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

Authors: Kenneth J. Goldman and Michael D. Anderson

A new high-level approach to interprocess communication in heterogeneous distributed systems is introduced. This approach, called I/O Abstraction, allows one to write each functional component of a distributed system as an encapsulated program that acts upon a set of local data structures, some of which may be published for external use. The functional components are separately configured by establishing logical connections among the published data structures.

In order to illustrate this approach, we describe the The Programmers' Playground, a high-level language "veneer" and protocol designed to support I/O abstraction in heterogeneous computing environment. Support for communication among programs written in multiple programming languages, protection of data and applications, and the uniform treatment of discrete and continuous data types for multimedia applications are discussed.

... Read complete abstract on page 2.

Follow this and additional works at: http://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
http://openscholarship.wustl.edu/cse_research/595
The Programmers' Playground: I/O Abstraction for Heterogeneous Distributed Systems

Complete Abstract:

A new high-level approach to interprocess communication in heterogeneous distributed systems is introduced. This approach, called I/O Abstraction, allows one to write each functional component of a distributed system as an encapsulated program that acts upon a set of local data structures, some of which may be published for external use. The functional components are separately configured by establishing logical connections among the published data structures.

In order to illustrate this approach, we describe the The Programmers' Playground, a high-level language "veneer" and protocol designed to support I/O abstraction in heterogeneous computing environment. Support for communication among programs written in multiple programming languages, protection of data and applications, and the uniform treatment of discrete and continuous data types for multimedia applications are discussed.

This technical report is available at Washington University Open Scholarship: http://openscholarship.wustl.edu/cse_research/595
The Programmers' Playground: I/O Abstraction for Heterogeneous Distributed Systems

Kenneth J. Goldman and Michael D. Anderson

WUCS-92-32

November 1992

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* and Michael D. Anderson†
Department of Computer Science
Washington University in St. Louis
St. Louis, MO 63130
kjg@cs.wustl.edu

Abstract

A new high-level approach to interprocess communication in heterogeneous distributed systems is introduced. This approach, called I/O Abstraction, allows one to write each functional component of a distributed system as an encapsulated program that acts upon a set of local data structures, some of which may be published for external use. The functional components are separately configured by establishing logical connections among the published data structures.

In order to illustrate this approach, we describe the The Programmers’ Playground, a high-level language “ veneer” and protocol designed to support I/O abstraction in a heterogeneous computing environment. Support for communication among programs written in multiple programming languages, protection of data and applications, and the uniform treatment of discrete and continuous data types for multimedia applications are discussed.

1 Introduction

A heterogeneous distributed system is a large collection of application programs that are written in many different languages and programming paradigms and that run on top of various operating systems and architectures. The applications are interconnected by a communications infrastructure that permits the users of these applications to collaborate interactively (possibly making use of real-time audio and video), to share data and resources, and to take advantage of various electronic services. Such heterogeneous systems are not designed from scratch to work as a tightly-coupled unit. Instead, they evolve over time, with applications coming and going, making use of each others’ services and cooperating for collaborative work and data sharing.

Writing programs for this kind of “open” heterogeneous environment is not easy. The presence of multiple programming languages and operating systems is one obstacle, but enforcing the use of a single common language and operating system is an impractical solution since different programming paradigms are better suited for different problems and individuals have personal preferences. By a similar argument, enforcing the use of a single low-level communication protocol is also impossible. For example, a multimedia application may have a variety of communication needs in order to handle continuous real-time video and audio, as well as discrete data. In addition, there is

*This research was supported in part by the National Science Foundation under grant CCR-91-10029.
†Current address: Brown University, Department of Computer Science, Providence, RI 02912. This author was supported at Washington University under NSF contract CDA-91-23643.
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.

1.1 Goals

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 this paper 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 that

- simplifies writing and configuring programs that interact in a heterogeneous system,
- builds on existing languages and operating systems,
- exploits high-bandwidth communication technology,
- permits the structure of system to change over time,
- offers protection for data and applications, and
- encourages the use of fundamental software engineering techniques.

Such an abstraction and supporting software would serve as an insulating layer between the programming language and the low-level communication protocols, and would provide a common uniform communication mechanism for diverse collections of both permanent and transient applications.

1.2 Related work

Traditionally, abstractions for interprocess communication have been provided by operating systems. For example, UNIX [2] provides the socket that supports a byte stream abstraction for communication among UNIX processes and is accessible through system calls from high level languages such as C [9]. More recently, the Clouds distributed operating system [6] provides a remote procedure call mechanism between objects in a system-wide virtual address space. In Clouds, binding of remote procedure calls is mediated by a name server. The resulting abstraction is useful for writing distributed applications in an object-oriented paradigm. A simpler example of operating system support for communication among applications is the "publish/subscribe" mechanism of the Macintosh operating system [4] that provides one-way asynchronous communication through files. The publisher writes the data into a special file, called an edition. Any number of subscribers may notify the edition manager that they want to subscribe to the data. Each time the publisher writes out a new edition, the subscribers are notified by the edition manager so that they may read in the data from the file. The publish/subscribe mechanism has a certain appeal since the publisher need not know about the subscribers and the communication pattern can be configured independently.

Some high level languages directly provide constructs that permit programs written in those languages to communicate among themselves. With these languages, distributed systems can be constructed in a way that insulates the programmer from the operating system. For example, Ada [18] provides the rendezvous and Argus [14] provides a remote procedure call mechanism. Some object-oriented languages allow entire objects, complete with their operations, to be passed
among processes. Others, such as Hermes [22], provide a facility whereby values may be passed across ports that may be connected from one module to another.

Recently, however, there has been growing interest in coordination languages [7] that provide a view of communication that is independent of particular programming languages or operating systems. Coordination languages are meant to simplify the task of setting up communication among programs written in different programming languages and running on different operating systems by separating computation from communication. This separation 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. Knowing the structure independently of the behavior of the components is also helpful in reasoning about the behavior of the system as a whole.\footnote{A more extensive discussion on the benefits of the separation of structure and function is given elsewhere [8].} Here, we are specifically concerned with how such a separation could be helpful in solving the software engineering problems associated with heterogeneous distributed systems. If the communication model is independent of the computation model, then we can offer programmers a uniform communication abstraction, regardless of their choice of programming language, programming paradigm, or operating system.

Several programming languages have been constructed that embrace the approach of separate computation and coordination languages. For example, Linda [1] is a coordination language that allows multiple processes, possibly written in different computation languages, to communicate through a shared tuple space. Linda provides out and in constructs that allow processes to insert and remove elements of a shared tuple space in order to communicate and coordinate their activities. This mechanism permits one to write a program that communicates with an abstract environment, not with particular other modules that must be known to that program. That is, since each program module interacts with its environment through the tuple space, it need not be concerned with establishing communication with particular other modules; each module finds the information it needs without regard to which other module placed it there. Although this global unstructured approach to communication frees the programmer from thinking in terms of senders and receivers, behind the scenes there must be some advance agreement among the programmers as to how the data will be named in the tuple space so that the appropriate processes may find it. Linda’s global tuple space also provides little help in specifying and understanding the logical structure of the system. In addition, protection of data in the tuple space has not been addressed. If a process can name a tuple, then it can see and remove the tuple. Therefore, Linda is more suitable for tightly coupled concurrent programs than it is for an open heterogeneous system in which processes enter and leave unpredictably.

A more typical approach to coordination languages, and the approach we adopt here, is to provide a configuration mechanism for establishing explicit relationships among program modules. For example, Darwin [10, 17, 12], a generalization of Conic [11, 12], is a configuration language that allows one to manage message-passing connections between the ports of various processes in a dynamic system. Each process is expressed in a computation language that is largely independent of the configuration language, except that it provides port declarations that comprise its connection interface for use by Darwin. Polylith [19, 20] is another system that takes the configuration approach. While the Darwin view of a configuration is expressed in terms of connections between communication ports, the Polylith view of a configuration is expressed in terms of a set of procedure call bindings. Polylith provides a set of “module interconnection constructs” for establishing procedure call bindings among modules in a distributed system. CONCERT [23] provides a coordination language in which the Hermes [22] distributed process model is embedded in several procedural programming languages by providing extensions to each language.
A high level coordination language might be implemented directly on top of each individual operating system and programming language. Alternatively, for ease of portability, one might implement a coordination language on top of a uniform set of system level communication constructs for heterogeneous distributed systems. For example, the Mercury system [13] provides a remote procedure call facility that spans multiple programming languages and operating systems. Each supported programming language is extended with a thin layer or "veneer" between the application program and the operating system. If one were to implement a coordination language on top of Mercury, then supporting that coordination language on a given platform would require only that Mercury be implemented on that platform. This would save rethinking the implementation of the coordination language for each platform. Of course, this implementation approach is only reasonable if the semantics of the coordination language is compatible with the semantics of the lower level communication mechanism. Furthermore, directly implementing the coordination language on each platform may result in the ability to take better advantage of the features offered by that platform for improved efficiency.

The work described here builds upon our previous exploration of the separation of structure and function in the specification and design of distributed systems [8]. In that work, an extended version of the I/O automaton model [15, 16] is used as the basis of a simulation system for describing automaton types, configuring them into systems and studying their executions. An important feature of the I/O automaton model is that each action of a system component is either an input action, an output action, or an internal action. Each output action is under the control of at most one component and input actions are always enabled (cannot be blocked). This allows one to construct system components in isolation in terms of a well-defined interface and to construct proofs of correctness in terms of local properties of executions that continue to hold when modules are composed into larger systems. Functional components are constructed separately from the system configurations in which they reside. In other words, one builds functional components and "plugs them together" by constructing a configuration that describes logical relationships among the components. The configuration is used to determine relationships based on the sharing of actions.

In this paper, our approach to configuring a distributed system differs from our previous work in that the logical relationships between different modules are not described in terms of shared actions, but instead are described as direct relationships between the state components of the modules. However, one could view our logical relationships between state components in terms of a canonical shared action system in which the set of actions in the external interface of each module directly corresponds to the set of state components that have logical relationships to the state components of other modules. Thus, an output action would be a report of the update of some local (but externally relevant) state component(s), an input action would represents the corresponding update in some other program, and an internal action would correspond to an update of a state component that is not related to the state of any other module. In order to model the communication that takes place when state components are changed, each output action reporting a state change would be shared with an automaton representing a "message system." The message system automaton would later produce an output action that would be an input to the module that must observe the change. The specification of the message system automaton, then, could be used to capture safety and liveness conditions related to the ordering and eventual delivery of each state change.
1.3 Rethinking program I/O

In this paper, we describe a new way of viewing program I/O that is significantly different from the approaches taken by previous coordination languages and communication constructs for heterogeneous systems. This view of program I/O encourages a new way of thinking about the design and implementation of distributed systems. Before describing our approach, however, we first explain what we mean by program I/O and discuss our motivation for taking a fresh look at this topic.

We define program I/O as the transfer of information between an encapsulated software component and its environment. This definition focuses on the point of information transfer from internal program state to the environment (output) or from the environment to the internal program state (input). Under this definition, a remote procedure call by itself is not program I/O, but it can be a means to program I/O. That is, a remote procedure call may perform program I/O (by passing some parameter value, for example), but it is the actual transfer of the parameter value from one module to another (not the RPC itself) that constitutes the program I/O. For program I/O to take place, we do not require a completed exchange. If a module makes some information externally available, that is considered output, whether or not another module makes use of the information.

The traditional abstraction for program I/O, the sequential byte stream, was designed to support single-threaded computations and was modeled after the communication technology available at the time. More recently, as higher-level communication constructs became important for the construction of large distributed systems, the remote procedure call was developed as an abstraction on top of the basic message-passing concept that fits well with existing sequential programming languages. However, there are several recent developments in both software design and communication technology that appear to justify taking a fresh look at program I/O.

First, if we accept that there is a trend toward the separation of computation from communication in the design of distributed systems, then perhaps it is worthwhile to think about the underlying communication model that could best support such a separation. As mentioned earlier, there already exist systems that provide separate configuration mechanisms based on established IPC mechanisms, such as streams and remote procedure calls. The approach taken in designing those systems was to adopt an existing I/O mechanism and then concentrate on the problem of separating the configuration from the computation. Our approach, however, is to take a wider view of the problem and to completely rethink the way a program's I/O interface is structured in order to find the best way to achieve a clean separation of structure and function.

Second, today's programming languages vary in the paradigms they support, and although the remote procedure call approach to interprocess communication fits well with the procedural paradigm, it is less clear how to integrate that approach with rule-based programs or dataflow programs. Thus, it behooves us to find some common denominator through which all programs, regardless of programming paradigm, may communicate.

Third, there has been a fundamental shift in the way users interact with computer systems today. Traditional user interfaces, written for character display terminals, were based on command sequences or structured sequences of questions or menus. Today's user interfaces, written for graphics workstations, are much less sequential. They present the user with a much broader view of the state of an application and provide simultaneously a multitude of options through direct manipulation. We argue that this fundamental change in our model of user interaction should be reflected in the structure of our software. If we view users as functional components that produce system input and consume system output, then much of today's software uses two distinct forms of communication: a high-bandwidth form for user/system communication and a sequential form for communication among software components. Are both forms really necessary? Perhaps we can extend the high-bandwidth view of program I/O (that has been so successful at the user level) to
the interactions between internal software components as well.

A fourth reason for taking a fresh look at program I/O is that advances in network technology have made it more reasonable to consider transporting large amounts of data from one module to another. In the past, programmers writing distributed applications needed to maintain very careful control on the amount of communication, but today we may be reaching the point where the advantages of providing a high level of abstraction to the programmer outweigh the disadvantages of asking the programmer to relinquish low level control over data transmission. In addition, these high speed networks are capable of supporting multimedia applications that involve not only discrete data types, but also continuous data types such as audio and video. One could construct a system with two separate IPC mechanisms, one for discrete data (such as RPC) and one for continuous data (such as a low-level stream abstraction), but it would seem better to have a unified abstraction that could accommodate both kinds of data.

1.4 A non-traditional approach

The approach to program I/O that we advocate in this paper is non-traditional in two respects. First, the approach is based on high-level relationships between the states of communicating modules, and second, the approach is based on a passive view of program I/O in which communication occurs implicitly as a byproduct of computation.

The traditional low-level view of program I/O dates back to the use of punched cards and paper tape. A module wishing to make some of its state information available outside of itself initiates some “output activity” in order to sequentialize that information into a sequence of bytes. A second module, wishing to make use of external information, initiates an “input activity” that captures the sequential information and makes appropriate changes in the state of that module. For example, in the C programming language [9], the printf routine is an output activity that sequentializes information about the program state into a stream of characters that may be read by another module using the scanf routine in order to capture the sequential information in the program’s state.

We prefer a programming model in which one can declare direct high-level logical connections between the state components of individual modules. The use of such declarative relationships between program states has been advocated for the visualization of concurrent programs [21]. Here, we advocate their use for interprocess communication in general. The low-level view of program I/O is a suitable abstraction of a communication wire, and communication in a distributed system is ultimately reduced to bits on a wire, but the I/O abstraction that is most useful to programmers is not necessarily the one that conforms to the physical reality of communication. We would rather not force the programmer to sequentialize state information on output and then reconstruct it on input. Instead, we would like to establish logical relationships among the state components of modules in the system. Another important advantage of our approach is that it addresses the need for a common denominator through which all programs, regardless of the programming paradigm, can communicate. Aside from the most strict of functional languages, all programming languages support some notion of a state components or variables. Thus, where it would be difficult for a procedural program and a rule-based program to communicate through a remote procedure call, it would seem perfectly natural to establish a logical relationship between a variable in the procedural program and some component in the state of the rule-based program.

We also claim that program I/O should be carried out in a passive way. In the traditional active view, I/O is accomplished by explicit requests that are made as part of the computation. The program, in the course of its computation, explicitly requests that some data be sent as output or received as input. We take the position that the programmer should be concerned only with the
explicit details of the computation and that I/O should be implicit, hidden from view. In a sense, output should be handled as a byproduct of computation and input should be treated as a modifier (or sometimes an instigator) of computation. That is, based on the logical relationships established between the state components of various programs, program I/O should happen "under the covers" as needed. In this way, the application programmer can construct each module in terms of local properties without being concerned with specifying when input and output activities should occur. This approach is particularly useful for multi-threaded programs, dataflow programs or rule-based programs, where adding input or output statements at appropriate points may be very difficult, since the control flow of the program is much less apparent in those programming paradigms. In addition, the ability to concentrate on the computation and separately define logical relationships between the state components of programs is in keeping with our desire to achieve a separation of structure and function.

1.5 I/O abstraction

Our non-traditional approach to program I/O, motivated by the objectives outlined in Section 1.1, is embodied in a new concept we call I/O abstraction. Briefly, I/O abstraction is the view that each module in a system presents to its environment a set of data structures that may be externally observed and/or manipulated. This set of data structures forms the external interface (presentation) of the module. Each module is written independently and modules are then configured by establishing logical connections between the data structures in their presentations. The connections are required to respect protection rules set up by each module for its own data structures. Therefore, the logical view is direct logical connections between the states of different programs. All I/O occurs implicitly ("under the covers") as the data structures are updated.

I/O abstraction is supported by a programming environment called The Programmers' Playground that is currently under development at Washington University in St. Louis. The Programmers' Playground takes its name from a metaphorical description of the I/O abstraction concept. The Playground consists of a collection of children (modules), each of which has a collection of toys (data structures). Children may come and go at will from the Playground, and may acquire or lose toys as they play. All interaction among the children takes place through the toys. That is, as one child manipulates a toy, other children may observe the change and react by manipulating that toy or other toys. Each child may establish rules (access rights) for the use of his/her own toys: who may use them and how they may be used. These rules are enforced by a Playground supervisor (connection manager).

1.6 Comparisons with other IPC mechanisms

Perhaps the closest programming model to I/O abstraction is the shared memory or shared object model. A fundamental difference between I/O abstraction and the shared object model is the way in which one views the program interface. In the shared object model, a program communicates with the outside world by invoking operations on external shared objects. The interface of a program using shared objects is essentially the set of procedure calls one might invoke on the shared objects. In this way, the relationship between communicating applications is defined indirectly in terms of the objects that they share and the actions they perform on those objects. In I/O abstraction, a program communicates with the outside world by modifying its own local variables that are made externally visible. The interface of a program using I/O abstraction is the presentation of its local variables. Consequently, the relationship between communicating applications is defined directly in terms of a logical connection between their local variables. Using I/O abstraction, a
programmer need not be concerned with external names for shared variables. Instead, one writes a
program with its presentation and separately establishes relationships between applications that are
to communicate. Another distinction between shared memory systems and I/O abstraction is that
one must actively read a shared variable in order to find out that its value has changed. However,
I/O abstraction supports the notion of a state change in one application causing computation to
take place in another.

I/O abstraction differs from the message-passing paradigm and the RPC paradigm in the degree
of separation of computation from communication. As described earlier, programs written using
I/O abstraction do not explicitly perform input or output as part of their computation. Instead,
a program is written in terms of computation on local data structures and I/O happens implicitly
on the basis of logical connections between the states of different programs. In this way, one need
not be concerned about when to wait for input and when to perform output in the course of
computation.

I/O abstraction is intended as a methodology for constructing applications programs so that
communication concerns are separated from computational concerns. It is not intended as a low-
level implementation technique. As such, one might imagine implementing part or all of a Play-
ground system on top of a shared memory or shared object system, or on top of an RPC-based
system. For example, I/O abstraction provides the programmer with an illusion that each time a
state change is made, it is reflected in all other modules to whom logical connections have been
made. One could choose to support this illusion by actually transmitting the values each time a
change is made, or by simply invalidating the remote copies and sending the new values only on
demand as in a shared memory system with caching. However, such an implementation may not
extend well to support continuous data types.

In the remainder of this paper, we describe how I/O abstraction is realized in The Programmers’
Playground. In Section 2, we describe the application programmers’ view of the Playground, both
from a local and global perspective. Then, in Section 3, we describe an implementation of the
Playground. In Section 4, we review The Programmers’ Playground in terms of its design goals.
We conclude, in Section 5, with the current implementation status and our plans for future work.

2 The Playground Programming Model

In this section, we describe the view of The Programmers’ Playground that is seen by the applica-
tions programmer. We begin with the local application view and then discuss the global view of
the system. In Section 3, we describe how this programming model is supported by the Playground
system.

2.1 The local view

Using I/O abstraction in The Programmers’ Playground does not require learning a new pro-
gramming language. Playground programs are written in familiar programming languages, such
as C++, where the choice of language used for a given application depends on the programmer’s
preference and the particular needs of that application. Each Playground program has a set of
local data structures, some of which may be published for external use. The following information
is associated with each published data structure:

- type information, supplied automatically,
• documentation information, supplied by the programmer to describe the published data structures, and

• access restrictions, supplied by the programmer to define how (and by whom) the data structure may be used.

In keeping with our high-level approach to program I/O, no explicit interprocess communication takes place in a Playground computation. There are no message sends and receives, and there are no remote procedure calls. Each program’s computation is expressed in terms of its local data structures, some of which may be published. The published data structures form a well-defined interface with an an abstract environment that is allowed to observe and modify those data structures. Logical connections between the published data structures are configured separately, not as part of program definitions. Therefore, the programs themselves need not be concerned with the particulars of the environment. As far as each individual program is concerned, it is acting on a set of local data structures, some of which may change “miraculously” during the computation.

Our division of the computational components from the communication structure provides considerable flexibility because the structure of the environment is hidden from the application. For example, the environment of an application might be a collection of other applications, a single graphical user interface, or a debugger that allows the user to act on the local state of the computation by direct manipulation of graphical objects, but the application is unable to tell the difference.

A module’s set of published data structures, known as its presentation, may change over time. The local data structures declared by a Playground application are constructed using a library of abstract data types provided as part of the Playground software for each programming language. In this way, programmers have a uniform view of data that spans different programming languages so that data can be shared easily across languages. The data types available to Playground programmers include base types (integer, real, character, etc.) and aggregates (record, set, multiset, array, sequence, etc.). Aggregates may be nested to arbitrary depth. The type information automatically supplied for each published data structure is expressed in terms of these base types and aggregates, and it is used at run-time by the Playground system in order to verify type compatibility when relationships between published data structures are established.

The documentation information associated with a published variable may be simply a descriptive name, or it may be a detailed specification of how the variable is used and its relationship to other variables of that module. The documentation information is an important part of the “open” systems approach taken by Playground, since it permits us to look at a module’s interface and understand how to make use of it.

Protection information is used to restrict the use of published data structures by other programs. This is useful not only for protecting against unauthorized use of sensitive data, but is also important for defending programs against unexpected changes to their data structures. Access privileges include read, write, and connect. Read access allows a module to observe the value of the data structure, write access allows a module to change the value of the data structure, and connect access allows a module (possibly a third party) to relate the data structure to a data structure of some other module. We will explain connections further in Section 2.2.

We assume an operating system in which each module is, directly or indirectly, created by some user (the owner) that may belong to some set of protection groups. Using a UNIX-style protection mechanism, we allow the access privileges for each published data structure to be specified separately for the owner, a group, and the rest of the world. For example, the owner might be given all four access privileges to a data structure, while the group is given only read and connect access, and the world is given only read access. Giving read access without connect access to the world
would mean that anyone could observe the data, but only if the owner or someone in the group established the connection for them. In all cases, the module publishing a data structure is able to read and write that data, whether or not the owner is granted these privileges.

As we mentioned above, no explicit interprocess communication takes place in a Playground computation. During execution, each Playground module acts upon its local data structures with the knowledge that the values of some of the published variables may be modified by its environment. Output is handled completely implicitly. Whenever a published data structure is changed, the new value becomes available to those modules having a logical connection to it. Depending on the type of application, a program may or may not need to be explicitly informed when one of its published data structures is changed by the environment. Therefore, three different input paradigms are supported in the current Playground design. These are passive observation, call-back functions, and update queues.

Passive observation is the most elegant of the input paradigms because it is completely implicit; external changes to a published data structure are not handled explicitly by the program but instead simply modify the future course of the computation. We expect passive observation to be most useful for iterative computations, such as a scheduler in which a change to the task queue would not need to be handled explicitly but would simply result in a modified behavior of the program.

Some applications, however, may need to react specifically to each input change. For this we provide two high-level mechanisms: call-back functions and update queues. A call-back function is a procedure that is associated with a particular data structure and is called whenever that data structure is modified by the environment. This provides support for an interrupt-driven style of programming. Alternatively, update queues provide an iteration capability for handling externally changed data structures. The programmer may create any number of update queues and assign each data structure to a particular queue. Whenever a data structure is externally modified, a reference to it is placed into the corresponding update queue. The update queues are used to iterate through the changed objects and handle the changes.

In some cases, data structures may be updated continually by the environment. For example, a data structure containing physical measurements for a process control application may have its values updated very rapidly. Some applications may need to be informed about each change to an object, even if the application may occasionally get behind in processing. Other applications may need only the most recent value and should not waste time on stale values. The Playground design supports each of these alternatives for both call-back functions and update queues.

2.2 The global view

A global view of a Playground system reveals multiple computational components (playground modules), each with a well-defined external interface expressed in terms of published data structures. Logical connections between published data structures determine the pattern of communication among modules. Each connection must relate data structures with compatible types and must obey the access restrictions published with those data structures.

The simplest kind of connection is between two data structures of the same type. A connection may be uni- or bi-directional, and unlimited fan-out and fan-in is permitted. For example, a bidirectional connection might useful for interactive or collaborative work, while a unidirectional connection with high fanout would be appropriate for connecting a video or audio source to multiple viewing or listening stations.

The current semantics of a Playground connection is asynchronous communication on update, although we are planning to impose data transmission ordering restrictions later. A connection from variable z in module A to variable z in module B means that whenever A updates the value of z, that
change is reflected (some time later) in variable \( z \) of module \( B \). Module \( A \) does not explicitly send the value to \( B \), and module \( B \) does not explicitly ask for the value. The communication happens implicitly as the result of the logical connection. Note that the same logical connection paradigm is used for both discrete and continuous data. For example, a connection between a video source and a video input and a connection between two integer variables are treated uniformly at the application level. The differences in the physical communication requirements of the two connections would be handled “under the covers” based on the data type information.

A connection between data structures of different types is allowed when one is an aggregate data type and the other’s type matches the element type of the aggregate. For example, a client/server application could be constructed by having the server publish a data structure of type \( \text{set}(T) \) and having each client publish a data structure of type \( T \). If a bidirectional connection is created between each client’s type \( T \) data structure and the server’s \( \text{set}(T) \) 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 data structure. As another example, a one-way connection from a data structure of type \( T \) to a data structure of type \( \text{sequence}(T) \) might be used for a producer/consumer application.

3 A Playground Implementation

We now discuss how I/O abstraction is supported in the design of the Playground system. Since one of our primary goals is to support heterogeneous distributed computing, The Programmers’ Playground is implemented not as a new programming language but instead as a layer (veneer) between each supported programming language and a protocol that runs concurrently with the application and interacts with the operating system in order to exchange data with other Playground modules on behalf of the application. The protocol for each application also interacts with a connection manager that is used to create logical connections among the data structures of different modules. In the Playground implementation, these logical connections are known as links. A logical overview of a Playground system is shown in Figure 1.

3.1 The local view

A Playground module consists of three parts: an application, a veneer, and a protocol. The application is written using the programming model described in Section 2.1 and has some set of data structures published as its presentation. These data structures are held in the veneer. All program I/O occurs in terms of these data structures and is handled by the protocol.

All of the Playground data types (base types and aggregates) are implemented in the veneer, and each published data structure is kept in the veneer so that it is accessible to both the application and the protocol. In addition, the veneer maintains locking information for concurrency control, as well as the documentation and protection information published with each data structure. Data structures are represented in the veneer using a hierarchical naming structure that permits different portions of a data structure to be updated (and locked) independently. Call-back functions and update queues are also registered in the veneer.

The protocol interacts with the connection manager in order to make published variables known to the outside and in order to know about connections established between its published data structures and those of other modules. In addition, the protocol interacts with the operating system to send the values of changed data structures and receive new values for data structures changed by other modules. These values are transmitted using a canonical representation that is independent of the particular programming language. The protocol also handles the exchange of locks on data structures between the veneers of different modules.
Figure 1: A Playground system

In a typical scenario, the application creates a set of data structures using the data types defined in the veneer and it publishes certain of these data structures. The set of published data structures is presented, by the protocol, to the connection manager which produces link information that is made available to the protocol. Whenever the application makes a change to a published data structure, the veneer first obtains a lock on the data structure and makes the change. Then the protocol is informed of the change by the veneer and forwards the new value to all other modules to whom an outgoing connection has been established from that data structure. This may involve sending the value of the entire data structure or only the updated portion. Upon receipt of a new value for a data structure, the protocol makes the appropriate change and invokes any call-back functions or update queue operations as required. All of this I/O happens implicitly as the result of an assignment to the published data structure. In fact, in our C++ implementation of Playground, we overload the assignment operator for the Playground data types in order to completely hide the I/O activity from the programmer.

Locks are used to prevent two applications from concurrently changing the data structures at the endpoints of a single logical connection. However, this does 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 \texttt{begin\_atomic\_step(obj\_list)} and \texttt{end\_atomic\_step()} provided by the veneer for encapsulating a set of changes as an atomic step. The \texttt{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. If atomic steps are nested, or if additional locks are acquired within atomic steps, there is a potential for deadlock. Thus, we encourage the use of simple atomic steps in which every object accessed within the atomic step is included in the object list. When possible, we “piggyback” lock transmission with data transmission. In order to minimize round trip lock requests, we use an adaptive scheme in which a lock either gravitates toward a frequent writer or is piggybacked with each value transmission in
the case of alternating updates by two communicating writers.

3.2 The global view

The global view of a Playground system is the set of modules and the connections between them. The connections are established by the connection manager, which is implemented as just another Playground application. The presentation of each playground module consists of the data structures published by the application, plus an externally readable data structure $P$ that holds a description of the application's presentation and an externally writable data structure $L$ that contains link information for that module. When the application is launched, $P$ and $L$ are automatically linked to the connection manager so that the connection manager is aware of each change to the application's presentation and so that the protocol for that application is aware of the logical connections established to that module. We emphasize $P$ and $L$ are used only by the protocol and are hidden from the application.

The connection manager publishes an externally writable set of presentation descriptions $P'$ that is linked to each module's data structure $P$ with an element to aggregate connection. Thus, $P'$ contains a set of presentation descriptions, one for each module in the system. The connection manager also publishes a set $L'$ of link sets, one link set for each module in the system. For a given module $m$, the element of $L'$ corresponding to $m$ contains the current connectivity (set of links) for $m$’s published data structures. This information is made available to $m$’s protocol by a connection automatically established between that element of $L'$ and $m$’s data structure $L$.

For establishing logical connections between the presentations of various applications, the connection manager publishes a set of connection requests $R$ which may be updated externally by any module in the system, most likely by a front-end application for the connection manager. For each connection request placed in this set, 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 link information $L'$. This change is reflected in the $L$ data structures of the protocols corresponding to the endpoints of the connection. In this way, each protocol is aware of each logical connection in which it is involved. Note that the connection manager is not a communication bottleneck. The connection manager simply sets up the connections that are then handled individually by the protocols associated with the connected data structures. It is also interesting to note that the connection management system itself uses the I/O abstraction mechanism to establish links in the system.

4 Discussion

We have claimed that I/O abstraction, as embodied in the Programmers’ Playground, provides a high-level abstraction that can span multiple programming languages and support, in a unified way, the communication needs of a variety of applications. In this section, we discuss I/O abstraction and the Programmers’ Playground in terms of the stated goals of this research.

I/O abstraction simplifies writing and configuring programs that interact in a heterogeneous computing environment by allowing each program’s computation to be expressed in terms of local variables that are published for others to observe and modify. This provides a common uniform mechanism for communication among diverse applications written in languages that support different programming paradigms. Each application can be constructed in isolation, with a well defined external interface. The applications programmer 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. Instead, the view is simply the local computation and an environment. Access protection is provided so that changes occur only to those published data structures that are expected to change. At this point, it is too early to tell which of the input paradigms supported by Playground will be most useful, or whether some additional alternatives might be needed. Only experience with many different kinds of applications will give us that answer. However, we feel that the more I/O can be separated from the computation, the easier it will be to construct distributed applications in a heterogeneous environment.

The Programmers' Playground builds upon existing languages and operating systems. Programmers write applications in a familiar programming language that is extended by the Playground veneer. The Playground protocol is constructed to use the lower-level communication facilities already provided by the operating system.

I/O abstraction exploits high speed network technology in two ways. First, it recognizes the fact that communication is becoming much less expensive, and so the programmer no longer needs such tight low-level control on the exchange of data between modules. This low-level control is relinquished in favor of a high-level communication abstraction that makes use of "behind the scenes" data transmission. Second, it takes advantage of high-speed networks by accommodating both discrete and continuous data types within the same high-level configuration mechanism. Although we have not yet done so, our plans are to implement continuous data types in the veneer and relate them with the usual Playground connection mechanism. In this way, users or programmers will be able to establish connections for continuous data with the same mechanisms they use for discrete data. This uniform view will be particularly useful for multimedia applications that use both discrete and continuous data.

The Programmers' Playground permits the structure of the system to change over time. Applications may come and go, and logical connections between data structures may change dynamically. In this "open" programming environment, the access privileges that are an integral part of the Playground design offer protection for data and applications. In addition, it is easy to see how authentication and encryption services could be integrated with the connection manager and protocols.

Finally, I/O abstraction encourages the use of proven software engineering techniques. Since each Playground module is written as an encapsulated unit, it is possible to reason about each module locally in terms of its well-defined interface, and it is possible to replace one module for another in the system configuration. Furthermore, I/O abstraction makes the state of a computation externally available and hides the structure of the environment from the application. Thus, it is possible to write a user interface for an application in a way that is completely decoupled from the application itself. Like other coordination languages, Playground provides a certain level of portability. Applications may be run on any architecture for which a compiler exists for that programming language and on any operating system on which the Playground protocol has been implemented.

From a software engineering point of view, a potential danger of I/O abstraction is exposing the representation of higher level abstract data types that are constructed from those provided by the Playground veneer. Access to instances of individual Playground data types are constrained by the standard set of operations provided for that type, but if we write a higher level data type whose representation is built from a collection of playground types, and if we publish that representation with write access, then it is possible for the environment to violate some consistency requirement on the high level data type by directly manipulating individual components of the representation through the operations provided for those lower level components. Sharing implementations of high level abstract data types among different programs solves the problem only when all of the programs are written in the same language. Alternatively, one might share the specification of
the high level data type and trust each application to access the representation according to that specification, but enforcement would be difficult. One way to ensure consistency would be to keep the sensitive data structure read-only, and publish a writable set of update requests. Thus, in a way that is very reminiscent of the remote procedure call, update requests could be inserted into that set by other modules and then processed locally by a call-back function in order to update the sensitive data structure. In any case, we would advocate a defensive programming style in which no update to an externally modifiable data structure could violate a consistency requirement. In a sense, this defensive style is analogous to the usual requirement of being prepared for arbitrary input. In practice, we are likely to see a combination of these methods.

5 Future Work

A prototype implementation of The Programmers' Playground is under development. The veneer is written for C++ and currently supports limited set of abstract data types. The protocol is implemented in C++ on top of UNIX using sockets. Continuous data types are not yet supported.

In the very near future, we are looking to expand the veneer to support our full range of data types, including continuous data. Following this, we plan to implement a number of different kinds of applications in order to experiment with various input paradigms. Potential classes of applications include collaborative work, multimedia applications, pipelined processing including image processing and interactive program visualization client/server applications including persistent data applications, and interaction with ongoing computations such as network monitoring and process control. We also plan to implement veneers for several high-level languages that support different programming paradigms.

As a testbed for our multimedia applications, we plan to use the high speed packet-switched network that is being deployed on the Washington University campus [5]. The network 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. Using the Playground, we plan to construct interactive multimedia applications that involve the use of discrete data types as well as real-time audio and video. We expect that the uniform treatment of logical connections for these data types will be helpful in designing these applications.

The completely asynchronous transmission-on-update semantics currently supported by Playground is inadequate for many applications. We are planning to investigate algorithms for achieving reliable causal ordering among data transmissions. Such algorithms may be similar in flavor to the atomic and causal broadcast algorithms of the ISIS system [3], but we expect to be able to take advantage of the additional information about the communication pattern that is available in the connection manager. Fault tolerance will be an important consideration in this work.

As Playground systems grow large, we will need to manage the large volume of presentation information available for Playground modules. One possibility we are considering is a distributed hierarchical connection manager and a distributed "Yellow Pages" for locating modules of interest.

The primary goal of I/O abstraction is to facilitate writing applications that interact in a heterogeneous computing environment by separating communication structure from computational concerns. The better we can achieve this separation, the closer we will be to a computing environment in which diversity of languages, programming paradigms, operating systems and architectures ceases to be a barrier to cooperation.
References


