Dynamic Reconfiguration with I/O Abstraction

Bala Swaminathan
Kenneth J. Goldman

WUCS-93-21

August 20 1993
Revised March 17 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130–4899

Dynamic Reconfiguration with I/O Abstraction

Bala Swaminathan
bs@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130

Kenneth J. Goldman*
kjg@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130

March 17, 1995

Abstract

Dynamic reconfiguration is explored in the context of I/O abstraction, a new programming model that defines the communication structure of a system in terms of connections among well-defined data interfaces for the modules in the system. The properties of I/O abstraction, particularly the clear separation of computation from communication and the availability of a module's state information, help simplify the reconfiguration strategy. Both logical and physical reconfiguration are discussed, with an emphasis on a new module migration mechanism that (1) takes advantage of the underlying I/O abstraction model, (2) avoids the expense and complication of state extraction techniques, (3) minimizes the amount of code required for migration and confines that code to a separate section of the program, and (4) is designed to permit migration across heterogeneous hosts and to allow replacement of one implementation by another, even if the new implementation is written in another programming language. The flexibility of the migration mechanism is illustrated by presenting three different paradigms for constructing reconfigurable modules that are supported by this new mechanism. A uniform specification mechanism is provided for both logical and physical reconfiguration.

Keywords: heterogeneous systems, dynamic reconfiguration, distributed systems, process migration

1 Introduction

The *configuration* of a distributed system consists of three parts, a *logical configuration* that defines the set of modules and their logical communication channels, a *physical configuration* that assigns

*Contact author: Kenneth J. Goldman, Department of Computer Science, Campus Box 1045, Washington University, One Brookings Drive, St. Louis, MO 63130-4899, (314) 935-7542, kjg@cs.wustl.edu

This research was supported in part by the National Science Foundation under grants CCR-91-10029 and CCR-94-12711.

modules to processors and assigns logical communication channels to physical paths in the network, and a *hardware configuration* that consists of a set of processors and physical communication links. *Dynamic configuration* is the act of modifying the configuration of a system while the modules are executing and interacting.

Logical reconfiguration may involve adding or removing modules, adding or removing logical communication channels, replacing one module by another, or redirecting communication. Physical reconfiguration may involve reassigning a module to a different processor (module migration) or assigning the communication to a different path in the network. Hardware reconfiguration may involve adding or removing processors, or adding or removing communication links. Reconfiguration is a planned activity, and does not include unplanned changes to the system due to hardware or software failures. However, some of the techniques for managing dynamic reconfiguration can be useful in building fault-tolerant systems.

Goscinski [6] points to several important benefits of dynamic reconfiguration. For example, if a new efficient algorithm is developed for a module that is part of a long-running or continuously-available application, dynamic logical reconfiguration makes it possible to replace the original module with the efficient version without having to start over or interrupt service. Dynamic physical reconfiguration is useful for migrating modules from one machine to another in order to perform routine maintenance or balance load. Dynamic physical reconfiguration can also be used to minimize computation time or message traffic of some applications by migrating processes to machines with special-purpose hardware or specialized databases at times when those processes have specific computational or data requirements. Furthermore, when advance notice is available about system shutdown, active processes in the machine can be relocated.

Dynamic reconfiguration must be handled carefully in order to preserve the logical consistency of the system. Changes to the logical configuration should appear atomic. For example, if a module A is replaced by a module B, no module should receive a message originating at A after receiving a message from B. Similarly, changes to the physical configuration should not alter the logical execution (results) of the executing system. For example, if a module on processor 1 is reassigned to processor 2, none of the module's output should be lost or duplicated as a result of the migration. These problems imply certain requirements for distributed environments that support reconfiguration. Hofmeister and Purtilo put forth the following requirements for reconfiguration in

heterogeneous distributed systems [8][1]:

(R1) communication across heterogeneous hosts

(R2) current configuration is accessible

(R3) bindings (interconnections) are not compiled into modules

(R4) no covert communication among modules

(R5) ability to add/remove modules and bindings

(R6) access to messages in transit

(R7) mechanism for synchronizing activities

(R8) access to module's state information

Item (R1) is necessary unless the system is homogeneous. Items (R2) through (R7) are necessary for both logical and physical reconfiguration. Item (R8) is necessary only for module replacement and physical reconfiguration.

Various strategies for different kinds of reconfiguration has been studied. For example, a Durra [1, 2] application can evolve during execution by dynamically removing processes and their ports and instantiating new processes and their ports without affecting other processes. Darwin [10, 14, 12], a generalization of Conic [11, 12], supports logical reconfiguration where the programmer adds code that adapts program modules to participate in reconfiguration. Both Durra and Darwin [2, 10] allow only adding or deleting processes and interconnections between them. PROFIT [9], a recent language that provides a mixture of RPC and data sharing for communication, permits dynamic binding of slots in special cases [7]. Argus [13] supports reconfiguration with two phase locking over atomic objects and version management recovery techniques. Some systems support physical reconfiguration, but support for module migration often has relied upon complicated and expensive techniques for the extraction of the module's state information [16]. Platforms like Polylith [15] support moving a process to another machine while the application is executing. In Polylith, configuration is expressed in terms of a set of procedure call bindings. The programmer specifies "reconfiguration points," that are used to automatically prepare a process

[1]This set of requirements does not appear in their paper, but was part of the conference presentation.

to participate during reconfiguration and special techniques are used to capture internal program state in order to accomplish the migration [8].

In this paper, we consider logical and physical dynamic reconfiguration in heterogeneous distributed systems whose modules are written using a new programming model called *I/O abstraction* [5]. Briefly, I/O abstraction is the view that each software module in a system has a set of data structures that may be externally observed and/or manipulated. This set of data structures forms the external interface (or *presentation*) of the module. Each module is written independently and modules are then *configured* by establishing *logical connections* among the data structures in their presentations. As published data structures are updated, I/O occurs implicitly ("under the covers") according to the logical connections established in the configuration. I/O abstraction is similar to the dataflow model, except that I/O abstraction modules can produce results autonomously, in addition to being able to react to input events, and communication can be bidirectional.

I/O abstraction is designed to simplify applications programming by treating communication as a *high-level* relationship between the states of communicating modules and leaving program I/O as an *implicit* activity. Since low level input and output activities are hidden, programmers need not be concerned with explicitly initiating communication activities, such as sending and receiving messages, and therefore need not be concerned with the particular communication primitives provided by the operating system or the network interface. The connection-oriented view of I/O abstraction also permits continuous media communication (such as audio and video) to be handled with the same high-level communication model used for discrete data.

Two properties make I/O abstraction particularly well-suited for supporting dynamic reconfiguration. The first of these properties is that I/O abstraction achieves a separation of computation from communication. Rather than writing applications programs that are peppered with explicit I/O requests, the applications programmer is concerned only with the details of the computation, and the communication is declared separately in terms of high-level relationships among the state components of different modules. Thus, the logical configuration is known to the system and can be modified independently, without involvement of the software modules. This is certainly a requirement for logical reconfiguration, but is also a necessity for physical reconfiguration (see R2–R5 above).

The second property of I/O abstraction that facilitates reconfiguration is that it exposes certain

state information (see R8) in the presentation of each module. This state information is accessible to the run-time system at all times. This ability to access state distinguishes I/O abstraction from many other programming models that provide access to configuration information. Because I/O abstraction already exposes certain local state information for each module, we are able to avoid some of the problems that have plagued other systems and accomplish physical reconfiguration in a straightforward and efficient manner.

In this paper, we are interested in the problem of *semantic migration*, in which the correctness of the observable behavior of the application is preserved, as opposed to *state migration*, in which the address space of the migrating module is transferred to the destination. That is, our point of view is that of the external environment as opposed to the internal structure of the migrated process. Advantages of semantic migration (in I/O abstraction) include (1) migrating without expensive state extraction, (2) migration across heterogeneous hosts, (3) "take over" by modules with the same presentation written in other programming languages, and (4) minimal (typically one line) change to the program code. In some kinds of applications, however, additional code may be need in the "migratable" program, or intermediate results from computation at the source processor may be lost during migration and must be recomputed.

The remainder of this paper is organized as follows. In Section 2, we describe *The Programmers' Playground*, a software library and run-time system we have designed to support the I/O abstraction programming model. This is followed, in Section 3 by a discussion of reconfiguration in Playground. The emphasis is on a detailed description of a module migration mechanism for supporting physical reconfiguration that takes advantage of the properties of I/O abstraction. Section 4 illustrates the flexibility of the process migration mechanism by describing three different paradigms for constructing relocatable modules that are compatible with our migration mechanism. The implementation of module migration in The Programmers' Playground is discussed in Section 5. We conclude in Section 6 with a discussion and possible directions for future research.

2 The Programmers' Playground

The Programmers' Playground is a software library and run-time system that supports I/O abstraction in terms of three fundamental concepts: data, control, and connections. It is not a

programming language, but rather a coordination language [4] designed to work with multiple computation languages in combination. We discuss data, control, and connections, and then describe a Playground implementation for C++ on the Solaris operating system.

2.1 Data

Data, the units of a module's state, may be private to a module or they may be *published* for access by other modules. Playground provides a library of data types for declaring publishable data. These include base types (integer, real, etc.), tuples, and aggregates (sets, arrays, etc.) of homogeneous elements. Aggregates and tuples may be arbitrarily nested and programmers may define aggregate types in addition to those provided in the Playground library.

Each Playground module has a *presentation* that consists of the set of data structures that have been published by that module. The set of presentation entries may change dynamically as the module runs. However, for the purposes of reconfiguration, we will assume that the presentation is fixed at initialization (although the values of the data structures in the presentation may, of course, change). Documentation and protection information may be associated with each published data item in order to assist in the configuration process and protect against unauthorized use. Data type information is automatically associated with each presentation entry so that type checking may occur at (re)configuration time.

It is helpful to think about a Playground module as interacting with an *environment*, an external collection of modules and users that are unknown to the module but may read and modify the data items in its presentation (as permitted by the protection information defined for the data items). A *behavior* of a module is a sequence of values held by the data items in its presentation. A module's behavior is the view that the environment has of the module.

The notion of behavior is useful for defining the correctness criteria for our reconfiguration algorithm. Let $\mathcal{B}$ be the set of all allowable behaviors of a module M with presentation P, and suppose that M is involved in a reconfiguration (logical or physical). If P_v is the value of P just before the reconfiguration, then M's behavior after the reconfiguration should be a suffix B' of an element of $\mathcal{B}$ such that B' begins with the presentation value P_v. Moreover, if the reconfiguration is only a physical reconfiguration, then M's behavior should not be distinguishable from a possible behavior in an identical system in which the reconfiguration did not occur.

2.2 Control

The control portion of a Playground module is divided into two components: *active control* and *reactive control*. The active control carries out the ongoing computation of the module, while the reactive control carries out activities in response to input from the environment. For example, in a simulation module, the active control would be responsible for the main loop that performs the computation for each event in the simulation, while the reactive control would handle changes in externally controllable parameters of the simulation.

The active control component of a Playground module is the control defined by the "mainline" portion of the module. To define the reactive control, one associates with each data item an activity to be performed when the value of that data item is changed by the environment. As a simple example, one might associate with data item x an enqueue operation for some "update queue" q. With each external update to x, the new value of x would be enqueued into q for later processing.

2.3 Connections

Relationships between the data items in the presentation of different modules are created by establishing *connections* between those data items.[2] The set of connections among published data items defines the pattern of communication among the corresponding modules. Connections are established separately from the definitions of module programs. One designs Playground modules with a local orientation, and later connects them together to form the logical configuration of the system. In this way, each module need not concern itself with the structure of its environment, but only with the behaviors exhibited at its presentation.

Playground supports two kinds of connections, *simple connections* and *element-to-aggregate* connections. A simple connection relates two data items of the same type. For example, an integer x in module A might be connected by a simple connection to integer y in module B. If the simple connection is unidirectional, then the semantics of the connection is that whenever A changes the value of x, item y in module B is updated with that value as well. If the simple connection is bidirectional, then an update of y's value by module B would also result in a corresponding update to x in A. Arbitrary *fan-out* and *fan-in* are permitted, meaning that multiple simple connections

[2] Connections are sometimes called *logical connections* to contrast them with *physical connections* such as links in a computer network.

Figure 1: A Playground system

may emanate from or converge to a given data item. For example, the integer x in the above example might also be connected to integer z in module C. Then, whenever the value of x is changed, y and z are both updated. Communication is asynchronous and pairwise FIFO.

Recall that an aggregate data type is an organization of a homogeneous collection of elements, such as a set of integers or an array of tuples. The *element type* of an aggregate is the data type of its elements. For example, if s is a set of integers, the element type of s is "integer." An element-to-aggregate connection results when a connection is formed between a data item of type T and an aggregate data item with element type T. For example, a client/server application could be constructed by having the server publish a data structure of type *set(T)* and having each client publish a data structure of type T. If an element-to-aggregate connection is created between each client's type T data structure and the server's *set(T)* data structure, then the server program will see a set of client data structures, and each client may interact with the server through its individual element. Details on element-to-aggregate connections may be found elsewhere [5].

2.4 Playground Implementation

A logical overview of a Playground system is shown in Figure 1. A running module consists of three components: the application process, the *protocol* process automatically launched with the application, and a block of *shared memory*, managed by the *veneer*, that is used by both the application process and the protocol. The veneer is a software layer between the application

and the protocol that defines the Playground data types and maintains locking information for concurrency control, as well as the documentation and protection information published with each data structure. Reactive control information is also registered in the veneer.

The application publishes a set of data structures as its presentation. These data structures are held in the shared memory so that they are accessible to both the application and the protocol. The protocol runs concurrently with the application and interacts with the operating system in order to exchange data with other Playground modules on behalf of the application. Connections are established by a special Playground application called the *connection manager*. The connection manager enforces type compatibility among the data items involved in a connection and also guards against protection violations by establishing connections only if they satisfy the access privileges defined for the relevant data items.

The protocol interacts with the connection manager in order to make its module's presentation information known to the connection manager and in order to learn about connections affecting its module. To facilitate communication between the connection manager and the protocol of each module, the presentation of each playground module includes an externally readable data structure P that holds a description of the application's presentation and an externally writable data structure L that contains link information for that module. Data structures P and L are used only by the protocol and are hidden from the application.

The connection manager publishes an externally writable set of presentation descriptions P' that is linked to each module's data structure P with an element-to-aggregate connection. Thus, P' contains a set of presentation descriptions, one for each module in the system. The connection manager also publishes a set L' of link update records, one link update record each module in the system. In order for modules, say a front-end to the connection manager, to know about the connectivity of the system, the connection manager publishes a set C' of connections (not shown in the figure). For a given module m, the element of C' corresponding to m contains the current connectivity (set of links) for m's published data structures.

For establishing logical connections between the presentations of various modules, the connection manager publishes a connection request data structure R which may be updated externally (using an element stream connection) by any module in the system, usually by a graphical front-end module for the connection manager. For each connection request placed in R, the connection man-

ager (through a reaction function) checks for type compatibility, verifies that the connection obeys the access protections established for the endpoint data structures, and adds the connection to its published link update record L'. This change is reflected in the L data structures of the protocols corresponding to the endpoints of the connection. In this way, each protocol is aware of each logical connection in which it is involved. Note that the connection manager is not a communication bottleneck. The connection manager simply sets up the connections that are then handled individually by the protocols associated with the connected data structures. It is also interesting to note that the connection management system itself uses the I/O abstraction mechanism to establish links in the system.

3 Dynamic Reconfiguration Mechanisms

Before proceeding with our discussion of dynamic reconfiguration, we note that the Playground programming environment, along with the connection manager, satisfies the requirements identified in Section 1. Requirement (R1), communication among heterogeneous hosts, is provided by the protocol in the Playground implementation. Requirements (R2) through (R5) are fulfilled by the fact that the logical configuration is maintained in the connection manager. The protocol and veneer provide access to messages in transit (R6) and provide a locking mechanism and causal broadcast algorithm for synchronizing activities (R7). Access to a modules state information (R8) comes "for free" with the presentation, although each module may keep private data that is not published in the presentation.

In this section, we present mechanisms for both logical and physical dynamic reconfiguration in The Programmers' Playground.

3.1 Logical reconfiguration

The logical configuration in The Programmers' Playground is maintained by the connection manager, with each protocol being aware of the connections incident to its module and carrying out the necessary communication under the covers as directed by the veneer. For a module M, we say that the set of modules having logical connections to M are the *peers* of M.

Logical reconfiguration in a running Playground system may involve invoking or terminating

modules, replacing one module by another, and creating and destroying connections. When a module is started, the protocol is automatically launched and links are established with the connection manager (see Section 2.4). When a module terminates, either after completion or due to some failure, the protocol sends out all the pending messages, regarding any updates, to its peers before quitting. The connection manager will then remove the process from all of its connections [5].

We treat replacing a running module by another module as a special case of physical reconfiguration (see Section 3.2), where the module is "moved" to the same machine with replacement of the code.

When a new logical connection is created, the endpoints become aware of it through the connection manager's presentation. The connection manager enforces type compatibility among the data items involved in a connection and also guards against protection violations by establishing connections only if they satisfy the access privileges defined for the relevant data items. When an existing connection is removed, again the endpoints become aware of it through the connection manager's presentation. The protocols of these endpoints send out all the pending messages on this connection.

3.2 Physical Reconfiguration

Physical reconfiguration in The Programmers' Playground involves the reassignment of a module from one processor to another[3]. Any mechanism to support this migration must stop the module's computation on the first processor, move any necessary state from the first processor to the second, and start the computation on the second processor. As defined earlier, the physical reconfiguration algorithm must guarantee that the behavior of each module involved in physical reconfiguration is the same as if the physical reconfiguration did not occur. Thus, every data transmission must occur in the appropriate order, with no such transmissions being lost or duplicated as the result of the migration. Some of the internal computation may be repeated, but the environment should not be able to tell.

Since we wish to avoid expensive state extraction techniques, moving the state information will be accomplished by moving the values in the presentation of the module. However, not *all* the local

[3]We assume that the assignment of a physical communication path to each logical connection is handled by lower level network protocols, so we do not consider that here.

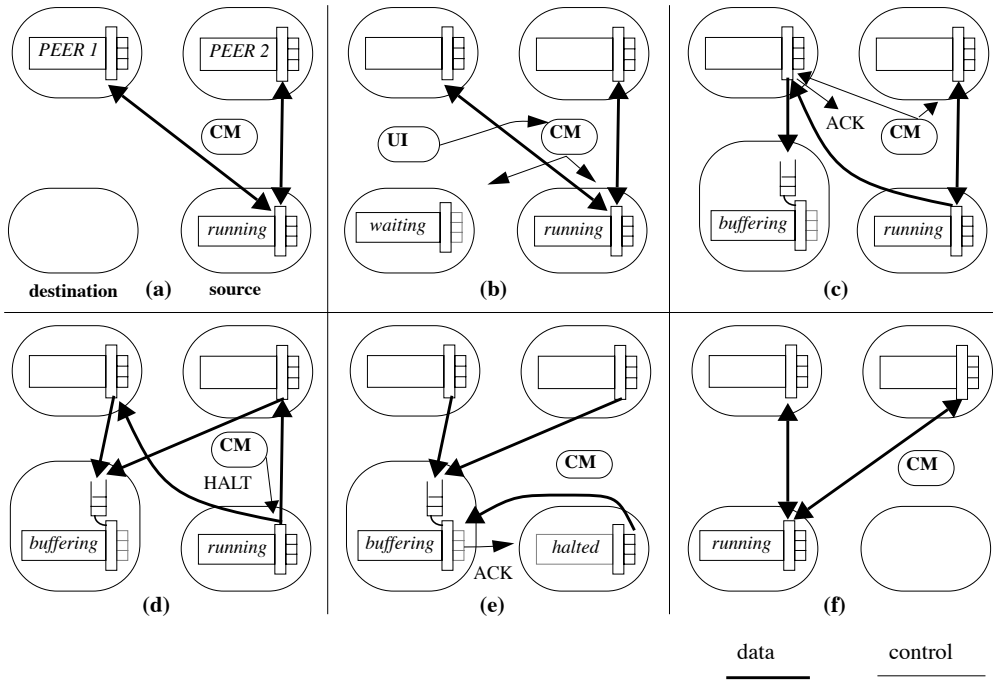

Figure 2: Messages during reconfiguration.

state information is necessarily exposed in the presentation of a Playground module. In fact, it is desirable to have a relatively narrow interface for interaction with the environment. If the most "important" data is exposed, though, that may be enough to restart the module. Otherwise, the module itself may provide reactive control to package up any remaining state information necessary (the *mobile* data) in order to move and restart the module. This is described in more detail in the module migration mechanism that we now present.

In the Playground programming environment, all the connections are logical; hence, during physical reconfiguration the connection information needs no changes. However, when a process is moved, messages from its peers somehow have to be received at the new location. When a module is to be moved, the destination module is launched in "waiting" state (see Figure 2a) at the destination machine, and an entry for the new module is created in the connection manager. In this state, only the protocol is active. The user, through the connection manager user interface, can request that the "running" module be taken over by the waiting module (see Figure 2b). The old module continues to execute normally until it receives a HALT command from the connection manager. The connection manager then updates the link information for the peers with the new

address of the module. This update is communicated to its peers using the normal I/O abstraction mechanism. Figure 2c shows *peer 1* having acknowledged the connection manager and starting to send messages to the new module, with *peer 2* still not having seen the update from the connection manager and still sending its messages to the old module. In both cases, the old module continues to send its messages to the peers. After all the peers acknowledge the connection manager, the connection manager sends a HALT command to the old module. Until this command is received, the old module keeps sending its messages to its peers. The peers send their messages to the new module in the destination machine only (see Figure 2d), where they are received and buffered by the new module, which awaits a *hand-off* message from the old module. The protocol of the old module, after receiving the HALT command from the connection manager, sends its encoded presentation (along with any other mobile data) to the destination as the hand-off message (see Figure 2e). The old module quits after receiving an acknowledgment of the hand-off message (see Figure 2f). Since each module is connected to the connection manager through the distinguished element connection, the corresponding entry for the old module is now deleted. The new module decodes the mobile data and its presentation according to the hand-off message and starts the program from the beginning. This completes the transfer, and the program runs with the transferred mobile data and presentation. Note that the connection manager participates only in the control part of reconfiguration and does not handle any data traffic.

The hand-off message is the actual transfer of control from the old module to the destination. This transfer must be handled carefully to ensure that the resulting behavior is one that could have occurred if the module was never moved. Consequently, the following steps are taken at the source machine just before sending the hand-off message (figure 2e).

(S1) If a reaction function is running, let it complete.

(S2) Suspend active control, respecting atomic changes to the presentation.

(S3) Run an optional *cleanup* routine provided by the applications programmer.

(S4) Close all open files[4].

(S5) Encode the presentation and mobile data, and hand-off to the destination machine.

[4] A distributed file system is assumed so that both the source and destination processors can access the file system.

At the destination, after receiving the hand-off message the following steps are taken:

(D1) Decode the presentation and mobile data.

(D2) Run an optional *restart* routine provided by the applications programmer.

(D3) Restart the program with the restored presentation and mobile data.

Playground modules can register a a cleanup routine and a restart routine with the protocol using a call `PGmobile(cleanup, restart)` to the veneer. The veneer calls the restart function if the program is being restarted as part of reconfiguration. The cleanup routine is registered in the veneer and invoked before relocation begins. Since an ongoing reaction function is allowed to complete before reconfiguration, it is desirable to register short reaction functions. Note that, a module need not take advantage of all the features of the reconfiguration mechanism. For example, it may not have active control or a restart routine.

4 Paradigms for Constructing Relocatable Modules

A module is *relocatable* if its behavior is unaffected by module migration. Our module migration mechanism described in Section 3.2 provides flexible support for writing relocatable modules. In this section, we identify certain guidelines for writing relocatable modules. A relocatable program under semantic migration must satisfy the following conditions.

(G1) The module must have a static presentation (only the values of the presentation entries, not the entries themselves, can change).

(G2) The module must have no covert communication with other modules.

(G3) Mobile data is declared globally. (This is an artifact of the current implementation but is not inherent to the basic mechanism.)

(G4) A `cleanup` function must save any internal data that is not declared mobile.

(G5) A `restart` function must restore any internal data that is not declared mobile.

(G6) When the module is initialized with a presentation and internal state that could have resulted from a behavior β of the module, then the module will exhibit some behavior β' such that $\beta\beta'$ is a correct behavior.

Note that our reconfiguration mechanism allows internal data to be stored and retrieved in separate cleanup and restart functions so that the programmer need not be concerned with reconfiguration throughout the code. We now present three different paradigms for writing relocatable modules. All three paradigms are supported by our module migration mechanism, yet they are very different in style and support a wide range of applications.

4.1 Reactive Paradigm

The reactive paradigm is useful for programs that are not autonomous in nature, but instead interact by invocation and response, such as a database server with many clients that access the data by remote procedure calls. A migratable server can be implemented in Playground using exclusively reactive control. Certain parts of the presentation can be designated for input and the others for output. The server module is reactively invoked when its input changes, and upon completion of the request, the result is written to an output variable (that might be connected to the module that made the request).

Migration is useful for this kind of application. For example, consider a server, performing computationally intensive math functions, that is started at a general purpose machine. When a specialized machine becomes available, the server can be moved to the specialized one without the knowledge of other modules in the system. As another example, data base query processing front-ends that react to user queries can be moved closer to the data base using the reactive paradigm.

Such modules have a static presentation and use only I/O abstraction for communication, so (G1) and (G2) are satisfied. When these modules save any global data used by the reation functions in the mobile data (G3, G4, G5), they become relocatable. The behavioral correctness (G6) follows from the fact that each invocation is processed by a reaction function and module migration does not interrupt reaction functions.

4.2 Active restart Paradigm

The active restart paradigm is useful for programs that have long computations that should be continued after migration. To accomplish this, the programmer may provide a restart routine, which could initialize the local data structures from the mobile data.

In detail, a relocatable program is always started with a call to the `PGinitialize` routine,

followed by a call to the `PGmobile` routine (see Section 3.2). The `PGmobile` call restores the presentation. When a restart routine is registered with the veneer, `PGmobile` routine will execute the restart routine. If the programmer knows that the program could be relocated later, he/she can have special Playground data structures in the program and store certain intermediate results along with the function that is executing in these mobile data structures. This way, the relocated program can jump to the corresponding function to avoid repeating earlier computation performed by the source module.

In order to allow the programmer to store the current results, a cleanup function can be provided by the programmer which will be invoked before the presentation is saved. The cleanup code can be used to either create a log, or to save certain important data structures from the heap into the mobile data structures. Programs handling time consuming database queries can store, in these data structures, information about how far they have searched so that it will not be lost when the programs move. The cleanup routine can also be used, as pointed out earlier, to close open streams and files.

4.3 Active Transaction Paradigm

Our migration algorithm is designed to respect atomicity of critical sections. Simulations and iterative calculations require active control. These programs read the presentation, take a step, and then update the presentation. Since request for relocation may arrive in the middle of an update, each update must be atomic. All the variables in the presentation must be updated (or none at all and the current step aborted).

Primitives for atomic access to a set of data structures are already provided in Playground for atomic updates to the presentation. If an atomic operation involving several published data structures are required, the programmer may use the functions **begin_atomic_step(obj_list)** and **end_atomic_step()** provided by the veneer for encapsulating a set of changes as an atomic step. At the end of the atomic step, all the objects in **obj_list** are modified as one atomic change.

The programmer of iterative or simulation applications can use the above primitives for the objects that are to be updated at the end of each step. Since the migration algorithm always waits for the completion of any atomic step in progress before migrating, this ensures that the modules will be restarted only from consistent states.

4.4 Discussion

The three paradigms discussed above are supported by the reconfiguration mechanism discussed in Section 3.2, and mixtures of these paradigms may also be useful. For example, modules written under the reactive paradigm and the active transaction paradigm need not register any restart or cleanup codes with their protocols, but nothing prevents the modules from registering these functions. In the active transaction paradigm, a cleanup and a restart function might be provided to save and restore partial results of a transaction. Similarly a program written in the active restart paradigm can also have reactive control by registering reaction functions with some of its variables.

As pointed out earlier in Section 3.1 we use physical reconfiguration, but with a slight twist, to accomplish module replacement. We treat the replacement of module A by module B as if it were a physical reconfiguration in which module A is "moved" to the same machine, except that the code is replaced. We require that both A and B have the same presentation type.

5 Implementation

As of this writing, we have a Playground implementation that includes a veneer for C++, a protocol that uses TCP socket communication on top of the Solaris (UNIX) operating system, and a connection manager. The veneer contains implementations for all the basic Playground data types, tuples, and aggregates (mappings, arrays, and general purpose aggregates).

A relocatable program adds a line `PGmobile (list)` after creating its presentation. This call creates a tuple called `mobileTuple` with each object in `list` as its fields. If a cleanup and a restart functions are provided in the `list` they are registered with the veneer, so that the veneer can call them before migration and just before execution in the migrated state. Users start the module normally or with the `migrate-wait` option. When the PGmobile function is called it creates a presentation entry `MIG` with `READ_ONLY` permission in the normal case or with `WRITE_ONLY` permission in the migrate-wait case.

Users specify migration by establishing a connection from the `MIG` variable in the running program to the `MIG` variable in the waiting program. The user interface already supports such a request, so we are able to reuse the connection specification for migration. When a migration request comes to the connection manager and the participating modules and peer protocols take

their steps (as described in Section 3.2), the veneer of the running module encodes the `mobileTuple` and sends it to the waiting module where it is decoded. Since `mobileTuple` is a Playground data type, the encoding and decoding of this structure comes for "free." The protocols and the connection manager communicate using the facilities already part of the Playground run-time system, resulting in a significant reuse of existing objects.

6 Conclusion and Future Work

The properties of I/O abstraction, particularly the clear separation of computation from communication and the availability of a module's state information, provide advantages for reconfiguration. In this paper, we presented mechanisms for both logical and physical reconfiguration that take advantages of these properties. Process migration techniques often do not take advantage of the way program modules are written [16] and may spend time capturing unnecessary state information. Low level state capturing is not always easy and may not work across different architectures or languages. As an alternative, we proposed *semantic migration* in which the meaning, not the low level bits, are migrated.

The Programmers' Playground is implemented not as a new programming language but instead as a layer (*veneer*) between each supported programming language and a *protocol*. In our current implementation, the veneer is written to support applications written in C++. The protocol is implemented in C++ on top of UNIX and uses TCP sockets for the underlying communication mechanism.

An important benefit of I/O abstraction is the potential for integrating discrete data and continuous data within one uniform configuration mechanism. As a testbed for this aspect of the work, we plan to use the high speed packet-switched network that is being deployed on the Washington University campus [3]. The network, called *Zeus*, is based on fast packet switching technology that has been developed over the past several years and is designed to support port interfaces at up to 2.4 Gb/s. The Zeus network will allows us to implement multimedia applications that communicate using real-time digital video and audio, as well as symbolic data. Since the network makes bandwidth guarantees, the module being moved might pre-send extra data to be processed at the peers just before sending the hand-off message to the destination machine. This way the relocation

could be smooth (undetectable) even for multimedia applications.

Our mechanism of reconfiguration does not facilitate open files to be moved when a process is relocated. One might imagine a "stable" attribute associated with certain objects provided in the veneer. Changes to these stable objects would be logged on stable storage that could be copied to the destination *with* the presentation. This would help in the relocation of database modules.

In this paper, we considered reconfiguration in the context of a static presentation. A program must publish its data structures before beginning execution and cannot add or remove entries from its presentation. A "changing" presentation can be achieved partially by using element-to-aggregate connections. For example, a server can publish one variable that is a set of addresses and any number of clients can talk to the server through an element-to-aggregate connection with this set. However, a fully dynamic presentation may be needed for other purposes, for example protection or temporary communication. The problem of writing relocatable modules with fully dynamic presentations remains to be addressed.

References

[1] M. R. Barbacci, D. L. Doubleday, C. B. Weinstock, and J. M. Wing. Developing applications for heterogeneous machine networks: The Durra environment. *Computer Systems*, 2(1), March 1988.

[2] Mario R. Barbacci and Jeannette M. Wing. A language for distributed applications. In *International Conference on Computer Languages, New Orleans, LA, USA*, pages 59–68, March 1990.

[3] Jerome R. Cox, Jr. and Jonathan S. Turner. Project Zeus: Design of a broadband network and its application on a university campus. Technical Report WUCS 91-45, Washington University, July 1991.

[4] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[5] Kenneth J. Goldman, Michael D. Anderson, and Bala Swaminathan. The Programmers' Playground: I/O abstraction for heterogeneous distributed systems. In *27th Hawaii International Conference on System Sciences (HICSS)*, pages 363–372, January 1994.

[6] Andrzej Goscinski. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, Reading, MA, 1991.

[7] Brent Hailpern and Gail E. Kaiser. Dynamic reconfiguration in an object-based programming language with distributed shared data. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 73–80, May 1991.

[8] Christine R. Hofmeister and James M Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, Pennsylvania*, pages 101–110, May 1993.

[9] Gail E. Kaiser and Brent Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, 14(2):201–264, April 1992.

[10] Jeff Kramer and Jeff Magee. Evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[11] Jeff Kramer, Jeff Magee, and Anthony Finkelstein. A constructive approach to the design of distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 580–587, May 1990.

[12] Jeff Kramer, Jeff Magee, and Morris Sloman. Configuring distributed systems. In *5th ACM SIGOPS European Workshop, St. Michel, France*, September 1992.

[13] B. Liskov. Distributed programming in Argus. *CACM*, 31(3):300–313, March 1988.

[14] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. In *International Workshop on Configurable Distributed Systems*, pages 102–117, March 1992. Imperial College of Science, Technology and Medicine, UK.

[15] James M. Purtilo and Pankaj Jalote. An environment for prototyping distributed applications. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 588–594, June 1989.

[16] Jonathan M. Smith. A survey of process migration mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.

APPENDIX

A An Example of a Migratory Program

```
0   // work.cc: Program to do some work 100 times
1   #include "PG.hh"     // include the Playground veneer library headers
2   PGreal in, out;      // the input and output variables
3   PGint counter = 0;   // local counter to keep track of the 100 count
4
5   // do the work once,  and increment counter
6   void do_work (PGreal* object) {
7     // SET out = the result of doing the work on *object
8     counter = counter + 1;
9   }
10
11  // The main line of the program
12  main (int argc, char** argv) {
13    PGinitialize (argc, argv);            // initialize the program
14    PGpublish (in,  "in",  WRITE_WORLD);  // let values come in
15    PGpublish (out, "out", READ_WORLD);   // send values out
16    PGreact (in, do_work);                // call do_work if "in" changes
17    PGmobile (&in, &counter); // only addition to make program relocatable
18    while (counter < 100) {               // wait for 100 computations
19      // .. local computation ..
20    }
21    PGterminate ();                       // cleanup
22  }
```

Figure 3: A relocatable program.

The module in Figure 3 does some computation 100 times. The number of times the computation is performed is maintained in the `counter` variable. A peer module that uses this "work" module will send its value into the `in` variable and will take its result from the `out` variable. Figure 4 shows the modules and their current logical configuration. The WORK modules is in running state and the NewWORK module is in migrate-wait state. When a connection is established from the variable MIG in WORK to MIG in NewWORK (by clicking the mouse at the source and dragging it to the destination), physical reconfiguration takes place. Since the `counter` variable is also requested to be packaged in the `PGmobile (&in, &counter)` call (line number 17), the migrated module will have its counter uptodate and the NewWORK will run for only the remaining

amount of computation. Note that although the variable `counter` is not part of the presentation, we specified it as part of the mobile data to be transferred upon relocation.

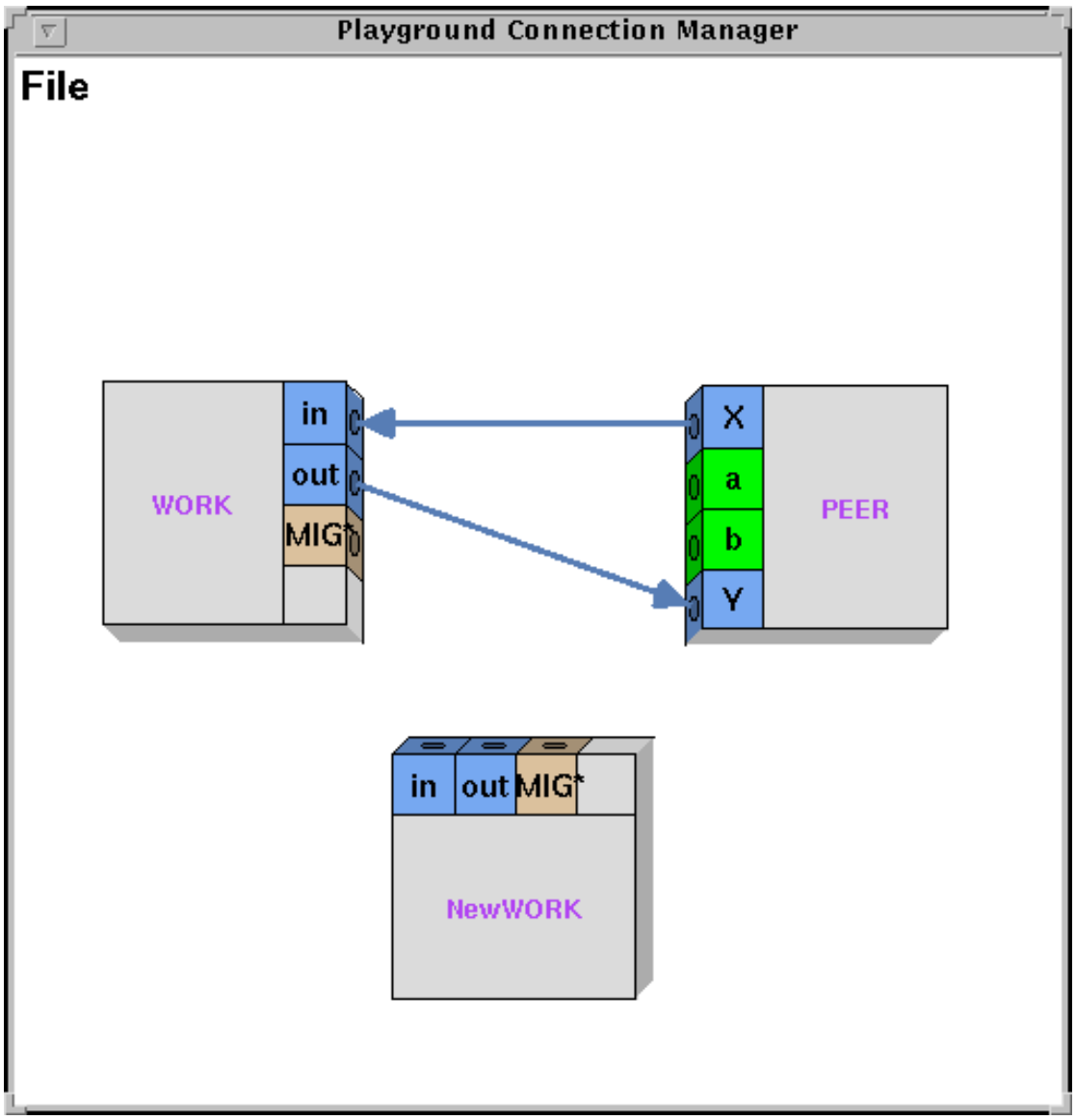

Figure 4: Interacting modules and their configuration.