Report Number: WUCS-TM-94-5

1994-01-01

Connection Management in Reconfigurable Distributed Systems

Bala Swaminathan

The Programmer's Playground takes a new approach to simplifying and supporting the construction of distributed applications. The approach, called I/O abstraction, separates the description of a system's communication structure from the descriptions of its computational components so that software modules written in existing programming languages cna be integrated flexibly and dynamically by both programmers and end-users. This separation is achieved by estabishing logical connectinos among the data interfaces of independent software modules. The logical connections provide a uniform high-level view of communication for both discrete and continuous data. The I/O abstraction approach inherits ideas from the I/O automaton model,... **Read complete abstract on page 2.**

Connection Management in Reconfigurable Distributed Systems

Bala Swaminathan

Complete Abstract:

The Programmer's Playground takes a new approach to simplifying and supporting the construction of distributed applications. The approach, called I/O abstraction, separates the description of a system's communication structure from the descriptions of its computational components so that software modules written in existing programming languages cna be integrated flexibly and dynamically by both programmers and end-users. This separation is achieved by estabishing logical connectinos among the data interfaces of independent software modules. The logical connections provide a uniform high-level view of communication for both discrete and continuous data. The I/O abstraction approach inherits ideas from the I/O automaton model, a formal model of distributed computing that provides compositionality properties and supports behavioral specifications of system modules. Implications of I/O abstraction for process migration and the ordering of events in a distributed system will be studied. Software supporting the I/O abstraction programming model will be constructed. A high speed ATM network developed at Washington University will be used as a testbed for the devlopment work. The availability of this campus network offers an unusual opportunity to construct novel distributed (multimedia) applications and to test our ideas in realistic settings. The connection management network protocol (CMNP), the underlying protocol for the ATM networks, will be formally studied by giving a formal specification.

Connection Management in Reconfigurable Distributed Systems

Bala Swaminathan

WUCS-TM-94-05

September 1994

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

THESIS PROPOSAL

Connection Management in Reconfigurable Distributed Systems

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

September 7, 1994

Abstract

The Programmers' Playground takes a new approach to simplifying and supporting the construction of distributed applications. The approach, called *I/O abstraction*, separates the description of a system's communication structure from the descriptions of its computational components so that software modules written in existing programming languages can be integrated flexibly and dynamically by both programmers and end-users. This separation is achieved by establishing logical connections among the data interfaces of independent software modules. The logical connections provide a uniform high-level view of communication for both discrete and continuous data. The I/O abstraction approach inherits ideas from the I/O automaton model, a formal model of distributed computing that provides compositionality properties and supports behavioral specifications of system modules.

Implications of I/O abstraction for process migration and the ordering of events in a distributed system will be studied. Software supporting the I/O abstraction programming model will be constructed. A high speed ATM network developed at Washington University will be used as a testbed for the development work. The availability of this campus network offers an unusual opportunity to construct novel distributed (multimedia) applications and to test our ideas in realistic settings. The connection management network protocol (CMNP), the underlying protocol for the ATM networks, will be formally studied by giving a formal specification.

Keywords: distributed computing, heterogeneous systems, broadcast, connection management, distributed programming environment

1 Introduction

A heterogeneous distributed system is a large collection of application programs written in many different programming paradigms and running on top of various operating systems and architectures. These are interconnected by a network that allows the users of these applications to share data and expensive resources. Such a loose coupling of distributed systems poses three significant problems [32], namely inconvenience to users, expense, and diminished effectiveness as substantial effort must be diverted to address the problems of heterogeneity.

Writing programs for an "open" heterogeneous environment is not easy. The presence of multiple programming languages and operating systems is one obstacle; however, enforcing the use of a single common language is an impractical solution since different programming paradigms are better suited for different problems. Low level communication primitives have several problems: type checking and parameter marshaling must be done explicitly by the user. The low level calls normally return a status value and different sorts of failures result in different values. A good program must be prepared for any status. The "other process" should know the types of messages used and the message organization. In addition, there is the problem of protecting one's own data and applications, as well as the problem of locating and making proper use of the data and applications of others.

What is needed is a high-level abstraction that can integrate programs written in multiple programming languages and support, in a unified way, the communication needs of a variety of applications. The goals of the Programmers' Playground [16] are to take a fresh look at the traditional mechanisms for interprocess communication in light of the shift from homogeneous to heterogeneous systems and to present an abstraction and supporting software environment that simplifies the construction of distributed applications, provides end-user configuration and integration of software modules, permits a dynamically changing communication structure, offers protection for data and applications, and supports existing programming languages and paradigms. This abstraction and supporting software will serve as an insulating layer between the programming language and the low-level communication protocols.

In this proposal, we first describe the general approach. Then we identify four related research topics that will form the focus of our work. This is followed by the status of our research, the design and implementation of the programming environment, the plan of completion, and the future research directions and integration.

1.1 Approach

Our approach, called *I/O abstraction*, is a model of interprocess communication in which each *module* in a system has a *presentation* that consists of data structures that may be externally observed and/or manipulated. An *application* consists of a collection of independent modules and a *configuration* of *logical connections* among the data structures in the module presentations. Whenever published data structures are updated, communication occurs implicitly according to the logical connections.

I/O abstraction communication is *declarative*, rather than *imperative*. The declarative approach simplifies application programming by cleanly separating computation from communication. Software modules written using I/O abstraction do not make explicit requests to establish or effect communication, but instead are concerned only with the details of the local computation. Communication is declared separately as high-level relationships among the state components of different modules.

I/O abstraction is based on three fundamental concepts: data, control, and connections. We present these concepts in the context of *The Programmers' Playground*, a software library, run-time system and programming environment we are constructing to support the development of distributed applications using I/O abstraction [16].

1.1.1 Data

Data (the components of a module's state) may be kept private or they may be *published* so that other modules may access the data. Playground provides a library of data types for declaring data structures that may be published. These include *base types* for storing integer, real, boolean, and string values, *tuples* for storing records with various fields, and *aggregates* for organizations of homogeneous collections of elements. Some aggregate data types (such as sets, arrays, and mappings) are provided in the Playground library, and the applications programmer may define others. Any Playground data type may be used in the field of a tuple or as the element type of an aggregate.

The presentation: Each Playground module has a *presentation* that consists of the data that it has published. The presentation may change dynamically. Associated with each published data item in a presentation are a *public name*, *documentation*, *access privileges*, and *data type*. The public name, documentation, and data type help users of the module understand its presentation. The data type information also permits *type checking* of logical connections. The access privileges are used to restrict the use of published data structures.

Protection: Access privileges include *read*, *write*, *insert*, and *connect*. Read access allows a module to observe the value of the data structure and write access allows a module to change the value of the data structure. Insert access allows a new element to be inserted into an aggregate as the result of an element-to-aggregate connection, described below. Connect access allows a module (possibly a third party) to relate the data structure to a data structure of some other module. Access protection may be changed dynamically.

The environment: A Playground module interacts with an *environment*, a collection of other modules that may be unknown to this module but that read and modify the data items in its presentation (as permitted by the access privileges).

1.1.2 Control

The *control* portion of a module defines how its state changes over time and in response to its environment. Insulated from the structure of its environment, a Playground module interacts entirely through the local data structures published in its presentation. A module may autonomously modify its local state, and it may react to "miraculous" changes in its local state caused by the environment. This suggests a natural division of the control into two parts: *active control* and *reactive control.*[1] Playground modules may have a mixture of both active and reactive control.

Active control: The active control carries out the ongoing computation of the module. Modules with only active control can be quite elegant, since input simply steers the active computation without requiring a direct response. Active control is analogous to the locally controlled actions of an I/O automaton.

Reactive control: The reactive control carries out activities in response to input from the environment. A module with primarily reactive control simply reacts to each input from the environment, updating its local state and presentation as dictated by that input change. Reactive control is analogous to the input actions of an I/O automaton.

Specifying control: The active control component of a Playground module is defined by the "mainline" portion of the module. Reactive control is specified by associating a *reaction function* with a presentation data item. This function defines the activity to be performed when that data item is updated by the environment.

1.1.3 Connections

Relationships between data items in the presentations of different modules are declared with *logical connections* between those data items. These connections define the communication pattern of the system. Connections are established by a special Playground module, called the *connection manager* (see Section 2.1), that enforces type compatibility across connections and guards against access protection violations by establishing only authorized connections.

Connections are declared separately from modules so that one can design each module with a local orientation and later connect them together in various ways. Connections define the sharing of state change information. However, if a simple asynchronous data transmission algorithm is used, state changes at a connection's endpoints do not necessarily appear to occur atomically. Data transmission ordering is discussed further in the research proposal.

2 Research Areas

The focus of this research plan is to further develop the I/O abstraction concept and The Programmers' Playground in order to support the construction of distributed applications with an emphasis on the goals outlined at the beginning of the proposal. This thesis will focus on configuration management aspects of the Playground environment: (1) connection management, (2) data transmission ordering, (3) module migration, and (4) the analysis of the connection management network protocol (CMNP), a low level ATM network protocol, to understand how logical

[1] RAPIDE[28], a rapid prototyping language for concurrent systems based on partially ordered event sets, is an example of another system that supports this distinction.

connections in the Playground map onto network connections.

2.1 Connection Management

So far, we have been assuming that the environment can change the values of the data items in a module's presentation and that the module's changes to the values of those data items are visible to the environment, but we have not yet described how this communication is established. Since the environment is really nothing more than a collection of (one or more) Playground modules, the question really becomes how to establish a relationship between the data items in the presentations of different modules.

The Playground connection manager handles adding new connections and deleting existing connections between the data items of different modules. The connection manager must provide type checking and must make sure that the access privileges are not violated. The different kinds of connections available (see below) must also be understood by the connection manager.

Playground supports two kinds of connections, *simple connections* and *element-to-aggregate* connections. A given data item may be involved in multiple connections of both kinds.

Simple connections: A simple connection relates two data items of the same type, and may be either *unidirectional* or *bidirectional*. The semantics of a unidirectional connection from integer x in module A to integer y in module B is that whenever A updates the value of x, item y in module B is correspondingly updated. If the 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 so that multiple simple connections may emanate from or converge to a given data item. If x in the above example is also connected to integer z in module C, then whenever x is updated, so are both y and z. Bidirectional simple connections are useful for interactive or collaborative work, while a unidirectional connection with high fan-out would be appropriate for connecting a video source to multiple viewing stations (see Section 6).

Element-to-aggregate connections: A Playground aggregate is an organized 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. As another example, a connection from a data structure of type *T* to a data structure of type *sequence(T)* might be used for a *producer/consumer* application.

Element-to-aggregate connections may take two different forms: *distinguished element* connections and *element stream* connections, with the choice being made when the aggregate is published.

Let x be an integer and s be a set of integers, and consider an element-to-aggregate connection from x to s:

A distinguished element connection from x to s causes a new element to be created in the aggregate s. All interaction for that connection takes place through that distinguished element and x, as if there is a simple connection between x and the distinguished element of s. The distinguished element is deleted when the connection is removed. Distinguished element connections are suitable for the client/server scenario described above. Like simple connections, they may be unidirectional or bidirectional, and permit arbitrary fan-out and fan-in. In the client/server example, arbitrarily many clients could be handled by multiple distinguished element connections to the same aggregate, each with its own distinguished element.

An element stream connection from x to s causes a new element (with the value currently held by x) to be created in s *each time x is updated.* Element stream connections are suitable for the producer/consumer scenario described above and are inherently unidirectional (from the element to the aggregate). Multiple fan-in is allowed, and could be used to allow many modules to produce elements for a single consumer, for example.

2.1.1 Related Work

Coordination languages [14] separate communication from computation in order to offer programmers a uniform communication abstraction that is independent of a particular programming language or operating system. The separation of computation from communication permits local reasoning about functional components in terms of well-defined interfaces and allows systems to be designed by assembling collections of individually verified functional components. There are many examples. Darwin [21, 30, 23] is a coordination language for managing message-passing connections between process "ports" in a dynamic system. Processes are expressed in a separate computation language that allows ports to be declared for interconnection within Darwin. In Polylith [34, 35], a configuration is expressed using "module interconnection constructs" that establish procedure call bindings among modules in a distributed system. CONCERT [45] provides a uniform communication abstraction by extending several procedural programming languages to support the Hermes [40] distributed process model. PROFIT [20] provides a mixture of data sharing and RPC communication through *facets* with data and procedure *slots* that are bound to slots in other facets during compilation. Extensions to PROFIT enable dynamic binding of slots in special cases [17]. Coordination languages can be implemented directly on top of each supported operating system and programming language, or for ease of portability, they may be implemented on top of a uniform set of system level communication constructs for heterogeneous distributed systems, such as the Mercury system [27] or PVM [13].

2.2 Data transmission ordering

Our current implementation of I/O abstraction (see Section 3) supports only asynchronous communication, but many applications require stronger properties of message ordering. We plan to implement a *causal ordering*[2] mechanism for the Playground environment.

Causal ordering is supported in the ISIS system [7], where the programmer declares the groups of processes to receive multicast messages. Efficiency of the algorithm (in terms of message header size) depends upon knowing these groups, because messages are ordered within each group. With I/O abstraction, the programmer does not declare process groups. The configuration is declared separately from the modules and may change dynamically. For example, bidirectional logical connections might be added between two formerly disconnected sets of modules that communicate among themselves using causal ordering. Provided that point-to-point connections are FIFO, when the first connection is added, no change is required, but when the second connection is added, there is a possibility of causal ordering violations, so the causal ordering algorithm must now treat the two sets as one.

In order to handle causal broadcast in a dynamically changing graph, we plan to exploit information available in the logical configuration graph. Since the biconnected components of the graph form a tree, by causally ordering the messages within each biconnected component (bcc), we will be able to guarantee that causal ordering for the entire system. There are many distributed algorithms for computing biconnected components [2, 19, 33]. These algorithms use some form of distributed depth first search [1, 25], and then use a probe-echo [3] algorithm to compute the biconnected components. They are inherently static, that is, a simple change to the original graph, like adding an edge, would involve starting the algorithms all over. We are interested in distributed bcc algorithms which can dynamically maintain the bcc information of a graph.

With this motivation, we have developed two incremental distributed algorithms, (1) a serialized algorithm where graph update requests occur one at a time, and (2) a concurrent algorithm where simultaneous graph update requests are permitted, for maintaining biconnected components in a dynamically changing graph [42]. We also plan to demonstrate the usefulness of our biconnected component algorithm by implementing the serialized version of the BCC algorithm on top of the Playground. The causal ordering mechanism will then be integrated with the BCC implementation to allow for incremental updates over the connection graph.

2.2.1 Related Work

The problem of computing biconnected components has been studied extensively. A number of sequential algorithms exist for dynamic graphs and there are several decentralized algorithms for static graphs, but the algorithms we present in this paper are, to our knowledge, the first distributed algorithms for finding biconnected components in dynamic graphs.

A distributed algorithm for finding biconnected components was given by Chang [9]. This algorithm has a message complexity of $4m-n$, where m and n are the number of edges and vertices in the graph respectively. The distributed algorithm by Ahuja and Zhu [2] has the same message complexity but improves on the message size bound. Hohberg [19] and Park et al. [33] present

[2]Suppose that information from process A is sent to processes B and C, process B makes a decision (possibly based on A's information), and C sees the result of that decision. Causal ordering would require that C observe A's information before seeing the results of B's decision.

distributed algorithms for finding biconnected components in a graph with a message complexity of $O(m + n \log n)$. These algorithms require the computation of a depth-first search tree. Hence, they are appropriate for a static graph, but the cost of recomputation of the depth-first search tree (for every change in topology) makes these algorithms impractical for a dynamic setting.

Tarjan and Vishkin proposed an optimal parallel algorithm on CRCW PRAM model [43]. This algorithm is also not incremental, but instead of using depth-first-search, it reduces the biconnectivity problem to the problem of computing connected components. Westbrook and Tarjan [44] proposed a sequential algorithm to compute biconnected components in a dynamically changing graph structure. A block forest of block trees is constructed using the biconnected components and the vertices in the graph. This block forest is used in maintaining the biconnected components of the original graph. Rauch [36] presented a sequential algorithm for maintaining biconnected components. This algorithm involves precomputation and "lazy" updating.

The algorithms we present are dynamic as well as distributed, and are designed to scale up for large systems. Update requests can be issued at any node in the system and in any order. The nodes have only local knowledge of the system graph and exchange information with other nodes by sending and receiving messages. Only one copy of the topology is maintained and it is distributed among the nodes in the system.

2.3 Reconfiguration and Process Migration

Module migration is the relocation of a running process from one host to another such that the move is transparent to the modules that interact with it. By supporting process migration, we can allow dynamic end-user manipulation of not only the logical configuration, but also the allocation of modules to processors. This is useful for load balancing, relieving network congestion and handling hardware maintenance without interruption of processing.

Data interfaces (such as I/O abstraction) can simplify run-time support for module migration and facilitate writing migratable modules. The trick is to provide a clean data interface that captures only the essential I/O behavior of a module and at the same time exposes enough state information so that the module can be relocated without the need for separate state extraction techniques. Once the state information is exposed, the next problem is determining when it is safe to move the module. The programmer might specify this explicitly by identifying safe points in the code (as in [18]), but we prefer confining the process migration code (cleanup and restart) to one section of the program and relying on existing mechanisms for specifying atomic steps to prevent untimely migration. If a module is moved between atomic steps, then its behavior will be unaffected.

In [41], we suggest three approaches to writing migratable modules that exploit data interface properties, localize process migration code, and use atomicity mechanisms of The Programmers' Playground. 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

[3]We assume that the assignment of a physical communication path to each logical connection is handled by lower

the module's computation on the first processor, move any necessary state and the code 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 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. Modules written in this way may include a procedure to initialize the local state from the presentation, and a cleanup function to be invoked before the module migration mechanism saves the presentation. We say that modules written in this fashion follow the *active restart paradigm.* Sometimes, the module itself may provide reactive control to package up any remaining state information necessary in order to move and restart the module. Such modules are said to be written in the *reactive paradigm.* Otherwise, the modules can be written in the *atomic transaction paradigm*, where the module accesses the presentation data items atomically using the primitives provided in the Playground implementation (see Section 3).

We plan to incorporate these algorithms into our run-time system in order to see their performance in practice. We also expect that some of these techniques may also be useful as part of lock recovery mechanisms for our concurrency control algorithms (see Section 6).

2.3.1 Related Work

An important property of I/O abstraction that facilitates reconfiguration is that it exposes state information 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. For example, a Durra [4, 5] application can evolve during execution by dynamically removing processes and their ports and instantiating new processes and their ports without affecting other processes. Darwin [21, 30, 23], a generalization of Conic [22, 23], supports logical reconfiguration where the programmer adds code that adapts program modules to participate in reconfiguration. Both Durra and Darwin [5, 21] allow only adding or deleting processes and interconnections between them. PROFIT [20], a recent language that provides a mixture of RPC and data sharing for communication, permits dynamic binding of slots only in special cases [17]. Argus [26] 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

level network protocols, so we do not consider that here.

expensive techniques for the extraction of the module's state information [38]. Platforms like Polylith [35] 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 to participate during reconfiguration and special techniques are used to capture internal program state in order to accomplish the migration [18]. Because I/O abstraction already exposes local state information for each module, we are able to avoid some of the problems that have plagued other systems and accomplish reconfiguration in a straightforward and efficient manner.

2.4 Network Signaling Protocol

Formal models are increasingly used in the design of communication systems. The precise specification and verification of systems help avoid design flaws that go unnoticed until the later stages of the design. Formal specification also provides users with an understanding of the services available in the system. Formally describing a service at an abstract level, from a protocol that provides that service, helps in standardization and customization of that service [12]. In this context we are working on a formal specification and a proof of a network protocol called Connection Management Network-node Protocol (CMNP) that is designed for use with the high speed ATM networks developed here at Washington University. The "users" of this protocol are those protocols that interface with the real world applications. The Playground Programming environment will interface with CMNP to provide the desired communication abstraction between user applications.

We define *network signaling* as the exchange of control messages among elements of a communication network to manage the network, and to initiate, maintain, and terminate connections between endpoints in a network. CMNP is a network signaling protocol that is being designed for the high speed ATM networking project at Washington University. The protocol is designed to address a wide range of services dominated by multipoint multimedia communications. CMNP is a distributed protocol and the network nodes cooperate to provide the service.

We believe that CMNP is the first network node signaling protocol that supports dynamic multimedia multicast connection set up. This is very suitable for the evolving Playground environment (see Section 6). In this area, we present a formal model of a simplified version of CMNP. We have modeled CMNP using the I/O automaton model of Lynch and Tuttle [29]. It is well suited for modeling asynchronous distributed systems. The purpose of the presentation is two fold: (1) to present a practically useful and correct multipoint network node signaling protocol, and (2) to show the power of formal modeling techniques, both in protocol modeling and verification.

2.4.1 Related Work

Integrated Service User Part (ISUP) of SS7 [8] is a network node protocol that has been employed in practice. It supports circuit switching in ISDN. Segall and Jeffe [37] proposed a connection set up protocol with local identifiers. Spinelli [39] presented a reliable route set up and reliable data transmission protocol and later modified this protocol to include self-stabilization. Both these

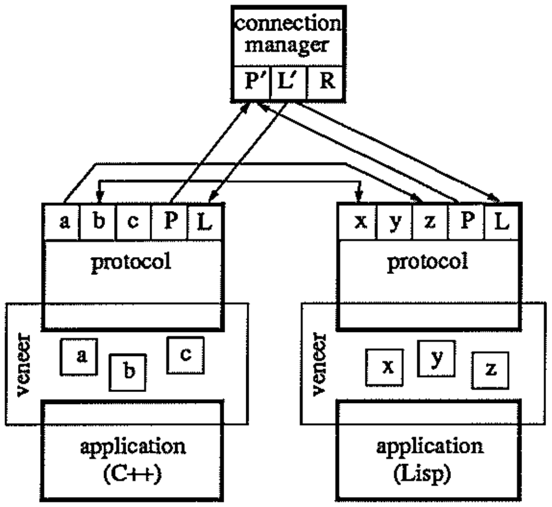

Figure 1: A Playground system

protocols are for point-to-point connections. Segall et al. [11] proposed a reliable multiuser tree set up protocol which can be used in a high speed network. There are two major limitations to their protocol. First, the node that initiates the connection set up must know the network topology. Second, the multicast tree is dynamic in the sense that any node in the tree can only leave the tree freely, but a new node cannot be added to an existing connection tree. Our protocol is really distributed and dynamic. Any node can join or leave a connection at any time and no node needs to keep global states. Cidon et al. [10] presented a connection set up protocol which can set up a point-to-point or limited multicast connection in a high speed network. Since the reservation phase is fast, the potential contention probability is reduced. Their protocol does not provide support for general multicasts and in order to support the protocol the underlying hardware must support source routing, an esoteric feature that many switching nodes do not have.

3 Design and Implementation

The Programmers' Playground is designed as a software library, run-time system, and programming environment that insulates the applications programmer from the operating system and the network. The version of the system described here supports applications written in C++ on top of the SunOS UNIX operating system with sockets as the underlying communication mechanism. A logical overview of a Playground system is shown in Figure 1.

Veneer: Each Playground module is written using I/O abstraction, as described earlier. Each module includes a software library called the *veneer* that serves as an abstraction barrier between the Playground module and its environment. The veneer defines the Playground datatypes, manages the presentation information, and handles reactive control. Each supported programming language requires its own Playground veneer.

Protocol: At module initialization, the veneer automatically launches a separate *protocol pro-*

cess that handles interprocess communication resulting from updates to published data. The module's veneer and protocol communicate through data structures in shared memory. The protocol also interacts with the *connection manager*, a special application that is used to create logical connections among the published data items of different modules. The protocol makes the module's presentation known to the connection manager and the connection manager informs the protocol of the connections established between the module's published data structures and those of other modules.

Connection Manager: The connection manager is itself implemented as a Playground module. It interacts with the protocol processes of other Playground modules through element-to-aggregate connections that are automatically set up by those protocol processes. These "bootstrap" connections are used by the protocols to convey presentation descriptions to the connection manager, and to learn about changes to the logical connection structure. (These presentation entries are shown in Figure 1 as `P` and `L`, respectively.) The connection manager also publishes a connection request data structure (shown as `R` in the Figure 1) and registers a reaction function with it. This data structure can be updated by modules such as a graphical configuration application. For each connection request, the 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 connection information. The protocols at the endpoints of the requested connection are advised of the new connection through the normal implicit I/O abstraction communication mechanism. Note that the connection manager is not a communication bottleneck since it simply sets up connections that are thereafter handled individually by the endpoint protocols.

Communication: Whenever the application updates the value of a published data structure, the veneer encodes the data and informs the protocol. The protocol then forwards the new value to all all other modules to whom an outgoing connection has been established from that data structure. Depending on the encoding scheme used, the entire data structure or only the updated portion is sent. Upon receipt of a new value for a data structure, the protocol forwards this update to the application veneer which updates the data structure and any necessary reactive control is handled. Note that all of this I/O happens implicitly, as the result of an (overloaded) assignment to the published data structure.

Atomicity: (see Section 6) Locks are used to prevent two applications from concurrently changing the data structures at the endpoints of a single logical connection. The lock (token) is held by the veneer before each update and released after the update. When the lock for a logical connection is not local, the protocol makes an external request for the lock on behalf of the veneer. The protocol is responsible for ensuring that at most one lock exists among the protocols participating in each logical connection, and it regenerates the token when it is lost (due to a partition, for example).

The locks alone do not prevent "blind" writes in which a value written by one module is obliterated without being observed by any other module. If an atomic read-compute-write for a published data structure is required, or if an atomic operation involving several published data structures is 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. The `obj_list` names the set of objects for which locks should be held for the duration for the atomic step. At the end of the atomic step, the locks are released and all the changed objects are forwarded to other applications as one atomic change.

4 Status

Playground and Connection Manager: As of this writing, we have a small Playground implementation that includes a veneer for C++, a protocol that uses TCP socket communication on top of the SunOS (UNIX) operating system, and a connection manager. The veneer contains implementations for all the basic Playground data types, tuples, and some aggregates (mapping, set, queue, and array). The protocol, launched with each application, automatically sets up a "primary" socket for providing presentation description information to the connection manager and receiving connection information from the connection manager through the usual implicit I/O abstraction communication mechanism.

All updates to the presentation data result in the necessary implicit communication according to the logical connections. These updates are caught by overloading the assignment operator for the Playground data types. Currently, incremental changes to aggregates result in the entire new value of the aggregate being sent by the protocol, instead of just the changed element(s). Whenever the application reads a Playground datatype , any pending input changes are handled so that the application sees recent and consistent data. The protocol's concurrency control algorithms (see Section 6) are not yet implemented, so race conditions for updates to data elements are possible. Simple connections and element-to-aggregate connections are supported. While type checking is enforced, access protection is not currently enforced.

Connections are made through a graphical front-end to the connection manager [31] that provides dynamic configuration of logical connections between the presentations of Playground modules.

Playground currently supports the development of distributed applications using I/O abstraction. However, we have not yet ported Playground to the ATM network, and we have not yet added datatypes to the veneer for supporting continuous media (such as audio and video, see Section 6).

Data Transmission Ordering: In [42], we have present two incremental distributed algorithms for computing biconnected components in a dynamically changing graph. The serial algorithm requires that the environment issue only one update request at a time. The concurrent algorithm allows the environment to make multiple concurrent update requests. The algorithm serializes the requests within each connected component with each node in a connected component having the same view of the update sequence, while allowing requests in different connected components to proceed in parallel. The algorithm uses logical clocks and collects timestamps from nodes in a connected component in order to achieve identical view of the update sequence across nodes. As the graph changes dynamically, ordering information is propagated to ensure consistency.

Our concurrent algorithm has a worst case communication complexity of $O(b + c)$ messages for an edge insertion and $O(b' + c)$ messages for an edge removal, and a worst case time complexity of $O(c)$ for both operations, where c is the maximum number of biconnected components in any of the connected components during the operation, b is the number of nodes in the biconnected component containing the new edge, and b' is the number of nodes in the biconnected component in which the update request is being processed.

Reconfiguration and Process Migration: Dynamic reconfiguration is explored in the context of I/O abstraction [41]. 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 strategies. Both logical and physical reconfiguration are discussed, with an emphasis on a new module migration mechanism that avoids the expense and complication of state extraction techniques. 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. The flexibility of our migration mechanism is illustrated by presenting three different paradigms: the reactive paradigm, active restart paradigm, and active transaction paradigm, for constructing reconfigurable modules that are compatible with this new mechanism.

In our work, 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. This is not a very serious restriction as a "changing" presentation can be achieved in the static case 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 dynamic presentation may be needed for other purposes, for example protection or temporary communication. One way to achieve this is to restart the program with its initial presentation, and only the values of variables that are still in the current presentation will be updated in the new machine.

We propose to implement the reactive paradigm for reconfiguration and integrate it with the Playground environment as part of this proposal.

5 Plan of Completion

The research project is divided into four tasks and will require twelve months to complete including write up of the dissertation. The task plan is shown in the table below.

Tasks	Expected completion time
Playground Implementation:	
Veneer with simple datatypes	August 1994
Veneer with aggregates (except sets)	August 1994
Playground Protocol	August 1994
Connection Manager:	
Simple connections	August 1994
Bootstrap connections with other applications	August 1994
Type checking	August 1994
Element to Aggregate connections	September 1994
Protection checking	September 1994
Playground documentation	October 1994
Reconfiguration:	
Module migration algorithms description	August 1993
Reactive paradigm	January 1995
Integration and Documentation	February 1995
Data Transmission Ordering	
BCC algorithms description	April 1994
Static causal ordering algorithm	March 1995
Serialized BCC algorithm	April 1995
Integration of serialized algorithm with causal ordering	May 1995
Documentation and technical write up	July 1995
Connection Management Network Protocol:	
CMNP specification	July 1994
CMNP verification	December 1994

6 Future Related Work and Integration

Note that I/O abstraction has a "read-always" semantics. Hence, traditional database concurrency control theory [6] does not carry over directly to I/O abstraction systems. Rather than actively performing an atomic read on an object, an application has continuous access to external updates through its presentation, so it is difficult to identify what would constitute a "read lock." In many applications, a mechanism (e.g., write locks) is required to prevent race conditions on updates to published data. However, this is not always required. For example, when only reactive control is used at the endpoints of bidirectional connections, it may be perfectly reasonable for both endpoints to write simultaneously.

Our work on concurrency control will concentrate on developing correctness conditions and mechanisms for specifying concurrency control requirements for I/O abstraction applications. Based on these conditions, we will develop concurrency control algorithms to support a range of concurrency control requirements. The usual concurrency control concerns, such as deadlock, will be considered. We expect close ties between the message ordering restrictions and the concurrency control correctness conditions.

In The Programmers' Playground, the connections should be designed to accommodate both discrete data (such as sets of integers) and continuous data (such as audio and video) in a single high-level mechanism, with *differences in low-level communication requirements handled automatically by the run-time system according to data type information.* Synchronization of media streams will play an important role in determining the usefulness of continuous media datatypes. The simplest example of media synchronization is the synchronization of audio and video information. ATM network protocols provide some synchronization services, but our main concern is in how the programmer will specify synchronization requirements and how the run-time system will implement those requirements in terms of the primitives provided by the network. We plan to develop multimedia datatypes whose semantics include synchronization requirements. For example, in addition to video and audio data types, we plan to support higher level objects such as media streams that can be composed and spliced together to form other media streams. Then, by treating real time as a primitive stream, we will be able to specify objects with complex synchronization and timing requirements.

It will be important to distribute the functionality of the connection manager in order to provide a scalable service. We plan to provide an evolving logical hierarchy of connection managers. Our plan is to have the connection manager operate with local knowledge, so that each user could have one connection manager. Connections between applications of different users is handled by first including the presentation of one connection manager into the other connection manager. For example, when a connection between modules of user A and user B is required, the connection manager of user A can become an ordinary application in user B's connection manager. Now user B's connection manager has access to the presentations of user A's modules.

The Programmers' Playground when integrated with proper concurrency control algorithms and multimedia datatypes and media synchronization mechanisms would simplify the construction of distributed multimedia applications, such as digital libraries, efficient video conferencing, and medical imaging.

References

[1] A. Aggarwal, R. J. Anderson, and M. Y. Kao. Parallel depth-first search in general directed graphs. *SIAM Journal of Computing*, 19:397–409, 1990.

[2] Mohan Ahuja and Yahui Zhu. An efficient distributed algorithm for finding articulation points, bridges, and biconnected components in asynchronous networks. In *Proceedings of the 9th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India. LNCS 405*, pages 99–108. Springer–Verlag, December 1989.

[3] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[4] 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.

[5] 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.

[6] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1986.

[7] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[8] CCITT. Specification of signaling system no. 7. *CCITT Blue Book*, 6, 1988.

[9] E. J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, 1982.

[10] Israel Cidon, Inder Gopal, and Adrian Segall. Fast connection establishment in high speed networks. *IEEE Transactions on Networking*, (August):469–481, 1993.

[11] Adrian Segall et al. Reliable multiuser tree setup with local identifiers. *IEEE Transactions on Communications*, 34(1):1427–1439, 1991.

[12] Alan Fekete. Formal models of communication services: A case study. *IEEE Computer*, 26(8):37–47, August 1993.

[13] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Sixth Annual Distributed-Memory Computer Conference*, pages 258–261, 1991.

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

[15] Kenneth J. Goldman, Michael D. Anderson, and Bala Swaminathan. The Programmers' Playground: I/O abstraction for heterogeneous distributed systems. Technical Report WUCS–93–29, Washington University in St. Louis, November 1992.

[16] 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.

[17] 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.

[18] 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.

[19] Walter Hohberg. How to find biconnected components in distributed networks. *Journal of Parallel and Distributed Computing*, 9(4):374–386, August 1990.

[20] 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.

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

[22] 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.

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

[24] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceeding of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 43–57, August 1990.

[25] K. B. Lakshmanan, N. Meenakshi, and K. Thulasiraman. A time-optimal message efficient distributed algorithm for depth-first search. *Information Processing Letters*, 25:103–109, 1987.

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

[27] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. In *Hawaii International Conference on System Sciences (HICSS)*, pages 178–187, January 1988.

[28] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3), June 1993.

[29] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3), 1989.

[30] 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.

[31] T. Paul McCartney and Kenneth Goldman. Visual specification of interprocess and intraprocess communication. In *Proceedings of the 10th International Symposium on Visual Languages*, October 1994. To appear. Also available as Washington University Department of Computer Science Technical Report WUCS-94-10.

[32] D. Notkin, A. P. Black, E. D. Lazowska, H. M. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *CACM*, 31(3):258–273, March 1988.

[33] Jungho Park, Nobuki Tokura, Toshimitsu Masuzawa, and Kenichi Hagihara. Efficient distributed algorithms solving problems about the connectivity of network. *Systems and Computers in Japan*, 22(8):1–16, May 1991.

[34] James M. Purtilo and Christine R. Hofmeister. Dynamic reconfiguration of distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560–571, May 1991.

[35] 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.

[36] Monika Rauch. Fully dynamic biconnectivity in graphs. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 50–59, October 1992.

[37] Adrian Segall and Jeffrey M. Jaffe. Route setup with local identifiers. *IEEE Transactions on Communications*, 34(1):45–53, 1986.

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

[39] John M. Spinelli. Reliable data communication in faulty computer networks. Ph.D. dissertation, MIT, 1989.

[40] R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, D.M. Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing.* Prentice-Hall, 1991.

[41] Bala Swaminathan and Kenneth J. Goldman. An incremental distributed algorithm for computing biconnected components. Technical Report WUCS–93–6, Washington University in St. Louis, September 1993.

[42] Bala Swaminathan and Kenneth J. Goldman. An incremental distributed algorithm for computing biconnected components (extended abstract). Technical Report WUCS–94–6, Washington University in St. Louis, February 1994.

[43] Robert E. Tarjan and Uzi Vishikin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[44] Jeffery Westbrook and Robert E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[45] Shaula A. Yemini, German S. Goldszmidt, Alexander D. Stoyenko, and Langdon W. Beeck. CONCERT: A high-level-language approach to heterogeneous distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 162–171, 1989.