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

Future distributed applications will need to support computing devices with a wide range of capabilities, varying network connectivity, increasing mobility of users, and a wide variation in load placed by clients on services. This paper presents DACIA[1]*, a framework for building adaptive distributed applications. In DACIA, distributed applications are viewed as consisting of connected components that typically implement data streaming, processing, and filtering functions. DACIA provides mechanisms for run-time reconfiguration of applications to allow them to adapt to the changing operating environments. Components can be moved to different hosts during execution, while maintaining communication connectivity with other components. New components can also be introduced along data paths, for example, to provide compression on low-bandwidth connections. Keeping communication overheads low is a significant challenge in designing component-based services. DACIA is designed so that communication costs among co-located components are similar to those of procedure calls. Performance results, as well as examples of adaptive services that can be built using DACIA are presented.*

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 personal digital assistants (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.

To manage this highly variable environment, *the distributed applications and the services provided have to adapt to 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 [15], 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 (Figure 1). The server 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.

We have encountered several problems 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

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

[1]Dynamic Adjustment of Component InterActions

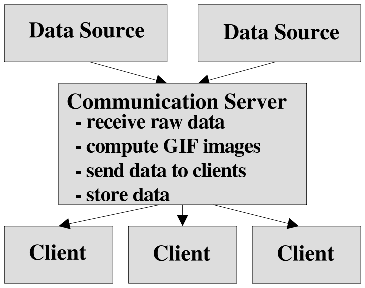

Figure 1: A collaborative application

bigger size than the size of the raw data. Therefore, the network links from the server to some clients can get congested.

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. In fact, we used this approach in the first version of UARC, but found that many clients get overloaded and are not able to keep up with the data arrived, if they are computing many images. Moreover, in some cases, several clients end up executing the same computation. A solution that addresses these shortcomings is to use a flexible architecture, in which the computing function can execute either on the server, or on the client, or on hosts situated close to clients.

In this paper we present DACIA, a framework that addresses the challenges of building *adaptive distributed applications that can change their structure at runtime*. An application with a modular architecture, in which various components implement individual functions, is able to 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.

Users are inherently mobile, regardless whether they work on their desktop computer or they use some mobile computing device to connect to distributed services. 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, restart the very same applications, and establish manually the same connections.

In a mobile component system, *persistent connectivity* between components is desirable. Our work 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.

We show that the overhead introduced by using modularity is minimal and that often we can actually reduce the communication costs in distributed applications. The cost of communication between co-located components is comparable with the cost of a local procedure call. We give an example of using code mobility and reconfiguration to improve the performance of a simple test application.

The remainder of the paper is organized as follows: Section 2 identifies the objectives we pursued in designing DACIA. Section 3 presents our approach to designing and implementing a framework for flexible component composition. Section 4 shows some performance results obtained with small test applications implemented using DACIA. Section 5 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:

- A modular architecture, that allows applications to be built through the composition of software components implementing individual operations;
- Support for application and user mobility;
- Persistent connectivity between mobile components;
- Low overhead for both local and remote inter-component communication;
- Runtime reconfiguration of an application through the dynamic adjustment of the connections between components and the location of execution of different components.

3 The Design 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.

3.1 A Component-Based Modular Architecture

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

Figure 2.a shows the graph structure for the collaborative system presented in Section 1. A communication server receives real-time streamed data from multiple data sources, it applies some computations (e.g., transforming raw data into GIF images, merging two data streams, filtering, etc), and then disseminates the resulted data to various clients. Multiple computations (the *Compute* module) 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.

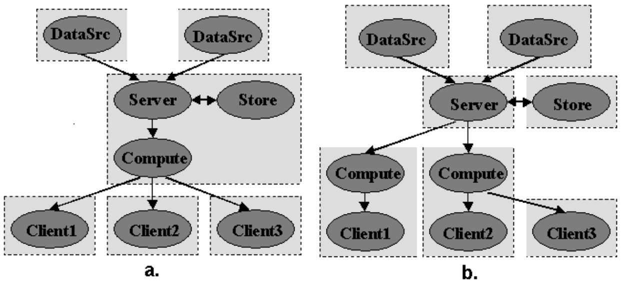

Figure 2: An application graph. Ovals represent components. Grey rectangles represent hosts. Components are connected through directed links, indicating the direction of the data flow within the application. Multiple graphs (a-b) may correspond to the same application.

Figure 2.b presents an alternative configuration that uses several *Compute* modules, located closer to the clients, in the same local network or even on the same hosts with the clients. 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. Depending on the network topology and on runtime conditions, either one of these configurations may be more efficient than the other ones.

There exist several distributed component architectures, that share some of the goals and design principles with our work, and also exhibit a range of differences. CORBA [10] provides a distributed object model that supports location transparency, interoperability, and portability, but does not support component mobility or communication connectivity for mobile components. These features are supported by DACIA. Our work focuses on providing support for adapting at runtime the structure of distributed systems and location of system components and services to the context in which they are being used, scale of use, location of users, and to available resources (e.g., CPU, network bandwidth, display).

Component-based distributed platforms have been also used in other contexts, such as resource management in large-scale distributed environments. The Globus project [6] focuses on the infrastructure needed to build *computational grids*, execution environments that enable an application to integrate geographically-distributed instruments, displays, and computational and information resources. The Darwin project [4] develops a set of customizable resource management mechanisms, based on hierarchical scheduling, a signaling protocol, and service brokers.

Our work has benefited greatly from a substantial body of knowledge accumulated in the context of other research projects targeting mobile environments, such as Bayou [3], Daedalus/BARWAN [5, 11], and GloMop [7]. These projects achieve adaptation in mobile environments primarily through application-specific transformations of the data stream according to the available bandwidth. DACIA adapts to resource constraints and changing requirements through application reconfiguration and component mobility. The Rover Toolkit [14] 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. In Rover, object migration is restricted to be between a user's desktop (server) machine and the user's mobile device (client). We allow components to move across all the machines in the system.

3.2 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. PROCs represent the basic building blocks for an application. They can be interconnected in multiple ways, according to certain rules and restrictions. PROCs are identified system-wide using a unique identifier obtained by combining the host ID where the PROC originated and a counter maintained by the host.

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

Unlike the object reference model used by other component architectures, like CORBA, to communicate between components, PROCs communicate by exchanging *messages* through *input and output ports.* Our choice is motivated by the fact that many of the applications we are considering exchange streamed data, for which an RPC-like invocation model is not appropriate. We support both the blocking semantics provided by RPC and *asynchronous message passing.* For *synchronous communication*, the PROCs must be located on the same host and, 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 following primitives are used to connect and disconnect PROCs and to communicate between them:

• *connectProcs(procID1, portNo1, procID2, portNo2)* - connect two PROCs. The PROCs can be local or remote.

• *disconnectProcs(procID, int portNo)* - disconnect two PROCs.

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

• *input(portNo, message, isSynchronous)* - receive a message on the specified input port.

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.

3.3 Engines and Applications

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. A DACIA distributed application (Figure 3) uses an engine on every host it runs on. We chose to use an engine per application per host (as opposed to sharing an engine running on a host between multiple applications) in order to minimize the cost of communication between PROCs and between PROCs and the engine. The engine and the PROCs run within the same address space, therefore the (synchronous) local communication translates into simple procedure calls.

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

• *connect(hostname, port, update)* - connects to an engine running on host *hostname*, listening on port *port.* If *update* is true, upon connecting, the engines exchange information about each other's local views of the PROCs in the system.

Similar to the *paths* in Scout [18], the way components are connected dictates the flow of data in the system. Although the graph structure of our architecture is similar to the *router graph* in Scout (*routers* are the units of program development), there are significant differences in terms of goals and underlying mechanisms. Scout is a communication-oriented operating system. A router graph resides within a single address space. Our system is used to implement distributed applications that span over multiple hosts, and where components are often loosely coupled. A router graph is defined at application build time, while in our case an application graph can change at runtime.

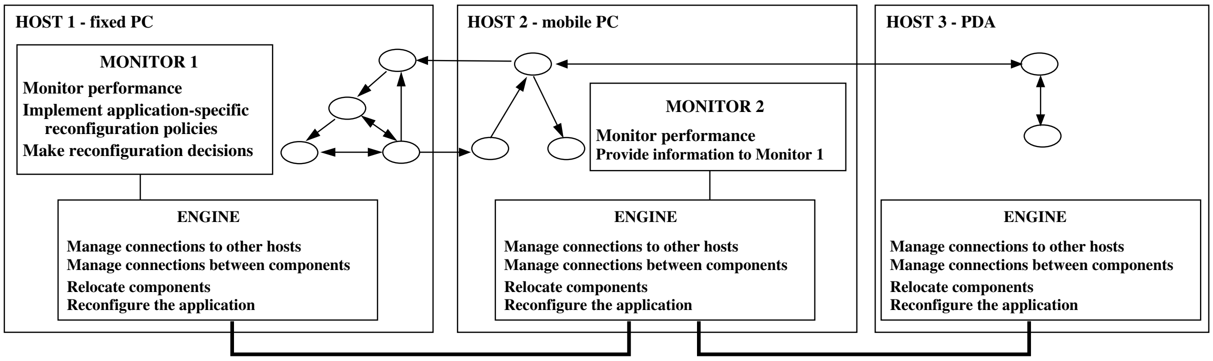

Figure 3: A DACIA distributed application is a directed graph of connected components (ovals represent components). An engine runs on every host. It manages the local components and the connections between components, both local and across different hosts. The monitor gathers performance data and implements application-specific relocation and reconfiguration policies.

3.4 The Monitor

The engine works in conjunction with a *monitor.* The monitor represents the part of an application that *monitors the application performance and makes reconfiguration decisions.* The monitor is not solely responsible for performance monitoring. The components and the engine may also collect some performance data. The monitor 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 engine and the PROCs are general-purpose and they can be reused to build multiple applications. The monitor is usually specific to the application and it incorporates relocation and reconfiguration policies applicable only to a particular application. A distributed application can use simultaneously multiple monitors, implementing different adaptations, sometimes based only on information locally available.

We separate administrative tasks such as maintaining the connections between components and migrating components (done by the engine) from the process of interpreting the semantics of an application and making reconfiguration decisions (done by the monitor). The approach used by FarGo [12] offers some similarities. The programming model proposed, called *dynamic application layout*, separates the programming of the layout of the application from the application logic. FarGo uses Java RMI to implement a reflective inter-component referencing model that allows the attachment of relocation semantics to inter-component references. An event-based monitoring service provides support for making runtime relocations decisions. The changes of an application layout in FarGo consist of finding the right place to execute components and migrating components at runtime. We go further, allowing an application to dynamically change the connections between components, to introduce new components, and to change its structure.

3.5 Component Mobility

One of the key features of our architecture is the ability to move components between hosts. *Code mobility* has received a great deal of attention in the research community. Several recent languages support various flavors of code mobility, such as Telescript [20], Obliq [2], and Sumatra [1]. Related to the work on code mobility, several systems use the term *mobile agent*, sometimes with different interpretations. For example, while in Telescript an agent is represented by a thread that can migrate among different nodes carrying its execution state, in TACOMA [13] agents are just code fragments associated with initialization data that can be shipped to a remote host. PROCs differ from mobile agents in the sense that they are not autonomous. They move as a result of commands issued by engines, according to adaptive policies implemented by monitors.

Existing mobile code systems offer two forms of code mobility [8]. *Strong mobility* allows migration of both the code and the execution state to a different computing environment. *Weak mobility* allows only the transfer of code across different computing environments.

We adopted the middle way between these approaches. We provide mobility at the component (object) level. We do not transfer the execution state of a PROC (e.g., program counter, stack content, thread status). However, a moving PROC carries with it the state of its data members, the messages received and not handled yet, and the state of its connections. By default, the state capture in DACIA is implicit (through Java serialization). To reduce the complexity and the overhead of component movement, the programmer has the ability to explicitly capture the state of a moving PROC, by writing customized serialization routines.

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

For PROCs to be reachable after they relocate, mechanisms for locating them are needed. Ideally, a location service should be present and it should be able to provide at any moment correct information about a PROCs location. In our current implementation, 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.

When a component moves, besides adjusting its connections, an additional problem is the access to the resources previously used by the component (e.g., a local file or a database). There are two types of components in the framework: *resource components* and *mobile components*. Resource components are usually fixed because they directly read/write a local resource. Mobile components access resources only via resource components. Resource components may support *replication* and *caching*, so that they can be accessed in disconnected mode.

3.6 Persistent Connectivity

A PROC is not required to be aware about the existence of other PROCs or about the structure of an application. For a PROC, the fact that the connected PROCs are local or remote is transparent. If context-awareness information is needed, PROCs can exchange information about each other and about other PROCs in the system through their 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.

In some cases, the temporary failure of a connection between engines can be made transparent to the PROCs. In this situation, 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. If the connection failure is permanent, the engine should either find alternative paths to deliver a message to a PROC, or notify the sender PROC after a timeout interval. Our implementation uses the latter alternative.

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[2]. 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.

3.7 Low Communication Overhead

To get the advantages of modularity and component mobility without a significant hit on performance, our framework attempts to keep communication overheads low when PROCs are co-located. When a remote PROC is relocated to the same host as a PROC it is connected to, we allow the two PROCs to be attached into the same address space (subject to security policies). Co-located PROCs can thus exchange data with low overheads, only slightly higher than local procedure calls within the same address space. In

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

the case of asynchronous communication, the cost of thread scheduling and queue management is added. For PROCs exchanging messages frequently, the cost of communication can potentially be reduced if the PROCs are co-located. Conversely, two PROCs doing some CPU-intensive processing without much interaction with each other may execute more efficiently if they are relocated on different hosts.

Section 4.1 presents some performance measurements for inter-PROC communication.

3.8 Adaptability through Runtime Reconfiguration

Several adaptive solutions have been proposed to address the variability encountered both in wired and mobile computing systems. Most of the existing work targets the improvement of network communication and bandwidth usage. Some of the solutions proposed are based on admission control and resource reservation [16], or on the use of proxies that perform data filtering or *on-demand dynamic distillation* (data type-specific lossy compression) [7, 22]. Odyssey [19] sees adaptation as a collaborative partnership between the system and individual applications. Conductor [21] 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 the *runtime reconfiguration* of the application. The reconfiguration consists of either reordering or relocating some components or replacing a set of components with a different set of components, possibly connected in a different configuration. As a result of reconfiguration, the application graph changes. A more efficient execution can be achieved through better usage of the available resources and optimized inter-component communication.

One of our goals is to provide mechanisms for dynamically reconfiguring an application (Figure 2 presents an example) and the support for actually making reconfiguration decisions. One way to do this is through the implementation of specialized monitors. A monitor uses information regarding the performance of the application and the resources available globally to automatically generate and choose among functionally equivalent configurations. We also provide a command-line interface (Figure 7) through which a system administrator can manually reconfigure an application by relocating PROCs, creating new PROCs, and changing the way existing PROCs are connected. This interface also allows the addition/replacement of a monitor while the application is running.

The primitives used either by a monitor or by a system administrator to reconfigure an application are the calls to *connectProcs()*, *disconnectProcs()*, as well as the primitives for moving a PROC to another host:

- *moveProc(procID, hostname)* - move a PROC to the specified host.
- *moveProc(procID, connection)* - move a PROC over the connection to another Engine.

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

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

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

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

3.8.1 Dynamic Reconfiguration Example

Using DACIA, we have implemented an adaptive multi-party communication application. The application consists of client (C) and server (S) PROCs. Through the servers, a client sends messages to the

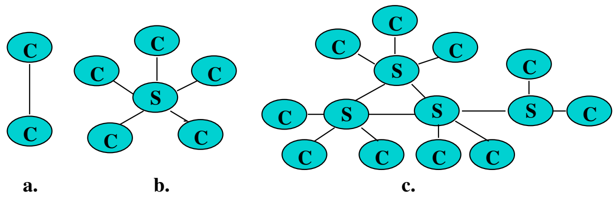

Figure 4: Adaptive multi-party communication. Servers are denoted by S, and clients are denoted by C. New servers are created as the number of participants grows.

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 4.a), without using a server. When a third client tries to join a communication group, a server module is spawned, and all the clients will connect to the server and will exchange data through it (Figure 4.b). Assuming that the clients are *C1*, *C2*, and *C3*, the sequence of operations[3] involved is:

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

When the number of clients on the server reaches an upper threshold N_{max}, a new server is spawned and it connects to the other server. The clients are distributed over the two servers. Ideally, the distribution should take into account clients' relative locations. When there is a large number of clients in the group, the application will contain several servers, connected to each other in a certain configuration, with the clients being assigned equally[4] to all the servers (Figure 4.c).

As clients leave the group, the load per server goes down, and thus it does not justify the usage of too many servers. When the load on a server goes under a lower threshold N_{min}, a server module is deallocated and its clients are distributed to other servers. Alternatively, two servers with load under N_{min} can be replaced by a single server supporting all their clients. Various algorithms can be implemented to allocate and deallocate server modules and to handle clients distribution.

The architecture and the adaptive algorithms presented can be used to implement various server-based group communication applications. The application reconfigures itself dynamically to scale up to a large number of clients and to reduce message latencies.

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

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

3.9 Implementation Issues

We have fully implemented a DACIA prototype in Java [9]. We chose Java because it is currently widely used, it runs on multiple platforms, and it is tightly integrated with World Wide Web-related technologies. Moreover, the Java class loader provides mechanisms for retrieving and dynamically linking classes in a running virtual machine. Our implementation uses TCP to communicate between engines. Alternatively, other transport mechanisms can be used.

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 5 presents the code for a simple PROC.

One of our goals 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 6 presents a simple DACIA application, consisting of an engine and two PROCs per host. These PROCs, of type *Chat* and *Forward*, respectively, have one input and one output each. The output of *p1* is connected to the input of *p2*. The engine connects to another engine running on a different host and having two similar PROCs. Subsequently, connections are established between PROCs running on the two hosts. A message originating at one of the *Chat* PROCs will be delivered to the other *Chat* through the interposed *Forward* PROC. The message

```
class Forward extends Proc {
  public Forward() {
   super("Forward", 2);  // name, 2 ports
  }
  public void handleMessage(Message msg, int port) {
    System.out.println("Message received: "
        + (String)msg.getData());
    output(0,msg,true); //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("Async message received: " +
          (String)msg.getData());
      output(0,msg,false); //port_no, message, async
    }
  }
}
```

Figure 5: 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.

```
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 Chat();
    Proc p2 = new Forward();
    Engine.addProc(p1);
    Engine.addProc(p2);
    Engine.connectProcs(p1, 0, p2, 0);
    // connect to another engine
    Engine.connect("AnotherHostName", 5000, true);
    // start the command-line shell API
    Engine.runShell();
    p1.start();  // triggers an action on a PROC
  }
}
```

Figure 6: 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 7).

exchange is triggered by calling the *start()* method on *p1*.

In addition to the programming API, we provide a command-line shell interface (Figure 7) for runtime management of the application. Through this interface, the user or system administrator 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.

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 (including queued up messages) is compacted before it moves and it is restored at the destination.

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 the messages received on the input port, and it sends to the output port the messages typed by a user in an input text area.

4 DACIA Performance

4.1 Micro-benchmarks

To determine the overhead of using our framework to execute an application, we compared the performance of inter-PROC communication with the cost of raw TCP in Java. Using a small application consisting of two PROCs, we determined the time needed for one PROC to send a message to the other PROC and receive a reply message, for the cases in which the PROCs are located on the same host (synchronous and asynchronous communication - columns 2 and 3 in Table 1, respectively) or on different hosts (column 6). We compared the results with the cost of two local procedure calls that copy the data (column 4), local communication using TCP (column 5), and remote request-reply using TCP (column 7). The remote communication results have been obtained for a pair of machines (PentiumII 200, 256 MB RAM, running Linux, and ULTRA SPARC 1, 128 MB RAM,

```
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 7: Command-line shell interface

message size (bytes)	local PROCs synchronous	local PROCs asynchronous	local procedure call	local TCP	remote PROCs	remote TCP
0	6.6	44	6.4	370	2040	990
1000	6.6	44	6.4	400	3900	2600

Table 1: Latencies (in microseconds) for inter-PROC communication and raw TCP, for a null message and for a message of size 1000 bytes. The results have been obtained by averaging over 10000 messages.

running Solaris) connected through a 10 Mbps LAN.

The cost of local communication is much lower than the cost of remote communication, either using PROCs or TCP. In the case of local inter-PROC communication, object references are passed through procedure calls and data is not actually copied. Therefore, the communication latency is much smaller (comparable with the cost of a procedure call) than in the case where TCP is used to exchange data between two co-located components. In the asynchronous case, the cost of switching threads is added. Remote communication between PROCs is slower than raw TCP, but within the same order of magnitude. The cost can grow significantly with the message size if Java object serialization is used to flatten the data objects in a message. Our current implementation is only a prototype and we intend to improve the performance of remote communication by avoiding, whenever possible, serialized read/write operations.

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, 121 msec, increases if the state size is larger, due to the cost of object serialization.

We also determined the impact of using DACIA on the communication throughput. Using the experimental setup described above, we measured the throughput of the message exchange between two PROCs located on different hosts. For various message sizes ranging between 1-5 kB, we obtained a throughput of 4.78-5.33 Mbps. These values are generally within 10-15% of the throughput obtained for a simple TCP connection between the same two hosts, which is 5.35-6.61 Mbps.

An adaptive application can take advantage of the differences in communication performance, by relocating and co-locating PROCs based on the frequency and the amount of data exchanged between them. Overall, the results presented show that the benefit of co-locating remote PROCs that exchange messages frequently 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.

4.2 Macro-benchmarks

To show basic component interaction and the benefits of application reconfiguration and component relocation, we used DACIA to implement the test case presented in Section 3.1 (Figure 2). A communication server receives data from various data sources, it applies some computations, and then disseminates the resulted data to various clients. Since different clients subscribe to different data, the *Compute* PROC executes a separate computation for each request.

We determined the average time to complete a client request, from the moment the client submits the request, until it receives the data. We repeated the experiment for the cases of a single client, or two clients that send requests concurrently. The configuration parameters for this experiment are:

• network bandwidth between server and client. We simulated a high capacity (approximately 800

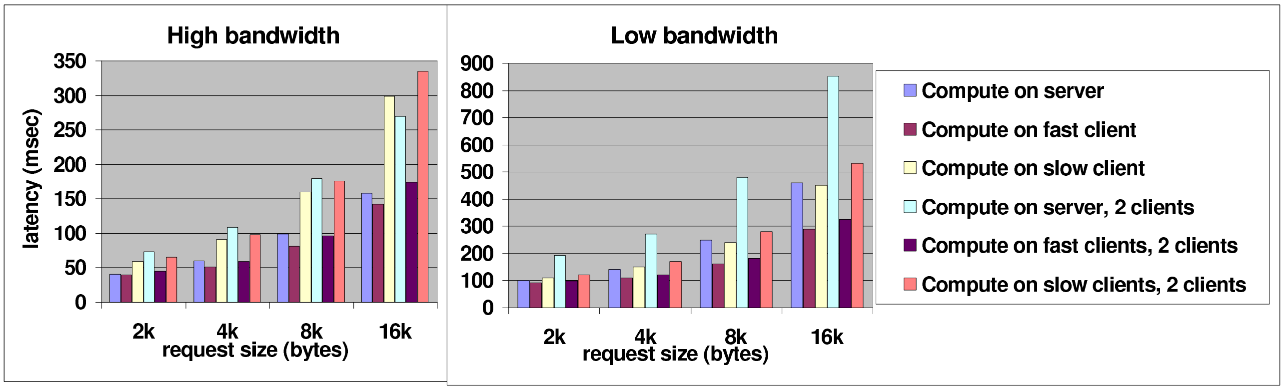

Figure 8: Average time to serve a request, for one or two clients, for the cases where the bandwidth is high or low, the Compute module is co-located either with the Server or the Client. Clients can run either on fast or slow machines.

kB/sec) network, and a low capacity (approximately 60 kB/sec) network.

• location of the *Compute* PROC - co-located either with the *Client* or with the *Server*

• size of the data requested, measured at the source - from 2kB to 16kB

• speed of the client machine - either comparable with the server machine or three times slower

• cost of applying a computation to a data item - 10ms for every 2kB of input data on a fast machine, 30ms/2kB on a slow machine

• amount of data produced by the *Compute* module - twice as high as the size of the input data

The results of the experiment[5], presented in Figure 8, show that:

• When a single client is connected to the server, for high link bandwidth, executing the computation either on the server or on a fast client gives similar results, better than in the case of running the computation on a slow client. For low bandwidth, due to the increased size of the computed data, executing the computation on the server gives latencies similar to the ones in the case of computing on a slow client.

• When two clients send different concurrent requests, due to the high CPU consumption for the *Compute* module and the large amounts of data exchanged, it is more beneficial for the clients' hosts to execute the computations. This can not be done if a client runs on a very slow machine. In this case, the computing function can be executed on a nearby client machine with higher capabilities.

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

[5]each data point has been obtained by averaging over 50 requests

5 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 changing the connections between components, relocating or replicating components, or by replacing a set of components with a different set of components, possibly connected in a different configuration.

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 a value close to the cost of a procedure call. 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 some simple applications. We showed that by using simple adaptive heuristics to reconfigure an 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 plan to address in the future:

- Policies and algorithms for optimizing application performance through dynamic reconfiguration;
- Formal framework for specifying components, their properties, rules for composing components and for defining equivalent composition schemes;
- Extensive deployment and experimental evaluation;
- Security infrastructure for our mobile component environment;
- Interoperability of PROCs and services, as in Sun's Jini [17].

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] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[3] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing Among Mobile Users. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec 1994.

[4] P. Chandra et. al. Darwin: Resource Management for Value-Added Customizable Network Service. In *Sixth IEEE International Conference on Network Protocols (ICNP'98)*, Austin, TX, Oct 1998.

[5] R.H. Katz et. al. The Bay Area Research Wireless Access Network(BARWAN). In *Proceedings Spring COMPCON Conference*, Feb 1996.

[6] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[7] 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.

[8] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), May 1998.

[9] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. *Addison Wesley, Reading*, 1996.

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

[11] T.D. Hodes, R.H. Katz, E. Servan-Schreiber, and L. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. In *Proceedings of the 3rd ACM/IEEE MobiCom*, Budapest, Hungary, Sep 1997.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] J.E. White. Telescript Technology: Mobile Agents. *Software Agents, J. Bradshaw, ed. AAAI Press/MIT Press*, 1996.

[21] 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.

[22] 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.