component creation, removal, connection, disconnection, or move), the change will be propagated only to the hosts (engines) that are *interested in the change*, i.e., they have PROCs that are connected to the PROCs involved in the change. Other hosts may have stale information. The programming API allows a monitor to explicitly propagate change information to all hosts connected to the host where the change originated. In most cases this is not necessary. It can also create a scalability problem in the case of large systems in which changes occur frequently and need to be propagated to many hosts, thus it is usually avoided.

Configuration changes are executed in an optimistic fashion, i.e., a change is first applied locally, then the information about the change is propagated to other hosts. Exception to this rule is the case when all components (one or two) involved in the change are remote to the host where the change was initiated. In such a case, a request to execute the change is

sent to the host (or one of the hosts) where the component(s) are located.

```
connectProcs(Host requester, Proc A, Port a, Proc B, Port b) {
  if (A is local) {
    if (B is local)
      connectLocal2LocalProcs(requester, A, a, B, b)
    else
      connectLocal2RemoteProcs(requester, A, a, B, b)
  }
  else {  // A is remote
    if (B is local)
      // similar with the case A local, B remote
      connectLocal2RemoteProcs(requester, B, b, A, a)
    else  // both A and B are remote
      forward the operation to host of A
  }
}

connectLocal2LocalProcs(Host requester, Proc A, Port a, Proc B, Port b) {
  if ((port a of A is not connected) && (port b of B is not connected)) {
    execute the connection locally
    increment A.changeSeqNo and B.changeSeqNo
    if (requester != localhost)
      send updated information about A and B to requester
  }
  else {
    if (requester != localhost)
      send error message to requester, together with updated PROC information
    return error
  }
}
```

Figure 3.5: The algorithms for connecting two PROCs (either local or remote) and for connecting two local PROCs

Figures 3.5 and 3.6 present the algorithm used for connecting two PROCs. If the PROCs are local, the operation is applied locally and the change sequence numbers for both PROCs are incremented. The updated information about the PROCs is propagated to the host requesting the operation, if it is not the local host. If one PROC is local and one is remote (Figure 3.6), if the operation can be executed, it is executed locally (partial connect – A's connection information is updated, but B's information is not), and the *changeSeqNo* of the local PROC is incremented. If the requester is the host where the remote PROC is located, the updated information about the local PROC is propagated to the requester (the connect operation has already been partially executed at that host). If the requester is a different host, the operation is propagated to the remote PROC's host. If the operation cannot be

executed due to the local PROC being already connected, the error is propagated to the requester. If the operation cannot be executed due to the remote PROC being connected, the local information about the remote PROC is verified against the information in the request for accuracy, and the request may or may not be executed.

```
connectLocal2RemoteProcs(Host requester, Proc A, Port a, Proc B, Port b) {
  if (info about B in the request has greater changeSeqNo than local info)
    update local information about B
  if (port a of A is not connected) {
    if (port b of B is not connected) {
      execute connection of A to B locally  // partial connect
      increment A.changeSeqNo
      if (requester != local host) {
        send updated information about A to requester
      if (requester != host of B)
        forward operation to host of B, including updated information about A
    }
    else {  // port b of B is connected
      if (port b of B is connected to port a of A, but port a of A is available) {
        // operation has already been executed at B's host
        execute connection of A to B locally
        increment A.changeSeqNo
        send updated information about A to host of B
        if (requester != local host) {
          send updated information about A to requester
        }
      }
      else {  // B is already connected -> operation fails
        if (requester != local host)
          send error message to requester
        return error
      }
    }
  }
  else  {  // port a of A is connected -> operation fails
    if (requester != local host)
      send error message to requester
    return error
  }
}
```

Figure 3.6: The algorithms for connecting two PROCs, one local (A) and one remote (B)

When an engine receives an error message about the failure of a PROC connect operation, it disconnects the PROCs locally and increments their *changeSeqNo* if the PROCs are local. When an engine receives an update about a remote PROC, it replaces the currently held information with the updated information, providing that the update does not contain

stale information (the *changeSeqNo* of the PROC in the update is higher than the local *changeSeqNo* of that PROC).

Figure 3.7 presents the algorithm used for disconnecting two PROCs. The updated information about the PROCs and the operation failure notifications are propagated and handled in a similar way with the case of the *connectProcs* operation.

We adopted this optimistic change execution solution despite the fact that it can lead to conflicting configuration changes. Our experience with implementing DACIA applications shows that the cases of concurrent conflicting changes are rare. Therefore, we chose to execute changes as early as possible, and to deal with conflicts when they occur.

The alternative would be that every time a configuration change is initiated, the change information is sent to the engines where the components involved in the change are located. These engines verify whether the operation can be executed, then they notify the initiator of the operation. The change can be committed only after all notifications are received. For example, consider the case in which the engine running on host H1 initiates a connection between PROC A, located on H1, and PROC B, located on H2. The connection information is sent to the engine running on H2, which verifies that B's port is available, then sends an acknowledgment to H1. At the receiving of the notification, H1's engine can commit the operation. This solution can lead to higher overheads in executing configuration changes. In the optimistic approach, the changes are executed right away.

In the following, we will investigate several cases of configuration changes, potential consistency problems that they may cause, and solutions to address these problems. Problems occur especially in situations when two configuration changes are applied concurrently at two different locations.

- **Component creation**

  The lack of information about a newly created component does not affect the execution of previously existing components and their interactions.

- **Component removal**

  Consider the case in which a PROC A, residing on host H1, is removed. At the same time, host H2 initiates an operation for connecting a PROC B, located on H2, to A. The connect operation is first executed at H2, and then propagated to H1. At H1, this operation fails, since A no longer exists. A failure notification is sent to H2. The

```
disconnectProcs(Host requester, Proc A, Port a, Proc B, Port b) {
  if (A is local) {
    if (B is local) {
      if (the PROCs are connected) {
        disconnect the PROCs locally
        increment A.changeSeqNo and B.changeSeqNo
        if (requester != localhost)
          send updated information about A and B to requester
      }
      else {
        if (requester != localhost)
          send error message to requester, together with updated PROC information
        return error
      }
    }
    else {  // A local, B remote
      if (info about B in the request has greater changeSeqNo than local info)
        update local information about B
      if (the PROCs are connected in the local view) {
        disconnect the PROCs locally
        increment A.changeSeqNo
        if (requester != local host)
          send updated information about A to requester
        if (requester != host of B)
          forward operation to host of B, including updated information about A
      }
      else {  // the PROCs are not connected
        send error message to requester if it is not host of A or host of B
        return error
      }
    }
  }
  else {  // A is remote
    if (B is local)
      // similar with the case A local, B remote
      execute disconnectProcs(B, b, A, a)
    else  // both A and B are remote
      forward the operation to host of A
  }
}
```

Figure 3.7: The algorithm for disconnecting two PROCs

removal operation is first executed at H1, then propagated to H2. At H2, the removal automatically results in removing the connection between A and B. If a message is sent from B to A before the conflict resolution at H2, a connection failure is reported, similar to the case when a message is sent on an unconnected port.

- **Connection establishment**

  A conflict can occur when two connect operations, involving the same PROC on the same port, are initiated at two different locations. Consider the following situation: PROCs A and B are located on H1, PROC C is located on H2. The following operations are executed:

  1. H1 initiates operation: connectLocal2LocalProcs(A, 1, B, 1). The change is not propagated to other hosts.
  2. H2 initiates operation: connectRemote2LocalProcs(A, 1, C, 0). The change is propagated to H1.
  3. The operation connectProcs(A, 1, C, 0) is received at H1. Since port 1 of A is already connected, the operation fails. A failure notification is sent to H2.
  4. The failure notification is received at H2. The connection between A and C is removed, and A is connected to B at H2.

  An interesting situation occurs if the scenario above is modified so that B is located on H2, and the change notifications are received both at H1 and H2 after the respective connect operations have been applied locally. The following operations are executed:

  1. H1 initiates operation: connectLocal2RemoteProcs(A, 1, B, 1). The change is propagated to H2.
  2. H2 initiates operation: connectRemote2LocalProcs(A, 1, C, 0). The change is propagated to H1.
  3. The operation connectProcs(A, 1, C, 0) is received at H1. Since port 1 of A is already connected, the operation fails. A failure notification is sent to H2.
  4. The operation connectProcs(A, 1, B, 1) is received at H2. This conflicts with the fact that A is connected to C. Since the problem is due to A, located on H1, and

the request to connect A to B, which originates at H1, has higher *changeSeqNo* for A, this request prevails. The connection between A and C is removed, and A is connected to B at H2.

In general, a host rejects any connection request that originates at a remote host and involves a local PROC that is already connected. In case of a conflict, the host where the PROC causing the conflict resides makes the ultimate decision and enforces its change operation.

If multiple connect requests are issued at various hosts, so that they depend circularly on one another, it is possible that all requests fail. Consider the case in which PROC A is located on H1, PROC B is located on H2, and PROC C is located on H3. The following operations are executed:

1. H1 initiates operation: connectLocal2RemoteProcs(A, 1, B, 0). The change is executed locally and propagated to H2.
2. H2 initiates operation: connectLocal2RemoteProcs(B, 0, C, 1). The change is executed locally and propagated to H3.
3. H3 initiates operation: connectLocal2RemoteProcs(C, 1, A, 1). The change is executed locally and propagated to H1.
4. The operation connectProcs(A, 1, B, 0) is received at H2. Since port 0 of B is already connected, the operation fails. A failure notification is sent to H1.
5. The operation connectProcs(B, 0, C, 1) is received at H3. Since port 1 of C is already connected, the operation fails. A failure notification is sent to H2. At the receiving of this notification, H2 disconnects B from C.
6. The operation connectProcs(C, 1, A, 1) is received at H1. Since port 1 of A is already connected, the operation fails. A failure notification is sent to H3. At the receiving of this notification, H3 disconnects C from A.
7. The failure notification from H2 is received at H1. H1 disconnects A from B.

- **Disconnection**

In most cases, the lack of updates about two components being disconnected does not affect the execution of the application at remote sites. In the worst case, a

connectProcs() operation can fail due to two PROCs being reported connected, when in fact they have already disconnected. In such a case, if the connect request comes from the site where the PROC causing the failure resides, it also contains up to date information about that PROC having been disconnected. The new PROC information overwrites the previous information. The old connection is removed and the new connection is established. Consider the following scenario: PROCs A and B are located on H1, PROC C is located on H2. Port 1 of A is connected to port 0 of B. The following operations are executed:

1. H1 initiates operation: disconnectProcs(A, 1, B, 0). The operation is executed at H1. The change is not propagated to other hosts, since both PROCs involved in the operation (A and B) are local to H1.
2. H1 initiates operation: connectLocal2RemoteProcs(A, 1, C, 0). The operation is executed at H1. The change is propagated to H2.
3. The operation connectProcs(A, 1, C, 0) is received at H2 from H1. This conflicts with the fact that A is connected to B at H2. If the new connect operation, received from the host where A, which causes the conflict, resides, has a higher *changeSeqNo* for A, then it overwrites the previous information about A being connected to B. A and B are disconnected, and A is connected to C at H2.

If the connect request comes from a host that is remote to the PROC in discussion, the request is considered invalid and it fails. Under normal circumstances, the latter case is avoided, since connection updates are usually propagated from one (or both) of the hosts where the PROCs involved in the connection are located.

There are cases in which a valid connect operation fails due to disconnect operations that were not propagated. In the previous example, if in Step 2 the operation connectProcs(A, 1, C, 0) is initiated at H2, the operation fails, since port 1 of A has not been disconnected at H2.

- **Component move**

The lack of updates or late updates about a component move does not affect concurrent connect or disconnect operations. The operations are commutative at the two

hosts where they originated. If an update is received at the old location of a component after a component move, the update will be propagated to the new component location. Chapter 4 provides more details on how component moves are handled.

Our optimistic approach to propagating configuration changes may cause some configuration operations to erroneously fail, e.g., the last examples presented above in the cases of PROC connect and disconnect operations. One way to address these potential shortcomings is that each engine periodically sends up to date information about the local PROCs to all other engines it is connected to. While this does not completely eliminate the possibility of errors, it reduces the likelihood that situations like the ones mentioned above occur.

CHAPTER 4

COMPONENT MOBILITY

One of the key features of our architecture is the ability to move components between hosts. The benefits of mobility are twofold. On one hand, the execution of an application can be made more efficient by dynamically changing the location where various parts of the application are executed. Thus the application can adapt to runtime changes in resource availability, application load, and patterns of interaction between components.

On the other hand, component mobility provides good support for mobile users and mobile applications. Mobile users connect from various points, using a variety of devices, having a wide range of connectivity, processing and display capabilities. Using DACIA, mobile users can move applications or parts of applications from one computing device to another, while maintaining seamless communication connectivity with other applications. At their new location, the applications continue their execution from where they left off. The users do not see any interruptions in the services accessed, and they do not need to manually re-establish connections with the communication parties. At the same time, the execution of components connected to the moving component is not affected. If so desired, a component's move can be made transparent to other connected components.

Through mobility, users can also share their previously private work with others, for instance by moving a GUI component from their personal desktop to a large touch-screen display, where several other users can access it.

Our work on component mobility addresses the problem of capturing the state of a component and restoring it at the destination. It also deals with the inherent unreliability of network connections and variations in connection quality characteristic to mobile envi-

ronments. The DACIA infrastructure hides transient network and communication failures from applications. Using a combination of techniques such as data pre-fetching, message buffering and retransmission, applications and implicitly their users can be given the illusion of a persistent end-to-end logical connection, even over an unreliable network connection. If disconnections persist, the communication middleware notifies the applications.

For PROCs to be reachable after they relocate, mechanisms for locating them are needed. Ideally, a location service should be present and it should be able to provide at any moment correct information about a PROC's location. In our current implementation, an engine finds out about a PROC's location either directly, by receiving notifications from the engine running on the same host with the PROC, or by querying other engines. We maintain a weak consistency of each engine's view of components' locations. When a PROC moves, the engine where the PROC was previously located sends notifications about the change only to the engines hosting PROCs connected to the moving PROC. Our system ensures that messages are delivered reliably during the period when hosts have inconsistent information about a PROC's location.

When a component moves, an additional problem is the access to resources previously used by the component (e.g., a local file or a database). There are two types of components in the framework: *resource components* and *mobile components*. Resource components are usually fixed because they directly read and write a local resource. Mobile components access resources only via resource components. Currently, DACIA does not allow resource components to move. In the future, we intend to explore the possibility of supporting *resource proxies*, that in conjunction with techniques such as *replication* and *caching* will allow resources to be accessed in disconnected mode, and will make the location of resources transparent to the components accessing them.

4.1 Moving Component Code and Data

DACIA provides mobility at the component (object) level. One of our concerns in implementing component mobility has been to reduce the overheads of component movement, thus the amount of data that has to be moved, and the time it takes to capture this data and to restore it at the destination. We do not transfer the execution state of a PROC (e.g., program counter, stack and registers content, thread status). However, a moving PROC

carries with it the state of its data members, the messages received and not handled yet, and the state of its connections.

DACIA uses Java serialization to move components across hosts. To reduce the cost of mobility, whenever possible, DACIA transfers only the data corresponding to a component, but not its code. If a PROC is moved to a host where an implementation of the PROC's class exists, only the data is serialized. A new instance of the component is created at the destination, and the serialized state is loaded into this instance. If the PROC's class implementation is not present at the destination, the whole component (code and data) is transferred. First the class code is moved using DACIA's dynamic loading capabilities. Then the instance of the component is transferred.

By default, the state capture in DACIA is implicit (through Java serialization). As a performance optimization, the programmer has the ability to explicitly capture the state of a moving PROC, by writing customized serialization routines. DACIA provides a pair of *pack()* and *unpack()* primitives for handling the state capture and restore for the base *Proc* class. A component writer needs to overload these methods to handle the specific state corresponding to a particular component. If the component spawned multiple threads that participate to its execution, these methods should also contain code for graciously stopping these threads and creating new threads at the destination.

Existing mobile code systems offer two forms of code mobility [28]. *Strong mobility* allows migration of both the code and the execution state to a different computing environment. *Weak mobility* allows only the transfer of code across different computing environments. We adopted the middle way between these approaches. A moving PROC does not carry its execution state in the form of stack content and thread status. However, aside from its data members, the soft state of a PROC that is transferred includes the status of its interactions with other PROCs, represented by its connections, and the messages received and not handled yet.

Similar to TACOMA [50] agents, PROC migration happens at well-defined times with respect to the execution of the PROC. Before a PROC moves, if a message is currently being handled, the handling routine completes. When a PROC moves to another host, all messages left in its message queue move with the PROC. A locking mechanism prevents an incoming synchronous call from being executed right away. Instead, the corresponding message will be sent to the new location of the PROC and it will be handled asynchronously.

If a message is received at the old location of a PROC after the PROC has moved, the engine forwards the message to the new location.

4.2 Moving Algorithm

One of the major challenges in implementing PROC mobility in DACIA has been to make the move transparent to communicating PROCs, allow other PROCs to send messages to the moving PROC while the move is ongoing, and ensure that these messages are delivered reliably to their destination. The FIFO order of message delivery and processing at the destination has to be preserved.

Initially, we considered creating a replica of the moving PROC at the destination, and running the two replicas in parallel for a while, until all connected PROCs are notified about the new location. Each replica handles the messages that it receives. Thus, no messages are lost. This solution could lead to an incorrect execution. The state of the moving PROC can change as a result of the old PROC replica handling a message, after the new replica is created and its state is initialized. The new PROC replica will not receive this state change. Moreover, the order of message handling by the two replicas may not be correct.

We decided to allow at any time only one version of the moving PROC, and to lock this PROC while it is moving, thus preventing other PROCs from sending messages to it. The message send operation will thus block while the PROC is moving. At the same time, a PROC move operation can not be executed while a PROC is receiving a message from another PROC.

Figure 4.1 presents the pseudo-code for the *output()* method executed by the PROC sending a message to another PROC. Figure 4.2 presents the pseudo-code for the *move()* method executed by the source engine for moving a PROC. Figure 4.3 presents the pseudo-code for the *receiveProc()* method executed by the engine where a PROC moves. Figure 4.4 presents the pseudo-code for the *receiveMoveNotification()* method executed by an engine that receives a notification about a PROC move.

In Figure 4.1, the receiver PROC has a *location* field which is set to *HERE* if the PROC resides on this host and to *AWAY* if it has already moved. Before sending a message, the sender PROC first acquires a lock on the receiver PROC (line A2). Then it tests whether the receiver has moved since the sender acquired a reference to this PROC (line A3). If

the *location* of the receiver PROC is set to HERE, the message send operation completes normally (line 8). Otherwise, the sender retrieves the *remoteProc* reference corresponding to the receiver (line A4), then sends the message to the new instance of the receiving PROC, at its new location (line A5).

(thread T1)

```
A1 : output(proc, msg) {
A2 :   synchronized(proc) {
A3 :     if(proc.location == AWAY) {  // the PROC has already moved
A4 :       new_proc = remoteProcs.get(proc.procID)
A5 :       remote_output(new_proc, msg)
A6 :     }
A7 :     else  // the PROC is local
A8 :       proc.input(msg)
A9 :   }
A10: }
```

Figure 4.1: The *output()* method executed by a PROC sending a message. *proc* represents the PROC receiving the message (the moving PROC). The message is delivered either to the local instance of the receiver PROC, if the receiver still resides on this host, or to the remote location, if the receiver has moved to another host.

The *move()* operation (Figure 4.2) starts by acquiring a lock on the moving PROC (line B2), in order to prevent the PROC from receiving a message while it is moving. Then it adds a reference to the moving PROC to the *remoteProcs* hashtable, and removes the reference to the local PROC from the *localProcs* hashtable. After the state of the moving PROC is sent to the remote host (line B6), the *location* field of the moved PROC is set to *AWAY* to notify potential senders that hold a local reference to the moving PROC that it has moved.

In an application that functions correctly, it is not possible that two distinct move requests for the same PROC are simultaneously issued. Therefore, two threads can not concurrently execute the *move()* method. If two concurrent move requests were allowed, after entering the critical section, the move procedure should test the *location* field to ensure that the PROC is still located on that host.

When a PROC is received at the remote destination host (Figure 4.3), a reference to the local PROC object is added to the *localProcs* hashtable, and the reference to the remote PROC object is removed from the *remoteProcs* hashtable. The location of the local PROC

(thread T2)

```
B1 : move(proc, host) {
B2 :   synchronized(proc) {
B3 :     // execute PROC move
B4 :     remoteProcs.add(new RemoteProc(proc, host))
B5 :     localProcs.remove(proc)
B6 :     sendProc(proc, host)
B7 :     proc.location = AWAY
B8 :   }
B9 : }
```

Figure 4.2: The *move()* method executed by a moving PROC. After the move completes, the *location* field is set to AWAY to notify potential senders that the PROC has moved. The PROC move and the message send (the output() method) to the moving PROC are executed in mutual exclusion.

is initialized to HERE. The operations on the two hashtables are protected by a lock in order not to interfere with the ongoing message send operations addressed to the moving PROC, or with potential subsequent move operations that are initiated before the data structures corresponding to the moving PROC are updated.

```
C1 : receiveProc(state) {
C2 :   proc = new Proc(state)
C3 :   proc.location = HERE
C4 :   synchronized(proc) {
C5 :     localProcs.add(proc)
C6 :     remoteProcs.remove(proc.procID)
C7 :   }
C8 :   notifyOtherHosts(proc.ID, thisHostName, ++proc.changeSeqNo)
C9 : }
```

Figure 4.3: The *receiveProc()* method executed by the engine receiving a PROC from a remote location. The data structures corresponding to the moving PROC are updated. Then other remote engines are notified about the new PROC location.

The following scenario gives an example of a problem that may appear in the absence of locks: the *receiveProc()* method executes up to and including line C5. At this point a reference to the local PROC exists in the corresponding hashtable. The thread executing the *receiveProc()* method is de-scheduled. Another thread executes a *move()* request. When line B4 is executed, the new remote PROC will replace the existing remote PROC with the same ID in the hashtable. Then line B5 is executed, followed by line C6. At this point, no

reference to the moving PROC exists either in the local or the remote PROC table. The use of the lock in line C4 prevents this from happening.

The destination engine sends notifications about the PROC move to other engines that are interested in this PROC[1] (line C8). A notification contains the ID of the moving PROC, the new location of the PROC, and the change sequence number that keeps track of the order of changes applied to the PROC (a change can be a move, connect or disconnect). The sequence number is used to prevent old PROC move notifications received out of order from being applied. Consider the following scenario: A PROC moves from host H1 to host H2, then it immediately moves to host H3. Another host H4 will receive move notifications both from H2 and from H3. Due to the communication asynchrony encountered in distributed systems, these notifications may be received out of order. The use of the *changeSeqNo* for a PROC prevents the notification from H2 from being applied at H4 after the notification from H3.

When an engine receives a PROC move notification (Figure 4.4), the notification is applied only if a more recent notification has not been previously received (the test in line D5). The updates to the PROC *hostName* and *changeSeqNo* are atomic, being protected by a lock. This prevents multiple updates that may satisfy the test in line D5 from being concurrently applied.

```
D1 : receiveMoveNotification(procID, hostName, changeSeqNo) {
D2 :   proc = remoteProcs.get(procID)
D3 :   if(proc != null)
D4 :     synchronized(proc)
D5 :       if(proc.changeSeqNo < changeSeqNo) {
D6 :         proc.changeSeqNo = changeSeqNo
D7 :         proc.hostName = hostName
D8 :       }
D9 : }
```

Figure 4.4: The *receiveMoveNotification()* method executed by an engine receiving a PROC move notification. The notification is applied only if a more recent notification has not been previously received.

The part of the moving algorithm presented in Figures 4.1 and 4.2 allows messages to be potentially received out of order at the destination. When a PROC moves from one host

[1] they have PROCs connected to the moving PROC

to another, messages in traffic may either be delivered directly to the new PROC location, or they can be first sent to the old location, and then forwarded from there to the new location. Consider the following scenario: PROC A, situated on host H1, is connected to PROC B, situated on host H2. A moves to host H3 while B sends a message M1 to A. Subsequently, B sends another message M2. M1 will be delivered to H1 after A has left, therefore it will be forwarded to H3. M2 is sent after H2 has received the move notification, therefore it will be delivered directly to H3. In this way, it is possible that M1 is delivered to A on host H3 after M2, thus compromising the message ordering semantics.

```
E1 : input(msg) {
E2 :   if (msg.ID == port[msg.portNo].lastMsg + 1) {
E3 :     handleMessage(msg)   // handle the message
E4 :     port[msg.portNo].lastMsg++
E5 :     // handle messages previously received out of order
E6 :     while(waitQueue.nextID(msg.portNo) == port[msg.portNo].lastMsg+1){
E7 :       // extract a message from the queue
E8 :       msg1 = waitQueue.nextMsg(msg.portNo)
E9 :       handleMessage(msg1)
E10:       port[msg.portNo].lastMsg++
E11:     }
E12:     if (waitQueue.isEmpty()) {
E13:       // stop the WaitQueueMonitor thread
E14:       wqMonitor.stop()
E15:       wqMonitor = null
E16:     }
E17:   }
E18:   else {
E19:     waitQueue.add(msg);
E20:     if (wqMonitor == null) {
E21:       // start a new thread to monitor the waiting queue
E22:       wqMonitor = new WaitQueueMonitor(waitQueue)
E23:       wqMonitor.start()
E24:     }
E25:   }
E26: }
```

Figure 4.5: To address the problem of messages being delivered out of order, the *input()* method of the receiver PROC uses message sequence numbers and a sliding window protocol.

To address this problem, the message reception (the *input()* method of the receiver PROC – Figure 4.5) uses message sequence numbers and a sliding window protocol. Con-

secutive message sequence numbers are assigned to messages sent along the same connection between two ports. When a connection between two PROCs is established, the message sequence number (*msg.ID*) is set to 1, and the last message received for the corresponding port (*port.lastMsg*) is set to 0. If a message is received out of order, it is not handled right away, but it is inserted in a waiting queue. If it is not already active, a *WaitQueueMonitor* thread is started to monitor the activity on the waiting queue.

When a message is received, the *input()* method checks the waiting queue for the eventual filling of a window (line E6), and eventually handles the messages previously received out of order. The waiting queue is sorted. All operations on the queue are synchronized. The *waitQueue.nextID(portNo)* call returns the lowest ID of a message received on the port indicated. If no such message exists in the queue, the call returns 0. The *waitQueue.nextMsg (portNo)* call returns the message with the lowest ID received on the port indicated, and removes the message from the queue. If the waiting queue becomes empty, the *WaitQueue-Monitor* thread is stopped. Thus all messages are handled in the order they were sent, even if they were received out of order at the destination.

Under normal operating conditions, in the absence of PROC moves, usually all messages exchanged using the same connection between two PROCs are handled by the same thread, therefore there are no ordering problems. Thus the *WaitQueueMonitor* thread never gets to execute.

The *WaitQueueMonitor* thread (Figure 4.6) ensures that messages in the waiting queue do not starve. Periodically, for each port for which a message exists in the queue, the thread checks whether a message has been subsequently received on that port (line F9), i.e., progress has been made towards filling the queued message's window. The *WaitQueue-Monitor* maintains a local vector containing IDs of last messages received that it is aware of (*lastMessage[]*). If a message has been received during the timeout interval, the local vector is updated (line F13). Otherwise, the corresponding connection between PROCs is considered faulty (ordered message delivery can not be achieved) and it is explicitly broken (line F11). The fault is seen by the application as the failure of the connection between PROCs.

```
F1 : run() {
F2 :   for(i = 0; i < myProc.nPorts; i++)
F3 :     lastMessage[i] = myProc.port[i].lastMsg
F4 :   while(true) {
F5 :     sleep(timeout)
F6 :     for (i = 0; i < myProc.nPorts; i++)
F7 :       if (waitQueue.nextID(i) != 0) {
F8 :         // there is a message in the queue that was received on port i
F9 :         if (myProc.port[i].lastMsg == lastMessage[i])
F10:           // no new message has been received since the previous check
F11:           myProc.disconnect(i)
F12:         else
F13:           lastMessage[i] = myProc.port[i].lastMsg
F14:       }
F15:   }
F16: }
```

Figure 4.6: The *WaitQueueMonitor* ensures that messages in the waiting queue do not starve. If such a situation occurs, the PROC connection where the starved message was received is considered failed, and the corresponding port is explicitly disconnected.

4.2.1 Correctness

The correct execution of a DACIA application during PROC moves and the guaranteed message delivery are based on the following assumptions:

1. At the beginning of their execution, both the *output()* and the *move()* procedures hold a local reference to the moving PROC. If another thread attempts to get a reference to the moving PROC while the move is ongoing, it will find it either in the *localProcs* or the *remoteProcs* hashtable.

2. A PROC cannot move from host H1 to host H2 and back to H1 while a message is being sent to the PROC on H1. In the worst case, if this happens, the message is not lost. It is first sent to H2, then it is sent back to H1 and delivered to the PROC.

3. Initially proc.location = HERE

In addition, we stated earlier that two threads can not concurrently execute the *move()* method for the same PROC, on the same engine. Only one *move()* operation and one or multiple *output()* operations can execute concurrently.

We will consider the following predicates in our subsequent reasoning:

- *P1: proc.location == HERE*
- *P2: localProcs contains a reference to the moving PROC*
- *P3: remoteProcs contains a reference to the moving PROC*
- *P4: the message is delivered to the local PROC before the move*
- *P5: the message is delivered to the remote PROC after the move*

Under the assumptions stated above, the behavior of an application with regard to PROC moves is governed by the following set of lemmas:

Lemma 1 *An engine E1 cannot send a message for a PROC to another engine E2 unless the PROC is or has been on E2.*

Proof

In Figure 4.1, if the engine E1 sends a message to a remote engine E2 (line A5), the reference to the remote PROC was obtained in line A4. Therefore P3 is true, thus the critical section in Figure 4.2 must have been executed. Therefore the PROC has moved to the remote host. When the message arrives at E2, it is possible that the PROC is still there, or it has subsequently moved to another host. □

Lemma 2 *An engine cannot receive a message for a PROC unless the PROC is or has been on that engine.*

Proof

Assume that the engine E2 receives a message from the engine E1. Therefore, at the moment of sending the message, E1 had information that the PROC resided on E2. This information could have been obtained either when the PROC moved from E1 to E2, or from a notification received from E2 when the PROC arrived at E2. □

Lemma 3 *In the absence of engine or network connection failures, assuming that a PROC ultimately stops moving, all engines interested in the location of the PROC will eventually find out the exact value of the PROC location.*

Proof

When a moving PROC is received by an engine (Figure 4.3), the engine sends move notifications to all interested engines, i.e., engines that have PROCs connected to the moving PROC (line C8). It is possible though that the PROC moves again before the notification is received. A new notification is sent about the latest PROC location. The use of sequence numbers in the move notifications ensures that old notifications received after more recent notifications are discarded. Therefore, if the PROC stops moving, the information about the exact location of the PROC will ultimately be disseminated to all interested engines. No other location updates will be applied later by these engines. □

Lemma 4 *If a message addressed to a PROC A is sent from an engine E1 to an engine E2, then either A is located on E2, or E2 has more recent information about A's location than E1.*

Proof

If E1 sends a message addressed to A to E2 (line A5 in Figure 4.1), then the latest location information that E1 has about A is that A resides on E2. This means that the most recent move notification that E1 has received is from E2. All other notifications that E1 might have received were sent prior to A being on E2 (they have lower sequence numbers). If A is no longer on E2 when the message for A is received by E2, then A has moved and the notification from the new host has not arrived yet at E1. Since A moved from E2, E2 knows the next location of A. Therefore E2 has more recent information about A's location. □

Lemma 5 *In the absence of engine or network connection failures, assuming that a PROC ultimately stops moving, a message sent to a moving PROC will be delivered to the PROC.*

Proof

In Figure 4.1, if the test in line A3 fails, then P1 is true, therefore the critical section in Figure 4.2 has not been executed. Line A8 is executed. The message is delivered to the local PROC. The predicate P4 is true.

If the test in line A3 succeeds, then P1 is false. This implies that the critical section in Figure 4.2 has already been executed, therefore P2 is false (line B5) and P3 is true (line

B4). The message is sent to the remote location (line A5). Assuming that the *output()* method succeeds there, the message is delivered to its destination.

If the PROC follows a sequence of moves from host to host, it ultimately stops moving, and the host location is correctly disseminated to the engines involved (Lemmas 1, 2 and 3), then the message will ultimately be delivered. If the message is received by a host after the PROC has moved from that host, the receiver host has more recent information about the actual PROC location than the sender (Lemma 4), therefore the message is getting closer to the destination PROC. Thus the message will eventually reach the host where the PROC resides. The predicate P5 is true. □

Lemma 6 *If multiple messages exchanged between two PROCs along the same connection are delivered, then they are delivered in FIFO order.*

Proof

Lines E2 - E3 in Figure 4.5 ensure that messages received in order are delivered in order. Messages received out of order are inserted into the waiting queue for later delivery (lines E18 - E25). Lines E6 - E11 ensure that messages previously received out of order are delivered in order, assuming that all messages with lower sequence numbers have been received.

The code in Figure 4.6 uses timeouts to detect failures. Assume that a PROC sends two messages M1 and M2 in this sequence. If M1 can not be delivered within a timeout interval started when M2 was delivered, then the connection between PROCs is considered failed (it can be due either to an engine or a network connection failure) and it is explicitly removed. □

4.2.2 Optimized Algorithm

Acquiring a lock corresponding to the destination PROC every time a message is sent (corresponding to a synchronized call for sending a message - line A2 in figure 4.1) is costly, and most of the time unnecessary, since the message exchange is much more frequent than PROC moves. Experimental data support this statement. On a particular machine (Pentium III 733 MHz CPU, 256 MB memory), a round-trip message exchange between two local PROCs took about .37 μseconds without acquiring a lock, and .78 μseconds if a lock was acquired.

We modified the algorithms in Figures 4.1 and 4.2 to avoid the overhead of acquiring locks. The sender PROC acquires a lock only if the receiver is undergoing a move. Figure 4.7 presents the pseudo-code for the *output()* method executed by the PROC sending a message. Figure 4.8 presents the pseudo-code for the *move()* method executed by the engine for the moving PROC.

The *location* field can have three values: HERE – the normal value when the PROC resides on this host, MOVING – while the PROC is undergoing a move, and AWAY – after the PROC has moved, but while other PROCs may still hold references to the local instance of the PROC. Initially *location* is set to HERE. A reference counter (*refcnt*) is used to inform the moving PROC that it is being accessed. The reference counter can be incremented and decremented by a PROC that sends a message to the moving PROC, or by the asynchronous thread, when it accesses a message from the queue.

```
A1 : output(proc, msg) {
A2 :   proc.refcnt++
A3 :   if(proc.location != HERE) {
A4 :     synchronized(proc) {
A5 :       if(proc.location == MOVING) {
A6 :         proc.refcnt--
A7 :         wait(proc)
A8 :       }
A9 :       new_proc = remoteProcs.get(proc.procID)
A10:       remote_output(new_proc, msg)
A11:   }
A12:   else {
A13:     proc.input(msg)
A14:     proc.refcnt--
A15:   }
A16: }
```

Figure 4.7: The *output()* method executed by a PROC sending a message. *proc* represents the PROC receiving the message (the moving PROC). A reference counter (*refcnt*) is used to inform the moving PROC that it is being accessed. The sender blocks if the destination PROC is in the course of moving (*proc.moving* == MOVING).

In Figure 4.7, before sending a message, a PROC increments the reference counter corresponding to the receiver PROC[2] (line A2). Then it tests whether the receiver is in the

[2]We assume that there are no concurrent updates to the reference counter, or the operations *refcnt++*, *refcnt--* are atomic. Otherwise, the accesses to *refcnt* need to be synchronized.

course of moving (line A3). If *location* is set to HERE, the message send operation completes normally, i.e., the message is delivered to the local PROC and the reference counter is decremented (lines A13 - A14). Otherwise, the PROC is either undergoing moving or it has already moved. The sender enters the critical section and tests again the *location* field. If it is set to MOVING (the move is ongoing), the sender decrements the reference counter (line A6), to allow the *move()* operation to proceed, then it blocks waiting for the move to complete (line A7). After the move completes, the sender thread can enter again the critical section, and the message is sent to the new instance of the receiving PROC, at its new location (lines A9 - A10).

The *move()* operation (Figure 4.8) starts by setting the *location* field to MOVING, in order to notify potential senders that the PROC is about to move. This is also a signal for the asynchronous thread of the moving PROC, which may be processing some messages from the queue. The asynchronous thread tests the *location* field every time it is about to start processing a new message from the queue. If *location* is not HERE, after the completion of the current message handling routine (if appropriate), it stops its execution.

```
B1 : move(proc) {
B2 :   proc.location = MOVING
B3 :   // notify the asynchronous thread
B4 :   while (proc.refcnt > 0)
B5 :     sleep(timeout)
B6 :   //execute PROC move
B7 :   remoteProcs.add(new RemoteProc(proc, host))
B8 :   localProcs.remove(proc)
B9 :   sendProc(proc, host)
B10:   synchronized(proc) {
B11:     proc.location = AWAY
B12:     notifyAll(proc)
B13:   }
B14: }
```

Figure 4.8: The *move()* method executed by a moving PROC. The *location* field is set to MOVING to notify potential senders that the PROC is about to move. The move is executed when no other PROCs are trying to send a message to this PROC, and no message handling routine is ongoing ($refcnt == 0$).

The moving routine periodically checks the reference counter, until it is zero (line B4). The *timeout* interval can be set to the same value for an entire application, or it can be

chosen based on the average expected duration of a message handling routine for a particular PROC. At one extreme, if *timeout* is set to a large value, then the test occurs less frequently. The downside in this case is that the move can be postponed for a longer time than is needed for the PROC to become ready to move. At the other extreme, if *timeout* is set to zero, the test of *refcnt* is done in a busy waiting loop.

When *refcnt* is zero, no other PROCs are trying to send a message to this PROC, and no message handling routine is ongoing. At this moment the move can be executed. After completion, *location* is set to AWAY, and all PROCs that might have been waiting are notified (line B12), so that they can resume sending messages to the moving PROC.

Most of the time, sending a message involves only checking the *location* flag in line A3 and updating the reference counter, with no additional overheads due to acquiring locks. Only if the PROC is moving, the sender of the message is blocked until the PROC move completes, and the message is sent to the new location of the PROC.

The algorithm presented above works correctly regardless of the sequence of interactions between threads executing message exchanges, message handling, or PROC moves. The relative ordering of the setting and testing of the *refcnt* and *moving* variables in lines A2, A3, A5, B2, and B4 is important. Several cases are possible:

- ```
  A2 :  proc.refcnt++
  A3 :  proc.location == HERE  -> continue output
  B2 :  proc.location = MOVING
  B4 :  while (proc.refcnt > 0)
  A13:  proc.input(msg)
  A14:  proc.refcnt--
        ...
  ```

  The sender PROC increments the reference counter, then tests the *location* variable before the moving PROC sets it to MOVING. The send operation completes. If the PROC attempts to move while the message is being sent or processed, it will block in the *refcnt* > 0 test (line 14) until *refcnt* is decremented.

- ```
  A2 :  proc.refcnt++
  B2 :  proc.location = MOVING
  B4 :  while (proc.refcnt > 0) -> wait in the test loop
  ```

```
B5 :  sleep(timeout)
A3 :  proc.location != HERE -> halt output
A5 :  proc.location == MOVING
A6 :  proc.refcnt--
      ...
```

The sender PROC increments the reference counter. Then the moving PROC sets *location* to MOVING. After testing this field in line A3, and then again in line A5, the sender will defer sending the message until after the move completes. If *location* is set to MOVING, to allow the move to proceed, it decrements *refcnt* (line A6). If the move procedure tests *refcnt* (line B4) before it is decremented, it blocks temporarily in the while loop. After the move completes, the message is delivered to the remote PROC location (line A10).

-
```
B2 :  proc.location = MOVING
A2 :  proc.refcnt++
B4 :  while (proc.refcnt > 0)
B5 :  sleep(timeout)
A3 :  proc.location != HERE -> halt output
A5 :  proc.location == MOVING
A6 :  proc.refcnt--
A7 :  wait(proc)
B6 :  execute PROC move
      ...
```

The moving PROC sets *location* to MOVING. Then the sender increments the reference counter (line A2). The move procedure tests *refcnt* (line B4) before it is decremented and it blocks temporarily in the while loop. The sender finds the *location* flag set to MOVING (line A5). It decrements the counter, then it blocks waiting to be notified when the move is completed.

-
```
B2 :  proc.location = MOVING
B4 :  proc.refcnt == 0 -> continue
     ...
B11:  proc.location = AWAY
```

```
A2 :  proc.refcnt++
A3 :  proc.location != HERE
A5 :  proc.location != MOVING ->  proc.location == AWAY
A10:  remote_output(new_proc, msg)
```

The moving PROC sets *location* to MOVING, then the move procedure completes and *location* is set to AWAY, before the sender increments the reference counter. The sender finds *location* set to AWAY (line A5) and it sends the message to the remote location.

4.3 Persistent Connectivity

Multiple virtual (logical) remote connections between pairs of PROCs are multiplexed over a single network connection between two engines. This has two benefits. On one hand, the cost of connecting and disconnecting PROCs is reduced. The more expensive operation, establishing a network connection, is executed only once. Connecting two PROCs is reduced to exchanging some control messages between engines and updating some data structures accordingly. On the other hand, the details of establishing and maintaining network connections are hidden to the application, being handled by the infrastructure. A connection between two remote PROCs is maintained as long as the corresponding engines are connected.

PROCs can move between hosts while maintaining *persistent connectivity* to other PROCs. The structure of the application does not change and the flow of data in the system is not interrupted[3]. Messages are reliably and orderly delivered during and after component relocation. The movement of a PROC is transparent to other PROCs. Unless explicitly required, a PROC does not know whether it is connected to a local or a remote PROC. The engine handles these details, as well as the message routing to their desired destination.

A logical connection between PROCs is maintained even if the underlying physical connection changes. Consider the case in which a PROC A, located on host H1, and connected to a PROC B, located on host H2, moves to host H3, which is not connected to H2. A connection will be established between the engines running on H2 and H3, transparently

[3]a small delay may be observed due to message forwarding

to A, B, and their respective applications. Messages addressed to A which are received at H1 after A has already moved will be forwarded to A's new location.

In some cases, the temporary failure of a connection between engines can be made transparent to the PROCs. When a network connection is broken, the engines will try to re-establish the connection[4] during a timeout interval. The timeout interval is read by the engine from a configuration file as part of the application initialization procedure. It can be modified by a system administrator during the execution of the application.

Assuming that the disconnection is temporary, an engine caches messages addressed to a remote PROC until the connection is re-established. If the connection failure is permanent, the engine should either find alternative paths to deliver a message to a PROC, or notify the sender PROC at the end of the timeout interval. Currently, our implementation uses the latter alternative.

The use of the timeout interval for re-connection allows a system administrator to briefly shut down an application running on one host, and immediately restart it, without other connected applications noticing it. The connection is re-established transparently, and neither one of the applications loses any state information. We have found this feature useful during the testing and debugging of a distributed application. At some point, a change is made to the code running on one host. The code is compiled. Then the application running on that host is stopped and restarted using the new code. Communicating applications running on other hosts are not impacted. All inter-component connections across hosts are maintained, and don't need to be explicitly re-established.

The seamless connectivity between DACIA components offers a great benefit to mobile users, who can move applications from one host to another without having to manually re-establish all the connections to other parties. It can also provide transparency of the location of a user, if so desired.

4.4 Is DACIA a Mobile Agent System?

DACIA differs from the goals of most mobile agent systems in the sense that PROCs are not designed to be *autonomous*. In autonomous agent systems, agents can initiate moves by themselves, in general by executing calls to move themselves at any time during their

[4] unless the connection has been explicitly shut down

execution. In contrast, PROCs move as a result of commands issued by engines, according to adaptive policies implemented by monitors. This difference in design goals helps achieve considerable simplicity and efficiency in DACIA. A PROC's state is typically smaller in size than those of similar autonomous agents. To handle autonomous move commands from anywhere within an agent's code, the agent's stack state generally has to be serialized and shipped to the remote host. On the other hand, in DACIA, commands to move a PROC issued by a monitor are asynchronous commands that can be executed only after the PROC has completely finished handling a message, thus avoiding the need to ship the stack state in most cases.

Another key difference between DACIA and mobile agent systems is that we consider PROCs to be interconnected and part of a distributed application. Often the actions of multiple PROCs can not be separated, and the end-to-end functionality of an application is achieved by applying the data processing routines corresponding to multiple PROCs situated along a data path. Considerable support is provided for maintaining communication links while PROCs move and for changing the structure of the distributed computation by introducing new PROCs or eliminating existing PROCs, in response to changes in environment.

The interactions between mobile agents are primarily asynchronous. Mobile agents operate most of the time in isolation. They interact rarely, usually through message exchange. The goal of each individual agent is to maximize its own performance. Thus, an agent may decide to move to a different host in order to be closer to the resources accessed (data or hardware resources), or to take advantage of spare processing power. Inter-PROC communication can be either synchronous or asynchronous. PROCs exchange data frequently. Synchronous communication usually yields significant reductions in the overhead of message exchange. The performance of a DACIA application is analyzed end-to-end. The goal of application reconfiguration or component relocation is to improve the performance of the whole application.

A final difference is that we consider a group of engines and PROCs that define a distributed computation to be part of the same trust domain. A major difficulty in deploying mobile agent systems is that of handling security [14], since agents are assumed to be capable of moving to arbitrary hosts (e.g., a database querying agent moving from host to host on the network). Most hosts are obviously reluctant to provide support for arbitrary agents

to execute in their environment. While security issues remain in DACIA, this problem is greatly alleviated, since an engine on a host can be limited to exchanging information only with other engines in the same trust domain.

4.5 Dynamic Code Loading

DACIA applications can evolve at runtime through the creation of new components. In the more common case, a new instance of a class already loaded into the application is created. DACIA also allows the creation of a component of a type previously unknown to the application. In some cases, an application developer creates a component of a new type off-line and makes it available at the site where an application is running. The application loads the class from the local host and creates an instance of the class. In some other cases, the class code can be downloaded from a remote host. A new instance can be either created locally or retrieved from the remote site.

One of the situations when the latter case is encountered is when moving a PROC to a host where the PROC's class implementation does not exist. In this situation, first the class' code is transferred to the destination host. Then the actual instance of the component is transferred as a serialized object.

The dynamic code loading facilities in DACIA make use of the Java class loader. An application uses a class loader object, as opposed to using the primordial class loader used by the Java Virtual Machine (JVM)[5]. The same class loader handles all requests to load a class, either locally or from a remote host. The primitive used for loading a class in DACIA is:

• *loadClass(className, hostName)* - loads a class from the specified host, and creates an instance of this class.

The class is loaded only if it is not already loaded in the JVM. If *hostName* is null, the class loader first attempts to find the class in the local file system. If it fails, it sends a request to other hosts that are connected to its engine. By default, it tries all connections one by one until the class is returned. Once the class data is received (as an array of bytes), it is converted into an instance of a *Class* object. The class code is also saved on persistent storage. Then an instances of this newly defined class is created and a reference to this

[5]a Web browser uses similar class loader objects to download the class files for an applet across a network

instance is returned to the application.

A Java class loader and the objects it creates from the class code it loads operate in the Java sandbox [32]. The class loader architecture contributes to Java's sandbox in two ways: a) it prevents potentially malicious code from interfering with well-behaved code, and b) it guards the borders of trusted class libraries, such as the Java API classes. The class loader architecture makes sure that untrusted classes can not pretend to be trusted. To protect the application from malicious code, the class loader and the application have to be written so that they limit the loaded objects' access to trusted code. In DACIA, loaded PROCs can not invoke methods on other PROCs or access their data. They can only send messages to them. Although it still poses security risks, message exchange reduces the security threats of running code obtained from a remote site.

4.5.1 Component Replacement

As an application of dynamic code loading, a PROC can be replaced with a newer implementation while an application is running. This allows the application to be upgraded with minimal disruption. The new implementation can fix some bugs existing in the previous version of the PROC, address some performance issues, or provide additional functionality.

The old and the new versions of the component have to be compatible, i.e., interchangeable with respect to their state information and their interaction with other components. Two types of compatibility are of interest: *strict compatibility*, and *upward compatibility*. Two versions are considered strictly compatible if they have the same number of ports, they implement the same functionality, and their state variables are equivalent. A version A is considered upward compatible with a version B if a component of type A can replace a component of type B, all interactions with connected components are preserved, and the state of the second component can be transferred to the first one.

A PROC provides a pair of methods, *getState()* and *setState()*, that are used for state transfer. In order for the replacement to be possible, the new component needs to understand the state encoding of the initial component.

The sequence of operations involved in component replacement is:

1. create a new PROC

2. interrupt the execution of the old PROC

3. disconnect the old PROC
4. capture the consistent state of the old PROC
5. transfer the state to the new PROC
6. connect the new PROC
7. start the new PROC
8. remove the old PROC

4.5.2 Monitor Replacement

Another benefit of dynamic code loading is that it allows to change at runtime the adaptive policy implemented by the application's monitor. Also, an application can be started without having a monitor, and a monitor can be added later. While an application is running, its execution environment or user requirements can change in a way that was not anticipated when the application was initially designed and implemented. As a result, the adaptive algorithms implemented by the application's monitor are no longer appropriate. In such a case, a new monitor can be developed and tested off-line. Then the monitor is loaded into the application, replacing the previously existing monitor.

The *stopMonitor()* call is used to stop the execution of the initial monitor. A monitor usually runs in a while loop. Before it stops, the monitor completes the execution of the current iteration of the loop. Thus, if a configuration change is ongoing, it will complete, so that the application is left in a consistent state. After the new monitor is loaded, the *startMonitor()* call will start its execution.

4.6 Application of Component Mobility: Application Parking

Through *application parking*, component mobility and persistent connectivity in DACIA can be used to support off-line operation of interactive applications. Initially developed in the context of groupware applications, application parking is suitable to any interactive distributed application. Using DACIA, a parked application is able to continue to maintain state and to participate, on a limited basis, to collaborations on the user's behalf, while the

user is disconnected or is not active. When the user reconnects, eventually from a different place, he can take over the control from the parked application.

A parked application can reside on the same computing device the user had been connected from, or it can move to a fixed host if the user's device is disconnected. Specialized hosts can provide *parking lot* services to mobile users. When the user's application moves to a different device, it maintains its connections to services and collaborative partners and it continues its execution. The ability to move applications without interrupting their participation to collaborative sessions is particularly appealing in mobile environments, in which users often change the point where they connect to the system or the device they use.

In current groupware applications (Figure 4.9.a), when a user disconnects, the disconnection is usually treated as long-term. Other users are aware that the user is no longer participating, but no further information is available regarding the duration of user's disconnection or whether asynchronous interactions are still possible. Furthermore, when the user reconnects, typically all the connections to collaboration services have to be manually re-established.

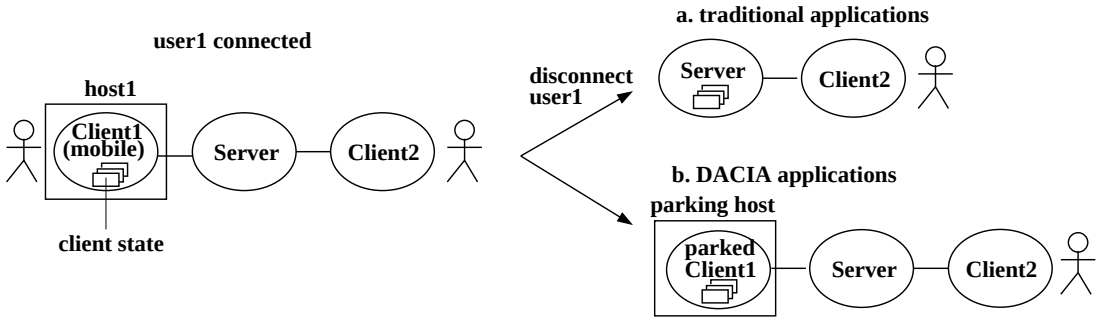

Figure 4.9: Using traditional groupware applications, when a user disconnects, its state has to be saved on the server. If the user later connects to a different server, the state has to be transferred between the servers and between the new server and client. Using DACIA applications, while the user is disconnected, its state is maintained by the parked client, which can continue to participate to collaborative activities.

Using DACIA, the user can park her client agent to a fixed, connected host. While the user is disconnected, a parked client (Figure 4.9.b) can continue to maintain state. Moreover, the parked application maintains its connections and it can interact with collaborative partners.

A user can delegate various degrees of autonomy to a parked application. For example, in the case of a parked chat application, the application's response to messages received from other collaborators could be a simple message (similar to the vacation email message) informing them that the user is not active. A more elaborate parked application could save messages, forward notifications to the user via email, or selectively notify other users of potential future activity schedule. The parked application's behavior can be set to change gradually, according to the duration of user inactivity. For example, after a timeout interval, it can potentially save its state to a server, and shut itself down. There is a tradeoff between the complexity of the parked application code and its ability to actively participate to collaboration.

4.7 An Example of Component Mobility

Figure 4.10 illustrates a simple example of component mobility, in the case of a chat-box application, that has been implemented using DACIA. Two chat-box users are involved in a session from their respective workstations. At some point, one of the users moves her application to a different host. The user issues a *move()* command using either the command-line interface or the graphical interface. The *Chat* PROC moves between the two machines and the users can continue to exchange messages without having to re-establish the connection. The move is transparent to the fixed user. The messages previously exchanged (the state of the moved PROC) are still displayed in the Chat window (the small grey window at bottom right). Messages sent while the move was undergoing are delivered to their destination.

Note that a PROC is allowed to move from one device to a different type of device that supports DACIA. For example, the *Chat* PROC can move to a DACIA-enabled PDA, where it presents a text interface to the user. The main requirement is that corresponding PROCs for different devices agree on the serialized state format so that a PROC move can be accomplished by transferring the serialized state from the engine on one device to the engine on another device. DACIA takes care of transparently restoring the connectivity between PROCs.

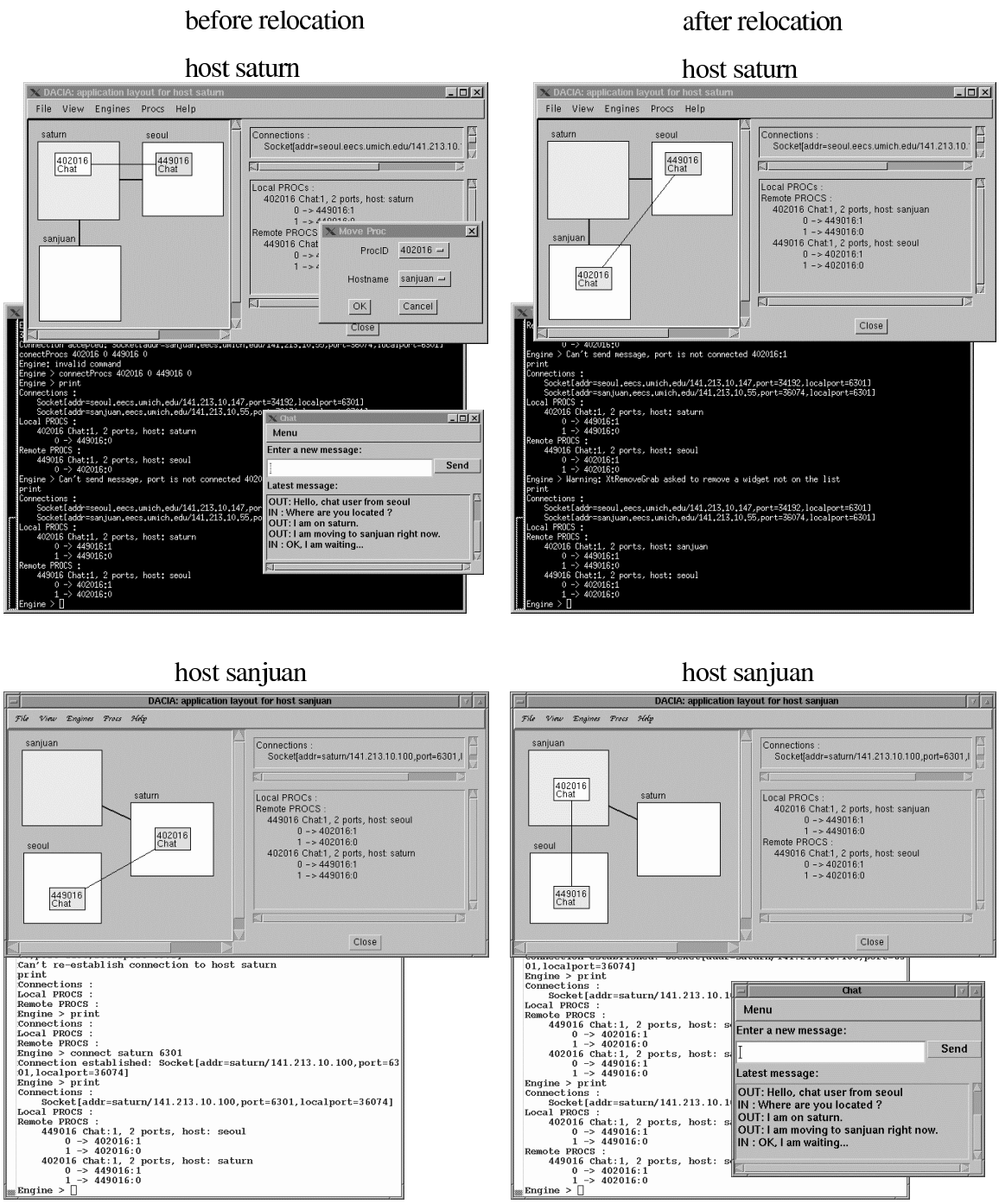

Figure 4.10: A Chat PROC moves from one host (saturn, top left) to another one (sanjuan, bottom right). All PROCs remain connected and continue to exchange data. The graphical interface windows on each host show the configuration of the application, both for the local and remote hosts. Squares represent hosts, and small labeled rectangles represent PROCs. The graphical interface in the top left corner shows two connected Chat PROCs, one situated on the local host (saturn) and one on the remote host (seoul). The terminal windows show application status information, as displayed by the command-line interface.

CHAPTER 5

DYNAMIC APPLICATION RECONFIGURATION

Dynamic reconfiguration represents the ability to modify the structure of an application while the application is running. It enables *application evolution without recompilation*, by allowing new components to be loaded and executed during runtime, as well as providing support for changing the location of existing components and the interactions between them. There are three characteristic types of evolution that we are considering: *corrective, adaptive*, and *perfective* [31]. Corrective evolution removes the effects of faulty behavior. Adaptive evolution alters an application in response to changes in the execution environment. Perfective evolution extends software functionality to meet changing application and user needs.

The ability to reconfigure applications at runtime has several benefits. A more efficient execution of a distributed application can be achieved by changing the way different parts of the application interact and their location of execution, thus taking advantage of the resources available system-wide. The cost of maintaining and upgrading existing applications is reduced by eliminating the need to stop and restart applications during maintenance operations. Runtime application composition and component mobility allow mobile users to access applications using a variety of heterogeneous devices, and to move applications between these devices.

5.1 Adaptability Through Runtime Reconfiguration

DACIA addresses the problem of adapting to hardware heterogeneity and changing application requirements and resource availability through the runtime reconfiguration of the

application. The reconfiguration consists of either reordering or relocating some components or replacing a set of components with a different set of components, possibly connected in a different configuration. Components can be connected or disconnected. New components can be loaded dynamically, and existing components can be removed or relocated to different hosts. As a result of reconfiguration, the application graph changes. The performance of the application can be improved through better usage of the available resources and optimized inter-component communication.

One of our goals is to provide mechanisms for dynamically reconfiguring an application and support for actually making reconfiguration decisions. DACIA provides an API that offers a set of primitives used to reconfigure an application:

- *connectProcs(procID1, portNo1, procID2, portNo2)* - connect two PROCs, using the specified ports. The PROCs can be local or remote.
- *disconnectProcs(procID, int portNo)* - disconnect two PROCs.
- *moveProc(procID, hostName)* - move a PROC to the specified host.
- *moveProc(procID, connection)* - move a PROC over the connection to another Engine.
- *load(className)* - load a component from the local host.
- *load(className, hostName)* - load a component from a remote host. If *hostname* is null, then the application will attempt to load the component from any one of the remote hosts it is connected to.
- *loadMonitor(className)* - load a monitor.

Using these primitives, an application developer can implement a specialized monitor that performs automated reconfiguration. A monitor uses information regarding the performance of the application and the resources available globally to automatically generate and choose among functionally equivalent configurations. DACIA also provides a command-line interface (Figure 6.4) through which an application user or system administrator can manually reconfigure an application by relocating PROCs, creating new PROCs, and changing the way existing PROCs are connected. This interface also allows the addition or replacement of a monitor while the application is running.

Carrying out the actual reconfiguration of an application raises a major challenge: providing atomicity of application reconfiguration when multiple changes are made at different hosts. We have to ensure that the resulting configuration is consistent with the initial application at all times. A distributed application should be reconfigured while it is running, without impacting its execution.

As an example of dynamic reconfiguration, consider the delivery of streamed data between two endpoints. The insertion of a pair of *Compress/Decompress* components at two points in the data-path should be done atomically, so that there are no inconsistencies in the data delivered end-to-end. A solution that can be applied in this case is to carefully write the sequence of operations carrying out the configuration change. Assume that *Sender* and *Receiver* are two PROCs exchanging data, being connected on their respective ports 0. The sequence of operations in Figure 5.1 allows the insertion of a pair of PROCs, *Compress* and *Decompress*, without corrupting the data exchanged. All the messages that had been sent by *Sender* prior to the disconnection will be first compressed and then decompressed, before being delivered to *Receiver*.

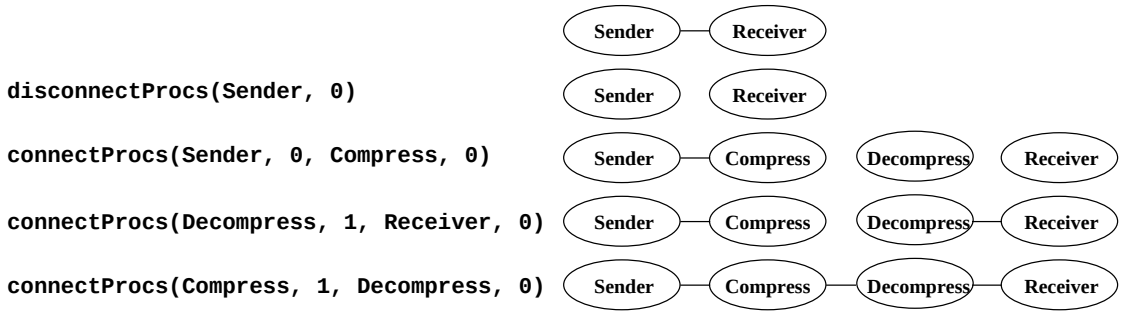

Figure 5.1: The insertion of a pair of *Compress/Decompress* components. The sequence of operations involved should maintain the end-to-end consistency of the data flowing through the system.

In the example above, the Compress and Decompress PROCs are simply introduced in the path between Source and Destination. The application works correctly, since all the messages sent by Sender either will be delivered uncompressed to Destination, or they will be buffered until the connections are re-established after the insertion of the new PROCs. In some other cases, when several PROCs are involved, messages may be in traffic through some communication links when connections between components are broken and re-established.

To ensure the consistency of the data flowing through the system, data delivery may have to be deferred at certain points, and some of the communication paths have to be flushed before changing the configuration[1]. An alternative solution is that the data messages carry some semantic information, allowing them to follow alternative paths down the road.

5.2 Types of Dynamic Structural Changes

The reconfiguration of a DACIA application consists of changing the structure of the application graph or relocating components. Complex application graph transformations are ultimately reduced to primitive structural changes, operating at the level of component or inter-component connections. For dynamic structural changes, it is not sufficient to specify the type of change and the target of the change. Other issues, such as the relative ordering of changes, the effect of changes on each other, and maintaining the application consistency during reconfiguration, need to be addressed.

This section focuses only on individual changes, viewed in isolation. We identified the following types of primitive structural changes that are involved in the runtime reconfiguration of a distributed application:

- **Component creation**

  Component creation supports perfective evolution by extending the functionality of an application. In many cases, adding a new component to a running application can be reduced to instantiating a new component of a type that has already been loaded into the application. However, there are cases in which component creation requires loading a new component of a type previously unknown to the application. Dynamic loading of components (or classes) often requires support from the operating system and programming language, in the form of dynamic class loaders and dynamic linking facilities.

  In order for a new component to execute correctly when added to a running application, the component should not make any assumptions about the current state of the system, or whether the system is in the initial state. The new component should discover the state of the system and synchronize its internal state with that of the

[1] such situations will receive an in-depth consideration in a subsequent section

system.

- **Component removal**

  When a component is removed from an application, special care has to be taken so that the component is in a safe state. The elimination of the component should not negatively affect the execution of other components, and data exchanged with other components should not be lost. These requirements are usually satisfied if the component is not connected with any other components at the moment when it is removed. In some situations, the state maintained by a component need to be transferred to the application or saved to stable storage prior to the component removal.

- **Component connection and disconnection**

  A connection specifies the type of communication between components (e.g., synchronous or asynchronous), as well as the means of handling connection failures. When two components are connected, usually the types of their interfaces have to be matched. Ports used by DACIA applications to communicate between components are not typed. This eliminates the need to perform type matching.

  Although connecting remote components and communicating between them is usually different from the case of local components, it is often desirable to specify a uniform connection syntax in both cases. The underlying mechanisms employed to implement connectivity may vary to accommodate various platforms or to take advantage of performance improvement opportunities.

  The disconnection of two components should not cause messages in traffic to be lost. Calls that have already been initiated should complete and reply messages should be eventually sent prior to the disconnection.

- **Component relocation**

  Moving a component from one host to another does not change the structure of the application and the state of its connections. The connections between components are persistent, regardless of the components' relative locations or the location changes. The state of the moving component and the interactions with other components (messages received, but not processed yet) are preserved and transferred to the

new location.

A change in the configuration of a distributed application usually consists of a combination of the above primitive changes. For instance, adding a component to an application involves creating the component and then appropriately connecting it with other components. Similarly, removing a component requires terminating its interactions with other components and disconnecting the component prior to deallocating the data structures corresponding to the component and freeing up the memory.

The primitive changes listed above can be used to construct higher-level operations, such as:

- **Implementation change**

  The implementation of a component can be changed, while maintaining its functionality (Figure 5.2). The interfaces exposed by the component are also maintained. The change can be motivated by performance reasons or by the need to adapt to specific hardware capabilities or to environmental changes that were not anticipated when the component was initially created and deployed.

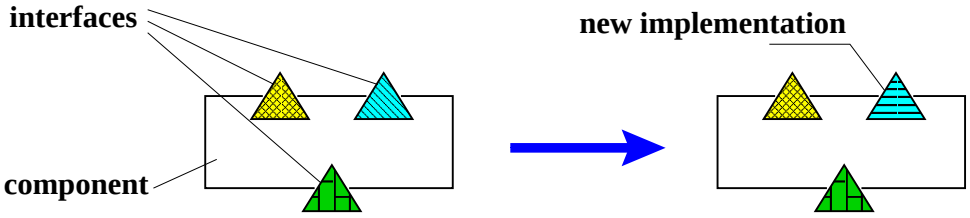

Figure 5.2: The implementation of an interface (the handling of input/output through that interface) can change at runtime. The change is transparent to other components, which may or may not be connected to the interface.

- **Interface change**

  A component can be customized by adding or removing interfaces. When an interface is added (Figure 5.3), the functionality of the component is extended. The functionality exposed through the other interfaces remains the same. Removing interfaces from a component may be useful to obtain a minimal implementation, to be used, for instance, in low-power devices with reduced capabilities (e.g., a PDA). When an

interface is removed, the component should not be engaged in communication with other components through this interface.

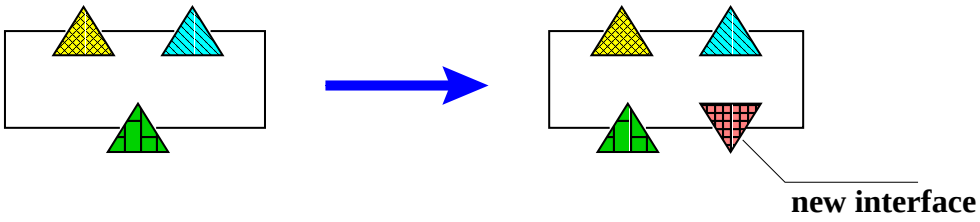

Figure 5.3: The functionality of a component can be extended while the component is running, by dynamically adding an interface. The other interfaces and the interactions with other components are not affected.

Both implementation changes and interface changes are ultimately reduced to **component replacement**. They require that the new component is loaded dynamically into the application and connected appropriately. It is desirable to perform the replacement without affecting the execution of other components or the interactions with the replaced component. If the component maintains state, the state has to be transferred from the old implementation to the new one.

5.3 Facets of Dynamic Reconfiguration

Dynamic reconfiguration supports the structural variability of distributed applications [103]. *Static variability* refers to the variability inherent in a family of related applications. For example, a family of applications might be functionally equivalent, but structural variability is necessary to support multiple platforms. *Dynamic variability* captures the set of all possible configurations that a running application may assume.

The semantic effects of changes on a particular application should be separated from concerns regarding the mechanics of change. In many cases, runtime application reconfiguration has two components. On one hand, the application itself or some external decision module specifies the desired configuration changes, as well as the circumstances under which these changes are applied. On the other hand, a *runtime system* or *configuration manager* performs the actual reconfiguration based on the specified changes, with or without the application involvement. The specification of configuration changes should be declarative,

in the sense that the configuration manager, not the application or its users, should be responsible for determining the specific order of applying various change operations.

There are several important aspects of dynamic reconfiguration. These determine the way configuration changes are reasoned and applied, and the complexity of writing reconfigurable applications and performing configuration changes.

- **Scope of reconfiguration** specifies the extent to which different parts of an application are affected by a configuration change. One approach, for example, requires that the execution of the whole application is blocked during the reconfiguration. In the case of a large-scale distributed application, the reconfiguration often affects only parts of the application. In fact, in most cases it is desirable to confine the reconfiguration to only a small area of the application, with minimal or no impact to the rest of the application and its users.

- **Reconfiguration policy** states how configuration changes are applied to an application. For example, one policy may instantaneously replace the previous functionality with new functionality, in one atomic operation. Another policy may gradually apply changes, so that the application goes through a sequence of intermediate states before reaching the final configuration. A different policy allows multiple versions of an application to run in parallel, while the transition from the initial to the final configuration is completed, and the correct execution of the final configuration is validated.

- **Change specification** influences the power and expressiveness of the reconfiguration mechanisms, as well as the complexity of executing configuration changes. The granularity of the entities involved in reconfiguration (e.g., code fragments, libraries, objects, components, connectors) and the level of abstraction at which changes are specified affect both the amount of information that must be effectively managed and the efficiency of the reconfiguration process.

  One way of specifying a configuration change is as a pair of initial and final configurations. This leaves the responsibility of determining the actual operations needed to perform the reconfiguration to the runtime system or configuration manager. In this case, the implementation of the configuration manager may be particularly challenging. Alternatively, reconfiguration can be specified as a set (ordered or unordered)

of reconfiguration operations. The configuration manager decides how to order these operations and what the necessary steps are for applying the changes and insuring that the reconfiguration executes correctly and the application consistency is not compromised.

- **Separation of concerns** reflects the degree to which the functional behavior of an application is separated from the issues concerning the dynamic change. The ability to alter either one of the above without affecting the other one increases as the degree of separation increases.

- **Degree of automation** represents the ability of an application to make reconfiguration decisions and execute the reconfiguration all by itself, without the user's intervention. Ideally, the application has all the information and capabilities needed to reconfigure itself, and the user's involvement is not necessary. However, there are cases when all environment changes that may occur during the execution of an application can not be anticipated when the application is initially designed and implemented. In such cases, an application user or system administrator has to explicitly intervene to reconfigure the application. An adaptive application and the supporting infrastructure should allow both automated and manual reconfiguration.

- **Failure semantics** specifies the behavior of an application if a failure occurs during the application reconfiguration. The failure should leave the application in a consistent state. Either the application is able to recover from the failure and complete the reconfiguration, or the failure is reported with respect to an externally visible consistent state. In the latter case, the failure is seen as either a failure in the initial configuration, or a failure in the final configuration. This may require grouping reconfiguration operations into atomic sets, so that if an operation fails, the whole set of changes is uncommitted to avoid leaving the system in an inconsistent state.

5.4 Requirements of Dynamic Reconfiguration

Runtime changes in the configuration of a distributed application have several requirements. Some of these requirements concern the performance aspects of reconfiguration and the impact of reconfiguration on applications and their users. Other requirements refer to

the correct execution of the application during and at the end of the reconfiguration and the maintenance of persistent application state. Additionally, often architectural changes must preserve several kinds of properties: structural (e.g., the application has a bus or a ring structure), functional, and behavioral (e.g., quality of service requirements have to be respected).

- **Performance**

  Dynamic configuration changes should be efficient. The efficiency is regarded along several dimensions. First, the time between the reconfiguration request and the execution should be minimal. Second, the time needed to perform the reconfiguration should be minimal. Third, the number of control messages exchanged and the size of these messages should be minimal.

- **Disturbance**

  Application disturbance with respect to the number of components affected by the reconfiguration should be minimal. Excessive disturbance may cause an application to fail its requirement for continuous service. Application disturbance can be reduced by designing loosely coupled component architectures.

- **Termination**

  For any reconfigurable application, all reconfiguration states should be *reachable*. An unreachable reconfiguration state can cause the reconfiguration process to fail, or the application to fail its functional requirements. Reconfiguration states should be reached in bounded time. These requirements insure that the reconfiguration process does not hang. They also contribute to the goal of minimizing the time needed to execute the reconfiguration.

- **Correctness**

  An application should execute correctly both during and after the reconfiguration. The state maintained by the application and its components should not be altered, and the integrity of data in traffic should not be compromised.

- **Automated reconfiguration**

Dynamic reconfiguration should be managed without user intervention whenever possible. When manual techniques are used, the risk of introducing errors in the process of reconfiguration is high. If changes are expressed at a higher level of abstraction, the probability of introducing errors is reduced.

In many cases, all the above requirements can not be met simultaneously. Tradeoffs can be made, for instance, between the efficiency of the reconfiguration and the application consistency. Application developers should have the ability to choose the set of requirements that represent their particular needs. The underlying infrastructure should provide the mechanisms and the API primitives necessary for implementing the desired reconfiguration algorithms.

5.5 Maintaining Application Consistency

The dynamic reconfiguration of an application should not compromise the application consistency. Consistency constraints are divided into *component consistency* and *system consistency* properties. The former are local and require that the state maintained by a component is not altered during the reconfiguration[2]. The latter are global and require that the end-to-end functionality of the application is maintained, and the integrity of the data maintained by the components or in traffic among the components is not compromised.

Figure 5.4 gives an example of a data integrity problem that might occur. Consider the data transfer between a component A and a component C, through an intermediate component B (Figure 5.4.a). Assume that at some point a pair of *Compress* and *Decompress* modules is inserted into this data path, as in Figure 5.4.b. If before the reconfiguration is executed data was available only at nodes A and C, the application will function correctly after the reconfiguration. The data at node A will first be compressed and then decompressed, before being delivered to node C. But if data existed at node B before the reconfiguration, after the reconfiguration this data will reach the decompression module without being first compressed. This will lead to the incorrect execution of the application.

To avoid these types of problems, our approach to reconfiguration first flushes the data maintained by certain components or in traffic along links between these components, before

[2]in case of component replacement, the state should be transferred from the old component to the new component

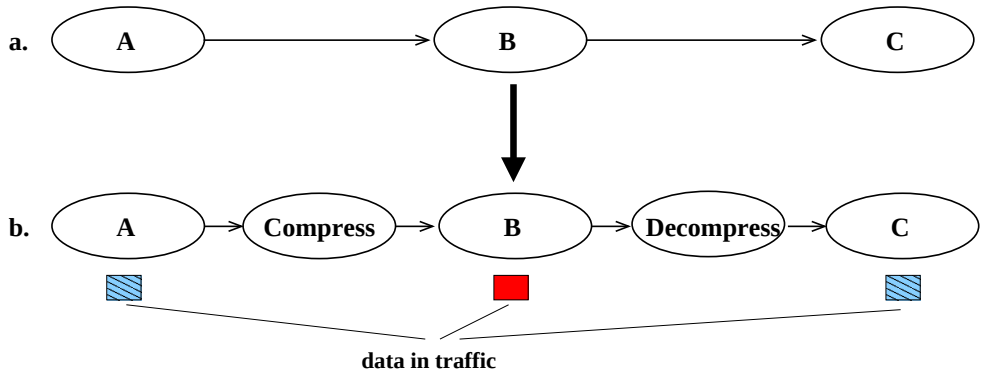

Figure 5.4: A pair of *Compress/Decompress* components is inserted into the data path between components A and C. The integrity of the data maintained by the application may be compromised. If data was present at node B before the reconfiguration, after the reconfiguration this data will be sent to the decompression component without being first compressed.

executing the reconfiguration. At the same time, it prevents new incoming message traffic from entering the region of an application that is about to undergo reconfiguration, by blocking the receiving of messages by some components. On the flip side, we attempt to minimize the number of components that are blocked, and the impact of reconfiguration on the application execution.

Maintaining application consistency leads to two problems: a) synchronizing the reconfiguration with the application execution, and b) managing the persistent state of individual components and that of the whole application. The former requires the specification of a sequence of states that must be attained prior, during, and after the reconfiguration, and the use of algorithms for safe transition between these states. The latter demands that the application state maintained by components and their interconnections should be preserved throughout the reconfiguration process.

In addition to the functional correctness of a distributed object system and its constituent components, consistency may refer to other attributes as well, such as quality of service and system security. Our reconfiguration solution currently does not address these issues. This represents some potential directions in which this work can be extended in the future.

5.6 Executing the Reconfiguration

5.6.1 Terminology and Assumptions

Before we get into the details of the approach employed by DACIA to perform dynamic reconfiguration, we need to introduce some terminology and to state the assumptions made by our algorithm.

The configuration changes that we consider are: *connecting and disconnecting components, creating new components, and removing components.* Moving components across hosts does not affect the overall structure of the application graph and the functional behavior of an application. Component mobility can be handled separately from other reconfiguration operations. In fact, Chapter 4 offers an in-depth presentation of the details of moving components.

The **changing set (CS)** is the set of components involved in reconfiguration. DACIA attempts to minimize this set of components affected by reconfiguration.

A **reactive chain** is a set of components executing dependent message exchanges. In the example in Figure 5.4.a, if component A sends a message to B, as a result of handling this message, B will send a message to C. In this case, A, B, and C form a reactive chain.

A reactive chain RC is formally defined as the set of tuples (A, a0, a1), where A is a PROC, a0 and a1 are ports of A[3], such that:

- (A, ϕ, a1) $\in$ RC, where a1 is one of A's ports (chain initiator).
- If (A, a0, a1) $\in$ RC, port a1 of A is connected to port b0 of B, and if B receives a message on port b0, B does not send out new messages as a result of handling this message, then (B, b0, ϕ) $\in$ RC (chain terminator).
- If (A, a0, a1) $\in$ RC, port a1 of A is connected to port b0 of B, and as a result of handling the message received on port b0, B may send out a message on port b1, then (B, b0, b1) $\in$ RC.

In practice, reactive chains are determined using the specifications for individual components and the application configuration information (the connections between components). The specification of a component states all dependencies between input and output ports,

[3] a0 and a1 may be equal to ϕ under certain conditions

i.e., whether handling a message received on port i may require that a message is sent on port j.

A component B is reachable from a component A with respect to a reactive chain RC iff ∃ ports a0, a1, b0, b1 of A and B, respectively, such that:

(A, a0, a1) ∈ RC, (B, b0, b1) ∈ RC and either

port a1 of A is connected to port b0 of B or

∃ (C, c0, c1) ∈ RC such that port a1 of A is connected to port c0 of C, and B is reachable from C with respect to RC.

A component C is in the middle of a reactive chain between components A and B iff ∃ a reactive chain RC such that A, B, C ∈ RC, C is reachable from A, and B is reachable from C with respect to RC.

A component can be in one of the following **states** (Figure 5.5):

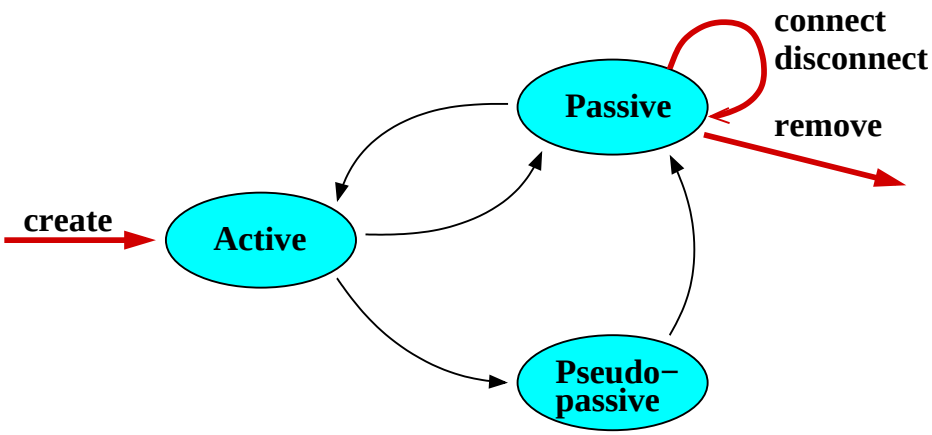

Figure 5.5: Component states. An *active* component can send, receive, or process messages unrestricted. A *passive* component can not receive or send messages, and it does not have any pending messages. A *pseudo-passive* component can receive messages only from a specific set of components, which are trying to become passive. When a component is created, it becomes automatically active. A component has to be passive before it can be connected, disconnected, or removed.

- **Active:** the component can send, receive, or process messages unrestricted. This is the normal functional state of a component.
- **Passive:** the component can not receive or send messages, and it does not have any pending messages, i.e., messages received and not handled yet, currently being placed

in its message queue. This is the state that the reconfiguration algorithm tries to achieve before executing the actual configuration changes.

- **Pseudo-passive:** the component can receive messages only from a specific set of components, which are trying to become passive. This is an intermediate state that a component reaches during the reconfiguration, before becoming passive.

For a component X we define the **sender set (SS)** as the set of components in CS that can send messages to X :

SS(X) = {Y ∈ CS | there is a direct connection from Y to X in the application graph}

The condition for component X to become passive is:

∀ Y ∈ SS(X), Y is passive

A **configuration manager** executes the application reconfiguration, based on the change operations requested by a monitor. The engine where the monitor executes assumes the configuration manager function. The runtime system (made up of the the engines where the application executes) has an interface with the application, in particular with its components, that allows it to bring the application to a safe state where the reconfiguration can perform correctly. In order to avoid the consistency problems mentioned in Section 5.5, before reconfiguration operations can be executed, all components involved in the reconfiguration have to become passive. All messages in traffic among these components are flushed.

The reconfiguration algorithm assumes that:

- All message handling routines complete in bounded time. Two cases are possible. If, as a result of handling a message, a handling routine sends another message asynchronously, the message send call returns right after the message is sent and the routine proceeds. If a message is sent synchronously, the sender waits until the handling routine completes in the receiving PROC and the call returns. It is thus possible to have a whole chain of messages being sent synchronously and associated handling routines. The first message handling routine should complete in bounded time in both cases.

- If reactive chains have cycles, the number of iterations over a cycle is finite. This requirement ensures that there is a finite number of message dependencies between

the PROCs in the system, and the PROCs will get to the passive state in finite time.

Without loss of generality, the reconfiguration algorithm assumes that there is at most one connection between a pair of components[4]. This is only a simplifying assumption. The algorithm works correctly even if this assumption does not hold.

5.6.2 Reconfiguration Algorithm

This section presents the algorithm used by DACIA to perform dynamic application reconfiguration. We first outline the steps of the algorithms. Then we detail the execution of these steps.

We use the example of application reconfiguration presented in Figure 5.6. In a multiparty communication application, client PROCs (Ci) are connected through intermediate server PROCs (Si). A client PROC has a single port (port 0). A server PROC has n ports. Every message sent by a client will be distributed to all other clients through the corresponding servers. The application graph in Figure 5.6.a. is transformed into the graph in Figure 5.6.b. A new server S4 is introduced and client C1 is assigned to S4. The application has to ensure that no messages are lost during the reconfiguration and the FIFO order of delivering messages with respect to their senders is not compromised.

The input of the reconfiguration algorithm consists of:

1. The initial application configuration[5]: the PROCs, their locations, and their interconnections.

2. The port input/output relationships for each PROC, e.g., if as a result of receiving a message on port i, the PROC may send a message on port j, then the pair (i, j) is an input/output relationship for the PROC. Together with the inter-PROC connections, these relationships are used to determine the reactive chains.

   In the example in Figure 5.6, client PROCs have no input/output relationships. The input/output relationships for a server PROC Si are :

   (i, j), $\forall\ 0 \leq i, j \leq portNo(Si), i \neq j$ (a message received on one port is sent on all other ports, if they are connected)

[4] one connection in each direction, or one bidirectional connection

[5] This configuration may not be accurate at the moment the reconfiguration operations are actually executed. The reconfiguration algorithm verifies the accuracy of the configuration information.

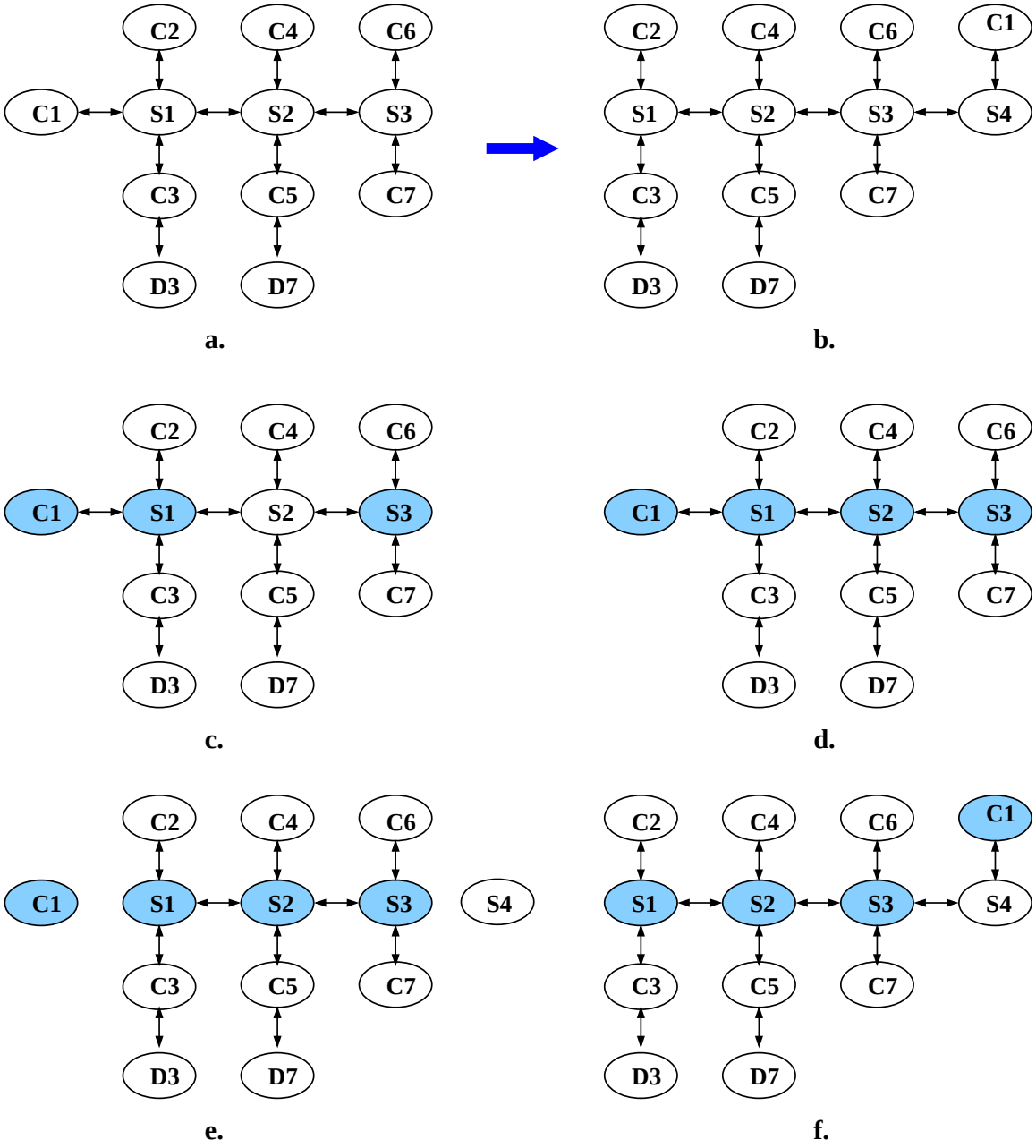

Figure 5.6: Execution steps for dynamic reconfiguration. The application graph in a. is transformed into the application graph in b.
c. The changing set CS is initialized to the set of components that will be connected, disconnected, or removed: CS = {C1, S1, S3}.
d. Components that are in the middle of a reactive chain between components in CS are added to CS. Thus, CS = {C1, S1, S3, S2}.
e. After all components in CS become passive, the reconfiguration proceeds. Components are disconnected, then the new component S4 is introduced.
f. Connections are established. After all reconfiguration operations complete, all components are activated and the application resumes its normal execution.

3. The configuration changes, specified as an ordered set. For instance, for the example considered, the configuration changes are:

```
disconnectProcs(C1, 0)
S4 = new Server()
connectProcs(S3, 3, S4, 0)
connectProcs(C1, 0, S4, 1)
```

The reconfiguration algorithm executes correctly if it meets the following requirements:

1. From the application's perspective, the reconfiguration is atomic, i.e., the application is either in the initial or in the final configuration, and its functionality corresponds to one of these two configurations.

2. All messages in traffic before the reconfiguration or generated by components during the reconfiguration are delivered to their respective destinations.

3. All messages are delivered in FIFO order with regard to their initial senders.

4. The application state maintained by the components is not altered, and the integrity of data in traffic is not compromised.

Before the actual execution of the reconfiguration begins, the configuration changes are validated, based on the specification of the set of changes and the existing application configuration. The configuration manager verifies that the changes can be executed. Validation actions can range from simple syntactic checks (e.g., whether components involved in changes exist, or whether a port to be connected is not already connected), to semantic checks that may be application-specific (e.g., whether the disconnection of some components will cause an application partition, or whether some components will receive duplicate messages following alternative paths). Depending on the specific reconfiguration policy adopted, the reconfiguration may fail or it may continue if one of the configuration changes is invalid and can not be completed as specified.

The reconfiguration algorithm executes the following steps:

1. Find the changing set CS

2. Verify the configuration information and lock the components in CS

3. Block the message exchange from components outside CS to components in CS

4. Flush all messages in traffic among components in CS

5. Execute the reconfiguration

6. Commit the configuration changes, activate and unlock all components

In every stage of the reconfiguration algorithm, all engines involved are aware about the progress of the reconfiguration and the current execution step. The engine where the configuration manager runs sends notifications to all other engines at the end of each step, and the engines send back acknowledgments[6]. The configuration manager proceeds to the next step only after it receives all notifications. Figure 5.7 shows the interactions between the configuration manager and other engines during each step of the algorithm.

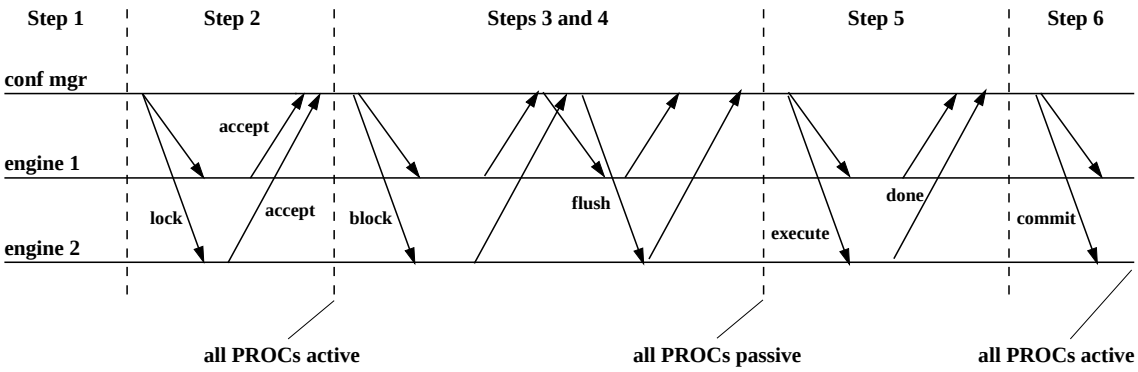

Figure 5.7: The interactions between the configuration manager and other engines during the execution of the reconfiguration algorithm.

Before executing the reconfiguration operations, all components involved in the reconfiguration (changing set CS) are brought to the passive state. We try to minimize the impact of reconfiguration on the execution of the application.

Step 1 is executed locally on the host where the configuration manager runs. The configuration manager determines the minimal set of components that need to be blocked to ensure the correct execution of the reconfiguration and that of the application. It starts with

[6]Step 4 can start on each engine right after step 3 completes, without the need to notify the configuration manager and the other engines.

the set of components that are directly involved in the reconfiguration, i.e., the components that will be connected, disconnected, or removed:

CS = {C1, S1, S3} (Figure 5.6.c.)

Then the configuration manager adds all other components that might be affected by the reconfiguration. It traverses the reactive chains to find all components that are in the middle of reactive chains between components already in CS. In fact, it is not necessary to calculate all reactive chains. The problem is reduced to a reachability problem: finding all components which are not in CS, but can be reached from components in CS through reactive chains, and from which other components in CS can be reached through the same reactive chains.

In the example, all reactive chains are of the form {(Ci, ϕ, 0), (Sj, s_{jk}, s_{jl})*, (Cm, 0, ϕ}, where k $\neq$ l (at the receiving of a message on a port, a server sends out the message on all other ports). If a client PROC Ci receives a message on port 0, the message is processed locally, and no output message is generated. Therefore, a reactive chains does not contain elements of the form (Ci, 0, 0). S2 is in the middle of the reactive chain between S1 and S3. Thus the changing set becomes:

CS = {C1, S1, S3, S2} (Figure 5.6.d.)

Components that belong to a reactive chain involving components in CS, but are not in the middle of a reactive chain between two components in CS, are not added to CS. For instance, {C2, S1, S2, S3, C7} is a reactive chain. S2 will be included in CS, but C2 and C7 will not be included.

In step 2, the configuration manager sends a *lock* message to each engine. This message contains the *changeSeqNo* known by the configuration manager for each PROC in CS located on the destination host. The engines acquire a lock on the local components in CS, to prevent these components from undergoing other configuration changes while the reconfiguration is in progress. First an engine verifies, based on the change sequence numbers, whether the configuration information the configuration manager has (PROCs locations and their interconnections) matches the local configuration. If the configuration is identical, then the engine attempts to lock the local PROCs in CS. If it is successful, it sends an *accept* message to the configuration manager. If either the configuration information is inconsistent, or a component is already locked by somebody else, the engine sends a *reject* message to the configuration manager.

At the receiving of a reject message, based on the reconfiguration policy implemented, the configuration manager can take one of the following actions: a. abort the reconfiguration and inform other engines to unlock their respective components; b. if a lock can not be obtained, back off and retry after a timeout interval, after previously re-evaluating the conditions that led to the reconfiguration in the first place; and c. if the configuration information is inconsistent, update the local configuration, re-evaluate the conditions for reconfiguration, and eventually retry to execute the reconfiguration.

Steps 3 and 4 of the algorithm ensure that all previous interactions between components in CS complete, changes of components' states as a result of these interactions are applied, and the state of the application is consistent. They also ensure that new interactions between components are not started. Thus livelock is avoided and all components in CS ultimately become passive.

In step 3, the configuration manager sends a *block* message to all engines. Each engine partially blocks the execution of local components in CS. Components are first made pseudo-passive. The pseudo-passive state allows components to progress towards the passive state. A pseudo-passive component accepts and handles incoming messages only from other components in CS. For example, S2 will accept messages from S1 and S3, but not from C4 and C5. A pseudo-passive component can not send new messages to other components; it can however send messages in response to handling pending messages.

The execution of active components that attempt to send messages to a pseudo-passive component is not blocked. For example, if C3 tries to send a message to S1, the delivery of the message is deferred. If the send call is synchronous, the sending thread is blocked. However, C3 can be accessed by other threads, and their execution is not impacted. If the send call is asynchronous, the delivery of the message to S1 is deferred, the send call returns, and the execution of C3 continues. At the same time, the communication between C3 and D3 may proceed without any interruptions.

In step 4, all pending messages maintained by components in CS are flushed. The components transition from the pseudo-passive state to the passive state. Other pseudo-passive components may be conditionally activated to receive and process messages sent by components that try to become passive. For example, as a result of handling a message received from C2, S1 sends the message to S2. S2 has to receive this message, and deliver it to C4, C5, and S3. Thus S3 is conditionally activated to receive this message. At the same

time, S2 does not have to accept new messages from C4.

In order for a component to become passive, its engine has to know when all component's potential senders in CS are passive. Multiple messages can be exchanged between the configuration manager and the other engines to inform each other when components' states change. The engines should also notify the configuration manager when their PROCs are idle, i.e., they are not processing any message and don't have any pending messages.

The reason for flushing messages in traffic among components in CS is to avoid application consistency and data integrity problems such as the one presented in Section 5.5. We avoid situations in which chains of interactions among components start in the initial application configuration and they are finalized after the reconfiguration.

Following we give a few examples of problems that may occur if messages are not flushed prior to the reconfiguration. Consider that C2 sends a message to S1. C1 is disconnected before receiving this message. The message is distributed from S1 to S2 to S3, and from here to C6 and C7 only. Then S4 and C1 are connected. C1 will not receive the message. Alternatively, if C1 sends a message to S1 before being disconnected, then C1 is connected to S4 before the message goes through all the servers, C1 will erroneously receive this message. It is also possible that the FIFO order of delivering messages that originated at the same client is compromised, for example if C1 sends a message M1 to S1, then a message M2 to S4 after the reconfiguration, and C6 receives first M2 and then M1.

To reduce the number of state transitions, components can be made passive according to their order in reactive chains. Thus, in many cases a component can go directly from the active state to the passive state, if it has no other pseudo-passive senders in CS.

For a component to become passive, all components in its sender set have to be passive. For the example considered, some of the sender sets are:

SS(S1) = {C1, S2}, SS(S2) = {S1, S3}.

The application graph may have cycles. It is possible that pseudo-passive components can not become passive, since they wait circularly for one another to become passive. In such a situation, all components become passive when there are no pending messages held by these components. The assumptions that message handling routines complete in bounded time and the reactive chains cycles are finite guarantee that the state when there are no pending messages can be reached.

The reconfiguration algorithm can be augmented with the algorithm in Figure 5.8 to

```
while (true) {
  sleep(timeout)
  determine PPS = set of pseudo-passive components
  success = true
  // setup phase
  for all components X in PPS
    if (X has no pending messages and no handling routine is ongoing)
      flag(X) = 1
    else {
      success = false
      break
    }
  if (success == false)
    continue
  // verification phase
  for all components X in PPS {
    if (flag(X) == 0} {
      success = false
      break
    }
    for each incoming connection of X
      // verifies if there are any messages sent, but not delivered yet
      if (lastSeqNoMsgSent != lastSeqNoMsgRcvd) {
        success = false
        break
      }
    if (success == false)
      break
  }
  if (success == true)
    break
}
for all components X in PPS
  state(X) = passive
```

Figure 5.8: Algorithm for handling circular dependencies between pseudo-passive components. In the setup phase, for all components in the pseudo-passive set PPS, a component's flag is set to 1 if it has no pending messages and no ongoing message handling routine. The verification phase ensures that no message has been received in the meantime by a component whose flag was previously set to 1, and no messages are in traffic between two components.

handle the situation in which cycles are present and to detect the moment when there are no pending messages held by any one of the pseudo-passive components, and all these components can become passive. Every pseudo-passive component has a flag. Initially all flags are set to 0. When a component has no pending messages and no message handling routine is ongoing, the configuration manager sets its flag to 1. If all flags have the value 1 and all channels between components in CS are empty, then all components can be made passive.

The algorithm executes periodically until it is successful. One iteration of the algorithm has three phases. First the set PPS of pseudo-passive components is calculated. The membership of the set may change between iterations, due to some components becoming passive. Next (setup phase), the status of all components in PPS is checked and their flags are set to 1 if appropriate. The third phase (verification) of the algorithm verifies that all the flags still have the value 1 at the end of the previous phase. While the algorithm executes the setup phase, it is possible that a component X that has already been checked and whose flag was set to 1 receives a message from a component Y. As a result of receiving this message, the flag of X is reset to zero. After sending this message, Y may enter in the state where it has no pending messages and no ongoing message handling routine. Later on during the setup phase, the verification algorithm sets Y's flag to 1. The verification phase ensures that this situation is captured and accounted for. It also checks whether there are any messages in traffic between two components (messages sent but not delivered yet), based on the sequence numbers of messages last sent and delivered for each connection between two components. If the verification phase succeeds, all components in PPS are made passive.

After all components in CS reach the passive state, in step 5 of the algorithm the configuration manager sends an *execute* message to all engines. Before any configuration change is made, each engine saves the initial configuration, as well as copies of the local components. This is necessary in order to allow the application to survive host or network failures, and the reconfiguration to roll back in case that such failures occur (see Section 5.6.4).

Then the reconfiguration operations are executed. First existing components are disconnected or removed, and new components are created. Then connections are established according to the final application configuration. After completing all reconfiguration oper-

ations, an engine sends a *done* message to the configuration manager.

After all configuration changes are executed, in step 6 the configuration manager sends a *commit* message to all engines. All configuration changes are committed, and the engines activate and unlock the local components in CS. The activation can occur in any order. The application resumes its normal execution, and all previously blocked messages are delivered.

Before changes are committed, all engines have to maintain copies of the local components as they were at the beginning of the reconfiguration. In case of a failure (see Section 5.6.4), these copies can be used to restore the initial application state. One of the non-blocking atomic commitment algorithms existing in the literature [5, 76] can be used in step 6 to ensure that the reconfiguration has completed, all engines involved have been notified, and they committed the changes.

5.6.3 Asymptotic Complexity

Let n be the number of components in the application graph, and m the number of connections between components.

In order to determine the changing set CS, the reconfiguration algorithm starts with the set of components that are involved in reconfiguration operations. Since the number of operations that can be applied to any component is bounded by a constant (there are four distinct operations, and the number of ports that can be connected or disconnected is bounded by a constant), the reconfiguration consists of at most O(n) operations. Each operation involves one or two components. Therefore the initial CS is determined in O(n).

To find the additional components that are in the middle of reactive chains between components in CS, we do not need to traverse all reactive chains. Instead, the problem is reduced to determining all additional components that can be reached using paths along reactive chains originating at components in CS. Also, there has to be a path from an additional component to a component in CS, following a reactive chain. This can be done in linear time with respect to the size of the graph (O(n+m)), using a breadth-first traversal.

Step 2 requires a message to be sent from the configuration manager to the other engines where components in CS reside. The engines verify the configuration information and acquire a lock for each component in CS. Then they send acknowledgments to the configuration manager. Therefore the complexity of step 2 is O(n).

Similarly, step 3 involves sending messages between the configuration manager and the

other engine, and updating the state for each PROC. Thus the complexity of this step is O(n).

For step 4 of the algorithm, we assume that there is at most one pending message for every component in CS. The same results hold if the number of pending messages for every component is bounded by a constant. The path followed by a message and the subsequent messages generated as a result of executing the message handling routine is in fact a reactive chain originating at the component where the message resides. The maximum length of this reactive chain is n. Therefore the complexity of flushing all messages held by the components in CS is $O(n^2)$. This worst case complexity can in fact be achieved, for instance, if all components are linked in a linear list: $1 \rightarrow 2 \rightarrow 3 \rightarrow ... \rightarrow n$, they each have one message, and all messages are propagated all the way to the last component. The length of the paths traversed by these messages are, respectively: n-1, n-2, ..., 1, for a total of $(n-1) + (n-2) + ... + 1 = n(n-1)/2 = O(n^2)$.

If the reactive chains may have cycles, a cycle is iterated at most k times, where k is a constant, under the assumptions at the beginning of section 5.6. The complexity of this step thus becomes $O(k*n^2) = O(n^2)$.

For most practical cases though, the complexity of step 4 is O(n+m). For instance, consider the case where the length of every reactive chain is bounded by a constant. The sum of the lengths of all reactive chains is at most equal to the number of edges m, multiplied by the number of inputs to a component (the same path can be traversed due to inputs received on multiple ports), which is bounded by the number of ports for the component, and therefore by a constant. Therefore the sum of lengths of all reactive chains is O(n+m). Thus the complexity of step 4 is O(n+m).

Executing a reconfiguration operation involves sending a message from the engine where the configuration manager runs to the engines where the components involved in the operation are located, therefore it can be done in constant time. There are four types of reconfiguration operations that can be applied to a component (connect, disconnect, create, and remove). The number of ports of every component that can be connected or disconnected is bounded by a constant. Thus there are O(n) reconfiguration operations possible. Therefore the complexity of step 5 is O(n).

Committing changes, activating and unlocking the components in step 6 is also done in O(n).

In every step of the algorithm, control messages are sent between the engine where the configuration manager runs and the engines where components are located. The total number of control messages exchanged is O(n).

In the above analysis of the complexity of the reconfiguration algorithm, after the changing set CS is determined in step 1, the algorithm involves only the components in CS. Beginning with step 2, n and m should in fact represent the size (number of vertices and number of edges, respectively) of the subgraph with vertices in CS. In the worst case, CS contains all components in the application, therefore n and m can be used as an upper bound for the size of CS.

To summarize, the reconfiguration algorithm executes in $O(n^2)$, where n is the number of components in the application. Flushing the messages held by the components involved in reconfiguration in step 4 of the algorithm is the most complex operation, in the worst case. For most realistic applications though, this can be done in O(n+m), where m is the number of connections between components. Thus the whole algorithm can execute in O(n+m), which is linear with respect to the size of the application graph.

5.6.4 Handling Failures

Failures may occur while a distributed application is undergoing reconfiguration. If a failure occurs either between two operations or during the execution of one reconfiguration operation, the application should either be able to recover from the failure and complete the reconfiguration, or it should report the failure with respect to the consistent state of the application either before the reconfiguration or at the end of the reconfiguration.

A situation that occurs under normal application execution, but might be interpreted as a failure by some hosts, is when an application voluntarily terminates or breaks the connection with another application running on a different host[7]. A request to disconnect an engine will be deferred until the application undergoing reconfiguration reaches a stable state. If a request is made during steps 1, 2, 3, or 4 of the reconfiguration algorithm, the configuration manager is informed and the reconfiguration is aborted. The disconnection is executed in the initial application configuration. If the request is made during steps 5 or 6, the disconnect operation is deferred until the reconfiguration completes. The disconnection

[7]such a decision can be made either automatically by the application or by its user

is thus executed in the final application configuration.

The failures that we are considering are the failure of one of the hosts (engine) where the application is running, and the failure of a network connection between hosts. An engine detects either one of these failures as a broken connection to another engine. The engine timeouts and attempts to re-establish the connection. If it succeeds, the application continues its normal execution. Otherwise, the remote engine is considered failed, and the corresponding PROCs are eliminated from the application. If the broken connection can be re-established, some messages need to be re-transmitted between the engine where the configuration manager executes and other engines. In the following, we will focus on the case where the connection can not be re-established. We first consider only the failures of engines where PROCs involved in the reconfiguration reside, but not the configuration manager. The failure of the configuration manager's engine is treated separately.

The following failure scenarios are possible:

1. A host or a network link failure occurs during steps 1, 2, 3, or 4 of the reconfiguration algorithm. No reconfiguration operation has been executed yet. The reconfiguration aborts, and the state of all reachable components is restored to active. The failure is seen as a failure in the initial configuration.

2. A failure occurs during step 5 of the reconfiguration algorithm. Some, but not all reconfiguration operations have already been executed. The application is rolled back to the initial configuration, using the information saved by the engines at the beginning. If an engine failed after a component had moved to that engine, the component is restored from the copy maintained at its initial location. The failure is observed in the initial application configuration.

3. A failure occurs during step 6 of the reconfiguration algorithm. All configuration changes have already been made. All reachable components are activate. The failure is seen as a failure in the final application configuration.

   If the commit protocol fails, there is no guarantee that the reconfiguration completed. The application is rolled back to the initial configuration, using the information saved by the engines at the beginning. The failure is observed in the initial application configuration.

The solution presented above assumes that the host where the configuration manager runs (primary host) does not fail. One solution that addresses the case where the primary host does fail is that before reconfiguration begins, a complete copy of the initial application configuration and the set of reconfiguration operations is saved on a different host (backup). This host uses heartbeats to detect the failure of the primary host. In case of a failure, a new configuration manager is started on the backup host. It initiates connections with all hosts where the application executes and assesses the current state of the reconfiguration algorithm. If the changes have not been committed yet, the reconfiguration is aborted. Using the information cached by the engines, the backup configuration manager restores the initial application configuration.

5.6.5 Conflicting Configuration Changes

Faults can occur during reconfiguration if multiple monitors simultaneously initiate different reconfigurations on overlapping sets of components. To prevent conflicting configuration changes, the configuration manager uses locks to exclusively reserve all components that are involved in the reconfiguration. Before the actual execution of the reconfiguration operations begins, the configuration manager verifies with the engines the location and connectivity information about the PROCs in the changing set. It also instructs the engines where the components in the changing set are located to lock these components. At their current hosts, only operations (move, connect, disconnect, or remove) initiated by the holder of a PROC's lock can be applied to the locked PROC.

If the algorithm encounters a component involved in another reconfiguration, one of the reconfiguration routines aborts its execution and restores the previous state of all the PROCs that have been marked for reconfiguration. The routine that gives up is determined based on the identifiers of the hosts (engines) where the two reconfigurations execute.

5.6.6 Discussion

We have described a reconfiguration algorithm that maintains the consistency of individual components during reconfiguration. It also ensures that the end-to-end integrity of the data exchanged between components is not compromised due to chains of interactions between components involved in configuration changes. A minimal set of components is selected and blocked during the reconfiguration to ensure that the above conditions are met. The

additional code for supporting dynamic reconfiguration introduces some runtime overheads. However, most of the mechanisms used for reconfiguration are not active during normal application execution, therefore the performance of the application is not affected.

The reconfiguration algorithm employed by DACIA compares favorably with previous approaches to dynamic reconfiguration, with regard to minimizing the impact of reconfiguration on the application execution, reducing the number of control messages exchanged, and reducing the application programmer's awareness of the reconfiguration.

In the solution proposed by Kramer and Magee (quiescent approach) [56], before the reconfiguration is executed, the components of an application are brought to the *quiescent state*. A quiescent component is not currently involved in transactions[8] with other components and it will not initiate new transactions (*passive state*). Additionally, no other components will initiate transactions with this component in the future. The set of components that need to be either passive or quiescent is not minimal. If dependent transactions are present, all components involved in such a transaction have to become at least passive. This is unnecessary in many cases. Our solution avoids this situation, by making passive only the components situated along a reactive chain between two components that are directly involved in reconfiguration.

A subsequent solution has been proposed in [34] (blocking approach). First the components that should not be involved in any transactions during the reconfiguration are identified (blocking set BSet). When these components are idle (not engaged in any transaction), they are blocked. To ensure that all required components progress to the blocked state, several transitions between the blocked and unblocked state may be needed.

The initial BSet is minimal compared to either the quiescent approach or our approach, and thus it causes less disruption. In many cases though, this set is extended dynamically to account for dependencies between components, i.e., whether the blocking of two components will interfere with each other. For instance, consider the case where A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D form a reactive chain (i.e., execute dependent transactions), A needs to be replaced, and there is a pending message at A. In the blocking approach, all components A, B, C, D need to be blocked. In our approach, only A and B are made passive, while C and D can continue their execution.

[8] a transaction is an exchange of information between two and only two components

Besides increasing the number of components that need to block, each change of the BSet requires notifying all other components about the change. The notification is not only a control message, but it contains the full membership of the BSet, thus increasing the runtime overhead of the algorithm.

Another limitation of the blocking solution is that it does not allow interleaved transactions – while handling a request, a component can not service any new request, even if it comes from a different connection and it is unrelated to the first one. DACIA does not have this limitation. Multiple threads can access an active component. While a thread may be blocked waiting to send a message to a passive component, other threads may proceed with their normal execution.

The quiescent approach requires application programmers to write code that allows a component to reach the passive state and maintain it. In contrast, the blocking approach uses *hooks* that are called when components receive certain control messages (e.g., transaction begin and transaction end). The application programmer only has to mark in the component's code the place where these hooks need to be invoked.

Similarly to the blocking approach, our solution does not require the application programmer to provide any additional code to support dynamic reconfiguration. In our model, the execution of a component often follows the sequence *receiveMessage() – handleMessage() – sendMessage()*, with the message queue operations interposed in the case of asynchronous communication. The application programmer only has to implement the handleMessage() method. The specific code used for reconfiguration is implemented either in the framework (engine) or in the base Proc class, from which all other PROCs inherit. State transitions may occur only between the three stages mentioned above, and not during the execution of a message handling routine.

Table 5.1 summarizes the comparison between the quiescent approach, the blocking approach, and our solution.

The paper by Kramer and Magee briefly mentions that an alternative solution to the reconfiguration problem may be designed by placing emphasis on connections that are established or removed, rather than on components. This approach was actually adopted in [105], and led to reducing the application disruption. At a high-level, this has in fact some similarities with the solution that we proposed, in the sense that a component may be active with respect to one connection, and passive with respect to another. The difference is in the

	Quiescent	**Blocking**	**DACIA**
Disturbance	High	Lowest - High	Low
Interleaved transactions	No	No	Yes
Programmer involvement	Yes	Minimal	No
Centralized	No	No	Yes
Control messages	n^2	n^2	n

Table 5.1: Comparison between the quiescent approach, the blocking approach, and DACIA reconfiguration solution.

fact that in the solution proposed in the latter paper the basic reconfiguration operations are connection creation and connection removal. Our solution is centered around components, not connections, and components are used to specify reconfiguration operations.

The reconfiguration algorithm employed by DACIA can be optimized if the application meets certain restrictions, or if additional knowledge about the application semantics exists. For instance, if there are no cycles of pseudo-passive components waiting for each other to become passive, the algorithm in Figure 5.8 does not need to be executed. Components only need to notify the configuration manager when they become passive. In case of isolated changes that do not affect each other or the execution of previously started interactions between components, if a component is added and/or connections are established, the components that are connected do not need to be passive prior to the change.

5.7 Examples of Runtime Reconfiguration

This section presents some examples of how DACIA applications can adapt to resource constraints and changing application requirements through dynamic reconfiguration.

5.7.1 Example 1: Support for Multiple Architectures

We have previously faced the challenges of building flexible collaborative applications in the context of the UARC [61] project, an experimental test-bed for wide-area scientific collaboratory work. Among the collaborative tools provided, there are several tools for visualizing various real-time or archived data streams. A communication server handles subscriptions from multiple clients and the distribution of data to these clients. The server

receives large amounts of raw data from various data sources, applies some computations (e.g., transforming raw data into GIF images), and then disseminates the resulting data to the clients. Figure 5.9.a. shows the DACIA graph structure corresponding to this application. The server caches the data (the *Store* module) for fault tolerance and for future access.

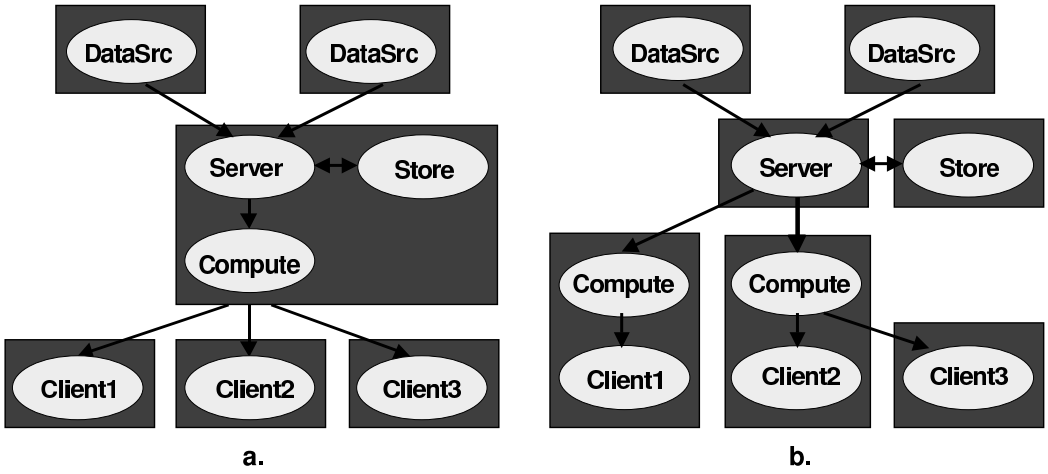

Figure 5.9: Alternative configurations for an application. Ovals represent components. Grey rectangles represent hosts. Components are connected through directed links, indicating the direction of the data flow within the application. Multiple graphs (a-b) may correspond to the same application.

We encountered several problems in using this system. First, the server handles inputs from tens of data sources and subscriptions from hundreds of clients, who can choose to view the data in different ways. Each different view requires a different computation task to run on the server. We found that with a large number of users, the server sometimes ran out of sufficient capacity to compute in real time the images for all the subscriptions. Second, most of the time the computations produce images with bigger size than the size of the raw data. Therefore, the network links from the server to some clients sometimes became congested.

Using an alternative architecture, where the server sends the raw data to the clients and clients do the image computations, can potentially alleviate the above problem. This architecture was in fact tried out in UARC after an expensive code redesign. Unfortunately, the experience was that some clients got overloaded if they computed many images. Since the

system was being used to support scientific collaboration, failure of some clients made group collaboration difficult, making the system seemingly unreliable for group collaboration.

Using DACIA, we simulated an adaptive version of this application. This version allows the computing function to be executed either on the server, on the client, or on any other host with a DACIA engine. The application structure can change at runtime, according to the load and resource availability. Figure 5.9.b. presents an alternative configuration created using DACIA, that uses several *Compute* modules located on the same hosts as the clients or on nearby hosts. The simplified reconfiguration policy that we implemented only took network bandwidth into account. Preliminary performance experiments (see Section 6.2.2) indicated that the system was indeed able to adapt better to network constraints via reconfiguration of *Compute* PROCs.

DACIA also allows additional changes to be applied to the application graph. Data caches can be placed at various points in the network, by introducing *Store* components. The server can store images instead of raw data. In this case, a *Compute* module should be placed between the *Server* and the *Store* module. A pair of *Compress/Decompress* components can be introduced at appropriate points in the data path. Depending on the network topology and on runtime conditions, either one of these configurations can be more efficient than the other ones.

5.7.2 Example 2: Multi-Party Communication

Using DACIA, we have implemented an adaptive multi-party communication application. The application consists of client (C) and server (S) PROCs. Through the servers, a client sends messages to the whole group. A server can be located on a different host than the ones where clients run. Initially, when there are only 2 clients, they are connected directly (Figure 5.10.a.), without using a server. When a third client tries to join a communication group, a server module is spawned, and all the clients will connect to the server and will exchange data through it (Figure 5.10.b.). Assuming that the clients are *C1, C2*, and *C3*, and the server is *S1*, the sequence of operations[9] involved is (arguments are procID's and portNo's):

[9] a negative value for the port number in *connectProcs()* allows to connect to any of the available ports of the specified PROC

```
disconnectProcs(C1, 0)
S1 = new Server()
connectProcs(C1, 0, S1, -1)
connectProcs(C2, 0, S1, -1)
connectProcs(C3, 0, S1, -1)
```

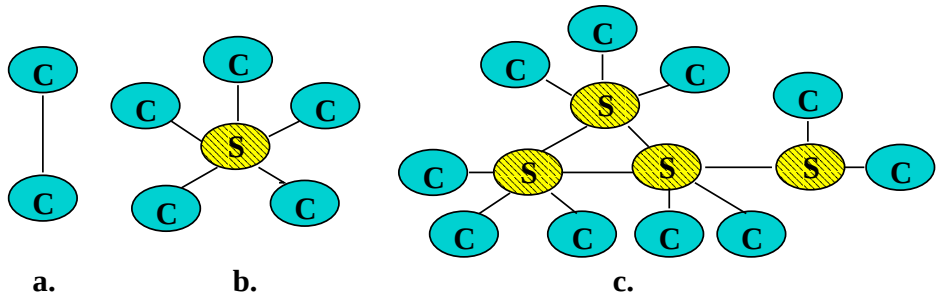

Figure 5.10: Adaptive multi-party communication. Servers are denoted by S, and clients are denoted by C. New servers are created as the number of participants grows.

Various adaptive algorithms can be implemented to allocate and deallocate server modules and to handle clients' distribution. For example, in our implementation, when the number of clients on a server reaches an upper threshold N_{max}, an engine spawns a new server, which connects to the existing servers. The clients are distributed over the two servers. Ideally, the distribution should take into account clients' relative locations. When there is a large number of clients in the group, the application will contain several servers, connected to each other in a certain configuration, with the clients being equally assigned[10] to all the servers (Figure 5.10.c.). Figure 5.11 presents the part of the monitor responsible for allocating new servers and balancing the load among servers.

As clients leave the group, the load per server goes down, and thus it does not justify the usage of too many servers. When the load on a server goes under a lower threshold N_{min}, a server module can be deallocated and its clients are distributed to other servers (code not shown).

DACIA only provides support for ordered delivery of messages along a channel between two PROCs. In multi-party communication, stronger guarantees such as totally ordered

[10] clients' locations are also a factor in choosing an appropriate server

```
// get the list of all the PROCs in the system
procs = Engine.getProcs();
// get the list of all the servers, sorted in decreasing order
servers = procSelect(procs, "server");
while(true) {
   // find the server with the highest load
   s1 = getServer(servers, 1);
   // if the server is overloaded, offload some clients to other servers
   if(s1.load() >= Nmax) {
      // find the server with the lowest load
      s2 = getServer(servers, 0);
      if(s2.load < Nmax-1) {    // can move PROCs to s2
         // find the number of PROCs to move
         moveSize = min((s1.load - Nmax + 1), (Nmax - 1 - s2.load));
      }
      else {    // need to spawn a new server
         // get the list of all the hosts in the system
         hosts = Engine.getHosts();
         // find a host that does not have a server
         h = getFreeHost(hosts, "server");
         // if there is no free host, resume the process later
         if(h == null) {
           sleep(checkInterval);
           continue;
         }
         // start a new server on host h
         s2 = Engine.createProc(h, "server");
         // connect the servers
         connectProcs(s2, 0, s1, -1);
         // add s2 to the list of servers
         servers.addElement(s2);
         // find the number of PROCs to move
         moveSize = s1.load/2;
      }
      // move moveSize PROCs from s1 to s2
      for(i=0; i<moveSize; i++) {
         // get any of the PROCs connected to s1
         c = s1.getConnectedProc();
         disconnectProcs(c, 0);
         // connect the client c to s2, on any available port
         connectProcs(c, 0, s2, -1);
      }
   }
}
```

Figure 5.11: An example of a simple monitor that balances load among servers. When the number of clients on one server reaches the threshold N_{max}, either some clients are assigned to one of the existing servers, if possible, or a new server is spawned to handle the excess clients.

delivery of messages may be required. To provide totally ordered message delivery using the current DACIA, a possible solution is to require that the graph formed by the servers does not have cycles (it is a tree) and one server acts as sequencer for group messages.

In our implementation of the application in Figure 5.10, the servers are stateless. They simply route messages and no consistency of state among the servers is required. If maintaining a group's state at the servers is required, currently the easiest way to do this is to provide a store component that maintains the group's state. In future versions, we plan to provide support for replicating components and maintaining consistency of their states.

The architecture and the adaptive algorithms presented can be used to implement various server-based group communication applications. As proof-of-concept, we implemented a multi-party chat-box application and a shared whiteboard application on top of this group communication service. The application reconfigures itself dynamically to scale up to a large number of clients and to reduce communication latencies.

CHAPTER 6

IMPLEMENTATION AND APPLICATIONS

6.1 Implementation Issues in DACIA

We have implemented the DACIA framework, as well as several applications, in Java [33]. We chose Java because it is currently widely used and it is tightly integrated with World Wide Web-related technologies. Java serialization provides a convenient way of moving both code and data across a network. In many cases, applications written in Java can be moved across hosts and they can execute on different platforms with minimal changes[1]. Moreover, the Java class loader provides mechanisms for retrieving and dynamically linking classes in a running virtual machine.

Our implementation of the framework and the inter-PROC communication mechanisms respects the architecture described in Chapter 3. We implemented a simplified version of the component movement algorithms presented in Chapter 4. The system ensures persistent inter-component connectivity during component moves, and tolerates transient network failures. We have not implemented yet the reconfiguration algorithm presented in Section 5.6. Dynamic reconfiguration is supported at the level of individual reconfiguration operations. All reconfiguration operations listed in Section 5.2 are currently supported.

We built our own message passing mechanism on top of TCP to communicate between engines. Alternatively, other transport mechanisms, such as Java Remote Method Invocation (RMI) [98], could be used. We decided not to use RMI in order to avoid the complexity of writing RMI programs, i.e., write an interface and an implementation for each object,

[1] More significant changes were needed for porting DACIA and DACIA applications to a minimal JVM, e.g., the one for small footprint devices such as PDAs.

register remote objects with the RMI registry, and generate *stubs* and *skeletons* using the *rmic* compiler. Moreover, with RMI a separate network connection may be established between any pair of objects that invoke a method on one another[2]. In DACIA, multiple logical connections between pairs of PROCs are multiplexed over the same network connection. RMI can still be used by particular DACIA components to communicate with non-DACIA applications.

6.1.1 Writing Applications

An important goal for our system has been to enable inexperienced users to build customized applications and write application-specific adaptation modules with only a small programming and configuration effort. We strove to put as much as possible of the functionality common to most DACIA applications into the framework, so that an application's code becomes very simple.

The DACIA *engine* implements all the mechanisms needed to build and reconfigure an application, e.g., creating, connecting, and moving components, maintaining connections between components, and managing message exchange. At the same time, the engine is a general-purpose class. It does not contain any application-specific code. Since every application uses a single engine, we implemented the engine as a static class. It does not have to be instantiated by an application. Its methods are invoked directly as class methods. This eliminates the need to pass references to an engine instance among the many objects in an application that invoke engine's primitives. When a DACIA application starts, the engine is initialized using either default parameters or the values contained in a configuration file (e.g., the file *Engine.config* that comes with the code distribution).

We provide application developers with a comprehensive API that contains primitives for creating and destroying PROCs, connecting applications running on different hosts, connecting and disconnecting PROCs, moving PROCs from one host to another, and registering and starting a monitor. Appendix B contains a comprehensive listing of this programming API. Using this API and assuming that the code for the PROCs (and for a monitor, if applicable) is provided, a simple distributed application can be written using 10-15 lines of code. Many of the primitives in this API are also used to write monitors that implement

[2]If an existing socket is in use by a call, then a new socket is created for a new call. Also, sockets are closed if they are not used for a certain period of time.

application-specific reconfiguration policies. Figure 5.11 presents an example of a simple monitor used by the multi-party communication application to allocate and deallocate servers, and to balance the load among servers.

```
public class DaciaApp {
   public static void main(String args[]) {
      if (args.length < 1) {
         System.out.println("Usage:  java dacia.DaciaApp [configFileName]");
         System.exit(1);
      }
      // initialize the engine
      Engine.init(args[0]);
      // instantiate two PROCs and connect them
      Proc p1 = new Chat();
      Proc p2 = new Forward();
      Engine.addProc(p1);
      Engine.addProc(p2);
      Engine.connectProcs(p1, 0, p2, 1);
      // connect to another engine, running on port 5000
      // the two engines will exchange and update their PROC information
      Engine.connect("AnotherHostName",5000,true);
      // start the command-line application interface - optional
      Engine.runShell();
      // start the graphical interface - optional
      Engine.displayGraph();
      // adds a monitoring routine - optional
      Monitor monitor = new AMonitor();
      Engine.setMonitor(monitor);
      monitor.start();
      // triggers an action on a PROC
      p1.start();
   }
}
```

Figure 6.1: A DACIA application. The application's engine connects to an engine running on another host. Subsequently, connections can be established between local and remote PROCs, using either the programming interface or the user command-line interface (see Figure 6.4).

Figure 6.1 presents a simple DACIA application, consisting of an engine and two PROCs per host. These PROCs, of type *Chat* and *Forward*, respectively, have two ports each. The output of *p1* is connected to the input of *p2*. The engine connects to another engine running on a different host and having two similar PROCs. Subsequently, connections can

be established between PROCs running on the two hosts. A message originating at one of the *Chat* PROCs will be delivered to the other *Chat* through the interposed *Forward* PROC. The message exchange is triggered by calling the *start()* method on *p1*.

6.1.2 Writing PROCs

We designed and implemented the PROC architecture so that it can be easily extended through inheritance, by simply adding component-specific data structures and methods for message handling. At the minimum, a subclass of the base *Proc* class has to implement a pair of methods for message handling, which are abstract in the parent class:

- *void handleMessage(Message msg, int portNo);* - handle a message received synchronously on the specified port.
- *void handleAsyncMessage();* - handle a message received asynchronously. The message is extracted from the message queue. It contains information about the port where it has been received.

For many PROCs, the implementation of these two methods is identical. In this case, a message is processed the same way regardless whether it is received synchronously or asynchronously. The difference concerns the threads that execute the message handling routine, and the decoupling between sender and receiver.

Usually a PROC implementation that subclasses the base *Proc* class contains data structures specific to that PROC. Multiple threads can concurrently execute message handling routines that access this data. The PROC developer is responsible for writing thread safe implementations for the message handling routines, so that race conditions are avoided, and data is not left in an inconsistent state.

We implemented a set of five basic PROCs that can be used to build data distribution services. These PROCs provide services for applying transformations to and filtering the input data, distributing data to multiple destinations, merging and synchronizing multiple input data streams, splitting the items in an input data stream and sending them alternately to multiple destinations. They can be easily extended by implementing in most cases only one method. These PROCs are:

- *Transform* - having one input and one output ports, it applies a transformation to messages received on the input port and sends the resulted data to the output port.

A subclass has to implement the method:

abstract Message handleMessage(Message msg);

Additionally, a *Transform* PROC can do some buffering, maintain state, or apply a transformation to a sequence of input messages, e.g., compute the maximum of tuples of incoming numbers, etc.

- *Filter* - having one input and one output ports, it filters the messages received on the input port and outputs only the messages matching the filter. It does not modify messages. A subclass has to implement the method:

  abstract boolean filterMatched(Message msg);

- *Merge* - merges two or more data streams, and sends the resulted data to the output port. It has two (or more) input ports and one output port. In most cases, it needs to synchronize the input streams and maintain state. A PROC that merges more than two input streams can be constructed by cascading *Merge* PROCs with two inputs. The following method has to be defined in a subclass:

  abstract Message Merge(Message msgs[]);

- *Replicate* - having one input and multiple output ports, it sends the messages received on the input port to all output ports. It can synchronize the rates at which it sends data to multiple output ports. A *Replicate* PROC with more than two outputs can be constructed by cascading several *Replicate* PROCs with two outputs.

- *Select* - splits a data stream into two or more streams going to different destinations. The selection function chooses one out of several output ports based on the content of the message received. A subclass has to implement the method:

  abstract void doSelect(Message msg);

A *Select* PROC can be replaced by one *Replicate* and two *Filter* PROCs. Thus, *Transform*, *Filter*, *Merge* and *Replicate* represent a minimal set of PROCs that can be used to implement any data distribution service.

Starting from the base *Proc* class, each of the above PROCs was implemented using 17-25 lines of code. For example, Figure 6.2 presents the code for the *Filter* PROC.

```
public abstract class Filter extends Proc {
   public Filter(String procName) {
      super(procName, 2); // name, 2 ports
      type = FILTER;
   }

   public void handleMessage(Message msg, int port){
      if(port == 1)
         // error condition: message received on the output port
         System.out.println("Filter " + id +
               " :  synchronous message received on port 1");
      else if(filterMatched(msg))
         output(1, msg, 1);
   }

   public void handleAsyncMessage() {
      Message msg = null;
      while(true) {
         // retrieve a message from the message queue
         msg = getMessage();
         if(msg.getPort() == 1)
           // error condition: message received on the output port
           System.out.println("Filter " + id +
               " :  asynchronous message received on port 1");
         else if(filterMatched(msg))
           output(1, msg, 0);
      }
   }

   // This abstract method should be defined in the subclass.
   public abstract boolean filterMatched(Message msg);
}
```

Figure 6.2: A *Filter* PROC applies a boolean filter to messages received on the input port and outputs only the messages matching the filter. A subclass has to implement the filtering method.

Component mobility is achieved through Java object serialization. In order to be mobile, a PROC should implement the Java *Serializable* interface. Since serialization and de-serialization can be expensive operations, we optimized them by overloading Java's serialization methods in the case of the base *Proc* class. The state of a PROC (including queued up messages) is compacted before it moves and it is restored at the destination. To reduce the size of the serialized object and the time to complete the serialization, references to connected PROCs and their ports are removed and replaced with numeric IDs. Running threads and their state are not serialized, but they are initialized at the destination. Additionally, applications programmers can write specialized routines that minimize the size of the serialized data representing PROC-specific state.

In most cases, the programming effort to transform a Java object into a mobile PROC is modest. It consists of adding a PROC wrapper to the object, implementing message handling routines and eventually writing methods for serializing and de-serializing the state of the object. Explicit serialization code can be more elaborate for complex objects. If the original object implements the *Serializable* interface, explicit serialization is not needed. There is a tradeoff between the programming effort needed to explicitly write serialization and de-serialization routines, and the efficiency gain obtained for PROC move operations. The component developer does not have to write any additional code to support mobility. All the mechanisms required for PROC mobility and persistent connectivity are implemented in the framework, i.e., in the engine and the base *Proc* class.

We used 26 lines of code to transform a Java object for a previously written multi-user Chat program (it included a graphical interface, with menus, input/output text areas, and buttons) into a PROC (Figure 6.3). The Chat PROC has two ports. It displays to an output text area the content of the messages received on the input port, and it sends to the output port the messages typed by a user in an input text area. The *pack()* and *unpack()* methods are used before/after moving the PROC to discard/restore the chat frame. The string *text* contains the state of the *Chat* PROC while it is moving.

6.1.3 Tools for Application Management and Dynamic Reconfiguration

A DACIA application can be reconfigured at runtime either automatically, as a result of a monitor's action, or manually, through commands issued by a user. In addition to the programming API, DACIA provides a command-line shell interface (Figure 6.4) for runtime

```
public class Chat extends Proc {
   ChatFrame frame = null;
   String text = null;

   public Chat() {
      super("Chat", 2); // name, 2 ports
      frame = new ChatFrame(this);
      frame.init();
   }
   public void handleMessage(Message msg, int port) {
      // display the message received in the output window
      frame.displayMessage((String)msg.getData());
   }
   public void handleAsyncMessage() {
      Message msg = null;
      while(true) {
         // retrieve a message from the message queue
         msg = getMessage();
         frame.displayMessage((String)msg.getData());
      }
   }
   // Send to the output port a message typed by the user in the input window
   void sendMessage(String message) {
      Message msg = new Message();
      msg.setData((Object)message);
      output(1, msg, 1);
   }
   public void pack() {
      text = frame.getText();
      frame.quit();
      frame = null;
   }
   public void unpack() {
      frame = new ChatFrame(this);
      frame.init();
      frame.displayText(text);
   }
}
```

Figure 6.3: Code added to a previously existing Java object to make it a mobile PROC, in the case of a *Chat* object.

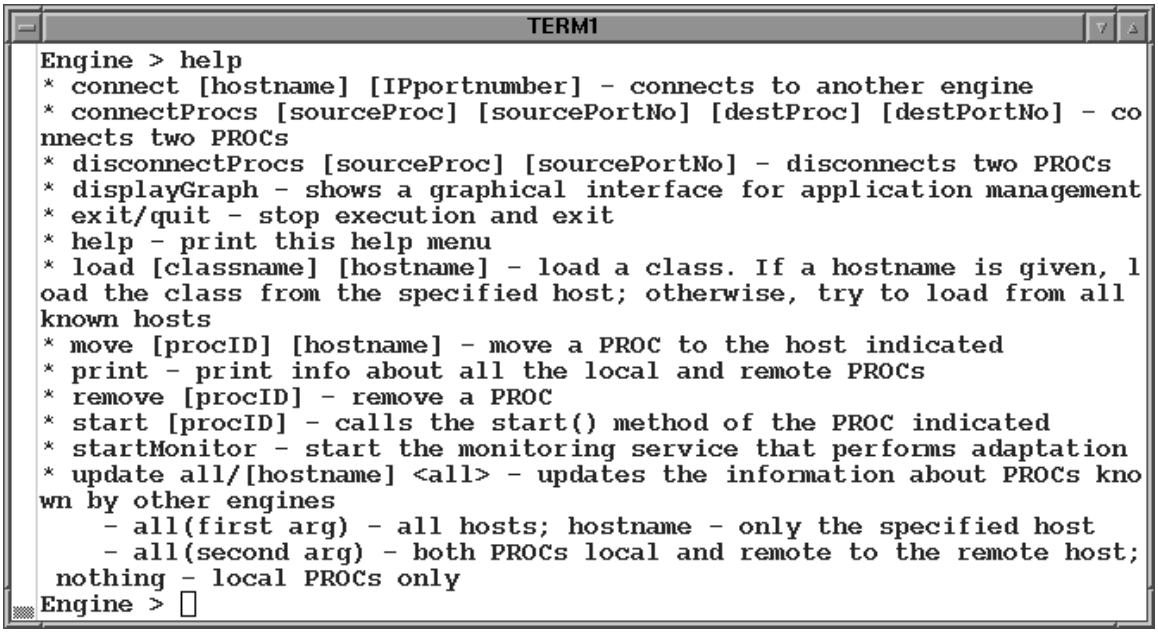

Figure 6.4: The command-line shell interface allows a user or system administrator to visualize in text mode the structure of a distributed application and to manually reconfigure the application.

management of the application. Through this interface, a user or system administrator interacts with the engine running on the local host. She can get information about the structure of the application (local and remote PROCs and their interconnections, and connections between engines), add or remove PROCs, manipulate the connectivity and the location of PROCs, and load and execute a monitor. The user can not access the PROCs directly, but only through the engine. This interface provides almost the same facilities offered by the programming API.

In most cases, the command-line interface is sufficient for experienced users and application developers, who can easily map the textual information to a spatial representation of the application. However, for ordinary users, this mapping might not be obvious. Furthermore, for large-scale applications that contain tens of components at the minimum, textual information about connections between components may be hard to read and understand. A graphical representation of an application is therefore needed.

We have built a graphical tool that provides an interactive environment for visualizing the structure of a distributed application and performing manual reconfiguration of the

application. This graphical interface offers all the functions that are available through the command-line interface. It can be started either during an application's initialization phase, or at any point during the execution of the application, by invoking the *displayGraph* command in the command-line window.

We considered the following requirements for the design and implementation of this graphical interface (GUI):

- **Graphical representation of the application's structure**

  The GUI should accurately and clearly represent an application as a graph of connected components, distributed over multiple hosts. The placement of hosts and components should provide good visibility of the application's structure.

- **Consistency and efficiency**

  The graphical information displayed should be consistent with the actual configuration of the application at all times. It should react to any changes in the application configuration and reflect the up-to-date information about components and links. Moreover, updating the graph should be efficient and should not introduce significant overheads.

- **Expressiveness and simplicity**

  The GUI should provide a rich set of commands that can be used to reconfigure an application. Its functionality should be as complete as the one provided by the command-line interface. The operations available through the graphical interface should be intuitive and easy to use even for a novice user.

- **Separation from the application**

  The use of the graphical tool in an application should be optional. Its goal is not to replace the existing command-line interface, but to enrich the application and to provide users with a wider selection of tools. A graphical display may not be available on certain computing devices, such as PDA's. The GUI should be independent and transparent to the application.

Figure 6.5 displays a DACIA application, as it is represented in the graphical interface. The GUI is divided into two parts. The *graph panel* graphically presents the structure

of the application. The *information panel* shows textual information about PROCs, their interconnections, and connections between engines[3], similar to the information displayed in the command-line interface using the *print* command. The information panel can be closed individually if so desired.

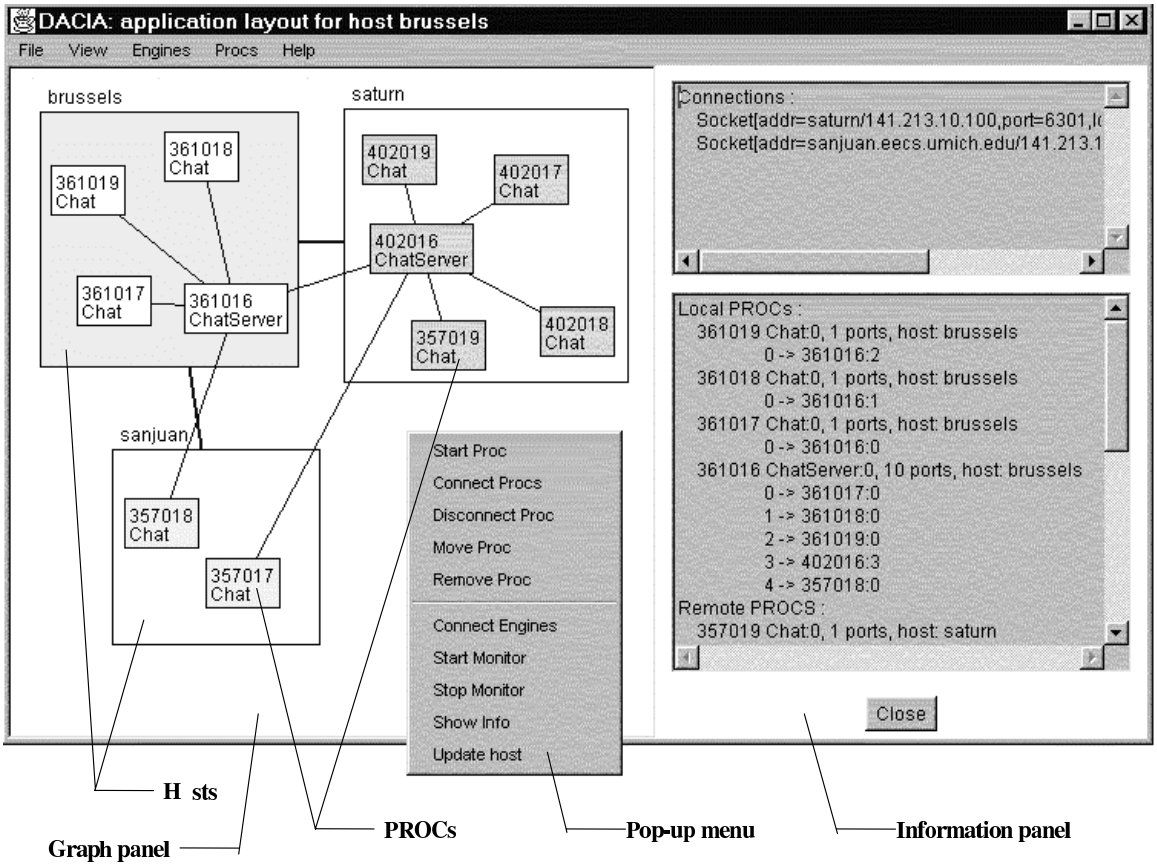

Figure 6.5: The graphical interface (GUI) provides an interactive environment for visualizing the graph structure of a distributed application and performing manual reconfiguration of the application.

The application presented in the figure resides on 3 hosts, represented by the larger rectangles. The local host (brussels) has a slightly darker color than the remote hosts (saturn and sanjuan). PROCs, represented by the smaller rectangles, are identified by their name (in most cases it corresponds to their type) and their unique numeric ID. The graph displays the connections between hosts and the ones between components.

[3]Only connections between the local engine and remote engines are displayed. By default, an engine has no knowledge about connections between remote engines.

Commands for modifying the application structure can be issued by selecting an option from one of several menus available (some of these commands are listed in the pop-up menu displayed in the figure). By right clicking on one of the PROCs or inside one of the hosts, a menu of operations available on the selected PROC, or on the host's engine, respectively, shows up.

Most commands require the user to input certain parameters. To simplify the user's task, the dialog boxes corresponding to various menu options provide drop-down lists of choices instead of input text fields, and only display the valid choices, whenever possible. Some of the inputs are automatically filled in certain situations (e.g., the selection of a PROC's ID and port number if the PROC's *Connect Procs* menu option is selected).

To address the problem of placement of hosts and PROCs inside the graph panel, we looked at work done in the areas of information visualization and graph layout. Similar research work deals with visualization of network topology, BGP routing tables, organization diagrams, and so on [45].

Although the user can manually change the position of hosts and PROCs in the graph, an initial automatic placement of nodes in the viewing window is necessary. The graph layout involves two phases: positioning the boxes that represent hosts and subsequently positioning the nodes that represent PROCs. Hosts are initially placed as vertices of an equilateral polygon. PROCs are randomly positioned inside the boxes representing their hosts. It is assumed that only a few (less than a dozen) hosts are involved in an application and they contain about the same number of PROCs. While a polygon-shaped graph avoids the overlapping of some links, it does not scale well for applications with a large number of hosts. As more hosts are added to the graph, they are placed in the blank area at the bottom of the graph, without affecting the position of existing hosts. When the user selects the *View*$\rightarrow$*Redraw* menu option, the graph is automatically regenerated and the positions of boxes are optimized.

6.2 Performance

In this section, we examine the performance of our current implementation of DACIA. We first investigate the impact of modularity on application performance. We compare the overheads of both local and remote inter-PROC communication with the default communi-

cation costs in the absence of our component-based architecture. We also determine the cost of component mobility in DACIA. Then we give an example of a simple DACIA application that showcases the performance benefits of dynamically reconfiguring an application.

6.2.1 Communication Overhead

To get the advantages of modularity and component mobility without a significant hit on performance, our framework attempts to keep communication overheads low when PROCs are co-located. When a remote PROC is relocated to the same host as a PROC it is connected to, the two PROCs are attached into the same address space (Figure 6.6). Co-located PROCs can thus exchange data with low overheads, only slightly higher than local procedure calls within the same address space. In the case of asynchronous communication, the cost of thread scheduling and queue management is added. For PROCs exchanging messages frequently, the cost of communication can potentially be reduced if the PROCs are co-located. Conversely, two PROCs doing some CPU-intensive processing without much interaction with each other may execute more efficiently if they are located on different hosts.

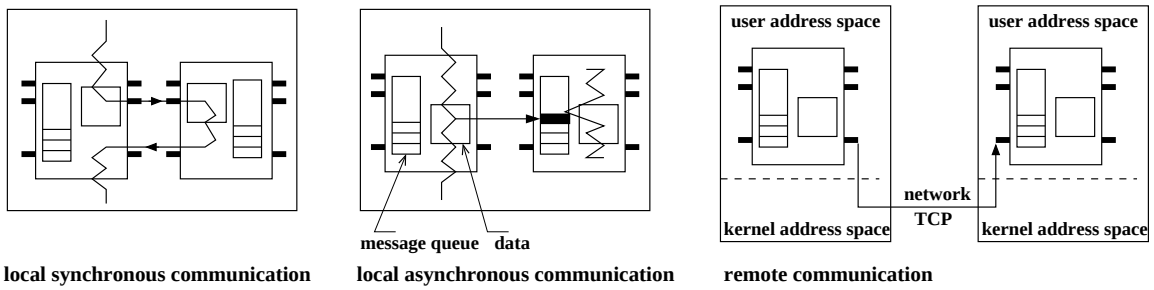

Figure 6.6: Inter-PROC communication. When the PROCs are located on the same host, they are in the same address space and message exchange translates into simple procedure calls. If they communicate asynchronously, the cost of thread scheduling and queue management is added. When the PROCs are located on different hosts, the communication overhead increases due to the cost of network communication and crossing user-kernel boundaries.

To determine the overhead of using our modular framework to execute an application, we compared the performance of inter-PROC communication with the cost of procedure calls in the local case, and with the cost of raw TCP in Java, both in the local and remote cases. Using a small application consisting of two PROCs, we determined the time needed for one PROC to send a message to the other PROC and receive a reply message, for the

cases in which the PROCs are located on the same host (synchronous and asynchronous communication - columns 2 and 3 in Table 6.1, top, respectively) or on different hosts (column 2 bottom table). We compared the results with the cost of one local null procedure call (column 4 top), local round-trip data exchange using TCP, both for a message (serializable object - column 5) and for a byte buffer (column 6), and remote request-reply using TCP, both for a message (column 3 bottom) and for a byte buffer (column 4 bottom).

message size (bytes)	local PROCs synchronous	local PROCs asynchronous	local procedure call	local TCP message	local TCP byte[ ]
0	1.1	9.8	.18	477	332
1000	1.1	9.8	.18	6867	431

message size (bytes)	remote PROCs	remote TCP message	remote TCP byte[ ]
0	2186	522	375
1000	5226	7232	526

Table 6.1: Latencies (in μseconds) for round-trip inter-PROC communication and raw TCP, for local (top) and remote (bottom) communication, for a null message and for a message of size 1000 bytes. Each data point has been obtained by averaging over 10000 or more messages.

For this experiments, we used a set of Pentium III machines with 733 MHz CPU and 256 MB memory, connected by a 100 Mbps switched Ethernet network. All the code used for testing was written in Java. We repeated the experiments for a null DACIA message (only a message header is sent), and for a message carrying 1KB of data.

The cost of both synchronous and asynchronous local communication is significantly lower than the cost of remote communication, using either PROCs or raw TCP. The size of the messages exchanged does not affect the latency in the case of local inter-PROC communication, since object references are passed through procedure calls and data is not actually copied. The cost of synchronous communication (.55 μs one way) is comparable to the cost of a few local procedure calls (.18 μs for 1 null procedure calls). In the asynchronous case, the cost of switching threads and message queue management is added. Even in this case, a message exchange between two PROCs takes only a few microseconds (4.9 μs), which is much less than in the case where TCP is used to exchange data between two co-located components. We use local TCP for comparison to give an idea of what the communication

costs would be if a uniform communication protocol such as TCP was used for both local and remote communication in a default implementation of a modular architecture.

The overhead of remote communication between PROCs is similar to the one of exchanging messages using raw TCP[4]. In fact, in the case of a 1KB message, inter-PROC communication is faster (5226 μs) than raw TCP message exchange (7232 μs). This is due to the fact that our solution for message exchange first writes the serialized message to a local byte buffer, and then writes the byte buffer to the socket, as opposed to doing the serialized write operation directly on the output stream corresponding to the socket. For a 1kB message, it takes 1154 μs from the moment a PROC invokes a message send operation until the message is actually written to the socket. At the same time, simply writing the serialized message to a local stream takes 418 μs.

Object serialization is the most expensive component of the communication overhead in the case of remote communication. Serialized message exchange is significantly higher then sending an equal amount of data as a byte buffer. The overhead of remote communication can grow significantly with the message size if Java object serialization is used to flatten the data objects in a message. The performance of remote communication can be improved by avoiding, whenever possible, serialized read/write operations.

Using the same experimental setup as above, we measured the time needed to move a simple PROC between two hosts. The average value obtained for a small PROC (the size of the serialized PROC is 725 bytes) is 4.4 ms. This increases if the state size is larger, due to the cost of object serialization. Out of the total time, it takes about 1.4 ms to serialize the PROC and to write it to the socket.

We also determined the impact of using DACIA on the communication throughput. Using the experimental setup described above, we measured the throughput of message exchange between two PROCs located on different hosts. The maximum throughput obtained is 6.87 Mbps. The maximum throughput obtained for a simple TCP connection between the same machines (we sent data as a byte[ ] using a Java implementation) is 32 Mbps. The throughput of local communication is 1,810,000 messages/s for synchronous communication and 204,000 messages/s for asynchronous communication, regardless of the message size (the data is passed by reference).

[4]messages are serialized in this case

An adaptive application can take advantage of the differences in communication performance, by relocating and co-locating PROCs based on the frequency and the amount of data exchanged between them. Overall, the results presented show that the benefit of co-locating remote PROCs that exchange messages frequently can outweigh the overhead of using our framework instead of simple TCP to communicate across multiple hosts. Moreover, the cost of moving PROCs across hosts can be kept low.

6.2.2 Impact of Relocation and Reconfiguration on Application Performance

To show basic component interaction and the benefits of application reconfiguration and component relocation, we used DACIA to implement the test case presented in Section 5.7.1 (Figure 6.7). A communication server receives multiple streams of raw data from various data sources, it applies some computations, and then it disseminates the resulted data to various clients. Since different clients subscribe to different data, the *Compute* PROC executes a separate computation for each request.

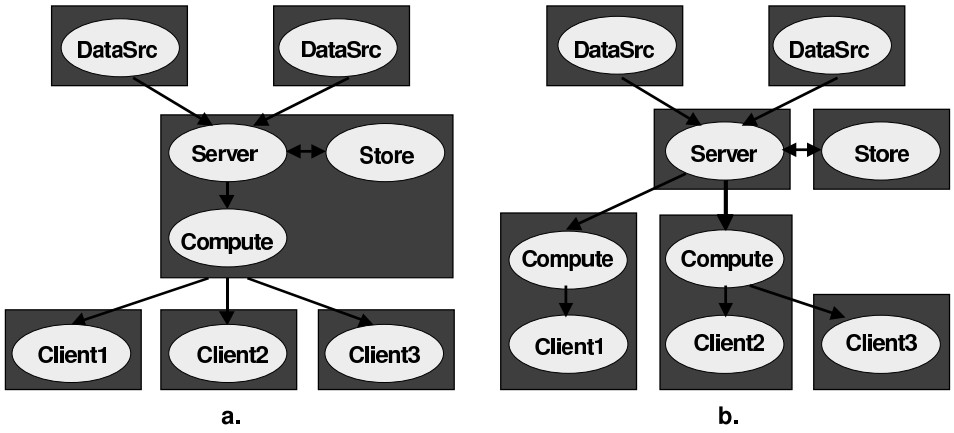

Figure 6.7: A communication server receives raw data from various data sources, applies some computations, and then disseminates the resulted data to various clients. The computations can migrate from the server machine (a) to the client machines (b) and back during the execution of the application.

We determined the average time to complete a client request, from the moment the client submits the request, until it receives the data. In this experiment, two clients send requests concurrently. We repeated the experiment for various sizes of the raw data requested (for

brevity, only the results for 8 KB and 16 KB are presented here), for the cases when the *Compute* PROC is either co-located with the *Server* or it is replicated and co-located with the *Clients*. We simulated a high capacity (approximately 800 KBps), and a low capacity (approximately 60 KBps) network between the server and a client. The time needed to execute the computation function depends on the size of the raw data and the characteristics of the machine. On a fast server machine, the computation takes about 5 ms for each KB of data. On a slow client machine, it takes about 15 ms for each KB of data. The size of the data produced by the *Compute* module is usually about twice the size of the raw data.

The results of the experiment[5], presented in Figure 6.8, show that adaptability and application reconfiguration can improve the performance of the application. In some cases, executing computations on the server is more efficient, while in other cases the response time is smaller if computations run on client machines. The size of the data affects both the time needed to compute images and the time it takes to move data over the network. For high bandwidth, the latter factor is less important and the computing time represents the bigger fraction of the total time to serve the request. In this case, it is more efficient to compute images on the fast server and then send the data (with increased size) over the fast network. When the bandwidth is low, it is better to send the raw data (with smaller size) over the network, and compute the images on the clients.

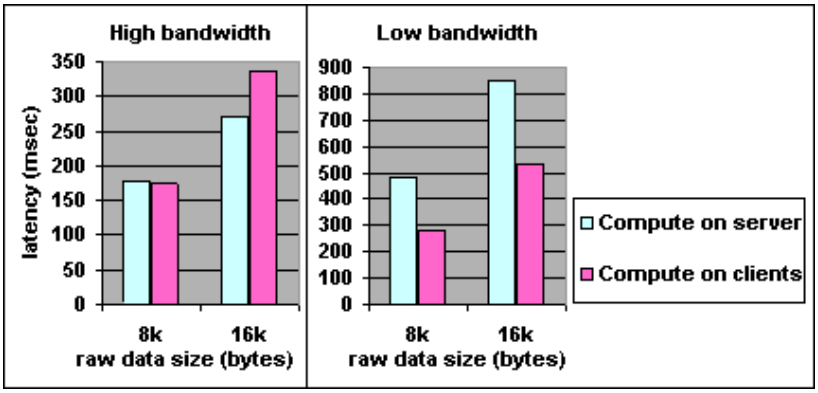

Figure 6.8: Average time to serve concurrent requests from two clients, for the cases where the bandwidth is high or low, and the Compute module is co-located either with the Server or the Clients. Clients can run either on fast or slow machines.

Resource consumption (e.g., bandwidth, CPU) and the application performance can be

[5]Each data point was obtained by averaging over 100 requests.

optimized by moving PROCs around according to the structure of the application, resource availability, and the pattern of communication between various PROCs. With DACIA, a system administrator or a specialized monitoring function can use the measured performance of the application or some heuristics to make decisions for relocating components and reconfiguring the application, in order to obtain a more efficient computation.

6.3 Applications

We used DACIA to develop several applications that showcase some of the features of DACIA, such as component mobility, runtime reconfiguration, and application parking.

Using the adaptive multi-party communication architecture presented in Section 5.7.2, we implemented two collaborative applications [65, 63]: a chat-box and a shared whiteboard. Both applications use the same Server PROCs and the same monitor. Only the client PROCs are different. If only two users are present, their client applications can be connected directly and the existence of Server PROCs is not necessary. In order to reduce communication latencies, these applications reconfigure themselves dynamically by introducing, removing, or relocating servers, and appropriately changing clients' allocation to various servers based on the number of clients and their locations.

Two showcase the performance benefits that can be obtained using our framework, we implemented an adaptive application that allows to move functionality between data servers and clients. The performance of an application can be improved by taking into consideration the computing resources available both on client and server machines, as well as the load placed by various clients on servers.

We also implemented a mobile web proxy that allows DACIA applications to communicate with web servers and access data residing on web sites. The proxy can be composed with data filters to build various data services. It also provides support for mobile web clients, that can start a web transaction from one host, and continue it on a different host.

We are in the process of porting the DistView toolkit [60, 62] to DACIA, by transforming the existing parts of the system into mobile PROCs. Our goal is to enhance the adaptability and reconfigurability of the existing system, and to add support for mobility.

We are also working on a PDA implementation of DACIA. Currently, we have a partial implementation of the system for PDAs that support the Java 2 Platform, Micro Edition

(J2ME) [97]. There is also ongoing work on porting DACIA to VisualAge Micro Edition [48]. This virtual machine will support applications that execute both under Palm OS and under Windows CE.

6.3.1 Chat-box

The multi-user chat-box application allows a group of users to exchange text messages. It provides an editing area for composing messages and a scrollable area for displaying a list of received messages (Figure 6.9). A message sent by one user will be distributed to the whole group through the servers. Servers do not maintain state for the messages exchanged. They simply route messages and no consistency of state among the servers is required. The application only ensures the FIFO delivery of messages between any two clients. Messages originating at different clients may arrive in any order to different destinations.

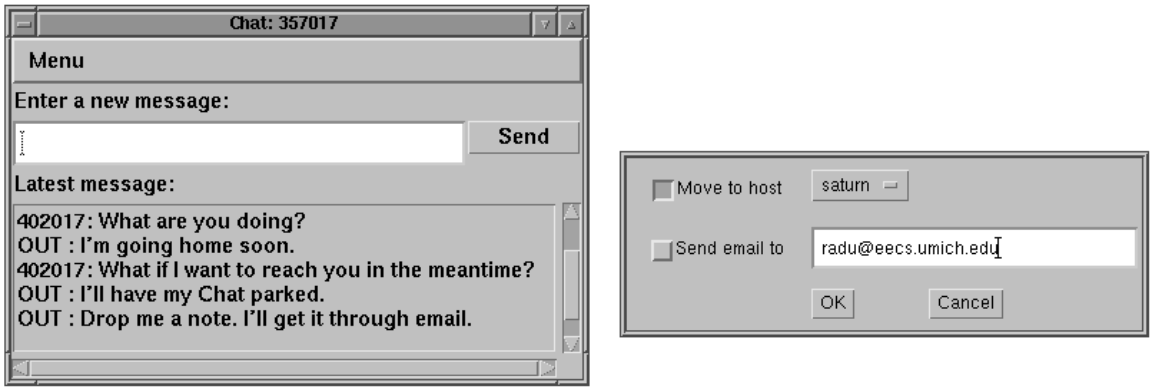

Figure 6.9: The multi-user chat-box application allows users to exchange text messages. A parked Chat client can be moved to a parking host (the right image represents the parking dialog). It can send email notifications to its user when messages are received.

Chat PROCs are mobile. When a Chat PROC moves to a different host, the state of the PROC is transferred as the list of text messages previously received. The frame of the chat client is not moved. Instead, it is initialized at the destination using default parameters. Thus the amount of state that needs to be serialized is reduced.

A Chat user can park her application while the user is not active or she is disconnected. A parked Chat client can reside on the same host the user had been previously connected from, or it can move to a *parking host* if the user's device is disconnected. While the Chat client is parked, it it still connected to a server, it can receive messages from other clients,

and update its internal state based on these messages. If desired, the user can set an email address where she can receive notifications from the parked chat when messages are received. The handling of messages received while a Chat is parked can be further improved. For instance, the parked Chat can filter the messages received based on their sender or priority, it can selectively send notifications to the user, or send some predefined replies to certain messages.

6.3.2 Whiteboard

Acting both as a shared notebook and a drawing board, the whiteboard (Figure 6.10) allows users to collaboratively draw figures, take notes, and import and share images. The basic drawing elements are line, point, and text. Raster images (e.g., .gif, .jpg) can be loaded from the local file system as the background.

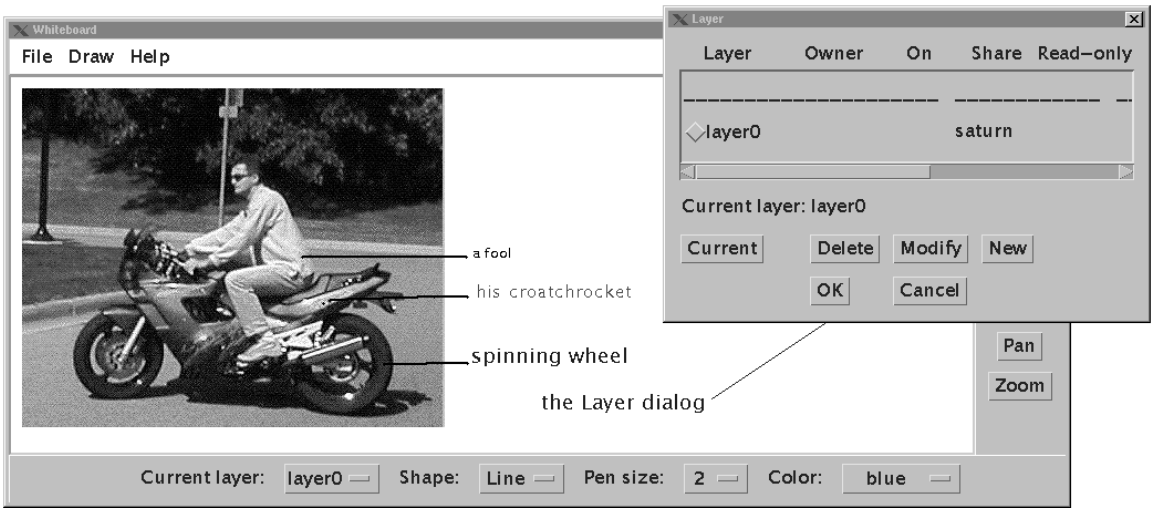

Figure 6.10: The shared whiteboard enables users to collaboratively draw figures, take notes, and import and share images. An image consists of multiple layers, that can belong to different users. The owner of a layer can set its visibility, shareability, and writability properties.

The graphical information is organized into *layers*. A user can create her own layers and send one or multiple layers to a remote whiteboard. The image displayed on the canvas contains multiple layers, overlapped in a particular order. Each layer has a *name* and an *owner*. The name is globally unique. The owner of a layer is the host where it was created. Each layer has three additional attributes: *visibility*, *shareability*, and *writability*. A user

can turn on/off a layer, make the layer shared or private, or make it read-only. Layers owned by other users are always read-only.

Currently, a whiteboard has two duplex ports, thus it can directly connect to two other peer whiteboards. When the *Send* command is invoked, the message is sent to both ports, if they are connected. If more than two connections are necessary, Server PROCs can be used to allow multiple whiteboard users to collaborate.

Whiteboard PROCs are mobile. The serialized state of the moving PROC contains all the data in all layers displayed by the whiteboard. Imported images are managed as bitmaps. Drawings and text are managed as objects, potentially reducing the size of the serialized state.

6.3.3 Adaptive Data Processing

FlexiImage is an adaptive client-server application that implements an image service. It allows data computations to be moved between client and server machines. Through the client application, a user requests various images from a data server. The server retrieves binary data from the disk, calculates an image based on this data, then it sends the image data to the client, which displays it.

Although this is not implemented yet, our goal is to incorporate this functionality into a web-based service. To do this, the *Server* and *Compute* components should run on the back-end of a web server, and the *Client* should be executed through a web browser, either as a plugin or as an applet (Figure 6.11.a.).

After receiving the original image, the user can request certain transformations to be applied to the image (e.g., rotation, scaling). For each such operation, a request is sent to the server. All image transformations are executed by the *Compute* component, then the data is sent to the client.

Executing all computations on the server may be inefficient due to the overhead of data transfers over the network, as well as the server overload when multiple clients request images simultaneously. To improve the efficiency of the application execution, in some cases the computations can be executed on the client machine (Figure 6.11.b.). The data can be cached on the client when an image is first accessed and all subsequent transformations are applied locally.

Our application makes decisions automatically over where to execute the computations,

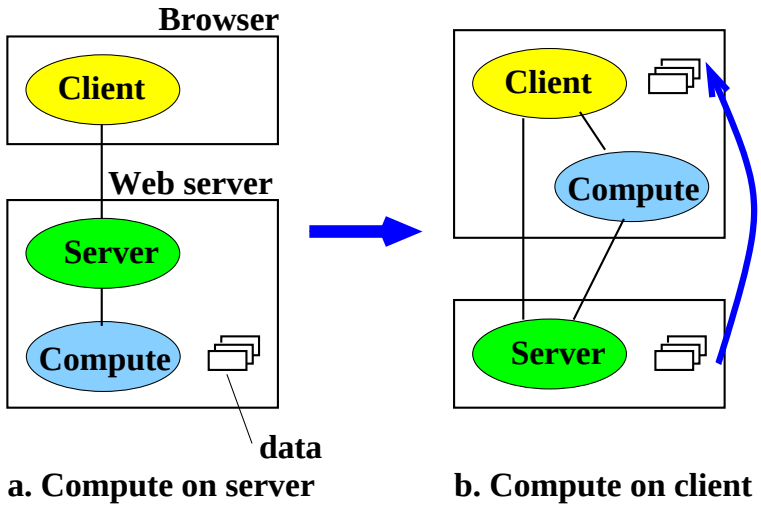

Figure 6.11: FlexiImage implements an adaptive image service that allows data computations to be moved between client and server machines. The location where image transformations are executed is chosen based on the relative speeds of the client and server machines and the load on the server machine.

based on the relative speeds of the client and server machines and the load on the server machine (the number of client requests currently served). To simulate faster or slower machines, the user can set the efficiency of the client machine. If the efficiency is 100%, the image computation on the client will be executed at normal speed. If the efficiency is 50%, the image computation is done twice as slowly, by artificially introducing delays.

The monitor decides where to execute the computation based on the evaluation of the condition:

$SC\_ratio/NumClients > Efficiency$

SC_ratio is the base speed ratio between server and client machines. By default, it is set to 1 for machines of the same type. NumClients represents the load on the server. Efficiency is the simulated efficiency of the client machine. If the above condition evaluates to true, then the image computations are executed on the server. Otherwise, the computations are executed on the client machine.

To assess the efficiency of adaptively determining where to execute computations and moving computations between clients and server, we repeated the execution of this application for the cases when (a) all computations are executed on the client machines, (b) all computations are executed on the server, and (c) the location where computations are

executed is determined at runtime. For this experiment, we used a set of Sun Ultra 10-440 machines with 256 MB memory, connected by a 10 Mbps network. Client efficiency is set to 50%. The image transformations executed are scaling at 1.5 ratio and rotation with 20 degrees. The size of the original JPEG picture used is 7969 bytes (160x200 pixels). Two clients send requests concurrently to the same server.

	Adaptive computation	Client computation	Server computation
Latency	12,395	13,880	18,153
Efficiency loss		11.98%	46.45%

Table 6.2: The cumulated time (in ms) necessary to execute two concurrent client requests. An efficiency gain is obtained by dynamically moving computations between clients and server (column 2) over the cases when all computations are executed on the clients (column 3) or all computations are executed on the server (column 4).

Table 6.2 presents the results of this experiment. The values represent the cumulated average time (in milliseconds) necessary to complete requests coming from the two clients. Each data point has been obtained by averaging over 10 measurements. The adaptive processing case performs better than both the client side (11.98% efficiency loss) and the server side processing (46.45% efficiency loss).

6.3.4 Web Proxy

We implemented a mobile client-side web proxy that acts as an *HTTP gateway* (see Section 3.5). The proxy allows DACIA applications to interact with web servers and access data residing on web sites. It works as an adaptor for web data. It talks the DACIA communication protocol on one side and HTTP on the other side. On the client side, the proxy can be accessed using either a common web browser or a customized PROC. We implemented a DACIA web client that can be used to display HTML data. Alternatively, other DACIA clients can only display data in text format.

The proxy only has general-purpose functionality. It does not interpret data received from web servers. A proxy can be composed with data filters to build specific data services (Figure 6.12). Some filters can extract text data from HTML pages, eliminating embedded objects (e.g., images) and links. Other filters can do data aggregation and analysis over time, or monitor web data for the occurrence of certain events (similar to web bots).

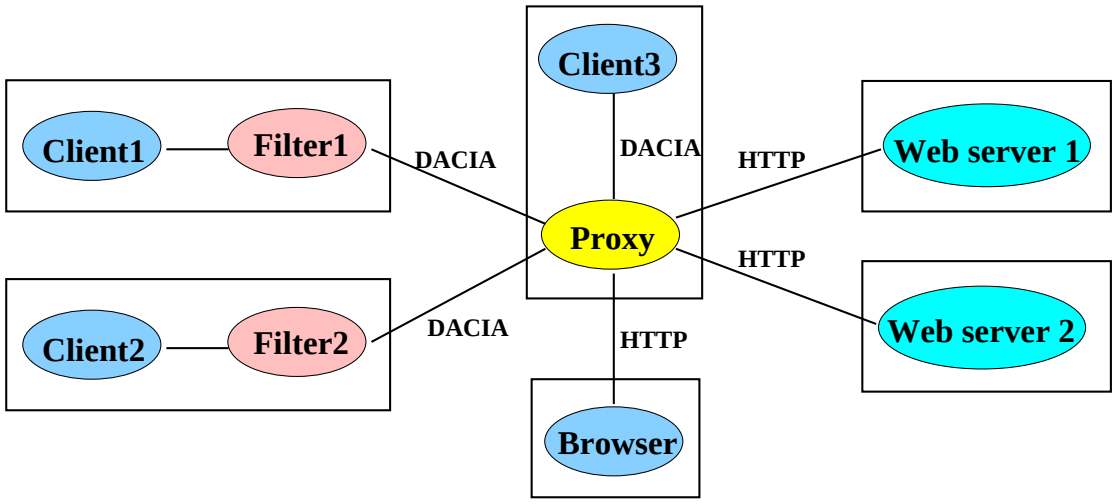

Figure 6.12: The mobile web proxy allows DACIA applications, as well as common web browsers, to interact with web servers. It can be composed with various filters (agents) to build specific data services. It can communicate using either HTTP or the DACIA protocol. It can be accessed using either a common web browser or a customized PROC.

A client-side application can be split into interface components and active components that maintain state and perform data processing. Filters act as users' agents. Through the Client PROC, a user connects to the agent, retrieves a list of services provided by the agent, and selects the desired service. Based on the service selection, the interface presented by the Client can change, in order to allow the user to specify preferences for the required information.

After the user chooses the information to be monitored, she can eventually disconnect from the agent and reconnect later, potentially from a different place, using a different type of Client PROC. The agent monitors the requested data, aggregates some information, and accumulates state. The information the user requested can be accumulated over time and delivered to the user at specific moments (push mode), or on request (pull mode). In certain cases, the user may want to be informed of the occurrence of specific events. If the user is disconnected when the event occurs, the agent will inform the user after she reconnects.

Proxies, filters, and clients are all mobile components. In many cases, clients are stateless. The state of a user session is maintained by the corresponding agent (filter). When a user is present, usually the agent is located on the same host. When the user is disconnected, the filter acts as a parked agent, and can eventually be moved to a different host.

We developed an agent for monitoring weather information. Through the proxy, the agent accesses weather information at *http://www.cnn.com/WEATHER*. The user can specify the city (cities) for which she wants the weather monitored, as well as certain events she wants to be informed about (e.g., the temperature reaching a certain high/low threshold, or the starting of rain). The agent filters the web data and only presents the relevant information to the user. If the user disconnects and later reconnects from a different place, the agent still maintains her preferences. The occurrence of certain events is cached while the user is disconnected, and she will be informed as soon as she reconnects.

An extended version of the web proxy allows mobile web clients (common web browsers or DACIA components) to start a web transaction from one host and continue it on a different host. For instance, consider the process of registering a user to a web site. Usually this requires the user to go through several screens and fill out various forms. Assume that the user has to stop at some point in the middle of filling out the forms (needs to leave the room, or lacks some information). The user can resume the process later, eventually from a different location.

As a user fills out various forms on a web site, some information about the transaction is passed back to the user. Usually this information is stored by the browser in the form of a cookie. In our case, the proxy caches all the state information for the transaction between the web server and the web client. The client is used only for user interaction. It does not maintain any state. If the proxy is fixed, the fact that the user moves to a different location is transparent to the web site. If the proxy moves, it has to transfer its state to the new location, resume communication with the web server, and eventually replay the previous data exchanges. In this way, the user only sees the information returned by the web server as a result of the last step of the transaction.

A user can access the history of interactions between the proxy and various web sites, on behalf of the user. The following commands are available for navigating the information cached by the proxy[6]:

- *http://!history/* - retrieve the history of pages accessed.
- *http://!read/#* - reload a specific page from the history. # specifies the index of the page in the history list.

[6]These commands are given as URLs, so that they can be accessed from existing web browsers.

- *http://!prev* - retrieve the previous page in the history.
- *http://!next* - retrieve the next page in the history.
- *http://!clear* - clear the history. All cached data is discarded.

A potential extension of this web proxy consists of adding support for collaborative web browsing. Assume that two web browsers use the same proxy to access some web pages. When one browser sends a request (HTTP GET), the proxy sends the web server reply to both browsers. Thus each browser will mirror the other one's actions. The proxy can move from one place to another, based on the clients' locations.

CHAPTER 7

CONCLUSIONS

7.1 Summary

This dissertation addresses the challenges of constructing flexible distributed applications and services that can adapt to variations in resource availability and application requirements, and to user mobility. The main contribution of this work is a mobile component framework, named DACIA, for building and executing reconfigurable distributed applications. Modular applications developed using DACIA can change their structure at runtime. They can dynamically load new components, change the way various components interact and exchange data, move components from one host to another, and replicate some components across multiple hosts.

There are several benefits of dynamic reconfiguration. A more efficient execution can be achieved by changing the way different parts of a distributed application interact and their location of execution, thus taking advantage of the resources available system-wide. The cost of maintaining and upgrading existing applications is reduced by eliminating the need to stop and restart applications during maintenance operations. Runtime application composition and component mobility allow mobile users to access applications using a variety of heterogeneous devices, and to move applications between these devices.

Chapter 3 presents the architecture of our framework, as well as the programming model used to build DACIA applications. An application is seen as a graph of connected components. The same application can be built in multiple ways, either by configuring differently the same set of components or by using different sets of components. Moreover,

the application can dynamically evolve from one configuration to another, without the need to be re-compiled or to stop its execution.

In Chapter 5, we identify the issues involved in the runtime reconfiguration of applications. We propose an algorithm for performing dynamic reconfiguration while preserving the consistency and correctness of the application and minimizing the application disturbance (Section 5.6). This algorithm executes in linear time with respect to the size of the application graph, and graciously handles failures that occur during the reconfiguration.

Dynamic reconfiguration is primarily achieved through the execution of adaptive functions implemented by monitors. A monitor represents the policy layer in a DACIA application. It monitors the application performance and makes reconfiguration decisions accordingly. The automated execution of adaptive monitors is complemented by the intervention of system administrators or users. DACIA provides a command-line interface and a graphical interface for runtime management of the application. Using these interfaces, presented in Section 6.1.3, users can manually issue commands to reconfigure the application.

Chapter 4 details the support component mobility in DACIA provides for mobile users and mobile applications. Logical connections between moving components are persistent. Messages are reliably and orderly delivered during and after component relocation. At the same time, the execution of components connected to the moving component is not affected.

Mobile users can move applications or parts of applications from one host to another, while maintaining seamless communication connectivity with other applications. At their new location, the applications continue their execution from where they left off. Users do not see any interruptions in the services accessed, and they do not need to manually re-establish connections with the communication parties. In Section 4.6, we show how component mobility and persistent connectivity can be applied to groupware applications, through application parking, to allow applications to participate to collaborations on the user's behalf, while the user is disconnected or is not active.

In Chapter 6, we give several examples of applications implemented using DACIA, that demonstrate the benefits of using our framework, both from the performance and from the usability standpoint. The experimental results presented in Sections 6.2.2 and 6.3.3 show that by using simple adaptive heuristics to reconfigure an application, the performance of the application can be significantly improved compared to the static case in which the

application structure is fixed.

The use of our modular architecture, as opposed to a monolithic approach, does not introduce significant overheads. In Section 6.2.1, we show that the costs of inter-component communication in DACIA are low. When components are on the same host, they are located in the same address space, and message exchange translates into simple procedure calls. Experimental results show that the cost of local communication equals the cost of a few procedure calls. The cost of remote communication in DACIA is close to the cost of raw message exchange using TCP. The performance of remote communication can be further improved by avoiding, whenever possible, serialized read/write operations. The benefit of co-locating remote components that exchange messages frequently can outweigh the overhead of using our framework instead of simple TCP to communicate across multiple hosts. Moreover, the cost of moving components across hosts can be kept low.

7.2 Future Work

The work presented in this dissertation leads to exciting new possibilities for future work in the areas of dynamic application reconfiguration and application mobility. Following we outline some of the topics that need to be addressed in the future:

- **Security infrastructure**

  Deploying mobile code raises important security concerns. A security infrastructure must weigh requirements of both the component owner and executing hosts. If the host environment is not protected, it may be exposed to a number of vulnerabilities caused by malicious components or programming errors. Thus, without proper precaution, a component can compromise system services and resources, potentially exposing sensitive data and enabling future attacks (e.g., through viruses or Trojan horses). Conversely, if the component is not protected, malicious environments may cause incorrect execution or expose private data and algorithms. In the extreme case, a malicious environment which modifies a component can compromise the hosts to which the component is later migrated.

- **Location service**

  Mechanisms for locating applications and components are needed. In our current

implementation, an application finds applications running on other hosts based on information hard-coded or read from a configuration file during initialization. When a component moves, the engine where the component was previously located sends notifications about the new location of the component. It also acts as the reference point for future inquiries. The alternative is to implement a location service (something in the sense of the Jini [99] lookup service) with which applications and individual components register, and which can provide at any moment correct information about components' location or about currently executing applications.

- **Policies and algorithms for dynamic reconfiguration**

  The DACIA framework provides only the mechanisms for performing application reconfiguration. Currently, application-specific reconfiguration policies are implemented by application developers as monitors. These monitors usually work only for the applications they were written for. A set of general-purpose adaptive policies is needed to enable, in certain cases, the optimization of application performance without semantic knowledge about the application. Such a policy may govern, for instance, the location of components based on the patterns of interaction and the amount of data exchanged between them, and the availability of computing and network resources.

- **Formal specification of components**

  The existing system can be extend with a formal framework for specifying components, their properties, interactions between components and rules for composing components and for defining equivalent composition schemes. The existing programming API and the command-line interface can be used as a starting point in the development of a configuration language for DACIA applications.

- **Deployment and experimental evaluation**

  More extensive deployment and experimentation is needed to assess the usefulness and ease of use of our mobile component framework to build and execute adaptive distributed applications. We will complete the porting of the DistView toolkit [60] to DACIA. We will also continue the ongoing work on the PDA implementation of DACIA using the Java 2 Platform, Micro Edition (J2ME) [97], and VisualAge Micro Edition [48]. One of the challenges of building hardware-dependent components that

can migrate across different types of devices lies in the efficient implementation of code that runs on multiple devices. One solution is to provide all the functionality needed in one single piece of code that can identify the device type and run everywhere. The downside in this case is that the implementation of a component is large and it often contains unneeded code. The alternative is to write multiple versions of the code for the same component, each of whom is minimal and provides only the functionality required by a particular device.

APPENDIX A

DACIA USER GUIDE

This section serves as a quick guide to compiling and installing DACIA, and running some simple applications that come with the DACIA distribution.

A.1 Installing DACIA

The DACIA code can be downloaded from
http://www.eecs.umich.edu/~radu/dacia/download.html.

You can download either the source code or the binary distribution.

To compile and execute DACIA applications, you need to have a Java Virtual Machine (JVM) version 1.2 or newer installed on your machine.

Assume that the *dacia.tar* archive you downloaded is placed in your /tmp directory. Unpack the archive:

tar xf dacia.tar

A directory /tmp/dacia/ will be created, which contains the dacia distribution. The code distribution contains the following packages:

- *dacia* - main package; contains the code for the DACIA framework
- *dacia.apps* - contains several applications, as follows:
  - *dacia.apps.chat* - the chat-box application;
  - *dacia.apps.wb* - the whiteboard application;
  - *dacia.apps.proxy* - a web proxy used by mobile web clients to start a web transaction from one host and resume it from a different host;
  - *dacia.apps.webclient* and *dacia.apps.webproxy* - another proxy implementation, used for filtering data obtained from web servers and building various data services. *webclient* contains the implementation for the weather monitoring agent.

Set your CLASSPATH environment variable to include the place where the *dacia* directory is placed (*/tmp* in this case).

To compile DACIA, run *make* in the main directory with one of the following arguments[1]:

[1] take a look at the *Makefile* to see what each argument does, and eventually modify this file if needed

- no argument - compiles only the main dacia package;
- *all* - compile all packages, including sub-packages;
- *clean* - remove class files and backup files created by emacs (files with names of the form *filename~*);
- *cleanall* - removes class files and backup files, including subdirectories.

Alternatively, you can run *make* only in a subdirectory.

A.2 Executing DACIA applications

Assuming that you have all binaries in place, you are now ready to execute some of the simple test applications provided with the code distribution. To run an example, use a command line such as:

java dacia.Main /tmp/dacia/Engine.config

or

java dacia.apps.chat.Main /tmp/dacia/chat/Engine.config <number of clients>

The main program gets as an argument the configuration file (*Engine.config*). Other application-specific command-line arguments can be used, e.g., the number of clients in the case of the *chat* application.

Figure A.1 presents a simple DACIA application (the file *dacia/Main.java*, consisting of an engine and two PROCs per host. These PROCs, of type *Chat* and *Forward*, respectively, have two ports each. The output of *p1* is connected to the input of *p2*. The engine connects to another engine running on a different host and having two similar PROCs. Subsequently, connections can be established between PROCs running on the two hosts. A message originating at one of the *Chat* PROCs will be delivered to the other *Chat* through the interposed *Forward* PROC. The message exchange is triggered by calling the *start()* method on *p1*.

Another simple application that uses a monitor can be found in *dacia/apps/chat*. For a good understanding of how this application is written and how it works, take a look at the following source files in this directory: *Main.java* (main program file), *Chat.java* and *ChatServer.java* (client and server PROCs), *ChatMonitor.java* (a monitor).

The Engine.runShell() call brings up the command line interface, which is described in more detail in the following section. Type *help* to get a list of commands. These are probably self-explanatory. Use *print* to see what other PROCs are in the system and how they are connected.

The Engine.displayGraph() brings up the graphical interface (GUI). Everything you can do in the command window you can also do using the GUI.

A.3 The command-line interface

DACIA provides a command-line shell interface (Figure A.2) for runtime management of the application. Through this interface, a user or system administrator interacts with the engine running on the local host. She can get information about the structure of the application (local and remote PROCs and their interconnections, and connections between engines), add or remove PROCs, manipulate the connectivity and the location of PROCs, and load and execute a monitor.

```
public class Main {
   public static void main(String args[]) {
      if (args.length < 1) {
         System.out.println("Usage:  java dacia.Main [configFileName]");
         System.exit(1);
      }
      // initialize the engine
      Engine.init(args[0]);
      // instantiate two PROCs and connect them
      Proc p1 = new Chat();
      Proc p2 = new Forward();
      Engine.addProc(p1);
      Engine.addProc(p2);
      Engine.connectProcs(p1, 0, p2, 1);
      // connect to another engine, running on port 5000
      // the two engines will exchange and update their PROC information
      Engine.connect("AnotherHostName",5000,true);
      // start the command-line application interface - optional
      Engine.runShell();
      // start the graphical interface - optional
      Engine.displayGraph();
      // adds a monitoring routine - optional
      Monitor monitor = new AMonitor();
      Engine.setMonitor(monitor);
      monitor.start();
      // triggers an action on a PROC
      p1.start();
   }
}
```

Figure A.1: A simple DACIA application. The application's engine connects to an engine running on another host. Subsequently, connections can be established between local and remote PROCs, using either the programming interface or the user command-line interface (see Figure 6.4).

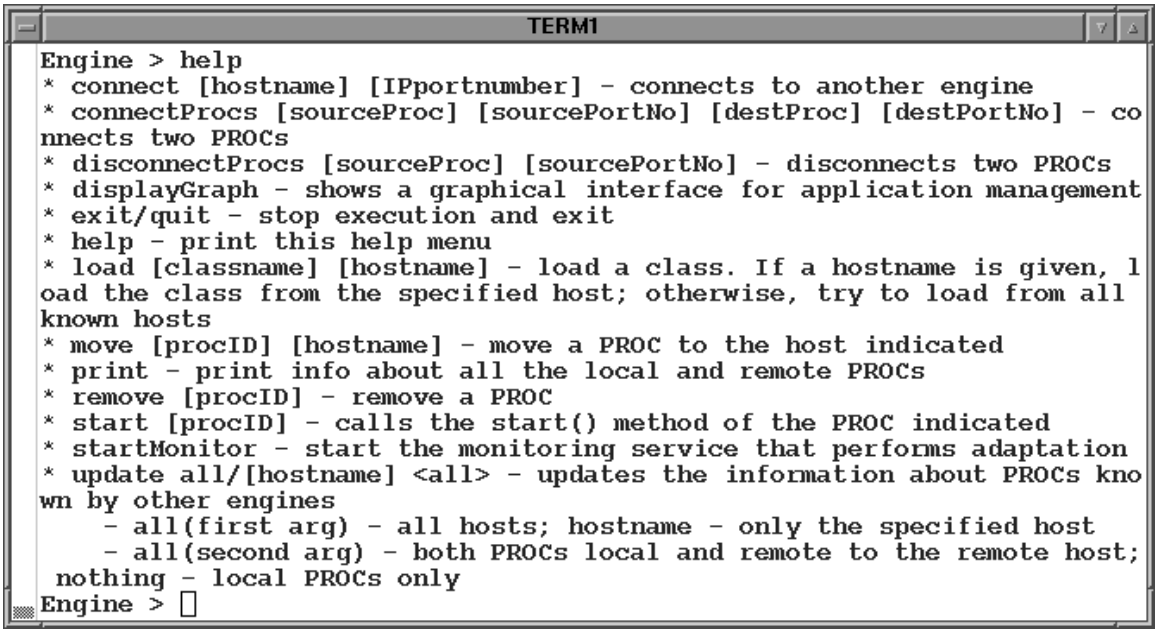

Figure A.2: The command-line shell interface allows a user or system administrator to visualize in text mode the structure of a distributed application and to manually reconfigure the application.

A.4 The graphical interface

DACIA comes with a graphical tool that provides an interactive environment for visualizing the structure of a distributed application and performing manual reconfiguration of the application. This graphical interface (GUI) offers all the functions that are available through the command-line interface. It can be started either during an application's initialization phase (add the line *Engine.displayGraph();* to the main program), or at any point during the execution of the application, by invoking the *displayGraph* command in the command-line window.

Figure A.3 displays a DACIA application, as it is represented in the graphical interface. The GUI is divided into two parts. The *graph panel* (left) graphically presents the structure of the application. The *information panel* (right) shows textual information about PROCs, their interconnections, and connections between engines[2], similar to the information displayed in the command-line interface using the *print* command. The information panel can be closed individually if so desired.

The application presented in the figure resides on 3 hosts, represented by the larger rectangles. The local host (brussels) has a distinct yellow color, while the remote hosts (saturn and sanjuan) are painted white. PROCs, represented by the smaller rectangles, are identified by their name (in most cases it corresponds to their type) and their unique numeric ID. Nodes belonging to different hosts are also assigned different colors. Currently,

[2]Only connections between the local engine and remote engines are displayed. By default, an engine has no knowledge about connections between remote engines.

up to 10 arbitrary colors are chosen for nodes. The graph displays the connections between hosts and the ones between components.

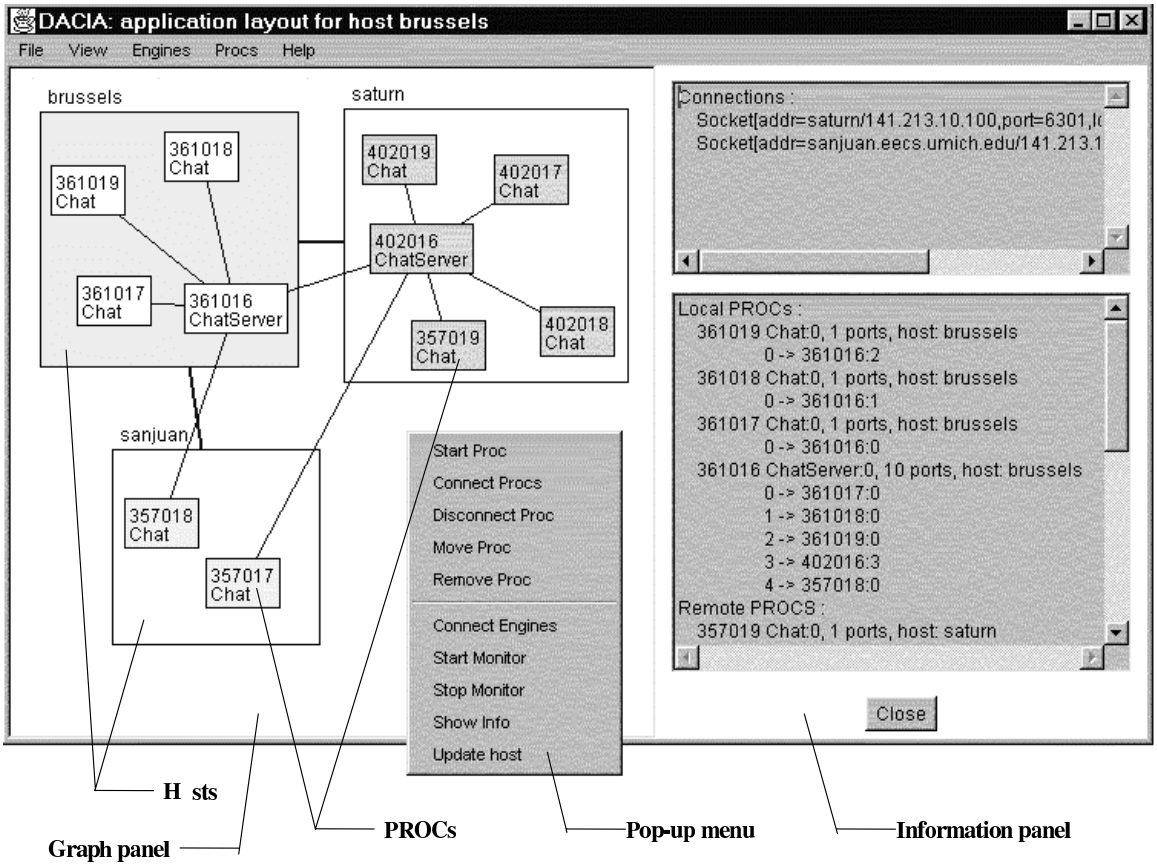

Figure A.3: The graphical interface (GUI) provides an interactive environment for visualizing the graph structure of a distributed application and performing manual reconfiguration of the application.

Commands for modifying the application structure can be issued by selecting an option from one of several menus available (some of these commands are listed in the pop-up menu displayed in the figure). By right clicking on one of the PROCs or inside one of the hosts, a menu of operations available on the selected PROC, or on the host's engine, respectively, shows up.

Most commands require the user to input certain parameters. To simplify the user's task, the dialog boxes corresponding to various menu options provide drop-down lists of choices instead of input text fields, and only display the valid choices, whenever possible. Some of the inputs are automatically filled in certain situations (e.g., the selection of a PROC's ID and port number if the PROC's *Connect Procs* menu option is selected).

The following menu options are available[3]:

- *File→Close* - close the graphical interface;

[3]Those annotated with a * have a dialog box associated with the command.

- *File→Quit* - stop the execution and exit;

- *View→Redraw* - automatically regenerate the graph;
- *View→Show Info* - display the information panel;
- *View→Preferences ** - allow user to customize parameters of Displaying the graph;

- *Engine→Connect ** - connect the local engine to another engine;
- *Engine→Start Monitor* - start the monitoring service that performs runtime adaptation;
- *Engine→Stop Monitor* - stop the monitor;
- *Engine→Update host ** - update information about PROCs;
- *Engine→Load class ** - load a PROC or a monitor;

- *PROC→Start Proc* - trigger an action on the selected PROC.
- *PROC→Connect Proc ** - connect two PROCs;
- *PROC→Disconnect Proc ** - disconnect two PROCs;
- *PROC→Move Proc ** - move a PROC to a selected host;
- *PROC→Remove Proc* - remove a PROC.

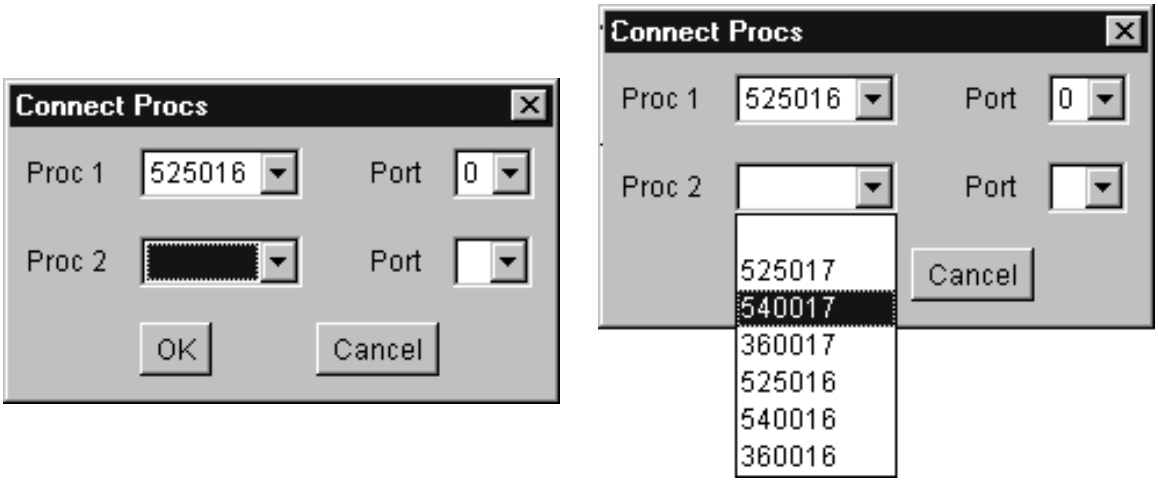

Figure A.4: The dialog box for the *PROC→Connect* command

Figure A.4 displays the dialog box for the *PROC→Connect* command. The user right clicks on PROC 525016 and invokes the command from a pop-up menu. The left figure shows

the initial dialog box with the default selection for the source ProcID already made. The source PortNo field lists only the available or unused port numbers of the selected PROC. In this example, since 0 is the only available port number, it is selected automatically. In the right figure, the destination ProcID field lists all the PROCs known by the Engine, excluding the source PROC.

The graph layout is initially generated using default settings, e.g., size of boxes and spacing between boxes. The user can change these values using the *View→Preferences* dialog box (Figure A.5). The graph also supports the manual customization of the display. The user can change the size of an individual box, move boxes and nodes around, resize the panel or scroll to view a specific region of the graph.

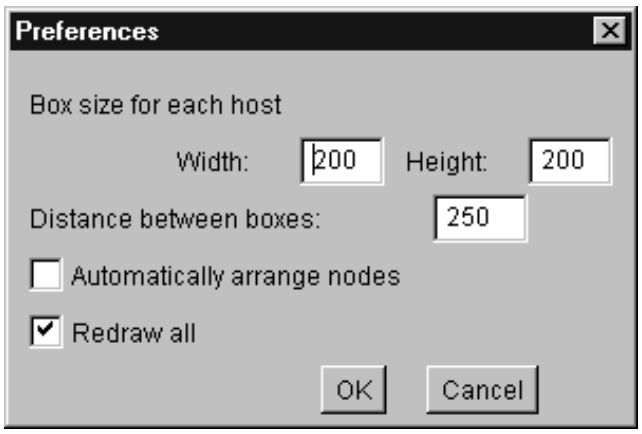

Figure A.5: The dialog box for the *View→Preferences* command

The user's actions are monitored to ensure the correctness of the graph. For example, boxes do not overlap. A node cannot be outside of all boxes at any time. If a node is dragged outside of all boxes, it will automatically go back to its host box. If a node is moved into a box corresponding to a host other than its own host, a confirmation box shows up asking the user if she wants the node to be moved to the new host. If the movement is confirmed, a *moveProc* command will be executed.

APPENDIX B

DACIA PROGRAMMING API

Primitive	Description
void init(String configFileName)	Initialize the engine, using information from the configuration file indicated.
Connection connect(String hostname, int port, boolean update)	Connect to the Engine running on the specified *hostname:port*. If *update* is true, exchange PROC info with the remote host.
void disconnectHost(Connection connection)	Close the connection between this engine and another engine.
void addProc(Proc proc)	Add a new PROC to the *localProcs* hashtable.
boolean removeProc(Integer procID)	Remove a PROC. Return *false* if the PROC does not exist.
void removeLocalProc(Proc proc)	Remove a local PROC.
boolean connectProcs(int procID1, int port1, int procID2, int port2, boolean propagate)	Connect two PROCs, local or remote. If *propagate* is true, inform the other engines about this connection. Return true if the connection is successful.
boolean connectProcs(Proc proc1, int port1, Proc proc2, int port2, boolean propagate)	Connect two local PROCs. If *propagate* is true, inform the other engines about this connection. Return true if the connection is successful.
boolean connectProcs(Proc proc1, int port1, RemoteProc proc2, int port2, boolean propagate)	Connect two PROCs, one local and one remote. If there is no conection to the remote host, a connection is opened. If *propagate* is true, inform the other engines about this connection. Return true if the connection is successful.
boolean connectProcs(RemoteProc proc1, int port1, Proc proc2, int port2, boolean propagate)	Connect two PROCs, one remote and one local. If there is no conection to the remote host, a connection is opened. If *propagate* is true, inform the other engines about this connection. Return true if the connection is successful.
boolean connectProcs(RemoteProc proc1, int port1, RemoteProc proc2, int port2, boolean propagate)	Connect two remote PROCs. Propagate operation to the corresponding hosts where the PROCs reside. Return true if the connection is successful.

Table B.1: Primitives provided by the Engine class

boolean disconnectProcs(int procID, int portNo, boolean propagate)	Disconnect two PROCs, local or remote. If *propagate* is true, inform the other engines about the disconnection.
boolean disconnectProcs(Proc proc1, int portNo1, boolean propagate)	Disconnect two PROCs, the first one being local. If *propagate* is true, inform the other engines about the disconnection.
boolean disconnectProcs(RemoteProc rp1, int portNo1, boolean propagate)	Disconnect two PROCs, the first one being remote. If *propagate* is true, inform the other engines about the disconnection.
void moveProc(Integer procID, String host)	Move a PROC to another engine/host.
void moveProc(Proc proc, Connection connection)	Move a PROC to another engine/host, using the connection indicated.
void setMonitor(Monitor m)	Set a monitor for this application. If another monitor exists, it will be replaced by the new monitor. The monitor needs to be subsequently started.
void startMonitor()	Start the Monitor.
void stopMonitor()	Stop the Monitor.
void update(String hostname, int allProcs)	Request updates about the PROCs known by other engines. *hostname* is the name of the remote engine to get the information from. If *hostname == all*, then request information from all known engines. If *allProcs == 1*, then request information about both PROCs local and remote to the remote host. If *allProcs == 0*, then request only information about local PROCs.
void runShell()	Start the command-line interface.
void displayGraph()	Start the graphical interface.
String connectionsInfo()	Return information about connections between this engine and other engines.
String procsInfo()	Return information about the local and remote PROCs.
String getHostName()	Return the name of the host where this engine is running.
Hashtable getLocalProcs()	Return the local PROCs table.
Hashtable getRemoteProcs()	Return the remote PROCs table.
Vector getConnections()	Return the list of connections to other engines.
Connection getConnection(String hostName)	Return the connection corresponding to the specified host, or null if there is no such connection.
Object getProc(int procID)	Return the local or remote PROC with the ID indicated, if it exists.
Proc getLocalProc(int procID)	Return the local PROC with the ID indicated, if it exists.
RemoteProc getRemoteProc(int procID)	Return the remote PROC with the ID indicated, if it exists.
int getPort()	Return the port number where this engine listens for connections from other engines.
Object loadClass(String name, String fromhost)	Load a class from a remote host, create and return an instance of that class. If *fromhost == null*, it attempts to load the class from any one of the engines it is connected to.

Table B.1: Primitives provided by the Engine class (continued)

Primitive	Description
Proc(String procName, int nports)	Construct a new PROC, having the name and the number of ports specified.
abstract void handleMessage(Message msg, int portNo)	Handle messages received synchronously. It is an abstract method; it has to be implemented in the subclass.
abstract void handleAsyncMessage()	Handle messages received asynchronously. It is an abstract method; it has to be implemented in the subclass. Messages are extracted from the message queue.
void output(int portNo, Message msg, int synchronous)	Send a message to another PROC through the specified port of this PROC. If *synchronous == 1*, then the communication is synchronous. If *synchronous == 0*, then the communication is asynchronous. Asynchronous output is recommended when the receiving PROC has multiple interfaces and it has to synchronize the data received on these interfaces.
void output(int portNo, Message msg)	Send a message to another PROC through the specified port of this PROC. The output type (synchronous or asynchronous) is given by the default type of the output port.
public Message getMessage()	Return the first message in the queue and remove it from the queue. If the PROC is about to move, return null.
public Message getMessage(int port, boolean remove)	Return the first message in the queue that has been received on the port indicated. Remove the message from the queue, if so required.
void printInfo()	Print information about this PROC and its connections.
void printInfo(DataOutputStream dos)	Print information about this PROC and its connections to the indicated output stream.
void start()	Start an action on this PROC. By default, it does nothing. Customized actions should be declared in the subclass.
void pack()	Prepare this PROC for moving to another Engine. Attempt to free as many objects as possible from this PROC, so that they are not serialized. Only the data representing state specific to this PROC should be kept. There is no need to submit general objects that can be restored at the destination by calling their constructor. This method needs to be overwritten in the subclass, to handle the customized structure of the subclass object. For PROCs with GUI, this method should dispose the GUI objects.
void unpack()	Restore the state of this PROC after moving from another Engine. This method needs to be overwritten in the subclass. It is the inverse of the pack() method.
int getID()	Return the ID of this PROC.
int getFreePort()	Return the first available port.
String getName()	Return the name of this PROC.

Table B.2: Primitives provided by the Proc class

Bibliography

[1] G. D. Abowd. Software Engineering Issues for Ubiquitous Computing. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, May 1999.

[2] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. *Mobile Object Systems: Towards the Programmable Internet, Lecture Notes in Computer Science 1219, Springer Verlag*, pages 111–130, Apr 1997.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] K. Amiri, D. Petrou, G. R. Ranger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[5] O. Babaoglu and S. Toueg. Understanding Non-Blocking Atomic Commitment. Technical Report UBLCS-93-2, University of Bologna, Laboratory for Computer Science, Italy, 1993.

[6] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges: An Application Model for Pervasive Computing. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MOBICOM 2000)*, pages 266–274, Boston, MA, Aug. 2000.

[7] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information Flow Based Event Distribution Middleware. In *Proceedings of the 19th International Conference on Distributed Computing Systems Middleware Workshop (ICDCS'99)*, pages 114–121, Austin, TX, May 1999.

[8] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Straer. Mole 3.0: A Middleware for Java-Based Mobile Software Agents. In *Proceedings of Middleware '98*, Lake District, U.K., Sep. 1998.

[9] V. Belloti and A. S. Bly. Walking Away from the Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team. In *Proceedings of the 1996 ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, pages 209–218, Boston, MA, Nov. 1996.

[10] R. Bentley and P. Dourish. Medium versus Mechanism: Supporting Collaboration through Customisation. In *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work (ECSCW'95)*, Stockholm, Sweden, 1995.

[11] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4):321–366, Nov 1998.

[12] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[13] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Resource Management for Value-Added Customizable Network Service. In *Sixth IEEE International Conference on Network Protocols (ICNP'98)*, Austin, TX, Oct 1998.

[14] D. M. Chess. *Security Issues in Mobile Code Systems.* Giovanni Vigna (Ed.): Mobile Agents and Security, pp. 1-14, Springer-Verlag, 1998.

[15] G. Chung and P. Dewan. A Mechanism for Supporting Client Migration in a Shared Window System. In *Proceedings of the Ninth User Interface Software and Technology*, pages 344–353, Boston, MA, Nov. 1996.

[16] J. E. Cook and J. A. Dage. Highly Reliable Upgrading of Components. In *Proceedings 21 st International Conference on Software Engineering*, pages 203–212, 1999.

[17] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.

[18] P. Dewan and R. Choudhary. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer Human Interaction*, 2(1):1–39, March 1995.

[19] P. Dourish. The Parting of the Ways: Divergence, Data Management and Collaborative Work. In *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work (ECSCW'95)*, Stockholm, Sweden, 1995.

[20] D. Duchamp. Issues in Wireless Mobile Computing. In *Proceedings of the Third IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL, April 1992.

[21] W. K. Edwards. Policies and Roles in Collaborative Applications. In *Proceedings of the ACM 1994 Conference on Computer-Supported Cooperative Work (CSCW '96)*, pages 11–20, Boston, MA, Nov.. 1996.

[22] M. Esler, J. Hightower, T. Anderson, and G. Boriello. Next Century Challenges: Data-Centric Networking for Invisible Computing - The Portolano Project at the University of Washington. In *Proceedings of MOBICOM '99*, Seattle, WA, Aug. 1999.

[23] G. Etzkorn. Change Programming in Distributed Systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 140–151, Imperial College of Science, Technology and Medicine, UK, 1992.

[24] G. Fitzpatrick, S. Kaplan, and T. Mansfield. Physical Spaces, Virtual Places and Social Worlds: A study of Work in the Virtual. In *Proceedings of 1996 the ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, pages 334–343, Boston, MA, Nov. 1996.

[25] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[26] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, Cambridge, MA, Oct. 1996.

[27] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2001.

[28] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), May 1998.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[30] B. Garbinato and R. Guerraoui. Using the Strategy Design Pattern to Compose Reliable Distributed Protocols. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 165–171, Phoenix, AZ, June 1997.

[31] C. Ghezzi, M. Jazayeri, and D.Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[32] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java^{TM} Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec 1997.

[33] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. *Addison Wesley, Reading*, 1996.

[34] K. M. Goudarzi and J. Kramer. Maintaining Node Consistency in the Face of Dynamic Change. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69, IEEE Computer Society Press, May 1996.

[35] S. Greenberg and M. Boyle. Moving Between Personal Devices and Public Displays. Nov. 1998.

[36] S. Greenberg and D. Marwood. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proceedings of the 1994 ACM Conference on Computer-Supported Cooperative Work, (CSCW '94)*, pages 207–217, Chapel Hill, NC, Oct. 1994.

[37] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *Proceedings of Mobicom '99*, Seattle, WA, Aug 1999.

[38] M. Hayden. The Ensemble System. Technical Report TR98-1662, Cornell University, Jan. 1998.

[39] M. Hayden and R. van Renesse. Optimizing Layered Communication Protocols. Technical Report TR96-1613, Cornell University, Nov. 1996.

[40] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[41] T. D. Hodes, R. H. Katz, E. Servan-Schreiber, and L. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. In *Proceedings of the 3rd ACM/IEEE MobiCom*, Budapest, Hungary, Sep 1997.

[42] C. R. Hofmeister and J. M. Purtilo. A Framework for Dynamic Reconfiguration of Distributed Programs. Technical Report UMCP TR3119, Computer Science Department, University of Maryland, College Park, 1993.

[43] O. Holder, I. Ben-Shaul, and H. Gazit. System Support for Dynamic Layout of Distributed Applications. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 403–411, Austin, TX, May 1999.

[44] S. E. Hudson and I. Smith. Techniques for Addressing Fundamental Privacy and Disruption Tradeoffs in Awareness Support Systems. In *Proceedings of 1996 the ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, pages 248–257, Boston, MA, Nov. 1996.

[45] B. Huffaker, E. Nemeth, and K. Claffy. A General-Purpose Network Visualization Tool. http://www.caida.org/tools/visualization/otter/paper/. In *Proceedings of the 9th Annual Conference of the Internet Society, INET'99*, 1999.

[46] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proceeding of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 187–200, New Orleans, LA, Feb. 1999.

[47] N. C. Hutchinson and L. L. Peterson. X-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[48] IBM Corp. VisualAge Micro Edition 1.3, http://www.embedded.oti.com/.

[49] R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - An Extensible Extension Language. *Software: Practice and Experience*, 26(6), 1996.

[50] D. Johansen, R. Van Renesse, and F. B. Schneider. An Introduction to the TACOMA Distributed System. Technical Report 95-23, Dept. of Computer Science, Univ of Tromso and Cornell Univ., June 1995.

[51] A. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[52] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. Computer Systems*, 6(2):109–133, 1988.

[53] C. Karamanolis and J. Magee. A Replication Protocol to Support Dynamically Configurable Groups of Servers. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 161–168, IEEE Computer Society Press, May 1996.

[54] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):21–30, July-August 1997.

[55] J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, pages 374–384, Tel-Aviv, Israel, 1990.

[56] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[57] J. Kramer, J. Magee, M. Sloman, and N. Dulay. Configuring Object-Based Distributed Programs in REX. *IEEE Software Engineering Journal*, 7(2):139–149, March 1992.

[58] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[59] J. H. Lee, A. Prakash, T. Jaeger, and G. Wu. Supporting Multi-User, Multi-Applet Workspaces in CBE. In *Proceedings of 1996 the ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, pages 344–353, Boston, MA, Nov. 1996.

[60] R. Litiu. The DistView Collaboratory Toolkit, http://www.eecs.umich.edu/distview/.

[61] R. Litiu and A. Prakash. Adaptive Group Communication Services for Groupware Systems. In *Proceedings of the Second International Enterprise Distributed Object Computing Workshop (EDOC'98)*, San Diego, CA, Nov. 1998.

[62] R. Litiu and A. Prakash. Stateful Group Communication Services. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 82–89, Austin, TX, June 1999.

[63] R. Litiu and A. Prakash. Challenges in Using a Mobile Component Framework to Develop Adaptive Groupware Applications. In *Proceedings of CBG 2000, the CSCW 2000 Workshop on Component-based Groupware*, Philadelphia, PA, Dec. 2000.

[64] R. Litiu and A. Prakash. DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. *Principles of Distributed Computing (PODC) 2000 Middleware Symposium*, Portland, OR, July 2000.

[65] R. Litiu and A. Prakash. Developing Adaptive Groupware Applications Using a Mobile Component Framework. In *Proceedings of the 2000 ACM Conference on Computer-Supported Cooperative Work, (CSCW 2000)*, Philadelphia, PA, Dec. 2000.

[66] S. Lu, K.-W. Lee, and V. Bharghavan. Adaptive Service in Mobile Computing Environments. In *Proceedings of IFIP IWQoS '97 (International Workshop on Quality of Service)*, New York, NY, May 1997.

[67] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–154, Sitges, Spain, Springer LNCS 989, Sept. 1995.

[68] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, SE-15(6), 1989.

[69] P. J. McCann and G.-C. Roman. Mobile UNITY: A Language and Logic for Concurrent Mobile Systems. Technical Report WUCS-97-01, Department of Computer Science, Washington University in St. Louis, Dec 1996.

[70] D. L. Métayer. Software Architecture Styles as Graph Grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23, ACM Press, 1996.

[71] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Journal of Information and Computation*, 100(1):1–77, 1992.

[72] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. Technical Report TR 91-32, Dept. of Computer Science, University of Arizona, Tucson, AZ, 1991.

[73] S. Mishra, L. L. Peterson, and R.D. Schlichting. Experience with Modularity in Consul. *Software Practice & Experience*, 23, 1993.

[74] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of OSDI '96*, pages 153–168, Oct. 1996.

[75] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latencies. In *Proceedings of SIGCOMM '96*, pages 73–84, Sep. 1996.

[76] S. Mullender. *Distributed Systems*, chapter Chapter 6.8: The Non-Blocking Atomic Commitment Problem. Addison-Wesley, 1993.

[77] G. C. Necula and P. Lee. Research on Proof-Carrying Code on Mobile-Code Security. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, 1997.

[78] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, Oct. 1997.

[79] M. Nuttal. Survey of Systems Providing Process or Object Migration. Technical Report 94/10, Dept. of Computing, Imperial College, UK, May 1994.

[80] Object Management Group. CORBA Services: Common Object Service Specification. Technical report, Object Management Group, July 1998.

[81] Object Management Group. The Common Object Request Broker Architecture (CORBA) Specification, Revision 2.4.1, http://www.omg.org/technology/documents/formal/corbaiiop.htm. Oct. 2000.

[82] ObjectSpace. Voyager, http://www.objectspace.com/products/voyager/. 2000.

[83] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[84] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, 17(4), 1992.

[85] J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 6(1):151–174, 1994.

[86] R. H. Katz et. al. The Bay Area Research Wireless Access Network(BARWAN). In *Proceedings Spring COMPCON Conference*, Feb 1996.

[87] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. Mach: A Foundation for Open Systems. In *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*, Sep. 1989.

[88] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(7):57–66, 1990.

[89] M. Roseman and S. Greenberg. Building Flexible Groupware through Open Protocols. In *Proceedings of the ACM Conference on Organizational Computing Systems*, California, 1993.

[90] A. Schill. DCE – The OSF Distributed Computing Environment. In *Proceedings of the International DCE Workshop*, Karlsruhe, Germany, Oct. 1993.

[91] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[92] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley and Sons, 1997.

[93] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[94] A. H. Shen and A. P. Dewan. Access Control in Collaborative Environments. In *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work, (CSCW '92)*, pages 51–58, 1992.

[95] I. Sommerville and G. Dean. PCL: A Configuration Language for Modelling Evolving System Architectures, Computing Department, Lancaster University. *ftp://ftp.comp.lancs.ac.uk/pub/proteus/PCL/PCL_overview.ps*.

[96] N. A. Streitz, J. Geisler, and T. Holmer. Roomware for Cooperative Buildings: Integrated Design of Architectural Spaces and Information Spaces. *Cooperative Buildings: Integrating Information, Organization, and Architecture, Springer-Verlag, Lecture Notes in Computer Science, 1370*, pages 4–21, 1998.

[97] Sun Microsystems. Java 2 Platform, Micro Edition (J2ME), http://java.sun.com/j2me/. July 2000.

[98] Sun Microsystems. Java Remote Method Invocation (RMI), http://java.sun.com/products/jdk/rmi/. Dec. 2000.

[99] Sun Microsystems. Jini Connection Technology, http://www.sun.com/jini/. 2000.

[100] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh. Mobile Agent Programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 190–197, Austin, TX, May 1999.

[101] R. van Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible Group Communication System. *Communications of the ACM*, Apr. 1996.

[102] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proc. of the 14th ACM symposium on Operating Systems Principles (SOSP '93)*, pages 203–216, Asheville, US, Dec 1993.

[103] I. Warren and I. Sommerville. Dynamic Configuration Abstraction. In *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, Springer LNCS, Sept. 1995.

[104] M. Weiser. Hot Topics: Ubiquitous Computing. *IEEE Computer*, Oct. 1993.

[105] M. Wermelinger. A Hierarchic Architecture Model for Dynamic Reconfiguration. In *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 243–254, IEEE Computer Society Press, 1997.

[106] M. Wermelinger. A Simple Description Language for Dynamic Architectures. In *Proceedings of the 3rd International Software Architecture Workshop*, pages 159–162, ACM Press, 1998.

[107] M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 111–118, IEEE Computer Society Press, 1998.

[108] J. E. White. Telescript Technology: Mobile Agents. *Software Agents, J. Bradshaw, ed. AAAI Press/MIT Press*, 1996.

[109] Xerox Palo Alto Research Center. ILU – Inter-Language Unification, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html. May 2000.

[110] M. Yarvis, P. Reiher, and G. J. Popek. Conductor: A Framework for Distributed Adaptation. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS '99)*, March 1999.

[111] B. Zenel and D. Duchamp. A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment. In *Proceedings of MobiCom '97*, Budapest, Hungary, Oct. 1997.

[112] H. Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, Apr. 1980.