DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications

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

Abstract

Current computing systems exhibit an impressive heterogeneity of hardware resources, software capabilities, and network connectivity. They have to accommodate various application and client demands and to adjust to the continuously changing load on network segments and intermediate processing nodes. We present DACIA[1]*, a framework for building adaptive distributed applications through the flexible composition of software modules implementing individual functions. An application is seen as a collection of components connected in a directed graph. Through runtime reconfiguration of the application graph, the application adapts to changes in the execution environment, resource availability, and application requirements. Thus, a more efficient execution is achieved. The location where various components are executed can be changed dynamically and the execution can be occasionally outsourced to nodes having specialized hardware or to less loaded sites on the network. Some components or groups of components can be replaced by other components or groups of components. DACIA supports application and user mobility, allowing parts of an application to move to different hosts at runtime, while maintaining seamless connectivity.*

1 Introduction and Motivation

The explosive growth of the World Wide Web and the proliferation of Internet applications and services pose significant challenges to system developers, namely dealing with the heterogeneity encountered within the end systems and across the network infrastructure, as well as handling the variety of user and application demands. Current computing systems exhibit an impressive heterogeneity in terms of hardware resources, from high-end machines, with significant computing power, memory, and graphic display capabilities, to simple devices, such as PDAs, that can only display text or primitive graphics. They are connected through network links whose characteristics in terms of delay, capacity, and error rate can vary by many orders of magnitude. They have to accommodate various application and client demands (e.g., image quality, latency, video frame rate, guaranteed data delivery) and to adjust to the continuously changing load on network segments and intermediate processing nodes. To manage this highly variable environment, *the communication infrastructure and the services provided have to adapt to the variations in resource availability and application requirements.* Heterogeneity and variability are even more stringent problems in mobile computing environments, which nowadays are becoming more and more ubiquitous.

To present a concrete example, consider UARC [9], 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 the subscriptions from multiple clients and the distribution of data to these clients (Figure 1). The server

[1] Dynamic Adjustment of Component InterActions

receives large amounts of raw data from various data sources, representing some instruments that collect data in real time. The server caches this data for fault tolerance and for future access. It also applies some computations transforming the raw data into GIF images. Then it sends the images to the interested clients using a group communication service.

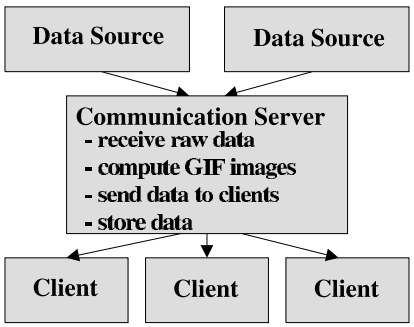

Figure 1: A collaborative application

We have encountered several problems with 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 of which requires a different computation task to run on the server. Consequently, the server may not have the capability to compute in real time the images for all the subscriptions. Second, most of the time the computations produce images with bigger size than the size of the raw data. Therefore, the network links from the server to some clients can get congested. Finally, these problems can potentially be alleviated by using an alternative architecture where the server sends the raw data to the clients and clients do the image computations. However, our experience from UARC was that some clients get overloaded if they are computing many images. To address this problem, some images can be computed on nearby hosts.

The work described in this paper allows us to change at runtime the way the application is structured, to move some of the functions of the application from one host to another, and to replicate some functions across multiple hosts.

Users are inherently mobile, regardless whether they work on their desktop computer or they use some mobile computing device. They run simultaneously multiple applications, some of which are part of a distributed computing environment, being connected to services, data sources, collaborative partners, etc. There are numerous situations in which a user would not want to shutdown all the applications he is running, cut all the connections with other parties, and quit all the login sessions he has established, in order for a minute later to move to a different place and restart the very same applications, establish manually the same connections, and also import the latest changes to some configuration files, that he has just made earlier.

Our work allows a mobile user to simply "pull" an application from the desktop of one computer and drop it on a new desktop. In this way, no manual restart is necessary and no connection has been broken and needs to be re-established. This seamless connectivity also provides transparency of the location of the user. Moreover, the user can move an application from some device to a totally different device, e.g., from a full-graphics computer screen to a PDA having a small text display. The application adapts to the new device and it presents to the user an interface appropriate to the device capabilities.

In this paper we present DACIA, a framework that addresses the challenges of building customized distributed applications that perform runtime adaptation and provides support for component mobility. The *modular architecture* adopted enables the *dynamic reconfiguration of a distributed application by reordering application components or moving components across machine boundaries.* We show that the overhead introduced by using modularity is minimal and that often we can actually reduce the communication costs in distributed applications. We give an example of using code mobility and reconfiguration to improve the performance of a simple test application.

The rest of the paper is organized as follows: Section 2 identifies the objectives we pursued in designing

DACIA. Section 3 examines existing systems and technologies covering areas related to our work. Section 4 presents our approach to providing a framework for flexible component composition. Section 5 shows some performance results obtained with small test applications implemented using DACIA. Section 6 presents our conclusions and outlines some of the current and future challenges for our work.

2 Design Goals

In designing DACIA, we considered the following issues concerning the design, implementation and execution of distributed applications for the Internet:

• **Managing Heterogeneity and Adapting to Variation**

Different distributed applications place different requirements on the communication infrastructure and supporting services. Users running the same applications may have different needs and different capabilities. The responsibility to adapt to changing demands and variations in resource availability is shared between the applications and the infrastructure. Adaptation is based on both local knowledge and information obtained from other entities along the execution path.

• **Modular Design**

To manage complexity, isolate failures, and facilitate the construction of customized and flexible configurations, a modular architecture offers an appropriate solution. In this architecture, applications are built through the composition of software components implementing individual operations. To allow flexible composition, components interact through standardized interfaces. The use of a modular architecture allows components to be reused across multiple applications.

• **Runtime Reconfiguration of the Application**

The simple composition of software modules implementing various functions of a system does not offer a satisfactory solution. The way these modules interact cannot always be fully determined statically and is greatly affected by variations in user needs and capabilities and the load on various nodes and network segments. *Runtime composition* and *dynamic adjustment of the order and the location of execution* of different components can significantly improve the performance of the system and the global usage of the available resources. In some cases, by deferring some operations, their execution can be avoided. Finding and using spare resources leads to more efficient scheduling of concurrent operations, thus optimizing the execution.

• **Support for Application and User Mobility**

An application or part of an application running on some computer should be able to move to a different computer without (visibly) stopping its execution or impacting other applications it communicates with. The movement should not affect the connectivity of the application. The same application should be able to execute on different devices with different capabilities (e.g., desktop computer, PDA, cell phone, etc). The application will present to the user an interface and a set of features corresponding to the capabilities of the device.

• **Performance**

The use of modularity should not introduce significant overheads compared with a monolithic implementation. Component mobility and application reconfiguration should contribute to reducing the overall cost of communication within a distributed application. The implementation of component mobility should be efficient and robust.

3 Related work

The idea of building computing systems through the composition of individual modules is not new; it has been used extensively in the design and implementation of systems ranging from layered operating systems [14] and network architectures [20] to more advanced distributed systems.

A frequently used approach is to view an application as a *protocol stack*. Horus [16] and its ML implementation, Ensemble [5], treat protocols as abstract data types that can be stacked on top of each other in a variety of

ways at runtime. Protocol modules have standardized top and bottom interfaces and they communicate with each other through message passing. Horus provides an object-oriented protocol composition framework; it supports objects for communication endpoints, groups of communicating endpoints, and messages.

The x-kernel [7] is an operating system kernel that provides an object-oriented framework designed to support the rapid implementation of efficient network protocols. It provides a uniform protocol interface and support library that allows the programmer to configure individual *protocol objects* into a *protocol graph* that realizes the required functionality.

Scout [12] is a communication-oriented operating system built on the foundation provided by the *path* abstraction. A path represents the flow of data from an I/O source, through the system, to an I/O sink. The units of program development in Scout are called *routers*. A router implements one or more *services* that can be used by other higher-level routers. Routers are organized in a *router graph*, defined at build time. A path is created incrementally by invoking a create operation on a router and specifying a set of *invariants* describing the properties of the desired path.

Bast [3] achieves flexible protocol composition by applying an object-oriented technology, the *Strategy design pattern* [2]. With Bast, a distributed system is composed of *protocol objects*, which are instances of *protocol classes* and have the ability to remotely invoke each other and to participate in various protocols. The Strategy pattern consists of objectifying an algorithm executed by a protocol object, i.e., encapsulating it into a *strategy object*, which is used by the *context object* represented by the protocol object. A strategy and its context are strongly coupled and the application layer only deals with instances of the protocol class.

FarGo [6] provides support for moving the components of a distributed application among multiple hosts during the execution of the application. The programming model proposed, called *dynamic application layout*, separates the programming of the layout of the application from the application logic. FarGo uses Java RMI to implement a reflective inter-component referencing model that allows the attachment of relocation semantics to inter-component references. An event-based monitoring service provides support for making runtime relocations decisions.

The Rover Toolkit [8] implements a distributed object model that provides a uniform view of objects at the OS level and a queued RPC mechanism for disconnected operation and object migration. For instance, simple GUI code can be migrated to a mobile client, where it uses queued RPC to communicate with the rest of the application running on the server.

CORBA [4] provides a distributed object model that supports location transparency, interoperability, and portability. Using an ORB, a client object can transparently invoke a method on a server object, without the need of being aware of where the object is located, its programming language, and its operating system.

To the best of our knowledge, there are no toolkits that provide support for building dynamically reconfigurable modular applications. Our DACIA framework allows the runtime reconfiguration of distributed applications by changing the location of execution of various components or by replacing a set of components with a different set of components. We specifically focus on minimizing communication costs when components are (re)located on the same host.

Several adaptive solutions have been proposed to address the challenges of both wired and mobile computing systems. Most of the existing work targets the improvement of network communication and bandwidth usage. The *adaptive service model* presented in [10] uses admission control and resource reservation and it allows the network and applications to (re)negotiate QoS depending on dynamic network conditions. Other approaches [1, 19] rely on the use of proxies that perform data filtering or *on-demand dynamic distillation* (data type-specific lossy compression). Odyssey [13] sees adaptation as a collaborative partnership between the system and individual applications. The system moni-

tors resource levels and notifies applications of relevant changes. Each application independently decides how best to adapt when notified. Conductor [18] enables the operating system to offer adaptation as a service to applications. It provides a general mechanism to select and dynamically deploy combinations of *adaptive agents* to multiple points in a network.

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

4 The Architecture of DACIA

DACIA (Dynamic Adjustment of Component InterActions) is a framework for building adaptive distributed applications in a modular fashion, through the flexible composition of software modules implementing individual functions. The objective of a modular approach is to manage complexity, isolate failures, and facilitate the construction of customized and flexible configurations. Customization enables the application developer to use only the modules necessary to achieve the desired functionality. Flexibility allows these modules to be combined in multiple ways. An application can be reconfigured dynamically at runtime, in order to obtain a more efficient execution.

4.1 A Model for Modular Design

An application is constructed by connecting in a particular configuration several components implementing various functions or parts of the application. The application can be seen as a directed graph of connected components. The links between components indicate the direction of the data flow within the application. The graph may have cycles and multiple paths may exist in the graph between two components.

The same application can be built in multiple ways, either by configuring differently the same set of components or by using different sets of components. Figure 2.a shows the graph structure for the collaborative system presented in Section 1 (Figure 1). A communication server receives real-time streamed data from multiple data sources, it applies some computations transforming the raw data into GIF images, and then disseminates the images to various clients. Multiple computations can be applied to the same data sets, corresponding to the particular clients' subscriptions. The server caches the data (the *Store* module) for fault tolerance and for future access.

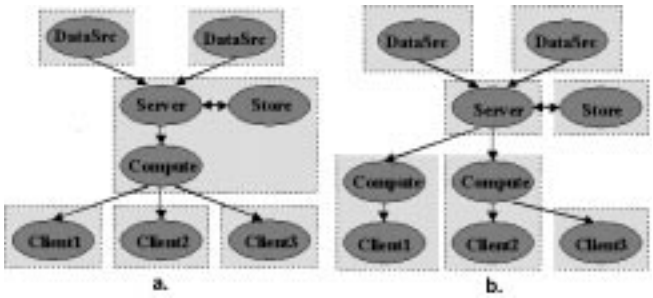

Figure 2: The application graph. Ovals represent components. Grey rectangles represent hosts. Components are connected through directed links, indicating the direction of the data flow within the application. Multiple graphs (a-b) may correspond to the same application.

Figure 2.b presents an alternative configuration for the application, that uses several *Compute* modules, located closer to the clients, in the same local network or even on the same host with the clients. The clients that are unable to perform computationally intensive tasks use *Compute* modules located on neighboring hosts. The communication server outsources the storage function to a different host.

Further changes can 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. Compression can be applied either to the

raw data or to images, by appropriately reordering these components. Depending on the network topology and on runtime conditions, either one of these configurations may be more efficient than the other ones.

4.2 PROCs

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 send them alternately to multiple destinations. Similar to the *paths* in Scout[12], the way PROCs are connected dictates the flow of data in the system.

PROCs represent the basic building blocks for an application. They can be interconnected in multiple ways, according to certain rules and restrictions. PROCs are mobile software components and they represent the unit of relocation.

There are certain differences between PROCs and objects in other component software architectures. PROCs are not just encapsulated objects. They are relocatable data objects. They are executable entities that may hold state, may be interrupted and restarted, and they are involved in communications with other entities. They are routing elements that dictate the structure of an application and the data paths.

PROCs communicate by exchanging *messages* through *ports*. An output port of a PROC is connected to an input port of another PROC. The communication can be either *synchronous* or *asynchronous*. For synchronous communication, the PROCs must be located on the same host and the thread that executes the *output()* method of the source PROC will also execute the *input()* method of the connected PROC and thus the action associated with that 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. (Alternatively, the message queue can be organized as a priority queue.) Asynchronous communication is generally used when multiple input streams have to be synchronized. It is also used when the execution of a PROC's action on some input takes a considerable amount of time and decoupling the actions of the two PROCs is desirable.

Our choice of using ports to communicate between PROCs, as opposed to the object reference model adopted by other component architectures, is motivated by the fact that most of the applications we are considering exchange streamed data. For streamed data, an RPC-like invocation model is not appropriate. In addition to the blocking semantics provided by RPC, we also support asynchronous message passing. Ports provide a clean way of specifying the one-to-one connections corresponding to a graph structure. They allow to distinguish among messages received from different sources, to synchronize and to combine various data streams.

4.3 The Engine

The *engine* is the most important part of DACIA. It 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. Although it belongs to DACIA's infrastructure, the engine is instantiated as part of each application. Every distributed application uses an engine on every host it runs on (Figure 3). We chose to use an engine per application per host (as opposed to sharing an engine running on a host between multiple applications) in order to minimize the cost of communication between PROCs and between PROCs and the engine. The engine and the PROCs run within the same address space, therefore the (synchronous) local communication translates into simple procedure calls.

A minimal application consists of an engine and a few PROCs, connected in some configuration. A distributed application is created by connecting engines running on multiple hosts and eventually connecting PROCs running on different hosts. An engine has global knowledge about all the PROCs in the system and the configuration

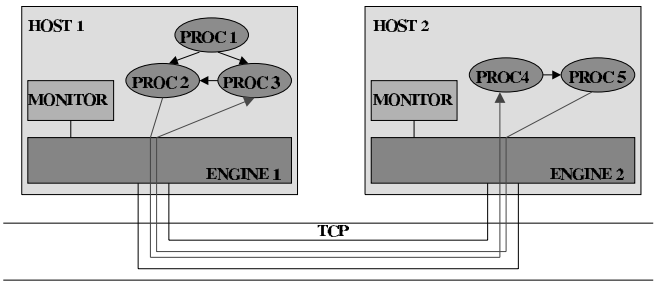

Figure 3: A DACIA distributed application. An engine runs on every host. It manages the local PROCs and the connections between PROCs, both local and across different hosts. The monitor implements application-specific relocation and reconfiguration policies.

of the application. When two engines connect to each other, they exchange information about each other's local views, so that they establish a consistent view of the system.

4.4 The Monitor

The engine works in conjunction with a *monitor*. The monitor represents the part of an application responsible for performance monitoring and making reconfiguration decisions. Performance monitoring is not the sole responsibility of the monitor. The PROCs and the engine may also collect some performance data. The monitor makes reconfiguration decisions and instructs the engine accordingly. The engine provides an API that allows the application reconfiguration. The engine is responsible for establishing and removing connections between PROCs and for moving PROCs to other hosts.

The engine is general-purpose and PROCs may be application-independent, being potentially reused to build multiple applications. The monitor is usually specific to the application and it incorporates relocation and reconfiguration policies applicable only to a particular application. Assuming that all the PROCs needed are provided, an application developer only needs to write a small main program and the customized monitor. An application can use simultaneously multiple monitors, implementing different adaptations. Currently, DACIA provides no coordination among multiple monitors.

4.5 Connectivity

In our current implementation, a PROC is not required to be aware about the existence of other PROCs or about the structure of an application. Thus PROCs are free from the task of handling references to other PROCs and they concentrate on their specific functionality. For a PROC, the fact that the connected PROCs are local or remote is transparent. If needed, PROCs can exchange information about each other and about other PROCs in the system through their input/output ports.

The engine maps virtual connections between PROCs to either local or remote physical connections, and handles data transfers accordingly. Multiple virtual remote connections between pairs of PROCs are multiplexed over a single network connection between two engines. The connectivity between remote PROCs is maintained as long as the corresponding engines are connected. Sharing physical connections reduces the cost of establishing network connections in a highly dynamic application, where PROCs often connect to each other or they are disconnected.

The failure of a connection between engines is transparent to the PROCs. Currently, when a network connection is broken, an engine caches messages addressed to a remote PROC until the connection is re-established, assuming that the disconnection is transient. We intend to address permanent connection failures by notifying the sender PROC after a timeout interval.

4.6 Component Mobility

PROCs can move between hosts while maintaining seamless connectivity with other PROCs. The structure of the application does not change and the flow of data in the system is not interrupted. The movement of a PROC is transparent to other PROCs. If a PROC moves to another host, all the messages left in the asynchronous message queue move with the PROC.

When a PROC moves, we maintain a weak consistency of each engine's view of the PROC's location. The engine where the PROC was previously located informs

the other engines about the change. If an engine receives a message addressed to a PROC that has moved, it forwards the message to the engine currently hosting the PROC and it informs the sender engine about the new PROC location. Presenting the details of ensuring the robustness of data exchange between moving PROCs is beyond the scope of this paper.

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

4.7 Runtime Reconfiguration

In many cases, an application can adapt to dynamic changes in load and the specific execution environment by reconfiguring itself at runtime. 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. A more efficient execution can be achieved through better usage of the available resources and optimized inter-PROC communication.

DACIA allows remote components to be relocated on the same host and thus to be attached into the same address space. Opposite, co-located components can be moved to different hosts and thus detached into different address spaces. Based on application specifics and the characteristics of the hardware (machines and networks) where the application runs, one or the other of the above actions may be beneficial. For two PROCs, situated on different hosts and exchanging messages frequently, the cost of communication can be significantly reduced if the PROCs are co-located. In this case, the communication cost is just the cost of some procedure calls within the same address space. Conversely, two PROCs doing some CPU-intensive processing without much interaction with each other may execute more efficiently if they are relocated on different hosts.

With DACIA, we attempt to minimize the effort for building customized adaptive applications, by putting as much complexity as possible in the infrastructure and as little as possible in the application itself. General-purpose adaptive policies can be embedded in the infrastructure. For example, filters that reduce the amount of data should be placed as early as possible in the data path. Also, a pair of *Compress/Decompress* components can be introduced at the ends of a slow network link. Additionally, application-specific adaptive policies can be implemented in the monitor.

4.8 Implementation Issues

We have fully implemented a DACIA prototype in Java. The engine is implemented as a static class. Our implementation uses TCP to communicate between engines. Alternatively, other transport mechanisms can be used.

One of the goals of our design was to enable inexperienced users to build customized applications with only a small programming and configuration effort. We offer application developers an API that provides some simple primitives for creating and destroying PROCs, connecting engines, 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 the monitor, if applicable) is provided, a simple distributed application can be written using 10-15 lines of code.

Figure 4 presents a simple DACIA application, consisting of an engine and two PROCs. These PROCs, of type *Forward* and *Chat*, respectively, each have one input and one output. The output of *p1* is connected to the input of *p2*. The engine connects with another engine running on a different host. Subsequently, con-

nections can be established between PROCs running on the two hosts. The message exchange starts by calling the *start()* method on *p1*.

In addition to the programming API, we provide a command-line shell interface (Figure 5) for runtime management of the application. Through this interface, the user interacts with the engine running on the local host and he can control the connectivity and the location of PROCs. The user can not access the PROCs directly, but only through the engine. This interface provides almost the same facilities offered by the programming API.

```
public class App {
  public static void main(String[] args) {
    if (args.length < 1) {
      System.out.println("Usage error:
          java dacia.App [config_file_name]");
      System.exit(1);
    }
    // initialize the engine
    Engine.init(args[0]);
    // instantiate two PROCs and connect them
    Proc p1 = new Forward();
    Proc p2 = new Chat();
    Engine.addProc(p1);
    Engine.addProc(p2);
    Engine.connectProcs(p1, 0, p2, 0, true);
    // connect to another engine
    Engine.connect("AnotherHostName", 5000);
    // start the command-line shell API
    Engine.runShell();
    // triggers an action on a PROC
    p1.start();
  }
}
```

Figure 4: A DACIA application. The application's engine connects to an engine running on another host. Subsequently, connections can be established between remote PROCs, using either the programming interface or the user command-line interface (Figure 5).

We designed and implemented the PROC architecture so that it can be easily extended through inheritance, by simply adding component-specific data structures and methods for message handling. We implemented a set of five basic PROCs that can be used to build data distribution services. These PROCs provide primitives for applying transformations to and filtering the input data, distributing data to multiple destinations, synchronizing input data streams, splitting the items in an input data stream and sending them alternately to multiple destinations. Based on the base PROC class, each of these PROCs has been implemented using 22-30 lines of code. They can be easily customized by overloading the message handling methods. Figure 6 presents the code for a simple PROC.

```
class Forward extends Proc {
  public Forward() {
   super("Forward",1,1);  // name, 1 input, 1 output
  }
  public void handleMessage(Message msg, int port) {
    System.out.println("Message received: "
        + (String)msg.getData());
    output(0,msg,1); // port_no, message, synchronous
  }
  public void handleAsyncMessage() {
    Message msg = null;
    while(true) {
      synchronized(msgQueue) {
        while(msgQueue.isEmpty() && !moving) {
          try msgQueue.wait();
          catch(Exception e)  System.out.println(
              "Forward exception:" + e.getMessage());
        }
        if(moving) return;
        msg = getMessage(true);
      }
      System.out.println("Message received: " +
          (String)msg.getData());
      output(0,msg,0); //port_no,message,asynchronous
    }
  }
}
```

Figure 6: A simple PROC with one input and one output. It prints the content of a message received on the input port and then sends the message to the output port.

Component mobility is achieved through Java object serialization. In order to be mobile, a PROC should implement the Java *Serializable* interface. Since serialization/deserialization can be expensive operations, we optimized them by overloading Java's serialization methods. The state of a PROC (only the data that is specific to that PROC) is compacted before it moves and it is restored at the destination. The developer of a customized PROC has the choice of either using the default Java serialization and paying the corresponding performance

```
connect [hostname] [portnumber] - 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
```

Figure 5: Command-line shell interface

penalty, or writing customized serialization code.

The Engine instantiating a PROC is device-aware and it can choose the appropriate constructor for the PROC. Thus, the PROC will have a representation corresponding to the capabilities of the device.

The programming effort to transform a Java object into a mobile PROC is modest. It consists of adding a PROC wrapper to the object, connecting the object to the message handling interface and eventually writing methods for serializing the state of the object. We used 23 lines of code to transform a Java object for a multi-user Chat program (it included a graphical interface, with menus, input/output text areas, and buttons) into a PROC. This PROC has one input and one output port. It displays to an output text area the content of a message received on the input port, and it sends to the output port the message typed by a user in an input text area.

5 DACIA Performance

5.1 Micro-benchmarks

To determine the overhead of using our framework to execute an application, we compared the performance of inter-PROC communication to the cost of raw TCP. Using a small application consisting of two PROCs, we determined the time needed for one PROC to send a message to the other PROC and to receive a reply message, for the cases in which the PROCs are located on the same host (synchronous and asynchronous communication) or on different hosts. We compared the results with the cost of a request-reply using TCP, for the same pair of machines located in a 10 Mbps LAN. We repeated the experiments for a null message and for a message of size 1000 bytes. For local communication, we used a PentiumII 200 machine with 256 MB RAM, running Linux. For remote communication, we also used a ULTRA SPARC 1 machine with 128 MB RAM, running Solaris. The testing code was implemented in Java.

size	local sync	local async	remote PROCs	TCP
0	6.6	44	8400	3600
1000	6.6	44	28000	8200

Table 1: Comparison of inter-PROC communication with raw TCP, for a null message and for a message of size 1000 bytes. From left to right, the columns represent: a) the message size, b) local PROCs, synchronous communication, c) local PROCs, asynchronous communication, d) remote PROCs, e) TCP. The results, given in microseconds, have been obtained by averaging over 10000 messages.

Table 1 shows that the cost of local communication is much lower than the cost of remote communication, either using PROCs or TCP. The message size does not affect the cost of local communication. In this case, object references are passed through procedure calls and data is not actually copied. In the asynchronous case, the cost of switching threads is added. Remote communication between PROCs is slower than raw TCP, and the cost grows significantly with the message size, mostly due to the cost of object serialization. Note that our current implementation is only a prototype and optimizations are possible.

Using the same two machines, we measured the time needed to move a simple PROC between two hosts. The

average value obtained for a PROC that does not carry any specific state, 130 msec, increases if the state size is larger, due to the cost of object serialization.

Overall, the results presented show that the benefit of co-locating remote PROCs that exchange messages frequently far outweighs the overhead of using our framework instead of simple TCP to communicate across multiple hosts. Moreover, the cost of moving PROCs across multiple hosts can be kept low.

5.2 Macro-benchmarks

The following experiment shows basic component interaction and mobility. It also performs adaptation and component migration according to a simple heuristic. We implemented a simple application that simulates the consumption of certain resources by several client applications, which may be located on the same host with the resources or on different hosts. The goal of the experiment is to show how the resource consumption can be optimized by moving components around according to the structure of the application, user requirements, and the pattern of communication between various components.

5.2.1 Experimental Setup

Figure 7 presents the structure of the application used for this experiment. We use three types of PROCs:

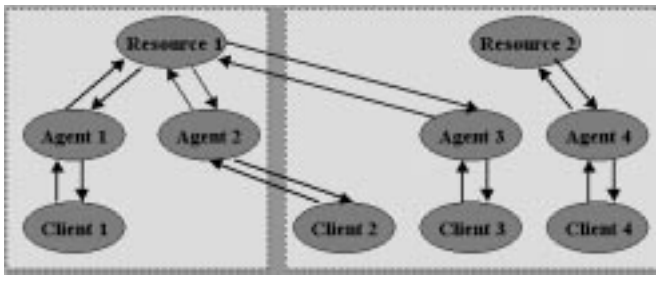

Figure 7: The application structure. Through their corresponding agents, clients submit requests for accessing various resources . The location of a client is fixed, while the resources and the agents can move from one host to another.

- **Client** - a PROC with a GUI (a frame with menus, buttons, input and output text fields). The user can submit requests in an input window. An output window displays status information. A Client PROC can not move to a different host, it has to stay on the machine where it started, in order to communicate with the user.
- **Agent** - acts on behalf of a client. Every client has a corresponding agent. An agent discovers and connects to various resources and it cooperates with the resources to complete the client's requests.
- **Resource** - represents the server side of the application. A particular resource may be available only at certain sites, and it can migrate from one site to another.

An agent can be located on the same host as its client (Agents 1, 3, and 4) or on a different host (Agent 2). Similarly, an agent can be located on the same host as the resource accessed (Agents 1, 2, and 4) or on a different host (Agent 3). An agent can move from one host to another according to the amount of data exchanged among the PROCs and the availability of bandwidth and processing power.

In our experiment, a client submits a sequence of requests to its agent. In the status window, it displays the time needed for the completion of each request, as well as the average time over all the requests. A request randomly targets either a resource local to the host where the client is, or a resource located on a remote host. A resource can move while a request is being served. Resource movements are triggered at random intervals, with values between 1 and 10 sec.

To complete a request, the agent sends a message to the corresponding resource. In this experiment, using the resource translates into using CPU cycles on the machine hosting the resource. The resource increments an integer counter 10000 times and then sends a reply message to the agent. This message exchange between the agent and the resource repeats 500 times. At the end,

the agent informs the client about the completion of the request.

We compared the case where the agent is fixed and located on the same host with the client with the case where the system adapts to the pattern of data exchange between the PROCs and it moves to the host where the resource is. For this experiment we used two machines located in the same 10 Mbps Ethernet LAN: saturn (PentiumII 200, 256 MB RAM, running Linux) and sanjuan (ULTRA SPARC 1, 128 MB RAM, running Solaris).

This experimental setup can be easily mapped to real applications. Consider a meta-service for web searching such as MetaCrawler [15]. A client is a web browser through which a user can request various searches. An agent is the search agent that queries various web servers to discover the information requested, and filters the results of the queries received from the web servers. Resources represent the web servers or the back-end databases behind these web servers. To complete a request, a search agent may connect to one or multiple web servers and exchange several messages with the web servers. While talking to a particular web server, the agent may be connected to various back-end nodes holding replicas of the web data, as in [17]. The agent can either run on the same host with the web browser or it can move to the host where the resource accessed is, in order to minimize the cost of communication with the resource.

5.2.2 Experimental results

Figure 8 presents the average time needed to complete a request, as well as the standard deviation for several measurements, for randomly generated requests. The first two bars correspond to the cases where all three PROCs (client, agent and resource) are located on the same host. The third bar is for the case where the client and the agent are located on the same host (saturn), and the resource accessed is always located on sanjuan. The fourth bar corresponds to the case where the client and the agent are co-located on saturn, while the resource accessed moves randomly between the two hosts. The fifth case is similar, with the difference that the agent may move once it detects that it is accessing a remote resource and that the communication with the resource is much more intense than the communication with the client.

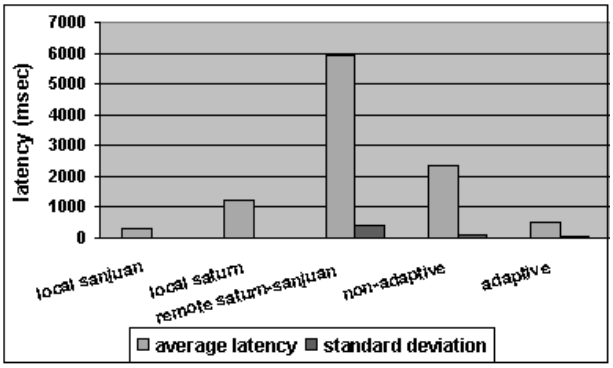

Figure 8: The time needed to complete a request for the cases where all the PROCs are fixed and they are located on the same host, or the PROCs are located on different hosts, or the resource accessed moves from host to host and the system either does or does not adapt to these location variations. The standard deviation is between .2% (second bar) and 11% (fifth bar) of the average.

For each case, a run of the experiment corresponds to a sequence of 100 to 1000 requests (depending on the speed of execution for 1 request), for which we calculated the average latency. We repeated a run five times and we calculated the average latency and the standard deviation for each one of the five cases outlined above.

The first two bars highlight the difference between the time needed to execute identical operations on the two hosts. The third bar shows that the latency increases dramatically if most of the communication between PROCs is done remotely. The last two bars prove that in the adaptive case the latency is most of the time lower than in the static case, and the simple heuristic employed is useful in reducing the latency, by taking advantage of the proximity of some of the PROCs. On average, the latency in the adaptive case is about 4 times lower than in the static one.

The non-adaptive case performs better than the remote case (third bar), but worse than both local cases (first two bars). The adaptive case performs better than the non-adaptive one. It also performs better than the

local case on saturn. The explanation is that in this case most of the communication between the agent and the resource is handled locally, thus bringing the average to a value between the values for local communication on the two machines, saturn and sanjuan.

This experiment relies heavily on the cost of network communication between various machines. Nevertheless, it shows that by using some simple heuristics, according to the specific structure of an application and the pattern of communication between PROCs, the execution time can be reduced significantly compared to the static case in which all the PROCs have fixed locations. Also, the frequent PROC movements across machine boundaries, under intense message exchange, prove the robustness and correctness of our implementation.

6 Conclusions

In this paper, we present a framework for building and executing adaptive distributed applications. Using DACIA, an application can be easily constructed through the flexible composition of existing software components. PROCs (Processing and ROuting Components) are the basic building blocks for an application. PROCs interact through standardized interfaces and they can be composed in a variety of ways, subject to certain rules and restrictions.

The novelty of our approach is that we allow the PROCs in an application to be composed in multiple ways, and we provide support for changing at runtime the structure of the application. An application can be reconfigured by replicating components, relocating some components, or by replacing a set of components with a different set of components, possibly connected in a different configuration. One of several equivalent composition schemes is chosen at runtime based on specific environment conditions, resource availability, and application requirements. Through runtime reconfiguration, a more efficient execution is achieved.

DACIA provides support for application and user mobility. PROCs can move between hosts while maintaining seamless connectivity with other PROCs. The structure of the application does not change and the flow of data in the system is not interrupted.

Micro-benchmarks show that when components are co-located, we are able to reduce the cost of communication between PROCs to the value of a few procedure calls. Depending on the pattern of communication between various PROCs in a distributed application, the overall cost of communication can be reduced through PROC relocation.

We demonstrated the benefits of using DACIA on a simple application. We showed that by using a simple adaptive heuristics to reconfigure the application, the execution time can be significantly reduced compared to the static case in which all the PROCs have fixed locations.

Following we outline some of the issues we are currently working on or we intend to address in the future:

• *Optimizing application performance through reconfiguration:* DACIA provides support for reconfiguring an application at runtime, according to some performance metrics and the availability of resources system-wide. The challenge is not to measure the performance of an application (several solutions exist for this), but to use this information to reconfigure the application. To do this, we need to specify semantically meaningful policies that are context-aware and can readily adapt to their environment.

• *Formal specification:* We are currently working on a formal framework for specifying components, their properties, and the complex interactions between components. This framework allows the specification of rules for composing components and for defining equivalent composition schemes. We intend to build tools that use a set of composition rules manually created to automatically derive new composition rules. This formal framework also provides support for defining and implementing adaptive policies used to reconfigure an application.

• *Deployment and experimental evaluation:* More experience is needed to evaluate the effectiveness of our model in building and executing adaptive distributed ap-

plications. We also need to assess the usefulness of component mobility to users by implementing some practical applications and testing them on a larger user population. We intend to use DACIA to implement and test larger-scale applications. We would like to determine the complexity and the overhead of performing adaptation and PROC relocation.

• *Interoperability of PROCs and services:* The programming model proposed can be extended by allowing PROCs to connect to services, as in Sun's Jini [11], as opposed to connecting to other PROCs. In this case, the engine will be responsible for connecting a PROC not to a specific PROC, but to one of potentially many PROCs implementing a specific function. A lookup service is needed to locate the PROCs implementing the desired service.

7 Acknowledgments

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

References

[1] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, Cambridge, MA, Oct. 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[3] B. Garbinato and R. Guerraoui. Using the Strategy Design Pattern to Compose Reliable Distributed Protocols. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 165–171, Phoenix, AZ, June 1997.

[4] Object Management Group. CORBA Services: Common Object Service Specification. Technical report, Object Management Group, July 1998.

[5] M. Hayden. The Ensemble System. Technical Report TR98-1662, Cornell University, Jan. 1998.

[6] 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.

[7] N. C. Hutchinson and L. L. Peterson. X-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[8] A Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, , and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[9] 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.

[10] S. Lu, K.-W. Lee, and V. Bharghavan. Adaptive Service in Mobile Computing Environments. In *Proceedings of IFIP IWQoS '97 (International Workshop on Quality of Service)*, New York, NY, May 1997.

[11] Sun Microsystems. Jini Connection Technology, http://www.sun.com/jini/.

[12] D. Mosberger and L.L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of OSDI '96*, pages 153–168, Oct. 1996.

[13] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, Oct. 1997.

[14] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. Mach: A Foundation for Open Systems. In *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*, Sep. 1989.

[15] E. Selberg and O. Etzioni. Multi-Service Search and Comparison Using the MetaCrawler. In *Proceedings of the 4th World Wide Web Conference*, pages 195–208, 1995.

[16] R. van Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible Group Communication System. *Communications of the ACM*, Apr. 1996.

[17] V.Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.

[18] M. Yarvis, P. Reiher, and G. J. Popek. Conductor: A Framework for Distributed Adaptation. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS '99)*, March 1999.

[19] B. Zenel and D. Duchamp. A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment. In *Proceedings of MobiCom '97*, Budapest, Hungary, Oct. 1997.

[20] H. Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, Apr. 1980.