The Programmers' Playground: I/O Abstraction for Heterogeneous Distributed Systems

Kenneth J. Goldman, Michael D. Anderson
and Bala Swaminathan

WUCS-93-29

June 1993
(supersedes WUCS-92-32, revised 2/94)

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

The Programmers' Playground:
I/O Abstraction for Heterogeneous Distributed Systems

Kenneth J. Goldman,* Michael D. Anderson and Bala Swaminathan
Department of Computer Science
Washington University
St. Louis, MO 63130
kjg@cs.wustl.edu
(314) 935–7542

Abstract

I/O abstraction is offered as a new high-level approach to interprocess communication. Functional components of a concurrent system are written as encapsulated modules that act upon local data structures, some of which may be published for external use. Relationships among modules are specified by logical connections among their published data structures. Whenever a module updates published data, I/O takes place implicitly according to the configuration of logical connections.

The Programmers' Playground, a software library and run-time system supporting I/O abstraction, is described. Design goals include high-level communication among programs written in multiple programming languages and the uniform treatment of discrete and continuous data types. Support for the development of distributed multimedia applications is the motivation for the work.

1 Introduction

Consider the vision of a global electronic infrastructure with sufficient communication bandwidth to support remote collaboration, information and resource sharing, and access to electronic services and broadcast media. This infrastructure will be *heterogeneous*, consisting of many computer architectures running various operating systems and supporting many different programming languages and programming paradigms. Furthermore, this infrastructure will be *dynamic*, evolving over time, with users coming and going in order to collaborate, share information, and to provide and use services electronically.

*This research was supported in part by the National Science Foundation under grants CCR-91-10029 and CDA-91-23643. A preliminary version of this work appeared in the *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994, pages 363-372.

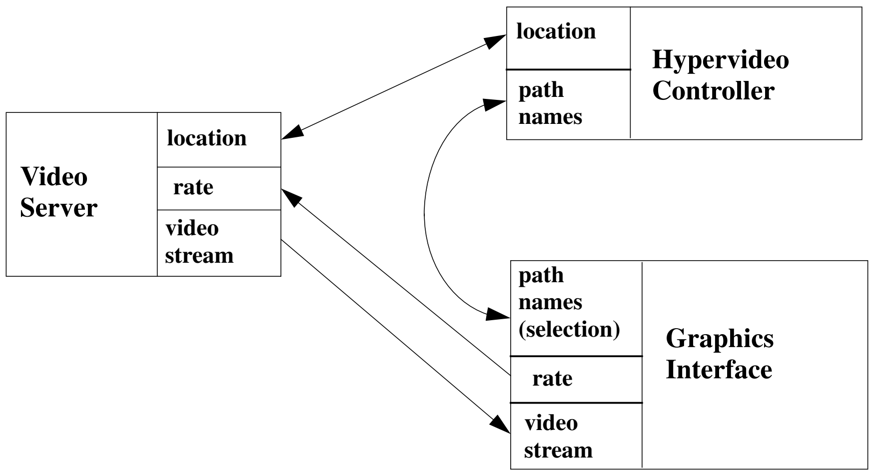

Figure 1: A hypervideo browser

Unfortunately, writing programs to communicate in a public heterogeneous environment is not necessarily easy. Obstacles include the presence of multiple programming languages, multiple operating systems, and multiple communication protocols. However, simplification of the task by enforcing the use of a single common language, operating system, and low-level communication protocols is an impractical solution. Different programming paradigms are better suited for different problems and individuals have personal preferences, and different applications require different kinds of communication services. In fact, a single multimedia application may have a variety of communication needs in order to handle continuous data (such as real-time video and audio), as well as discrete data. To be successful, an open computing environment must also provide access protection and privacy, as well as paid access to electronic resources and services.

2 Motivation and Goals

The long-term objective of this work is to facilitate the construction and smooth integration of diverse of distributed multimedia applications. A simple example of a distributed multimedia application is the hypervideo browser shown in Figure 1. The application consists of a hypervideo controller module that interacts with two other modules: a video server and a graphics interface.

The video server has a data interface that includes the current video location (frame number), the rate of play, and the video stream itself. The hypervideo controller internally maintains a simple data structure for a directed graph whose vertices denote video segments (start and stop frame numbers) and whose edges are labeled with sets of path names such that each vertex has at most one incoming and at most one outgoing edge labeled with a given path name. The intent is that viewing the sequence of video segments denoted by the vertices of a labeled path conveys a particular idea or message. The hypervideo controller supplies a video location (connected to the video server) and a list of path names labeling the incoming and outgoing edges of the vertex for the segment currently being viewed. At all times, one of these path names is marked as the current "selection." The graphics interface has a rate of play output (controlled graphically and connected to the play rate of the video server), a video stream input (connected to the video stream of the server and visible in a video window on the display), and a set of names (connected to the path names of the hypervideo controller and selectable on the display).

As the hypervideo controller executes, it traverses an externally-selected path in the graph. At each vertex, it updates the video location to the start position in that vertex. This is seen by the video server, which begins playing video from that location at the specified rate. As the video is played, the server continually updates the position information, which is seen by the hypervideo controller. When the end of the video segment for the current vertex is reached, the hypervideo controller reacts by moving to the next vertex along the currently selected path.

This hypervideo controller is a fairly simple application, but one might imagine extending the hypervideo controller to be a sophisticated hypervideo editor, where the graph structure may be changed by direct manipulation in the graphics interface.

It is important to notice that the hypervideo controller itself does not store any video data, but simply maintains a graph data structure. Also, notice that the video location is updated by both the hypervideo controller and the video server, and both must react to changes. A different form of interaction occurs with the path selection. Selected by the user interface, the choice of path is noticed passively by the hypervideo controller. Traversal to the next vertex is made according to the currently selected path, but no reaction is necessary at the time the selection is changed.

This application is configured from multiple independent modules, incorporates both discrete and continuous data, and can be highly interactive, depending upon the duration of the video segments. The video server is not specific to this application and could be used by multiple clients in a variety of applications.

Motivated by the requirements of distributed multimedia applications, this paper describes an abstraction and supporting software that

1. simplifies the construction of distributed applications,
2. provides end-user configuration and integration of software modules,
3. is designed for high-bandwidth communication technology,
4. provides uniform treatment of discrete and continuous data,
5. permits a dynamically changing communication structure,
6. offers protection for data and applications,
7. supports existing programming languages and paradigms,
8. is designed for scalability and modularity,
9. rests on a formal foundation, and
10. is compatible with industry's model of communication services.

Our abstraction and supporting software is designed to serve as an insulating layer between the programming language and the low-level communication protocols. It hides the complexity of the underlying heterogeneous infrastructure in order to provide a common framework for communication.

We have stated that our long-term goal is to facilitate the construction and smooth integration of diverse distributed multimedia applications. Our tools are not yet to the point of being able to support development of applications like the hypervideo browser. However, in this paper, we provide an abstraction and implementation design that is the first major step toward realizing this goal.

The remainder of this paper is organized as follows. In Section 3, we present *I/O abstraction*, a connection-oriented model of interprocess communication in which independent modules interact with an abstract environment. Then, in Section 4, we provide a simple example to illustrate the I/O abstraction concepts. This is followed, in Section 5, by some general comparisons to other communication models. Section 6 describes an implementation of *The Programmers' Playground*, a software library and run-time system designed to support I/O abstraction. We conclude with a summary and some directions for future work.

3 I/O Abstraction

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*. One declares direct high-level logical connections among the state components of individual modules, as opposed to directing communication within the control flow of the module. The use of declarative relationships between program states has been advocated for the visualization of concurrent programs [30]. Here, we advocate it for interprocess communication in general. Declaring high-level relationships among the state components of software modules makes implicit communication possible. Once the high-level relationships between state components are declared, if a particular state change in one module should be reflected in the state of another module, then this can be recognized by the system and the necessary communication can be handled implicitly. Thus, output is essentially a byproduct of computation, and input is handled passively, treated as a modifier (or an instigator) of computation.

This 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.

The I/O abstraction programming model has its roots in the formal I/O automaton model of Lynch and Tuttle [22]. An *I/O automaton* is a state machine with a *signature* consisting of a set of *input actions* and a set of *locally controlled actions* (divided into *output actions* and *internal* *actions*). Locally controlled actions are under the control of the automaton, while input actions may occur at any time. Automata may be composed such that when an output action of one automaton occurs, all automata having a same-named action as an input action make a state transition simultaneously. A *behavior* of an I/O automaton is a sequence of input and output actions that may occur in an *execution* of that automaton. The I/O abstraction programming model is designed to benefit from the useful characteristics of the I/O automaton model (such as compositionality properties) that are helpful in reasoning formally about distributed systems.[1]

I/O abstraction is based on three fundamental concepts: data, control, and connections. It is difficult to discuss these concepts in detail without reference to particular mechanisms for supporting them. Therefore, we present them in the context of *The Programmers' Playground*, a software library, run-time system and programming environment we have designed to support the development of distributed applications using I/O abstraction.

3.1 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 sequences) 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

[1]Relevant similarities between I/O abstraction and the I/O automaton model are noted in the discussion, but prior familiarity with the I/O automaton model is not necessary.

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. Using a UNIX-style protection mechanism, the access privileges for each published data structure might be different for the owner, a designated group, and the rest of the world. For example, worldwide read access without connect access would allow any module to read the data, but only if a (trusted) module with connect access establishes the connection on its behalf.

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).

Behaviors and specifications: A *behavior* of a module is a sequence of values held by the data items in its presentation. It is the view that the environment has of the module, and (symmetrically) is the view that the module has of its environment. A Playground module can be described in terms of a *behavioral specification* including: the data items in the presentation, the behaviors that may be exhibited by the module, and any assumptions made about the allowable behaviors of the environment. Dividing the presentation into input (write-only) data items and output (read-only) data items can simplify the task of constructing a behavioral specification and such a division can be enforced using access protection. Behavioral specifications are similar to the *schedule module* specifications in the I/O automaton model, except that I/O automaton behaviors are sequences of actions, while our behaviors are sequences of state changes at the presentation.

3.2 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 cause by the environment. This suggests a natural division of the control into two parts: *active control* and *reactive control.*[2] 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. For example, in a discrete event simulation, the active control would be the iterative computation that simulates each event. External updates of simulation parameters could affect the course of future iterations, but would not require any special activity at the time of each change. 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. For example, a data visualization module could be constructed so that each time some data element changes, the visualization is updated to reflect the change. In the above discrete event simulation, one might add reactive control to check the consistency of simulation parameters that are modified by the environment. 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 that data item is updated by the environment. As a simple example, one might associate with data item x an enqueue operation for some local queue q. With each external update to x, the new value of x would be enqueued into q for later processing by the module.

3.3 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

[2] RAPIDE[21], a rapid prototyping language for concurrent systems based on partially ordered event sets, is an example of another system that supports this distinction.

the system. Connections are established by a special Playground module, called the *connection manager*, 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 are 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.*

If we liken the data items in the presentation of a Playground module to the actions in the signature of an I/O automaton, then just as like-named actions in automaton signatures define the sharing of actions, 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. We are currently working to support various data transmission ordering requirements, such as atomic and causal ordering, in the context of I/O abstraction.

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.

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.

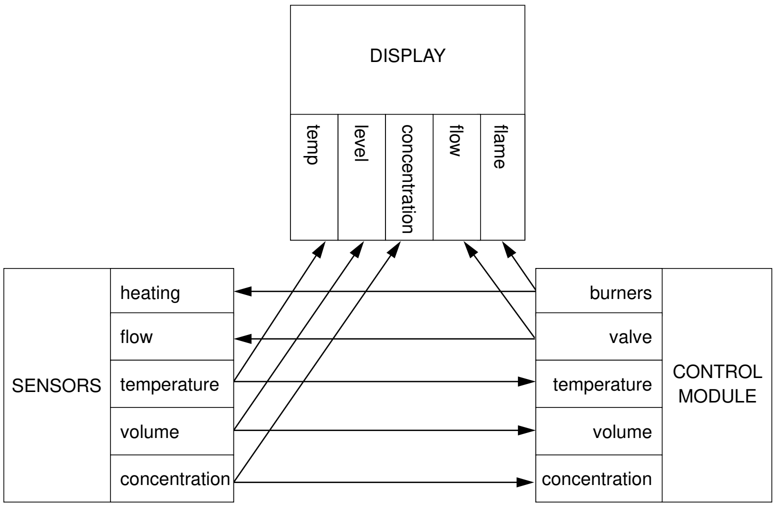

Figure 2: Configuration for a process control simulation

4 Example: Process Control Simulation

The following simple application illustrates some of the I/O abstraction concepts supported by the Programmers' Playground. It is not intended to illustrate all of the power of the model (for example, element-to-aggregate connections are not illustrated), but is meant to provide a feeling for how software modules are written using I/O abstraction. The example is a process control simulation for a factory in which maple sap is concentrated under low heat to produce maple syrup. The application consists of three modules, configured as shown in Figure 2.

The "sensors" module outputs simulated changes in readings that would be reported by sensors in the factory (liquid temperature, volume, and concentration) based on the current conditions, including the current flow of sap into the tank and whether or not the liquid is being heated. The "control" module, based on current readings, regulates the valve controlling the flow of sap into the tank and also turns the burners under the tank on and off. The "display" module is simply a graphical representation of the conditions in the factory.

Example code for the sensors and control modules are shown in Figures 3 and 4, respectively. The first line of the sensors module includes the Playground header file. It then declares its program name (for use by the connection manager) and declares five Playground data items (flame, flow, temp, level, and conc). Two ordinary procedures are defined to compute the new temperature

```
#include "PG.hh"
PGprogram("Sensors");

PGbool flame;
PGint  flow;
PGreal temp, level, conc;

real newTemp (bool flame, int flow, real level, real temp)
 { ... returns the new temperature ... }

real newLevel (bool flame, int flow, real oldlevel, real temp)
 {  ... returns the new liquid volume ...  }

main() {
  PGinitialize();

  PGpublish(flame, WRITE_ONLY, "heating");
  PGpublish(flow,  WRITE_ONLY, "flow");
  PGpublish(temp,  READ_ONLY,  "temperature");
  PGpublish(level, READ_ONLY,  "volume");
  PGpublish(conc,  READ_ONLY,  "concentration");

  real oldtemp;
  while ((conc < 4.0) || (flame == ON) || (flow > 0))
    {
      oldtemp = temp;
      temp = newTemp (flame, flow, level, temp);
      level = newLevel (flame, flow, level, oldtemp);
      if (level > 0) conc = input_volume / level;
    }
  PGterminate();
}
```

Figure 3: Sensors module in the process control application

```
#include "PG.hh"
PGprogram("control module");

PGbool flame;
PGint  valve;
PGreal temp, level, conc;

void watchTemp() {
   if ((temp >= high_temp) && (flame == ON))
      flame = OFF;
   else if ((temp < low_temp) && (level > low_level) && (flame == OFF))
      flame = ON;
}

void watchLevel() {
   if ((level <= low_level) && (flame == ON)) flame = OFF;
   valve = ...
}

main() {
  PGinitialize();

  PGpublish(flame, READ_ONLY,  "burners");
  PGpublish(valve, READ_ONLY,  "valve");
  PGpublish(temp,  WRITE_ONLY, "temperature");
  PGpublish(level, WRITE_ONLY, "volume");
  PGpublish(conc,  WRITE_ONLY, "concentration");

  PGreact(temp, watchTemp);
  PGreact(level, watchLevel);

  while (conc < target_concentration) usleep(100);

  flame = OFF; valve = 0;
  PGterminate();
}
```

Figure 4: Control module in the process control application

and level based on the current conditions. The mainline publishes the five Playground data items, specifying external names and access protection for each. The main loop performs the simulation, calling the newTemp and newLevel procedures repeatedly until the desired concentration is reached, the flow has stopped, and flame has been turned off. Each time through the loop, it computes the new concentration based on the current volume and the cumulative additions of sap to the tank. These new values are sent to other modules implicitly by the run-time system, according to the logical connections.

While the sensors module simulation uses *active* control, the control module uses primarily *reactive*. Like the sensors module, it creates and publishes some Playground data items. It defines two procedures (watchTemp and watchLevel) as reaction functions that will be executed whenever the presentation variables temp and level are updated, respectively. These procedures are responsible for adjusting the burner and valve to keep within the desired set points. The main loop in the control module simply waits for the liquid to reach the desired concentration.

An important observation is that neither module makes any reference to any other module in the system. The sensors module, for example, does not have to connect explicitly to the two other modules and send each of them updated readings when the simulation changes the values. In fact, the programmer need not know anything about how communication of data values is accomplished. Each module is written independently in terms of its data interface with its own variable names. The configuration is handled separately as in Figure 2. Communication is handled implicitly by the run-time system on the basis of the logical connections.

5 Related Work

I/O abstraction is a new approach to interprocess communication that is designed specifically to satisfy the needs of distributed multimedia applications. However, it is natural to ask why existing models fail to satisfy these needs. Figure 5 summarizes the key properties of I/O abstraction that relate to our research goals and indicates their presence in the following communication models (listed here with some example implementations): datagrams [24], streams [2, 29, 25], remote procedure call [5, 20, 35], shared memory [18], shared dataspace [1, 31], shared objects [7, 8, 14],

	I/O	message-passing			shared data			
	abstraction	datagrams	streams	RPC	memory	dataspace	objects	dataflow
PROGRAM MODULES:								
active (autonomous)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	
reactive	$\surd$	$\surd$	$\surd$	$\surd$				$\surd$
abstract environment	$\surd$		bind			open		$\surd$
CONFIGURATION:								
connection-oriented	$\surd$		$\surd$					$\surd$
user-configurable	$\surd$		bind	bind			bind	$\surd$
COMMUNICATION:								
implicit	$\surd$			$\surd$	$\surd$	$\surd$	$\surd$	n/a
direct	$\surd$	$\surd$	$\surd$	$\surd$				$\surd$
bidirectional	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	
multiway	$\surd$		bind	groups	$\surd$	$\surd$	$\surd$	$\surd$
continuous streams	$\surd$		$\surd$	non-blk				$\surd$

Figure 5: Properties of I/O abstraction and their presence in other communication models.

and dataflow [12, 33]. The rest of this section discusses the importance of these properties to our goals and explains the comparisons made in the table.

Three high-level remarks about the table are in order. First, we offer this table with the hope that it will assist the reader in making comparisons with systems not specifically addressed here. However, the table is intended only as a starting point for such comparisons, as specific systems based on these models vary. Second, since one communication model can often be used to implement another (indeed, our implementation of I/O abstraction is based on message passing), we compare the models only on the abstraction they provide directly, and not on what properties could be achieved by adding other mechanisms or abstractions on top of these models. Finally, we only consider properties that are important to our goals. We are by no means claiming that I/O abstraction subsumes all the other models, as the other models each have different properties that are not considered here.

Program Modules: Both *active* and *reactive* control are important for simplifying distributed applications. Simulations need active control, while many server applications would require reactive control. Shared memory systems tend not to have reactive control, since the communication is indirect, while dataflow systems tend to provide only reactive control. Writing program modules in terms of an *abstract environment*, besides simplifying programming, is important for decoupling programs from the communication structure in order to provide end-user configuration and integration and is useful for letting the run-time system deal with the low-level communication requirements. Modules written in terms of an abstract environment can be written independently, without knowledge about the behavior of other modules in the system. In datagram communication, modules must know the network addresses of the other modules with which they communicate. With RPC, modules must be aware of the interfaces of other modules in order to make the appropriate procedure calls and expect certain results. Furthermore, RPC requires that the environment be understood in terms of the procedural paradigm, making it more difficult to integrate rule-based or dataflow programs into an RPC-based application. In most shared data systems, programs modules must agree on shared variables. In shared dataspace systems, it is an open bulletin-board approach. Coordination languages (see discussion below) for message-passing or shared data can provide a decoupling of communication from computation by manipulation of binding tables.

Configuration: A *connection-oriented* configuration is useful for providing a dynamic communication structure where communication requirements can be declared and protection can be enforced on a connection-by-connection basis. The connection-oriented approach not only provides users with an intuitive and uniform communication model, but it also provides system designers with opportunities for analysis and optimization based on configuration information[3]. Stream-based message-passing systems and dataflow systems tend to be connection-oriented, while the others listed tend not to be. Providing *end-user configuration* is a central goal of our research. Most of the models do not directly support end-user configuration, but in many cases reconfiguration is possible through additional coordination languages that change binding tables dynamically (noted by "bind" in the table).

Coordination languages [10] 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 [16, 23, 17] is a coordination language for managing message-passing connections

[3] For example, in a Playground application configured so that some presentation entries lack logical connections, dead code elimination techniques could be used to optimize away the computation of unused values

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 [27, 28], a configuration is expressed using “module interconnection constructs” that establish procedure call bindings among modules in a distributed system. CONCERT [36] provides a uniform communication abstraction by extending several procedural programming languages to support the Hermes [32] distributed process model. PROFIT [15] 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 [13]. 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 [19] or PVM [9].

Communication: Having *implicit* communication means that the programmer need not think about when to initiate communication. Communication occurs as a byproduct of computation. In the message-passing model, communication is inherently explicit (sends and receives). For dataflow, the implicit communication category is not applicable, since dataflow modules are not autonomous. *Direct* communication is important for modules that must react to input from another module. Shared data systems use indirect communication, where a module must access a shared variable in order to learn that the value has changed. Reactive control (see above) requires direct communication. *Bidirectional* communication is important for interactive applications, where there a lot of sharing. The dataflow model does not support bidirectional communication well.

Multiway communication is important for interactive applications with multiple participants. Multiway communication is not part of the datagram model since each message is an independent event. However, multiway communication be achieved with streams, provided that there are indirect bindings for the streams and a copy mechanism exists. Some kinds of multiway communication, notably for multicast to process groups implementing distributed servers, is supported in RPC systems. Indirect multiway communication is possible in shared data systems; however, coordination among the processes is required so that all readers have an opportunity to see the data

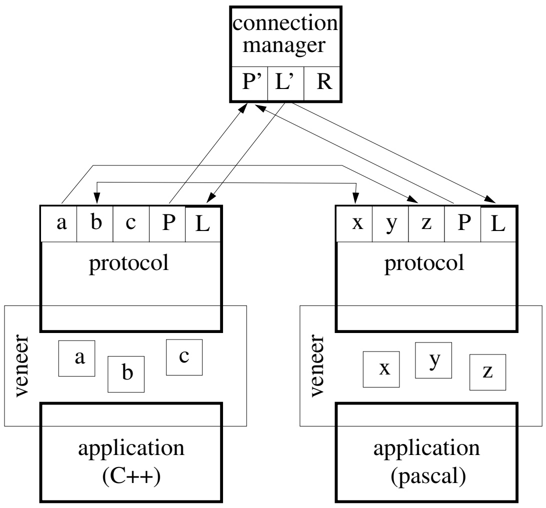

Figure 6: A Playground system

before it is overwritten. *Continuous streams* are the natural abstraction for continuous datatypes such as audio and video. Dataflow and message-passing streams support this, and non-blocking RPC's provide a (somewhat unnatural) mechanism for continuous streams. Datagram and shared data systems are not well-suited for continuous streams.

6 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 6.

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 process* 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 6 as `P` and `L`, respectively.) The connection manager also publishes a queue of connection requests (shown as `R` in the Figure 6) into which modules, such as a graphical configuration application, may enqueue connection requests using an element-to-aggregate (element stream) connection. For each connection request, the connection manager 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 updates the data structure and any necessary reactive control is handled. Note that all of this I/O happens implicitly, whenever a module updates the value of a published data structure.

Atomicity: 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.

Current status: 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 (set, queue, and array). The protocol, launched with each application, automatically sets up a "main" socket through which it provides presentation description information to the connection manager and accepts connection information from the connection manager. Updates are transmitted 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 the presentation, any pending input changes are handled so that the application sees recent and consistent data. Reactive control is not fully implemented. In our current design, reactive control resides in the same process with the active control, so each module's active control must periodically access the presentation in order to give the veneer an opportunity to handle the external updates. The protocol's concurrency control algorithms are not yet implemented, so race conditions for updates to data elements are possible. Simple connections and element-to-aggregate connections are supported. Access protection is not currently enforced.

7 Summary and Future Work

We have offered I/O abstraction as a high-level communication abstraction that can span multiple programming languages and support the communication needs of a variety of applications. Each module's computation is expressed in terms of local data structures. These may be published in a well-defined data interface through which the application interacts with an abstract environment. The environment may observe and/or modify the published data structures. Logical connections between the published data structures are configured separately from the application programs and may be changed dynamically. The application need not be concerned with explicitly sending data to and receiving data from other modules, and need not be concerned with coordinating its activities with specific processes. Access protection is provided so that changes occur only to those published data structures that are expected to change. An important benefit of I/O abstraction is the potential for integrating discrete data and continuous data within one communication model.

The connection-oriented flavor of I/O abstraction is particularly well-suited for ATM networks, where a straightforward implementation of logical connections would be to allocate the corresponding network bandwidth for data transmission. Logical connections that have arbitrary fan-out could be handled with multicast connections in the network. As a testbed for this work, we plan to use the high speed packet-switched network that is being deployed on the Washington University campus [6]. The network, called *Zeus*, is based on fast packet switching technology that has been developed at Washington University over the past several years and is designed to support port interfaces at up to 2.4 Gb/s. The Zeus network will allow us to implement multimedia applications that communicate using real-time digital video and audio, as well as discrete data.

Natural directions for further development include writing veneers to support more programming languages, implementing process migration [34], as well as extending the protection mechanism to support authentication, encryption and connection-based accounting services. Research questions remain in data transmission ordering, concurrency control, and program verification. We are working on new algorithms for causal and logically synchronous ordering of data transmission, building on [4, 11] for example, but exploiting the connection information available in the connection manager.

The concurrency control assumptions in I/O abstraction differ from those of classical concurrency control theory [3]. In a sense, we have a "continuous read" semantics that may have interesting implications for concurrency control algorithms.

We expect that useful techniques for the verification of Playground programs will be developed on the basis of commonalities between the I/O automaton model and I/O abstraction.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] M.J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1987.

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

[4] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[5] A.D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Jerome R. Cox, Jr., Mike Gaddis, and Jonathan S. Turner. Project Zeus: Design of a broadband network and its application on a university campus. *IEEE Network*, pages 20–30, March 1993.

[7] William J. Dally and Andrew A. Chien. Object-oriented concurrent programming in cst. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, September 1988.

[8] Partha Dasgupta, Richard J. LeBlanc, Jr., and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, pages 34–44, November 1991.

[9] 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.

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

[11] Kenneth J. Goldman. Highly concurrent logically synchronous multicast. *Distributed Computing*, 6(4):189–207, 1991. Earlier version in proceedings of the 3rd International Workshop on Distributed Algorithms, Nice, France, Sprinter–Verlag LNCS 392.

[12] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*, May 1991.

[13] 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.

[14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[15] 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.

[16] Jeff Kramer and Jeff Magee. The evolving philosophers problem. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[17] Jeff Kramer, Jeff Magee, and Morris Sloman. Configuring distributed systems. In *Proceedings of the 5th ACM SIGOPS European Workshop*, September 1992.

[18] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

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

[20] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[21] 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.

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

[23] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 102–117, March 1992.

[24] J. Postel. User datagram protocol. Technical Report RFC 768, USC Information Sciences Institute, August 1980.

[25] J. Postel. Transmission control protocol: Darpa internet program protocol specification. Technical Report RFC 793, September 1981.

[26] Dick Pountain. *A Tutorial Introduction to Occam Programming.* INMOS, Limited, March 1986.

[27] 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.

[28] 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.

[29] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[30] Gruia-Catalin Roman and Kenneth C. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10):25–36, October 1989.

[31] Gruia-Catalin Roman and H. Conrad Cunningham. A shared dataspace model of concurrency — language and programming implications. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 270–279, June 1989.

[32] 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.

[33] P. A. Suhler, J. Biswas, K. M. Korner, and J. C. Browne. TDFL: A task-level dataflow language. *Journal of Distributed and Parallel Programming*, 9:103–115, June 1990.

[34] Bala Swaminathan and Kenneth J. Goldman. Dynamic reconfiguration with I/O abstraction. Technical Report WUCS–93–21, Washington University in St. Louis, August 1993.

[35] J. White. A high-level framework for network-based resource sharing. In *Proceedings of the National Computer Conference*, pages 561–570, 1976.

[36] 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.