Infrastructure Support for Mobile Collaboration

Radu Litiu and Amgad Zeitoun
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122, USA
{*radu, azeitoun*}*@eecs.umich.edu*

Abstract

Future groupware systems will need to extend collaboration beyond the desktop. They will need to support computing devices with a wide range of capabilities, varying network connectivity, and increasing mobility of users. We have designed and implemented a component-based framework for building reconfigurable distributed applications that address the specific needs of mobile environments. This paper focuses on the use of this framework to support mobile collaboration. Components of a groupware application can move across heterogeneous devices while maintaining persistent logical connectivity with groupware services and other users, even during transient network failures. Users do not see any interruptions in the services accessed, and they do not need to manually re-establish connections with the communication parties. New collaboration features can be more easily implemented. For instance, users may "park" their client agents temporarily at a fixed host while they are disconnected. The parked agent can continue to maintain connectivity with other group members on behalf of the user, if desired. We show how this framework can be used to structure distributed applications that adapt their behavior and their interface to the context in which they execute. We give examples of groupware applications we have implemented.

1. Introduction

Ubiquitous computing [1] promises to help organize and mediate social interactions wherever and whenever these situations might occur. Improved wireless communication capabilities, continued increase in computing power, and improved battery technology have lead to the large scale adoption of a wide range of mobile computing devices, in addition to traditional desktop-based system. Mobile collaboration has become the norm, rather than the exception. Novel collaborative paradigms need to be developed to take into account the intermittent application, resource, and user availability, the variability in device capabilities, and the unreliable network connectivity in mobile computing environments. A major goal is to ensure that a user's applications are available, in a suitably adapted form, wherever the user goes. At the same time, user's applications should be able to participate, on a limited basis, to collaborations on the user's behalf, even while the user is disconnected or is not active.

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 effectively used if the assumptions no longer hold. For best performance and functionality, different system architectures may be required as we go 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. We believe there is a need to develop techniques for designing flexible groupware systems that adapt better to user mobility and resource availability.

Consider the following scenario: Bob is in his office working on the final details of a presentation he will give the following day at a trade show. He uses a shared editor to collaboratively work together with his colleagues Alice and Paul. Each of them uses a set of resources (e.g., data files, images) available only on the local machine. Besides the editor, the collaborative session they established also runs a videoconferencing tool. Bob realizes he is running out of time and has to leave for the airport, and the presentation is not ready yet. He "parks" on a corporate server the applications running on his workstation, turns his computer off, and heads for the airport. Once at the airport, Bob still has some time left before the flight departure. He finds a hot-spot with a wireless access point. Using his high-end PDA (Personal Digital Assistant), he connects to the corporate server and pulls the parked editor. Some of the slides reflect the recent changes made by his colleagues. He is able to view and play the whole presentation, but has limited capabilities for editing only the textual information. The application realizes Bob is connecting from an untrusted network and automatically loads a pair of encryption/decryption modules into the data flow. In addition, the application senses the weak connectivity of Bob's PDA and loads compression and decompression modules. Meanwhile, Alice and Paul do not realize that Bob has been disconnected and now he is connected from a different place.

We have developed a component-based framework, called DACIA (Dynamic Adjustment of Component InterActions), that addresses some of the challenges occurring in the scenario above. DACIA provides a seamless collaborative environment where network connectivity, computing devices, and user's behavior keep changing. It provides support for building adaptable groupware systems. DACIA can be used to develop groupware applications that support features such as *dynamic reconfiguration*, *persistent connectivity with user mobility*, and *off-line collaboration*.

Dynamic reconfiguration allows applications to adapt to the continuously changing user demands, load on network segments and intermediate processing nodes, and variations in device capabilities and resource availability. A modular architecture of an application, in which various components implement individual functions, can easily mutate its structure at runtime to adapt to different conditions. 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.

Persistent connectivity encounters great challenge when users are mobile. Mobility provides a flexible way to weave computing and collaboration into user's daily activities. In a mobile environment, users connect from various points, using a variety of devices. Operating in a mobile environment raises the problem of dealing with the inherent unreliability of mobile network connections and variations in connection quality. Hence, masking transient network and communication failures becomes an important issue for some applications in a mobile environment. We show how persistent connections are preserved in our DACIA framework. By maintaining logical connections between moving components, a mobile user can simply "pull" an application or application component from one computing device and drop it on another computing device. Manual re-establishment of connections is not needed as all connections are automatically re-established transparently while maintaining application's states.

Off-line collaboration provides a great opportunity for disconnected users to participate to collaborative activities on a limited basis. In DACIA, such realization is applicable. It would be nice to provide support to users so that they do not have to close all the sessions with other parties and quit all the collaborative applications, in order to move to a different place short time later, restart the very same applications, and manually establish the sessions. DACIA allows a mobile application to be parked while its user is disconnected or idle. A parked application can continue to interact, with some limitations, with other parties on behalf of the user. The 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 reconnects, eventually from a different place, he can take over the control from the parked application.

The rest of the paper is structured as follows. Section 2 presents related research work. Section 3 gives an overview of the architectural features of DACIA. Section 4 shows the support provided by DACIA for building collaborative applications for mobile environments. We give examples of several groupware applications we have implemented. Finally, Section 5 presents some concluding remarks and directions for future work.

2. Related Work

Our work enables the design of more flexible and adaptable CSCW (Computer Supported Collaborative Work) systems for mobile environments. Several researchers have pointed out the importance of *flexibility and adaptability* in CSCW systems [2, 3]. 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 [4, 5]. In fact, there are many dimensions of flexibility and adaptability in CSCW systems, such as: access control [6, 7], concurrency control [8], coupling of views [9], and extensible architectures [10, 11].

A common observation in 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. DACIA is complementary to the above work and focuses on providing infrastructure support for dynamically adapting the architecture of CSCW systems and the location of system components and services to the context in which they are being used, scale of use, location of users, and available resources.

Other researchers have emphasized the importance of considering *resources* (e.g., CPU, display, and network) in CSCW systems design. Groupware systems need to be designed to allow tradeoffs between context awareness and available resources [12]. There is a cost to providing more awareness information in terms of information overload, screen real-estate, network resources, privacy, etc. In a world where users are surrounded by a multitude of devices (e.g., computers, PDAs, cell phones, large wall displays, sensors, etc.) and overwhelmed by the wealth of information presented to them, *user attention* has also been identified as a valuable resource [13]. There have been debates over the merits of centralized architectures, peer-to-peer architectures, and replicated services in building groupware systems. The goal of our work 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.

The need to support *user mobility* has also been pointed to recently. The work in the cooperative buildings area assumes that users are mobile inside buildings and the work should be possible anywhere users are (coffee tables, walls, desktops, etc.), rather than users having to work on a standard desktop [14]. Belloti and Bly argued that CSCW systems must be designed to support mobility, since mobility can be critical to many work settings [15]. They concluded that CSCW systems must accommodate mobility rather than seek to eradicate it via desktop collaboration tools. DACIA simplifies building of groupware applications in which clients are mobile. We extend the existing work on *mobile code* [16] and *mobile agent systems* [17–19] and apply it to the construction of distributed applications through the dynamic composition of software components.

The goal of *ubiquitous computing* (or ubicomp, for short) [1] 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

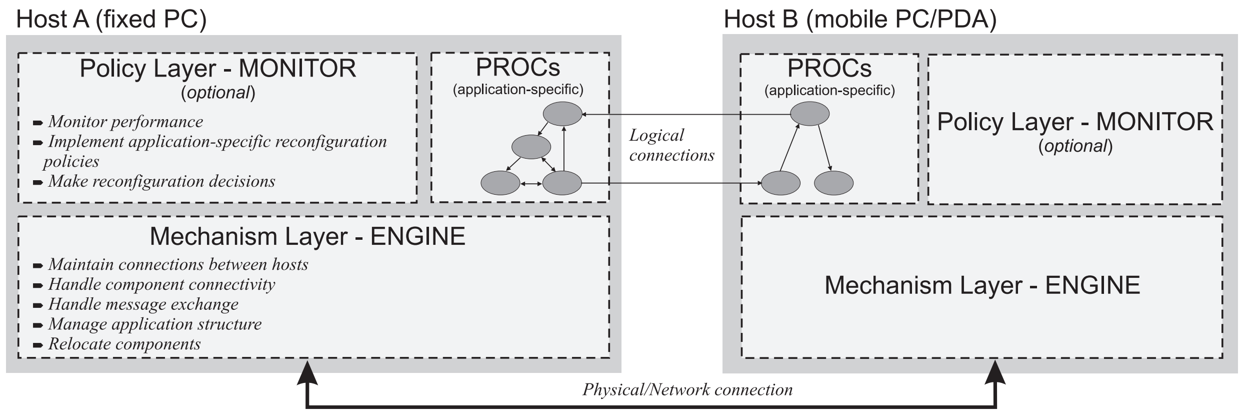


Figure 1. A DACIA distributed application is a directed graph of connected components (ovals represent components). An engine runs on every host. It manages the local components and the connections between components, both local and across different hosts. The monitor gathers performance data and implements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

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. *Context-aware applications* [20] can follow their users as they move around a building, and can adapt to the characteristics of the environment where they execute. They can change their behavior based on knowledge of the user's current location, using, for instance, active badges. Novel software engineering solutions are needed to provide the functional features required by ubiquitous and pervasive computing [21, 22].

Application adaptation based on reconnecting components has been employed in ad hoc and sensor networks. For example, MagnetOS [23] provides a single system image of a unified Java virtual machine across a set of ad hoc nodes that comprise the system. Application's components are dynamically placed at different nodes. The main goal of MagnetOS is to provide power-aware computing.

3. Flexibility in Mobile Computing Systems

In this section, we briefly describe the component-based architecture of DACIA. We discuss the support provided for dynamic application reconfiguration and for component mobility. We give examples of using mobility and supporting disconnected users that wish to collaborate, with some limitations, during their disconnection. Details of architectural issues in DACIA are presented elsewhere, and are not the subject of his paper.

3.1. Component-based 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 can be seen as a directed graph of connected components. The links between components indicate the direction of the data flow within the application.

In DACIA, a component is a *PROC* (Processing and ROuting Component). 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 alternately send them to multiple destinations. A PROC is identified system-wide using a unique identifier.

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 1) 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.

PROCs communicate by exchanging *messages* through *ports*. Communication can be either *synchronous* or *asynchronous*. In the case of asynchronous communication, the messages received by a PROC are inserted into the PROC's *message queue*. Every PROC has a thread that handles the messages in the queue, usually in FIFO order. DACIA provides a lightweight solution to local communication, by co-locating local PROCs within the same address space. Thus, message exchange translates into simple procedure calls. The engine maps virtual connections between PROCs to either local or remote network 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 engine may work in conjunction with a *monitor*[1]. The monitor represents the part of an application that keeps track of 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. Engines and PROCs are general-purpose and they can be reused to build multiple applications. Engines provide the mechanisms for

[1]The use of a monitor is optional.

reconfiguration, while monitors implement application-specific policies for reconfiguration.

3.2. Dynamic Application Reconfiguration

DACIA provides mechanisms for dynamically reconfiguring an application. These mechanisms can be used to connect and disconnect PROCs, introduce new PROCs in the data paths or eliminate PROCs, and move PROCs across hosts. As a result of reconfiguration, the application graph changes. Even when users are not mobile, application reconfiguration can be desirable to enable groupware applications to better adapt to available resources and to the context of their execution. A more efficient execution can be achieved through better usage of the available resources and optimized inter-component communication. Our solution to runtime reconfiguration ensures that the application executes correctly both during and after the reconfiguration.

There is a separation between reconfiguration policies and mechanisms. An application developer can implement customized policies in monitors. System administrators can also manually reconfigure applications based on changing runtime requirements and resource availability, using either a command-line or a graphical interface. Through this interfaces, the user or system administrator can issue commands to change the application structure or to move parts of the applications from one host to another.

3.3. Component Mobility and Persistent Connectivity

Traditional collaboration paradigms, in which users interact using their desktop computers, are too rigid to provide adequate support for novel environments, in which mobility has become ubiquitous. Fixed hardware, combined with mobile devices, form a set of resources with distinct interaction and availability characteristics. Mobility is not restricted to the mere use of mobile computing devices such as laptop computers and PDAs. Non-conventional devices, such as video cameras, touch-screen interactive displays, biometric devices, etc., support the collaborative experience. Sensors and active badges have been used together with telemetry software and a location system to implement *context-aware applications* [20]. *Cooperative buildings* [14] use upgraded versions of mundane objects such as walls, tables, and chairs as computing and display devices.

DACIA allows application components to move from one computing device to another. Our work addresses the problem of capturing the state of a component and restoring it at the destination. To reduce the overhead of component movement, thus the amount of data that has to be moved, we do not transfer the execution state of a PROC (e.g., program counter, stack and registers content, thread state, etc.). 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. A locking mechanism synchronizes message exchange with the PROC move. Our algorithm guarantees that messages in traffic are reliably and orderly delivered to the moving PROC.

In many cases, the temporary failure of a connection between engines can be made transparent to PROCs and applications. When a network connection is broken, the engines will try to re-establish the connection during a timeout interval. Assuming that the disconnection is temporary, an engine caches messages addressed to a remote PROC until the connection is re-established. The use of a timeout for reconnection allows an administrator to briefly shut down an application running on one host, and immediately restart it, without other connected applications noticing it. All inter-component connections are transparently re-established, and neither one of the applications loses any state information. We have found this feature useful during the testing, debugging, and upgrading of distributed applications.

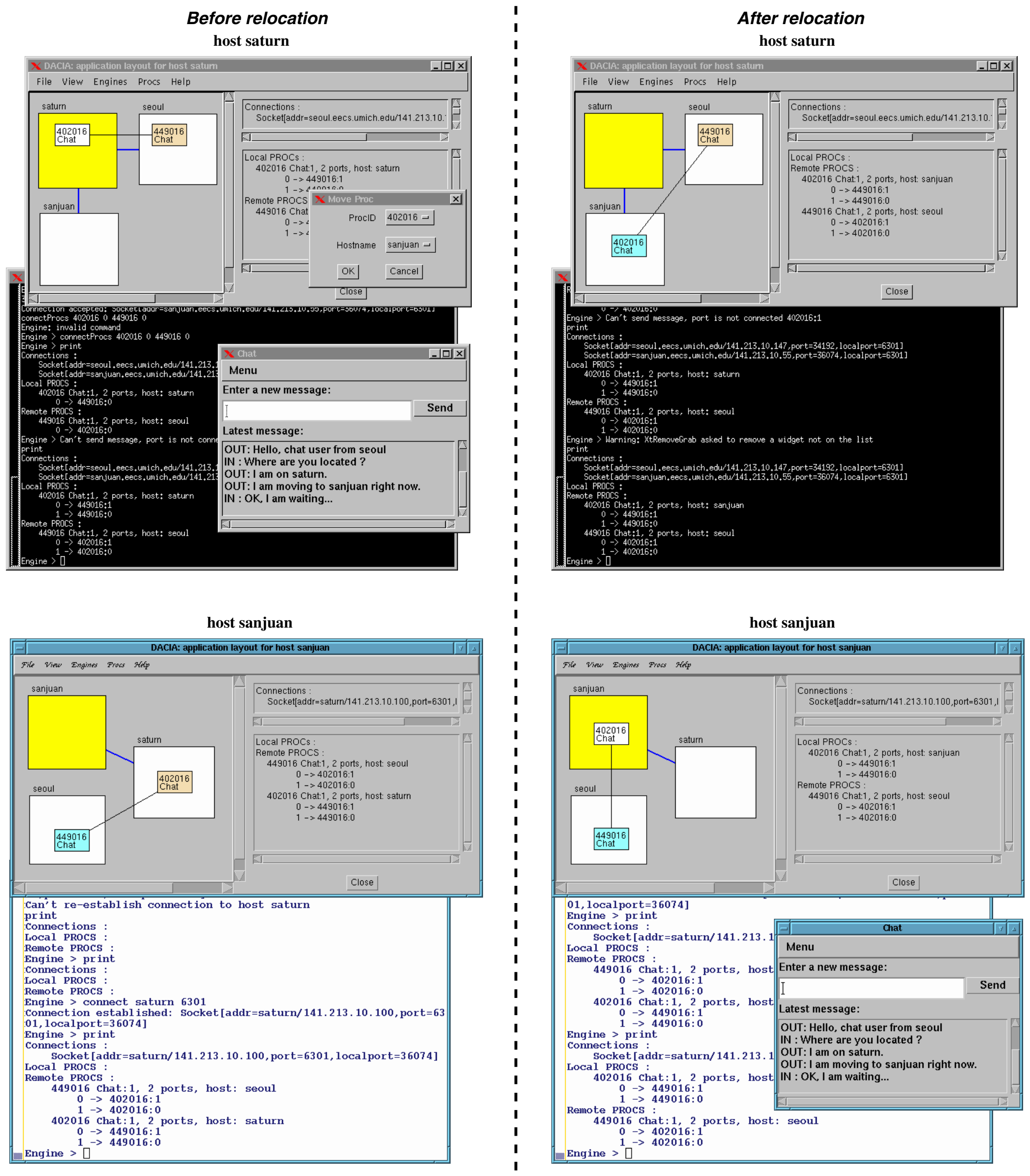
A logical connection between PROCs is maintained even if the underlying physical/network connection changes. A PROC can move between hosts while maintaining persistent connectivity to other PROCs. The move is transparent to communicating PROCs. The structure of the application does not change and the flow of data in the system is not interrupted. The seamless connectivity between DACIA components offers a great benefit to mobile users, who can move applications from one device to another without having to manually re-establish all connections to other parties. It can also provide transparency of the location of PROCs and users, if so desired.

Through mobility, users can 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 interact with it.

3.4. Example of Component Mobility

Figure 2 illustrates a simple example of component mobility, in the case of a chat-box application, 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 *Chat* PROC moves between the two machines and the users can continue to exchange messages without having to explicitly 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.

A PROC is allowed to move from one device to a different type of device that supports DACIA. Eventually, the PROC adapts its functionality and its user interface to the capabilities of the device. For example, the *Chat* PROC can move from a desktop 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.



Before relocation, host saturn:

```
Connections :
   Socket[addr=seoul.eecs.umich.edu/141.213.10.

Local PROCs :
   402016 Chat:1, 2 ports, host: saturn
          0 -> 449016:1
          1 -> 449016:0
Remote PROCS
   449016 Chat
          0 -> 4
          1 -> 4
```

```
Connection accepted: Socket[addr=sanjuan.eecs.umich.edu/141.213.10.55,port=36074,localport=6301]
conectProcs 402016 0 449016 0
Engine: invalid command
Engine > connectProcs 402016 0 449016 0
Engine > print
Connections :
    Socket[addr=seoul.eecs.umich.edu/141.213.1
    Socket[addr=sanjuan.eecs.umich.edu/141.213
Local PROCS :
    402016 Chat:1, 2 ports, host: saturn
        0 -> 449016:0
Remote PROCS :
    449016 Chat:1, 2 ports, host: seoul
        0 -> 402016:0
Engine > Can't send message, port is not conne
print
Connections :
    Socket[addr=seoul.eecs.umich.edu/141.213.1
    Socket[addr=sanjuan.eecs.umich.edu/141.213
Local PROCS :
    402016 Chat:1, 2 ports, host: saturn
        0 -> 449016:1
        1 -> 449016:0
Remote PROCS :
    449016 Chat:1, 2 ports, host: seoul
        0 -> 402016:1
        1 -> 402016:0
Engine > 
```

```
OUT: Hello, chat user from seoul
IN : Where are you located ?
OUT: I am on saturn.
OUT: I am moving to sanjuan right now.
IN : OK, I am waiting...
```

After relocation, host saturn:

```
Connections :
   Socket[addr=seoul.eecs.umich.edu/141.213.10.

Local PROCs :
Remote PROCS :
   402016 Chat:1, 2 ports, host: sanjuan
          0 -> 449016:1
          1 -> 449016:0
   449016 Chat:1, 2 ports, host: seoul
          0 -> 402016:1
          1 -> 402016:0
```

```
        0 -> 402016:0
Engine > Can't send message, port is not connected 402016:1
print
Connections :
    Socket[addr=seoul.eecs.umich.edu/141.213.10.147,port=34192,localport=6301]
    Socket[addr=sanjuan.eecs.umich.edu/141.213.10.55,port=36074,localport=6301]
Local PROCS :
    402016 Chat:1, 2 ports, host: saturn
        0 -> 449016:1
        1 -> 449016:0
Remote PROCS :
    449016 Chat:1, 2 ports, host: seoul
        0 -> 402016:1
        1 -> 402016:0
Engine > Warning: XtRemoveGrab asked to remove a widget not on the list
print
Connections :
    Socket[addr=seoul.eecs.umich.edu/141.213.10.147,port=34192,localport=6301]
    Socket[addr=sanjuan.eecs.umich.edu/141.213.10.55,port=36074,localport=6301]
Local PROCS :
Remote PROCS :
    402016 Chat:1, 2 ports, host: sanjuan
        0 -> 449016:1
        1 -> 449016:0
    449016 Chat:1, 2 ports, host: seoul
        0 -> 402016:1
        1 -> 402016:0
Engine > 
```

Before relocation, host sanjuan:

```
Connections :
   Socket[addr=saturn/141.213.10.100,port=6301,l

Local PROCs :
Remote PROCS :
   449016 Chat:1, 2 ports, host: seoul
          0 -> 402016:1
          1 -> 402016:0
   402016 Chat:1, 2 ports, host: saturn
          0 -> 449016:1
          1 -> 449016:0
```

```
Can't re-establish connection to host saturn
print
Connections :
Local PROCS :
Remote PROCS :
Engine > print
Connections :
Local PROCS :
Remote PROCS :
Engine > connect saturn 6301
Connection established: Socket[addr=saturn/141.213.10.100,port=63
01,localport=36074]
Engine > print
Connections :
    Socket[addr=saturn/141.213.10.100,port=6301,localport=36074]
Local PROCS :
Remote PROCS :
    449016 Chat:1, 2 ports, host: seoul
        0 -> 402016:1
        1 -> 402016:0
    402016 Chat:1, 2 ports, host: saturn
        0 -> 449016:1
        1 -> 449016:0
Engine > 
```

After relocation, host sanjuan:

```
Connections :
   Socket[addr=saturn/141.213.10.100,port=6301,l

Local PROCs :
   402016 Chat:1, 2 ports, host: sanjuan
          0 -> 449016:1
          1 -> 449016:0
Remote PROCS :
   449016 Chat:1, 2 ports, host: seoul
          0 -> 402016:1
          1 -> 402016:0
```

```
01,localport=36074]
Engine > print
Connections :
    Socket[addr=saturn/141.213.1
Local PROCS :
Remote PROCS :
    449016 Chat:1, 2 ports, host
        0 -> 402016:1
        1 -> 402016:0
    402016 Chat:1, 2 ports, host
        0 -> 449016:1
        1 -> 449016:0
Engine > print
Connections :
    Socket[addr=saturn/141.213.1
Local PROCS :
    402016 Chat:1, 2 ports, host
        0 -> 449016:1
        1 -> 449016:0
Remote PROCS :
    449016 Chat:1, 2 ports, host: seoul
        0 -> 402016:1
        1 -> 402016:0
Engine > 
```

```
OUT: Hello, chat user from seoul
IN : Where are you located ?
OUT: I am on saturn.
OUT: I am moving to sanjuan right now.
IN : OK, I am waiting...
```

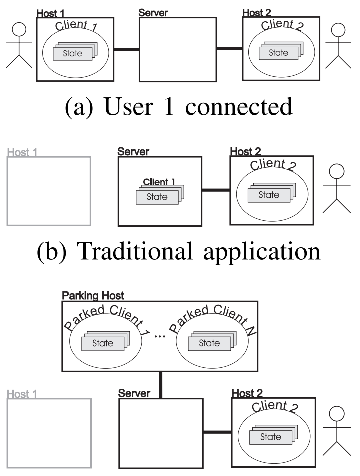
Figure 2. A Chat PROC moves from one host (saturn, top left) to another one (sanjuan, bottom right). All PROCs remain connected and continue to exchange data. The graphical interface windows on each host show the configuration of the application, both for the local and remote hosts. Squares represent hosts, and small labeled rectangles represent PROCs.

3.5. Application Parking

Through *application parking*, component mobility and persistent connectivity 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 host* services to mobile users. When the user's application moves to a different host, it maintains its connections to services and collaborative partners, and it continues its execution.

In current groupware applications (Figure 3.a), when a user disconnects, the disconnection is usually treated as long-term. The state of user's application has to be saved on a server (Figure 3.b). When the user reconnects, typically all connections to collaboration services have to be manually re-established. Using DACIA, the user can park her client to a fixed, connected host. While the user is disconnected, a parked client (Figure 3.c) can continue to maintain state.



(a) User 1 connected

(b) Traditional application

(c) DACIA application

Figure 3. Application parking. (a&b) In traditional groupware applications, when a user disconnects, the state of her application has to be saved on a server. If the user later connects to a different host, the state is transferred between servers and between the new server and client. (c) Using DACIA, the user can park her client to a fixed, connected host. While the user is disconnected, a parked client can continue to maintain state and its connections, and it can interact with collaborative partners.

Moreover, the parked application maintains its connections and it can interact with collaborative partners. 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.

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 informing them that the user is not active (similar to the vacation email message). A more elaborate parked application could save messages, forward notifications to the user via email, or notify users of potential future activity schedule. The parked application's behavior can be made 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 the collaboration.

4. Building Adaptive Groupware Applications with DACIA

4.1. Graphical Interface for Application Management

Tools are needed to allow a system administrator to visualize the component-based structure of DACIA applications (local and remote PROCs and their interconnections, and connections between engines), and to reconfigure the application. We have developed both a command-line interface and a graphical interface for application management. In many 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.

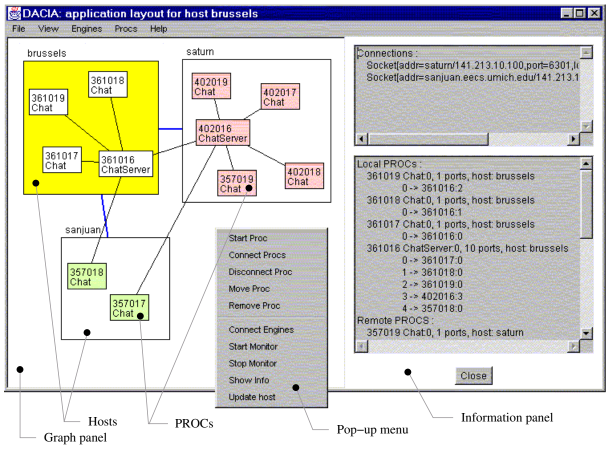
We have built a graphical tool that provides an interactive environment for visualizing the structure of a distributed application and performing manual reconfiguration of the application. We considered the following requirements for the design and implementation of this graphical interface (GUI):

- **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.
- **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. 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. 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. The GUI should be independent of and transparent to the application.

Figure 4 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]. The application presented in the figure resides on 3 hosts, represented by the larger rectangles. The local host (brussels, top left) 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. Commands for modifying the application structure can be issued by selecting an option from one of several menus available.

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.

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



```
Connections :
  Socket[addr=saturn/141.213.10.100,port=6301,l
  Socket[addr=sanjuan.eecs.umich.edu/141.213.1
```

```
Local PROCs :
  361019 Chat:0, 1 ports, host: brussels
         0 -> 361016:2
  361018 Chat:0, 1 ports, host: brussels
         0 -> 361016:1
  361017 Chat:0, 1 ports, host: brussels
         0 -> 361016:0
  361016 ChatServer:0, 10 ports, host: brussels
         0 -> 361017:0
         1 -> 361018:0
         2 -> 361019:0
         3 -> 402016:3
         4 -> 357018:0
Remote PROCS :
  357019 Chat:0, 1 ports, host: saturn
```

Figure 4. The graphical interface (GUI) provides an interactive environment for visualizing the graph structure of a distributed application and performing manual reconfiguration of the application.

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.

4.2. Structuring Distributed Applications

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 5) can be classified according to their characteristics and functionality as follows:

- *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 other applications. 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.

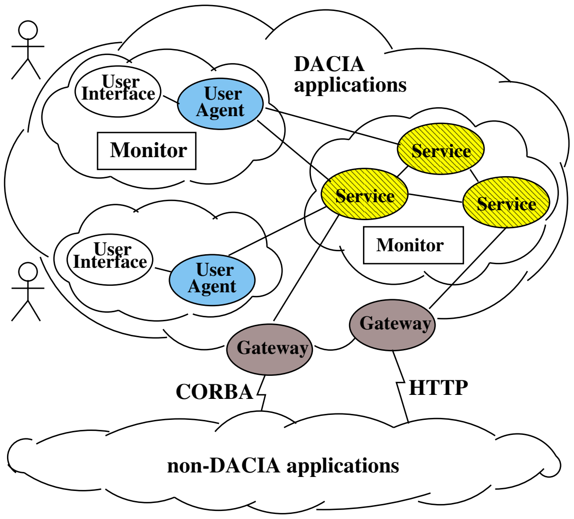


Figure 5. A DACIA collaborative application may contain PROCs of various types: User Interface, User Agent, Service, and Gateway.

A client-side application consists of User Interface and User Agent PROCs. If desired, they can be combined into a single PROC. This is the case of the *Chat* client used by the application in Figure 2. The system can be potentially enhanced by adding sensors that detect a user's presence next to a host, as in [20]. 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 User Interface and User Agent PROCs is useful if the client is expected to run with different interfaces on various devices. It simplifies development—similar to the separation of Model 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. The user uses the unique PROC identifiers in order to locate and reconnect to its agent.

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, in Figure 5, one monitor can manage the interactions between PROCs implementing a service, while another monitor controls a client-side application.

4.3. Multi-party Communication

Using DACIA, we have implemented an adaptive multi-party communication application that can be used as a basis for building various groupware systems. The application consists of client (C) and server (S) PROCs (Figure 6). 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

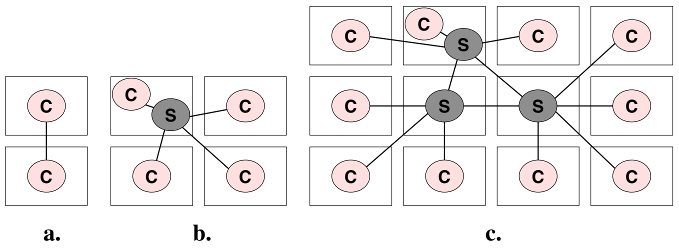
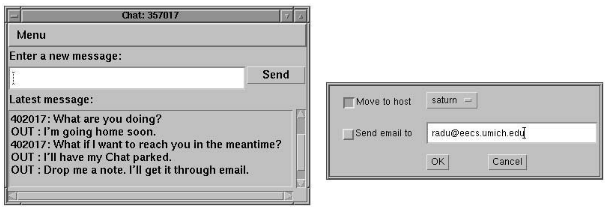


Figure 6. Adaptive multi-party communication. Servers are denoted by S, and clients are denoted by C. Rectangles represent hosts. New servers are created as the number of participants grows.



```
402017: What are you doing?
OUT : I'm going home soon.
402017: What if I want to reach you in the meantime?
OUT : I'll have my Chat parked.
OUT : Drop me a note. I'll get it through email.
```

Send email to: radu@eecs.umich.edu

Figure 7. 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.

connected directly (Figure 6.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 6.b).

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, an engine spawns a new server, which connects to the existing servers. The clients are distributed over the two 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 equally assigned to all servers[3] (Figure 6.c). As clients leave the group and the load on a server goes below a lower threshold, a server module is deallocated and its clients are distributed to other servers. Alternatively, two servers with load under the lower threshold can be replaced by a single server supporting all their clients.

DACIA only provides support for ordered delivery of messages along a channel between two PROCs. Messages originating at different clients may arrive in any order at different destinations. In multi-party communication, sometimes stronger guarantees such as totally ordered message delivery may be required. To provide totally ordered 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 6, the servers were stateless. They simply routed messages and no consistency of state among the servers was 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 to the system, that maintains the group's state. In future versions of DACIA, we plan to provide support for replicating components and maintaining consistency of their states.

The application structure and the adaptive algorithms presented can be used to implement various server-based group communication applications. Below, we describe two applications we developed on top of this group communication service: a multi-party chat-box application and a shared whiteboard. Little programming effort is required to develop these specific collaborative applications, as well as other applications. In fact, we keep the base application in Figure 6 as it is. Client components (C) are seen as user agents. The personalization consists of adding user interface components corresponding to the chat-box and whiteboard applications, respectively. These interface components are responsible for interpreting the group messages exchanged as either chat messages or updates to a shared drawing, and for appropriately presenting the information to the user. All collaboration features (e.g., group membership, awareness, notifications) are implemented between user agents. The same monitor can be used for dynamically reconfiguring both applications to scale up to a large number of clients and to reduce communication latencies.

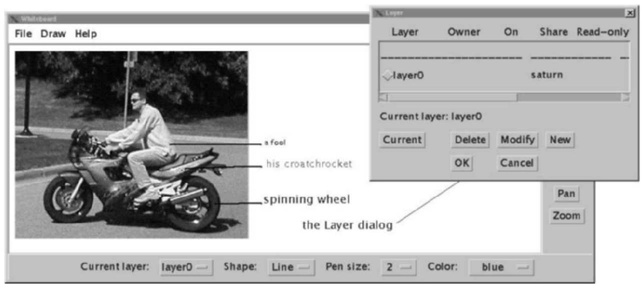
[3]clients' locations are also a factor in choosing an appropriate server

4.3.1. 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 7). A message sent by one user will be distributed to the whole group through the servers.

Chat PROCs are mobile. When a PROC moves to a different host, its state 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. The interface component usually changes when the Chat client moves to a different type of device, e.g., from a desktop to a PDA.

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 it can 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.

4.3.2. Whiteboard: Acting both as a shared notebook and a drawing board, the whiteboard (Figure 8) allows users to



Current layer: layer0

Current layer: layer0 Shape: Line Pen size: 2 Color: blue

Figure 8. 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.

collaboratively draw figures, take notes, and import and share images. The basic drawing elements are line, point, and text. Raster images (e.g., .gif and .jpg) can be loaded from the local file system as the background.

The graphic 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 or servers. 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.

4.4. Easy Application Development

We have implemented the DACIA framework, as well as several applications, in Java. 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.

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. 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.

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

  public Chat() {
    super("Chat", 1); // name, 1 port
    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); // port 0, message, 1-synchronous
  }
  public void pack() {
    text = frame.getText();
    frame.quit();
    frame = null;
  }
  public void unpack() {
    frame = new ChatFrame(this);
    frame.init();
    frame.displayText(text);
  }
}
```

Figure 9. Code added to a previously existing Java object to make it a mobile PROC, in the case of a *Chat* object.

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. For the multi-user chat-box application in Section 4.3.1, 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 9). The Chat PROC has one port. It displays to an output text area the content of the messages received on the port, and it sends to the 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.

The programming API can also be used to write application-specific adaptive monitors. For example, we used 54 lines of code to write the monitor used by the multi-party communication application in Section 4.3 to allocate and deallocate servers, and to balance the load among servers.

5. Conclusions

Our work focuses on the use of reconfigurable component-based applications to support the specific needs of mobile users. We are particularly concerned with the application and resource availability and the intermittent connectivity in a mobile collaboration environment. In this paper, we presented DACIA, a mobile component framework used to develop applications that can adapt to environmental changes. They can dynamically reconfigure by loading new components, changing the way components interact and exchange data, and moving components from one host to another. DACIA provides support for application and user mobility across heterogeneous devices, and it enables persistent connectivity between moving components.

We presented a taxonomy of components that can be used to structure collaborative applications. We showed several reconfigurable groupware applications implemented using DACIA. Applications include groupware clients that relocate based on user's location, and mobile clients that can be parked while their users are disconnected. A parked client may continue to participate, on a limited basis, to collaborations on the user's behalf. DACIA has also been used to build collaboration services that adapt to available resources and the number of users. Such reconfigurable services are useful even when users are not mobile.

We are currently extending DACIA to address the security concerns of both mobile components and the hosts where they execute. We are also in the course of developing a set of general-purpose adaptive policies that optimize the performance of distributed applications through dynamic reconfiguration and component relocation. Future challenges include using and evaluating DACIA-enabled collaboration tools in real collaboration settings, and providing access control features for users to control the applications' structure and the mobility of their components.

6. Acknowledgments

This work has been supported in part by the National Science Foundation under Grant No. ATM-9873025.

References

[1] M. Weiser, "Hot Topics: Ubiquitous Computing," *IEEE Computer*, Oct. 1993.

[2] 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.

[3] M. Roseman and S. Greenberg, "Building Flexible Groupware through Open Protocols," in *Proceedings of the ACM Conference on Organizational Computing Systems*, California, 1993.

[4] 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.

[5] S. Greenberg and M. Boyle, "Moving Between Personal Devices and Public Displays," Nov. 1998.

[6] W. K. Edwards, "Policies and Roles in Collaborative Applications," in *Proceedings of the ACM 1994 Conference on Computer-Supported Cooperative Work (CSCW '96)*, Boston, MA, Nov.. 1996, pp. 11–20.

[7] 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)*, 1992, pp. 51–58.

[8] 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)*, Chapel Hill, NC, Oct. 1994, pp. 207–217.

[9] P. Dewan and R. Choudhary, "Coupling the User Interfaces of a Multiuser Program," *ACM Transactions on Computer Human Interaction*, vol. 2, no. 1, pp. 1–39, March 1995.

[10] G. Fitzpatrick, S. Kaplan, and T. Mansfield, "Physical Spaces, Virtual Places and Social Worlds: A study of Work in the Virtual," in *Proceedings of the 1996 ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, Boston, MA, Nov. 1996, pp. 334–343.

[11] 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)*, Boston, MA, Nov. 1996, pp. 344–353.

[12] S. E. Hudson and I. Smith, "Techniques for Addressing Fundamental Privacy and Disruption Tradeoffs in Awareness Support Systems," in *Proceedings of the 1996 ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, Boston, MA, Nov. 1996, pp. 248–257.

[13] E. Horvitz, A. Jacobs, and D. Hovel, "Attention-sensitive Alerting," in *Proceedings of the 15th Conference on Uncertainty and Artificial Intelligence*, Stockholm, Sweden, July 1999, pp. 305–313.

[14] 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*, pp. 4–21, 1998.

[15] 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)*, Boston, MA, Nov. 1996, pp. 209–218.

[16] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. on Software Engineering*, vol. 24, no. 5, May 1998.

[17] D. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[18] 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.

[19] 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)*, Austin, TX, May 1999, pp. 190–197.

[20] 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.

[21] 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.

[22] 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)*, Boston, MA, Aug. 2000, pp. 266–274.

[23] R. Barr, J. Bicket, D. Dantas, B. Du, T. D. Kim, B. Zhou, and E. Sirer, "On the Need for System-Level Support for Ad hoc and Sensor Networks," in *ACM Operating Systems Review*, April 2002.