# Challenges in Using a Mobile Component Framework to Develop Adaptive Groupware Applications

Radu Litiu and Atul Prakash
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122, USA
Email: {radu,aprakash}@eecs.umich.edu

## ABSTRACT

There is a need to develop groupware systems that adapt to available resources and support user access from a variety of fixed and mobile computing devices and information appliances. In this paper we present DACIA, a component-based framework that provides support for building reconfigurable groupware systems. We specifically focus on the challenges of using mobile components to build adaptive collaborative applications that work across multiple devices and support user mobility. We explore DACIA's applicability to building groupware applications that can be reconfigured at run-time to adapt to changing user demands and resource constraints.

## INTRODUCTION

Users increasingly work in environments with varying resource constraints or where mobility is required. Groupware systems will need to be developed that work in such environments. The variability occurs along several dimensions:

- User and application demands;
- User mobility and intermittent connectivity;
- Hardware heterogeneity and network variability.

It is difficult to design a one-size-fits-all groupware system that works well under all potential usage situations. Groupware systems often end up making significant assumptions about the environment and must be redesigned to be used effectively if the assumptions no longer hold. We believe that there is a need to develop techniques for designing flexible groupware systems that adapt better to user mobility and available resources.

[0]This work is supported in part by the National Science Foundation under Grant No. ATM-9873025.

We have developed a component-based framework, called DACIA[1], that addresses some of the above challenges. DACIA provides support for building adaptable groupware systems. To illustrate its potential applicability, we have used DACIA to build groupware applications with the following types of features:

• **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 re-established transparently.

• **Dynamic application reconfiguration**: A groupware 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.

• **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. When the user reconnects, eventually from a different place, he can take over the control from the parked application.

A detailed presentation of the design and implementation of DACIA, as well as more extensive examples of using it to build adaptive groupware applications, can be found in a companion CSCW paper [4].

[1]Dynamic Adjustment of Component InterActions

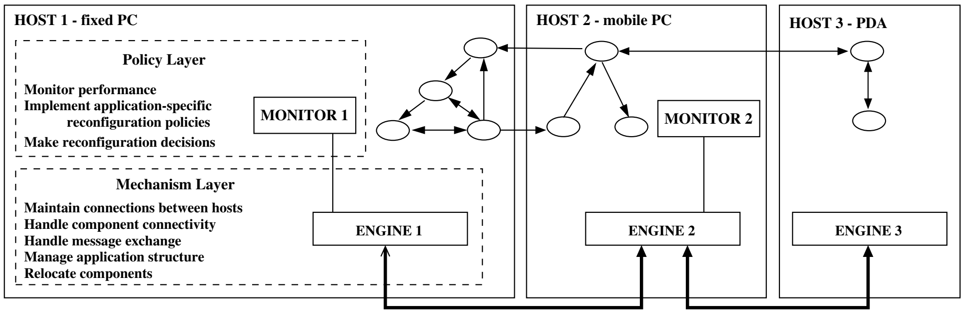


Figure 1: A DACIA distributed application is a directed graph of connected components (ovals represent components). An engine runs on every host. It manages the local components and the connections between components, both local and across different hosts. The monitor gathers performance data and implements application-specific relocation and reconfiguration policies.

In this paper, we briefly present the architecture of DACIA, and we show how it can be used to build and execute flexible groupware applications for mobile computing environments. The primary focus of the paper is to lay out some of the issues raised by the use of a mobile component framework to develop reconfigurable applications that adapt to available resources and to changing user requirements.

## 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 (Figure 1) 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.

In DACIA, a component is a *PROC*[2]. 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 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 communicate by exchanging *messages* through *input and output ports*.

[2] Processing and ROuting Component

The *engine* decouples the 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. In the current DACIA implementation, a DACIA distributed application requires an engine on every host it runs on.

The engine works in conjunction with a *monitor*. The monitor represents the part of an application that *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.

The engines and the PROCs are general-purpose and they can be reused to build multiple applications. Engines provide the mechanisms for reconfiguration while monitors implement application-specific policies for reconfiguration. The monitor is usually specific to the application and it incorporates relocation and reconfiguration policies applicable only to a particular application. The monitors are optional – applications can be also reconfigured manually by system administrators, using a command-line interface. Through this interface, the user or system administrator can manually reconfigure an application by relocating PROCs, creating new PROCs, and changing the way existing PROCs are connected.

## MOBILITY AND INTERMITTENT CONNECTIVITY

One of the key features of our architecture is the ability to move components between hosts. 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. We maintain a weak consistency of each engine's view of components' locations and their connections. When a PROC moves, the engine where the PROC was previously located will send notifications about the change only to the engines hosting PROCs connected to the moving PROC.

PROCs can move between hosts while maintaining *persistent connectivity* to other PROCs. This seamless connectivity offers a great benefit to mobile users, who can move applications from one host to another without having to re-establish all the connections to other parties. The structure of the application does not change and the flow of data in the system is not interrupted[3].

A potential application of component mobility and persistent connectivity is *application parking*. 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. When the user's application moves to a different device, it maintains its connections to services and collaborative partners and it continues its execution.

## DYNAMIC APPLICATION RECONFIGURATION

DACIA provides mechanisms for dynamically reconfiguring an application. These mechanisms can be used to dynamically connect and disconnect PROCs, introduce new PROCs in the data paths, and move PROCs across hosts.

Even when users are not mobile, application reconfiguration can be desirable to make groupware systems adaptable to available resources. 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.

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

### Reconfiguration Example: Multi-Party Communication

Using DACIA, we have implemented an adaptive multi-party communication application that can be used as a basis for building a range of groupware systems. 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 the clients run.

Initially, when there are only 2 clients, they are connected directly (Figure 2.a), without using a server. This is a typical architecture for two-party communication tools. When a third client tries to join the communication group, a server module is spawned, and all the clients will connect to the server and will exchange data through it (Figure 2.b).

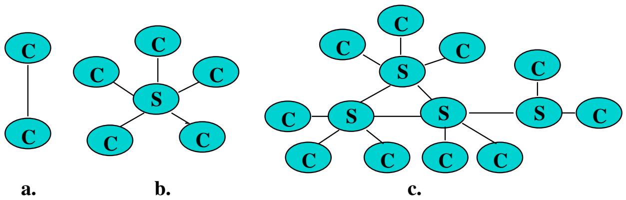


Figure 2: Adaptive multi-party communication. Servers are denoted by S, and clients are denoted by C. New servers are created as the number of participants grows.

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. 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 assigned equally[4] to all the servers (Figure 2.c). 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 is deallocated and its clients are distributed to other servers.

As proof-of-concept, we implemented a multi-party chatbox application on top of this group communication service. The

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

application reconfigures itself dynamically, based on the implemented adaptive policy. The same architecture can be used to implement other groupware applications requiring multi-party communication infrastructure.

## ONGOING WORK AND OPEN ISSUES

Our work on designing and implementing adaptive groupware applications using DACIA is in a preliminary stage. Some future challenges include the extensive deployment and evaluation of DACIA-enabled collaboration tools in real collaboration settings, and providing access control features for users to control the mobility of their components. In this section we present some of the issues raised by the use of a component-based framework to develop flexible groupware applications.

### Designing Groupware Applications with DACIA

Our preliminary experience with using DACIA in building groupware applications indicates that certain types of PROCs are likely to be useful. The PROCs used to develop groupware applications (Figure 3) can be classified according to their characteristics and functionality as follows:

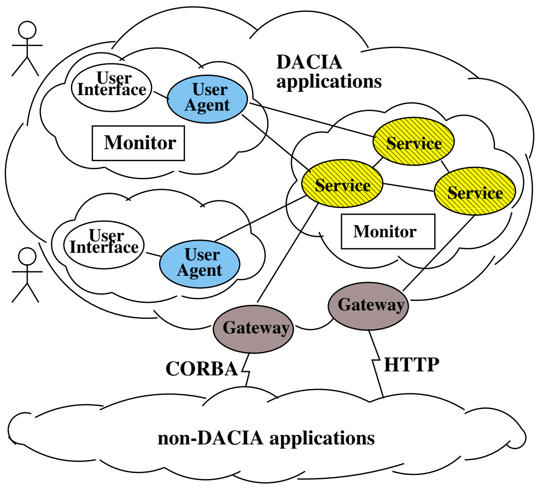


Figure 3: A DACIA collaborative application may contain PROCs of various types: User Interface, User Agent, Service, and Gateway. Via gateways, DACIA applications can interact with non-DACIA components.

- *User Interface PROCs* represent interfaces between human users and applications.
- *User Agent PROCs* are persistent representations of users of a groupware 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* perform the actual computation in a collaborative application. They can be used to implement various services, such as data processing and distribution, caching and storage, group and session management, etc. A particular service can be implemented in various ways, using different sets of components, connected in multiple 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. The reconfigurable multi-party chat application in Figure 2 uses only User Interface and Service PROCs. The system can be potentially enhanced by adding sensors that detect a user's presence next to a host, as in [3]. 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. The separation also simplifies client parking. While a user is disconnected, its corresponding agent can still participate to collaborative activities on behalf of the user. The User Interface is not needed in such a case. In order for a user to be able to locate and reconnect to its agent, DACIA provides unique PROC identifiers.

### Division of Functionality and Coordination

In a DACIA application, the engine provides the *mechanisms* used to build and reconfigure applications, while the monitor implements the *policy* layer responsible for making reconfiguration decisions. One challenge is to draw the line between the general-purpose infrastructure functions that should be handled by the engine and the application-specific functions that have to be implemented by monitors.

Measuring application performance and making reconfiguration decisions based on these measurements raises several questions. What are the parts of an application that can and should collect performance data? How is this data communicated to a monitor and how does a monitor makes sense of it? If multiple monitors are used, implementing different reconfiguration policies, how do their decisions affect each other?

The monitor is not solely responsible for performance monitoring. The PROCs and the engine may also collect some performance data. For instance, 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, are performed by the engine. Application-specific metrics, such as the amount of memory used, or the amount of computation, are handled by the PROCs. Specialized API functions are used to communicate to the monitor the values of these latter measurements.

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 (Figure 1, HOST3) it is desirable that the computation running on a host is as lightweight as possible. In this situation, the adaptive function can be executed by a monitor running on a different host.

Different parts of a large-scale collaborative application may fall under different administrative domains. They employ different coordination policies, resource management routines, and reconfiguration algorithms. 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.

A formal framework is needed to specify interactions between components and rules for composing components, and for defining semantically equivalent composition schemes. This formal framework will also provide support for defining and implementing adaptive policies used to reconfigure an application.

**Support for Multiple Devices**

A component can be written to be hardware-dependent. In this case, when a component moves from one host to another, it can be made to exhibit different capabilities, depending on the host. For example, on a high-end host, a component can have a full-fledged graphical representation and it can present a GUI interface to the user. On a PDA, the component only displays information in text format and it eventually sends visual or sound alerts to inform the user about important changes in the events monitored by the component. A user should be able to move a component from one device to another without having to re-instantiate it or to connect again to other components.

The challenge in providing this component versatility 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. Moreover, small devices have limited memory and processing power and they will benefit from executing components with a small footprint, implementing only the minimal functionality appropriate to their capabilities.

An alternative allows for multiple versions of the code for the same component, each of whom is minimal and provides only the functionality required by a particular device. When a component moves to a different device, it is instantiated locally, using the code available in local libraries. The application will transfer between hosts only the necessary state of the component. A requirement on component developers will be to implement routines for marshalling/unmarshalling the state that are efficient and work across multiple platforms.

**Security Policies for Components**

The security infrastructure of our mobile component environment 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 manipulate the behavior of hosts to which the component is later migrated. Not only may the modified component operate incorrectly, but it can compromise the hosts to which it is migrated.

Several solutions have emerged that allow a precise definition of the rights of mobile code. Through the enhanced security model of Java 1.2 [2] and *trust management* approaches [1], hosts and developers specify precisely the set of rights assigned to a component. Each potentially sensitive operation is only allowed when it is consistent with the component rights. Another way to address the safe operation of mobile code is to establish its correct operation prior to execution. In *Proof-Carrying Code* [5], code consumers (e.g., hosts) verify whether mobile code adheres to predefined safety rules. These rules, called *safety policies*, allow a consumer to define the requirements for safe behavior.

While significant effort has been expended in addressing the security concerns of executing hosts, protecting components from malicious environments has received little attention. With regard to component security, there are three goals of the component owner: *execution integrity*, *data integrity*, and *secrecy*. Execution integrity ensures that the execution of the component is consistent with its definition. Data integrity ensures that the component data can not be modified, or in a lesser guarantee, modification of data is detectable by its owner. Finally, secrecy ensures that internal state and the component code itself is visible only to authorized parties.

We intend to further investigate the requirements and capabilities of the participants in mobile code environments. Using this analysis, we will design and construct a service from which stated security requirements, called *policies*, can be defined and distributed. The mobile code environment will be augmented with facilities for the run-time evaluation, reconciliation, and enforcement of policies stated by the component owner and executing host. Existing security techniques and models will be analyzed and, as appropriate, integrated within our environment.

## SUMMARY

In this paper, we presented DACIA, a mobile component framework that allows applications to reconfigure dynamically by loading new components, changing the way components interact and exchange data, and moving some components from one host to another. DACIA provides support for application and user mobility and it enables persistent connectivity between moving components. To illustrate DACIA's use, several reconfigurable groupware applications have been implemented.

We highlighted some of the directions in which we intend to extend our work and some of the open issues regarding the use of mobile components to build adaptive groupware applications. One of our concerns is the division of functionality between various parts of modular application and the coordination of a large-scale distributed application. Challenges arise from the need to support multiple devices with a wide range of capabilities and to handle the connectivity of mobile devices. The use of mobile code also raise important security concerns.

We believe that DACIA provides a useful platform for experimenting with new kinds of groupware applications. More extensive deployment and experimentation is needed to assess the usefulness and ease of use of a mobile component framework to build and execute adaptive collaborative applications.

## REFERENCES


1. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems Security. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects, Springer-Verlag*, 1999.
2. 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.
3. A. Harter, A. Hopper, P. Steggles, A. Ward, and Paul Webster. The Anatomy of a Context-Aware Application. In *Proceedings of Mobicom '99*, Seattle, WA, Aug 1999.
4. 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 (to appear), Dec. 2000.
5. 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.