Developing Adaptive Groupware Applications Using a Mobile Component Framework

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

A need exists to develop groupware systems that adapt to available resources and support user mobility. This paper presents DACIA, a system that provides mechanisms for building such groupware applications. Using DACIA, components of a groupware application can be moved to different hosts during execution, while maintaining communication connectivity with groupware services and other users. DACIA provides mechanisms that simplify building groupware for domains where users are mobile. New collaboration features can be also more easily implemented. DACIA is also applicable to non-mobile environments. We show its applicability to building groupware applications that can be reconfigured at run-time to adapt to changing user demands and resource constraints, for example, by relocating services or introducing new services. This paper describes the architecture of DACIA and its use in building adaptable groupware systems.

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**: Collaborative groups can vary significantly in group size. 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.

• **User mobility and intermittent connectivity**: Users are increasingly mobile. They connect from various points, using a variety of devices. In some cases, it is desirable that a user's applications be able to participate in collaborative activities on behalf of the user on a limited basis, even while the user is disconnected or inactive. In domains such as cooperative buildings [21], users can move from one work area to another. In such domains, 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 establish manually the sessions.

• **Hardware and network variability**: The devices used range from high-end machines, with significant computing power, memory, and graphic display capabilities, to simple devices such as PDAs (personal digital assistants), that can only display text or primitive graphics. Network links characteristics in terms of delay, capacity, and error rate can vary significantly. The ideal architecture of the system depends on the available computing and network resources.

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.

We have developed a 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 can continue to interact, with some limitations, 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, he can take over the control from the parked application.

In this paper, we explore the use of DACIA to model and develop adaptive collaborative systems. The current version of DACIA is implemented using Java and runs on standard desktops as well as on PDAs (e.g., Windows CE devices) that support PersonalJava. It has been used to implement several prototypes of groupware applications that illustrate support for mobility and reconfigurability.

The outline of the paper is as follows. We first present related work, followed by the architecture of DACIA. Next, we illustrate the support provided by DACIA for building collaborative applications for mobile environments. Then we show how DACIA can be used to build and execute reconfigurable groupware applications that adapt to available resources. Finally, we present some concluding remarks and directions for future work.

[1] Dynamic Adjustment of Component InterActions

RELATED WORK

DACIA enables the design of more flexible and adaptable CSCW systems. The importance of flexibility and adaptability in CSCW systems has been discussed by several researchers [3, 19]. It is important to understand where this paper fits in that context. We believe that in fact there are many dimensions to flexibility and adaptability in CSCW systems. Some of these dimensions include:

- access control [8, 20];
- concurrency control [7, 11];
- coupling of views [6];
- extensible architectures [10, 16].

The key point that these researchers make is that there are significant tradeoffs in CSCW system design along many dimensions and many of these tradeoffs in fact cannot be made *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 support for adapting the architecture of CSCW systems and location of system components to the usage context, scale of use, location of users, and to available resources.

The importance of considering resources has been pointed out by other researchers. Hudson and Smith point that CSCW systems may need to be designed to allow tradeoffs between context awareness and available resources (CPU, display, network) [14]. 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. In DACIA, our goal is to provide the mechanisms to CSCW system designers so that the systems and their architecture can be more easily reconfigured, at run-time if desired.

The work of Chung and Dewan [5] has some goals similar to ours, such as migrating applications to make better use of the available resources, and accommodating heterogeneous

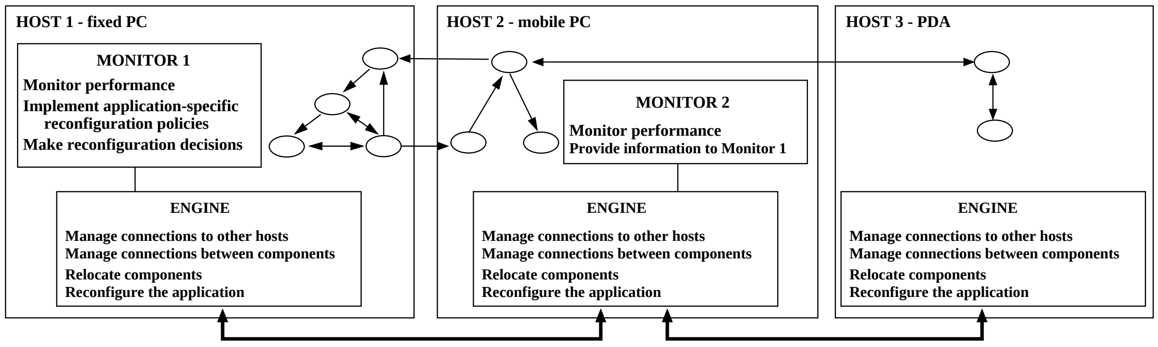

HOST 1 - fixed PC
MONITOR 1
Monitor performance
Implement application-specific reconfiguration policies
Make reconfiguration decisions
ENGINE
Manage connections to other hosts
Manage connections between components
Relocate components
Reconfigure the application
HOST 2 - mobile PC
MONITOR 2
Monitor performance
Provide information to Monitor 1
ENGINE
Manage connections to other hosts
Manage connections between components
Relocate components
Reconfigure the application
HOST 3 - PDA
ENGINE
Manage connections to other hosts
Manage connections between components
Relocate components
Reconfigure the application

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.

environments. Their approach targets the specific application domain of centralized shared Windows systems. It is based on the migration of a X Windows client that receives inputs generated by multiple users, and the migration of the events logged at a particular site. DACIA targets a broader class of applications, in which application components, including services, can move from one host to another to adapt to changing usage conditions.

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 [21]. 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 [2]. 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 of groupware applications in which clients are mobile.

Code mobility [9] has received a great deal of attention in the distributed systems research community. Several recent languages support various flavors of code mobility, such as Telescript [22], Obliq [4], and Sumatra [1]. In TACOMA [15], the term *mobile agent* is used to denote code fragments associated with initialization data that can be shipped to a remote host. TACOMA agents do not have the ability to migrate once they started execution. In contrast, DACIA's components can move during the execution of an application.

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 represent the basic building blocks for an application. 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 communicate by exchanging *messages* through *input and output ports*. We support both the blocking semantics

[2]Processing and ROuting Component

```
connect [hostname] [IPportnumber] - connect the local engine to another engine
connectProcs [sourceProcID] [sourcePortNo] [destProcID] [destPortNo] - connect two PROCs
disconnectProcs [sourceProcID] [sourcePortNo] - disconnect two PROCs
exit/quit - stop execution and exit
help - print a help menu
move [procID] [hostname] - move a PROC to the host indicated
print - print information about the local and remote PROCs and the application configuration
start [procID] - trigger an action on the PROC indicated
startMonitor - start the monitoring service that performs runtime adaptation
update [hostname/all] <allProcs> - updates the information about PROCs known by other engines
```

Figure 2: Command-line shell interface. A programming API offers application programmers similar primitives. For example, two PROCs can be connected using the call: *boolean connectProcs(int procID1, int port1, int procID2, int port2)*.

provided by RPC and *asynchronous message passing*. For *synchronous communication*, the PROCs are located on the same host, in the same address space. To reduce overheads, the thread that executes the action associated with the source PROC will also execute the message handling routine of the connected PROC. In the asynchronous case, 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.

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 approach used by FarGo [13], called *dynamic application layout*, also separates the programming of the layout of the application from the policy logic. The changes of an application layout in FarGo consist of finding the right place to execute components. We go further, allowing an application to dynamically change the connections between components, to introduce new components, and to change its structure.

The monitors are optional – applications can be also reconfigured manually by system administrators, for example by using the shell interface in Figure 2. 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. A programming API is also available with similar functionality and is typically used by the monitors to make reconfiguration changes.

Further details about implementation aspects of DACIA can be found in [18].

MOBILITY AND INTERMITTENT 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* [12]. *Cooperative buildings* [21] use upgraded versions of mundane objects such as walls, tables, and chairs, as computing and display devices.

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 but not yet handled, and the state of its connections. When a PROC moves to another host, all the messages left in its message queue move with the PROC. Java object serialization, with extensions for serializing buffered messages, is used to transfer a PROC's state to another engine. 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.

The temporary failure of a connection between engines is made transparent to the PROCs. When a network connection is broken, the engine caches messages addressed to a remote PROC until the connection is re-established, assuming that the disconnection is temporary.

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]. The movement of a PROC is transparent to other PROCs. The virtual connection between PROCs is permanently maintained, even if the underlying physical connection changes. Messages addressed to a PROC that has left a host will be forwarded to the PROC's new location.

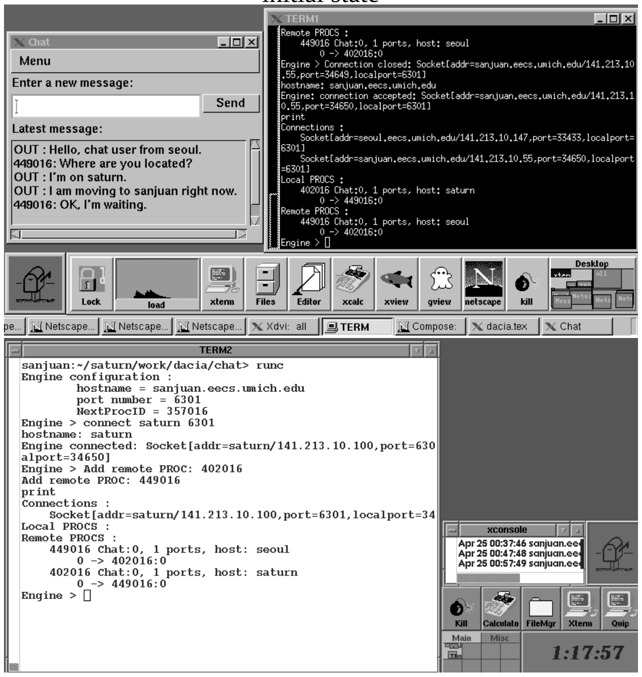
This seamless connectivity offers a great benefit to mobile users, who can move applications from one host to another without having to re-establish all the connections to other parties. It can also provide transparency of the location of a user, if so desired.

A Simple Example of Component Mobility

Figure 3 illustrates a simple example of component mobility, in the case of a chatbox application, that has been implemented using DACIA. Two chatbox users are involved in a session from their respective workstations. At some point, one of the users moves to a different device, which is currently allowed to be another desktop or a Windows CE PDA that runs PersonalJava (available from www.javasoft.com). The user issues a *move()* command using the command-line

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

initial state

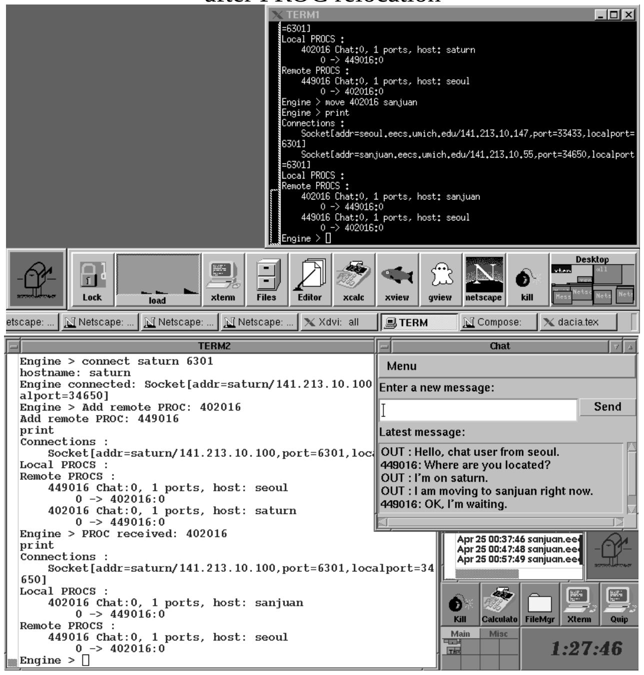


```
OUT : Hello, chat user from seoul.
449016: Where are you located?
OUT : I'm on saturn.
OUT : I am moving to sanjuan right now.
449016: OK, I'm waiting.
```

```
Remote PROCS :
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
Engine > Connection closed: Socket[addr=sanjuan.eecs.umich.edu/141.213.10
.55,port=34649,localport=6301]
hostname: sanjuan.eecs.umich.edu
Engine: connection accepted: Socket[addr=sanjuan.eecs.umich.edu/141.213.1
0.55,port=34650,localport=6301]
print
Connections :
    Socket[addr=seoul.eecs.umich.edu/141.213.10.147,port=33433,localport=
6301]
    Socket[addr=sanjuan.eecs.umich.edu/141.213.10.55,port=34650,localport
=6301]
Local PROCS :
    402016 Chat:0, 1 ports, host: saturn
        0 -> 449016:0
Remote PROCS :
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
Engine > 
```

```
sanjuan:~/saturn/work/dacia/chat> runc
Engine configuration :
        hostname = sanjuan.eecs.umich.edu
        port number = 6301
        NextProcID = 357016
Engine > connect saturn 6301
hostname: saturn
Engine connected: Socket[addr=saturn/141.213.10.100,port=630
alport=34650]
Engine > Add remote PROC: 402016
Add remote PROC: 449016
print
Connections :
    Socket[addr=saturn/141.213.10.100,port=6301,localport=34
Local PROCS :
Remote PROCS :
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
    402016 Chat:0, 1 ports, host: saturn
        0 -> 449016:0
Engine > 
```

after PROC relocation



```
=6301]
Local PROCS :
    402016 Chat:0, 1 ports, host: saturn
        0 -> 449016:0
Remote PROCS :
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
Engine > move 402016 sanjuan
Engine > print
Connections :
    Socket[addr=seoul.eecs.umich.edu/141.213.10.147,port=33433,localport=
6301]
    Socket[addr=sanjuan.eecs.umich.edu/141.213.10.55,port=34650,localport
=6301]
Local PROCS :
Remote PROCS :
    402016 Chat:0, 1 ports, host: sanjuan
        0 -> 449016:0
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
Engine > 
```

```
Engine > connect saturn 6301
hostname: saturn
Engine connected: Socket[addr=saturn/141.213.10.100
alport=34650]
Engine > Add remote PROC: 402016
Add remote PROC: 449016
print
Connections :
    Socket[addr=saturn/141.213.10.100,port=6301,loca
Local PROCS :
Remote PROCS :
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
    402016 Chat:0, 1 ports, host: saturn
        0 -> 449016:0
Engine > PROC received: 402016
print
Connections :
    Socket[addr=saturn/141.213.10.100,port=6301,localport=34
650]
Local PROCS :
    402016 Chat:0, 1 ports, host: sanjuan
        0 -> 449016:0
Remote PROCS :
    449016 Chat:0, 1 ports, host: seoul
        0 -> 402016:0
Engine > 
```

```
OUT : Hello, chat user from seoul.
449016: Where are you located?
OUT : I'm on saturn.
OUT : I am moving to sanjuan right now.
449016: OK, I'm waiting.
```

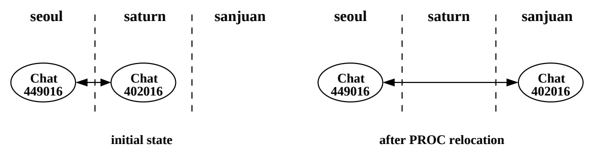


Figure 3: A Chat PROC moves from one machine (saturn, top) to another one (sanjuan, second from top). All the PROCs remain connected and they continue to exchange data. The terminal windows on each screen show application status information for both local and remote PROCs, as displayed by the command-line interface. Machines can be desktop machines that support Java or PDA devices (e.g., Windows CE) that support PersonalJava JVM.

interface given previously (see output in the terminal window). The *Chat* PROC moves between the two machines and the users can resume communication without having to re-establish the connection. The messages previously exchanged (the state of the moved PROC) are still displayed in the window.

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 the PROCs.

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 interact with it.

Application/Client Parking

Our work is particularly concerned with the application and resource availability and the intermittent connectivity in a mobile collaboration environment. Our goal is to ensure that a user's applications are available, in a suitably adapted form, wherever the user goes. At the same time, we want to allow a user's applications to participate on a limited basis in collaborations on the user's behalf, while the user is disconnected or not active.

A potential solution is *application parking*, in conjunction with *persistent connectivity*[4], to address these issues. A parked application is able to continue, with some limitations, to interact with other parties on behalf of the user while the user is idle. 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.

[4] here we refer to logical connectivity

In current groupware applications (Figure 4.a), when a user disconnects, the disconnection is typically treated as long-term. Other users are typically aware that the user is no longer participating but no further information is available as to how long or whether asynchronous interactions are still possible. Furthermore, when the user reconnects, typically all the connections to collaboration services have to be re-established.

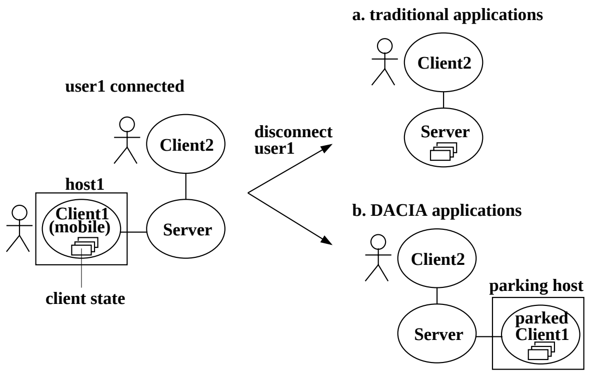


Figure 4: Using traditional groupware applications, when a user disconnects, its state has to be saved on the server. If the user later connects to a different server, the state has to be transfered between the servers and between the new server and client. Using DACIA-enabled applications, while the client is disconnected, its state is maintained by the parked client, which can continue to participate to collaborative activities.

Using DACIA, the user can park his client agent to a fixed, connected host. While the user is disconnected, a parked client (Figure 4.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 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 in the collaboration.

DYNAMIC APPLICATION RECONFIGURATION

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. As a result of reconfiguration, the application graph changes. A more efficient execution can be achieved by better using the available resources and optimizing inter-component communication.

DACIA provides mechanisms for dynamically reconfiguring an application. These mechanisms are the same as shown in Figure 2. They can be used to dynamically connect and disconnect PROCs, introduce new PROCs in the data paths, and move PROCs across hosts.

There is a separation between reconfiguration policies and mechanisms. An application developer or a system administrator can implement customized policies in monitors or can manually reconfigure the application based on changing runtime requirements and resource availability.

Below, we give two examples of using DACIA to implement reconfigurable groupware services that also apply to situations where users are not mobile.

Example 1: Need for Multiple Architectures

We have previously faced the challenges of building flexible collaborative applications in the context of the UARC [17] project, an experimental testbed 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, it applies some computations (e.g., transforming raw data into GIF images), and then disseminates the resulting data to the clients. Figure 5.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.

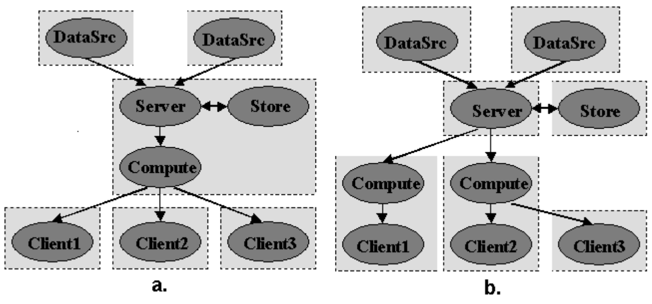


Figure 5: Alternative configurations for an application. Ovals represent components. Grey rectangles represent hosts. Components are connected through directed links, indicating the direction of the data flow within the application. Multiple graphs (a-b) may correspond to the same application.

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 produced images with bigger size than the size of the raw data. Therefore, the network links from the server to some clients also sometimes got congested.

The above problems can be potentially alleviated by using an alternative architecture where the server sends the raw data to the clients and clients do the image computations. 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 implemented an adaptive version of this application. This version allowed the computing function to be executed on the server, on the client, or on any other host with a DACIA engine. The application structure could change at runtime, according to the load and the resource availability. Figure 5.b presents an alternative configuration created using DACIA, that uses several *Compute* modules located on the same or nearby hosts as the clients. The simplified reconfiguration policy that we implemented only took network bandwidth into account. Preliminary performance experiments (not presented in this paper) indicated that the system was indeed able to adapt better to network constraints via reconfiguration of Compute PROCs.

DACIA can also allow 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 may be more efficient than the other ones.

The insertion of new processing nodes, such as a pair of *Compress/Decompress* components is done as follows. Assume that *Sender* and *Receiver* are two PROCs exchanging data, being connected on their respective ports 0. The following sequence of operations allows the insertion of a pair of PROCs, *Compress* and *Decompress*, without corrupting the data exchanged. All the messages that are sent by *Sender* after the disconnection will be first compressed and then decompressed, before being delivered to *Receiver*. In DACIA, messages are buffered for a limited duration to allow reconfigurations that require disconnecting and reconnecting PROCs.

```
disconnectProcs(Source, 0)
connectProcs(Sender, 0, Compress, 0)
connectProcs(Decompress, 1, Receiver, 0)
connectProcs(Compress, 1, Decompress, 0)
```

Example 2: Multi-Party Communication

Using DACIA, we have implemented an adaptive multi-party communication application that can be used as a basis of 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 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). Assuming that the clients are *C1*, *C2*, and *C3*, the sequence of operations[5] involved is (arguments are procID's and portNo's):

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

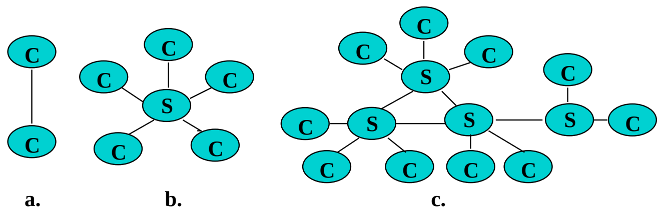


Figure 6: Adaptive multi-party communication. Servers are denoted by S, and clients are denoted by C. New servers are created as the number of participants grows.

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. The clients are distributed among the servers. Ideally, the distribution should take into account clients' relative locations. Figure 7 presents the part of the monitor responsible for allocating new servers and balancing the load among servers.

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

```
// 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 7: The monitoring routine 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.

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. Alternatively, two servers with load under N_{min} 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. In multi-party communication, sometimes stronger guarantees such as total ordered 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 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 that 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.

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

CONCLUSIONS

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. Applications include groupware clients that relocate, based on the user's location, and mobile clients that can be "parked" while their users are disconnected. A parked client may continue to interact with other parties in specified ways on behalf of the user. 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 believe that DACIA provides a platform for experimenting with new kinds of groupware applications. Some future challenges include implementing a library of useful PROCs, using and evaluating DACIA-enabled collaboration tools in real collaboration settings, providing access control features for users to control the mobility of their components, and supporting component replication.

ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under Grant No. ATM-9873025. The authors would like to thank Tom Finholt for his useful comments on an earlier draft of the paper. We also would like to thank Prasun Dewan for suggesting the multi-party communication application as a potential candidate for use of DACIA.

REFERENCES

1. 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.
2. 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.
3. 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.
4. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
5. 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 11–20, Nov. 1996.
6. 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.
7. 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.
8. 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.
9. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), May 1998.
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 1996 the ACM Conference on Computer-Supported Cooperative Work, (CSCW '96)*, pages 334–343, Boston, MA, Nov. 1996.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. R. Litiu and A. Prakash. DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. Technical Report CSE-TR-416-99, University of Michigan, EECS, Dec 1999.
19. M. Roseman and S. Greenberg. Building Flexible Groupware through Open Protocols. In *Proceedings of the ACM Conference on Organizational Computing Systems*, California, 1993.
20. 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.
21. 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.
22. J.E. White. Telescript Technology: Mobile Agents. *Software Agents, J. Bradshaw, ed. AAAI Press/MIT Press*, 1996.