AN ENVIRONMENT FOR PROTOTYPING DISTRIBUTED APPLICATIONS

James M. Purtilo† Pankaj Jalote

Department of Computer Science and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

ABSTRACT
Designing a distributed application is an extremely complex task. Proper facilities for prototyping distributed applications can be extremely useful in evaluating a design, and also in understanding the effect of different parameters on the performance of an application. We describe an environment for prototyping distributed applications, that supports different communication primitives with specified delays, and provides primitives to aid debugging and evaluation. The environment also supports heterogeneous computation in which processes can execute on different hardware. Different source languages can be used for coding different modules of the processes. The system has a centralized control and monitoring facility which is based on the Suntools window system.

† Research supported by Office of Naval Research contract N00014-87-K-0307, and by the University of Maryland Institute for Advanced Computer Studies.

INTRODUCTION
A distributed system is one in which a number of loosely coupled computing elements are interconnected by a communication network. Processes communicate by message passing, and no shared memory is available. Over the years, distributed systems have proliferated in the form of both local and wide area networks. However, to take advantage of these systems, we must also make facilities for writing distributed programs easily available to developers.

The complexity of designing a distributed application arises for a number of reasons. The area is relatively new, and there is little experience in designing such applications. Distributed systems inherently display non-deterministic behavior due to the varying speed of execution of processes, making verification and validation of distributed software extremely difficult. Evaluating such systems is also much more complex as compared to sequential systems, since new issues like blocking time, deadlock, and communication overhead become important. A number of different communication paradigms are available for interprocess communication, for example, remote procedure call [Nels81], asynchronous message passing, and CSP type synchronous communication [Hoar78], and the decision about which communication paradigm will result in best performance depends on both the application and the distributed environment. Complexity is further increased by the many possible configurations of processes on processors.

All this means that the current knowledge about distributed systems is insufficient for properly designing and evaluating a distributed application. Prototyping could be a useful approach that can help in designing such an application. However, little work has been done in providing environments that will allow rapid prototyping of distributed applications, and also provide means to study the effects on performance of different choices. The available systems and languages are generally inflexible and do not permit experimentation with different communication primitives (a language supports its own favorite communication paradigm), or different configurations. Clearly, an environment that permits flexible, rapid prototyping and evaluation of distributed applications would be extremely useful for designing distributed applications, and would enhance our understanding of the effects of different factors on such applications.

In this paper we describe an environment for prototyping distributed applications. The environment supports different communication primitives, and has facilities to aid debugging and evaluation. The environment also supports heterogeneous computation in which processes can execute on different hardware, and different source languages can be used for coding different modules of the processes. It has a centralized control and monitoring facility that is based on the Suntools window system.

The environment is general purpose and has other capabilities such as supporting shared memory, path expressions for synchronizations, a host of tools to automate or semi-automate different activities, and graphical and window based user interfaces. It is based in a Sun Microsystems workstation environment, using Sun windows for display and user interfaces.
REQUIREMENTS FOR A PROTOTYPING ENVIRONMENT

Let us first discuss the facilities and primitives that an environment for prototyping should support. Then we will describe the design of our environment and how it supports the primitives.

Support different communication primitives. A number of different primitives for interprocess communication have been proposed. Different systems may support different primitives. Hence, a prototyping environment should allow for all commonly used primitives. In addition, as the performance of an application depends on the choice of the communication medium, it is desirable to have means to easily change the underlying communications network.

Support heterogeneity. Heterogeneous computing is now an active area of research. Most organizations have many different types of machines. We would like any prototyping system to allow processes to execute on different machines. In addition, a truly heterogeneous system should also allow different source languages for different processes of a distributed application.

Centralized monitoring and control facility. For experimentation and analysis, it is extremely important to be able to control the I/O for different processes from one console, even if the processes are executing on different processors. Monitoring the state of different processes, to get a "global view" of the system on one console can be extremely useful in validating and understanding the behavior of the distributed algorithm. This property is important for the environment if it is to be used effectively.

Debugging Support. To debug any program, users need information about the implementation as well as the exhibited behavior of the program in question. Debugging is difficult enough a task for serial programs run entirely on a local node. However, when debugging in a distributed environment, the task of keeping track of many threads of control across many machines can be very complex and tedious. We need for the debuggers on each host to interoperate so that users may localize faults effectively.

Different Communication Delays. Communication between processes can have different delay characteristics depending on the underlying communication network. The prototyping environment should support different delay characteristics for communication between different processes so that realistic prototypes can be built.

Real time clocks. There should be support for real-time clocks. This will allow real-time distributed algorithms to be implemented, and timing properties to be studied.

Instrumentation. The environment should provide primitives for instrumenting a distributed application, to provide data about performance measures, delays, message interaction etc. This can be very useful for evaluating and verifying a distributed application.

Process allocation. The environment should provide simple methods to allocate processes to different machines, and provide flexibility to the designer.

SUPPORT FOR PROTOTYPING

The environment we describe here supports most of the requirements discussed above. For prototyping, the system distinguishes module implementations from the interconnection of modules in a particular application. The modules are implemented and compiled separately. To construct an application, the modules are then interconnected in the desired manner using the Module Interconnection Language (MIL) of the system. With the MIL a user can design an application using the program modules. The MIL also allows dynamic binding between modules, which makes modification of an application easy — often only the interconnection specification needs to be changed. While our underlying interconnection system provides for transparent use of arbitrary application languages, we have extended the C language to include constructs for interprocess communication and instrumentation, specifically taking advantage of our interconnection system. In this section we describe these language features, along with the necessary parts of the MIL.

Source Language

We have augmented the language C to provide the basic primitives for communication, and instrumentation. The code first goes through a pre-processor that converts these primitives into necessary system calls. For communication, we allow remote procedure call, and both synchronous and asynchronous message passing.

The use of Remote Procedure Call (RPC) for distributed programming has been promoted by many [Lin85, Nels81, YaJT88]. In our system, a call to a remote procedure appears exactly the same as a call to a local procedure. Later, by using MIL, the call is bound to the remote procedure. So at the source language level no special primitive is needed for RPC.

In the source language, the communication primitives for message passing are of the form

\[
\text{send (name, message) [delay = nn] } \\
\text{receive (name, message)}
\]

In the send command, the name is not the name of the target process (although that can be done, if desired), but rather a variable name specifying an output port that is bound to the target process using the MIL specifications. The name in the receive command is also bound later to a particular sender. This technique gives the designer flexibility in choosing naming methods for communication. Moreover, changing
Module Interconnection

Once functions are implemented using the modified C language, a distributed program can be constructed from these functions. First, for each function to be used separately in the system (e.g., code for a process), a module is defined. The definition of a module contains the name of the module; optionally the specification of its interfaces (an interface must be specified if the module is to be used by other modules); a list of all other services it uses; an implementation (object code file); and different optional attributes, such as the target machine. An application is created using the modules by specifying all bindings needed between different modules.

As an example, consider a function called main, which calls a procedure proc (which could be a remote procedure that resides on a different host machine). Two modules are defined in the system, and the module definitions are given below:

module main {
  uses: proc (parm.list)
  implementation: "main.o"
  host: mimsy
}

module proc {
  interface: proc (parm.list)
  implementation: "proc.o"
  host: tove
}

The first definition specifies a module called main, which uses another module called proc whose object code is in the file main.o, suitable for execution on the host machine called mimsy. It has no interface definition, as no other module uses this module. The module specification for proc has an interface definition, as it is used by the module main. Note that the module proc is specified for execution on a different machine than the module main.

To form the program, we need to create an application by importing different modules and then binding their interfaces. We bind the module specification in the uses part to the interface specification of that module. In the above example, an application is created as follows.

application demo {
  import main
  import proc
  bind main.proc proc.proc
}

This will create an application called demo that uses two modules — main and proc. Since main invokes the module proc, we need to specify their relationship explicitly. This is done by the bind clause, which specifies that the specification in main of proc is to be bound to the interface specification of the module proc. The module and application definitions are compiled with the MIL compiler, then linked together for execution. At run time, the system creates two processes for this application, one for each module. However, the process corresponding to module proc is blocked until the RPC call is made by the process executing main.

The specification of modules that communicate by passing messages is similar. In this case we have an intermediate module with which the sender and the receiver bind themselves. For example, if we want a process P1 that will send a message to another process P2, the code for each process (written in the modified C language), will include the proper send and receive commands. Suppose the send command is send (name1, msg), and the receive command is receive (name2, msg). Module specifications for the two processes are given below:

module P1 {
  interface: name1
  implementation: "P1.o"
  host: mimsy
}

module P2 {
  interface: proc (parm.list)
  implementation: "proc.o"
  host: tove
}

The message in the commands is any general data structure.

In the MIL specification of a module, an attribute can be given that specifies the number of buffers allowed. This allows the user to experiment with different bounded message queues. By setting the buffer size to 0, we can get synchronous message passing. With the send command there is an optional delay attribute. Using the delay feature, different communication delays can be obtained in the application.

Markers are provided in the language for instrumentation. There are two timing data that are collected during execution of processes for which user specification is needed. One is the blocking time for a receive. If the blocking time of a receive is to be measured, this is specified by putting a "Q" before the receive command. The blocking time data is collected only for the receive statements marked by the user.

The other timing information of use is the total execution time between two statements in the process. Here again the user has to mark the statements. (For the complete process we just mark the start and end statement of the process.) This is done by naming a statement block, and identifying the start and end of that block, as shown below:

name S.L.: stmt
stmt
stmt
EL.: stmt

S.L identifies the start of the statement block named name; EL identifies its end. The execution time of the process for statements between S.L and EL will be recorded by the environment. For a given process we require that the names of the blocks be distinct.
module P2 {
    uses: name2
    implementation: "P2.0"
    host: tove
    buffer_size: 5
}

application demo {
    import P1
    import P2
    bind P1.name1 P2.name2
}

In addition to interface, uses, host, and implementation specifications in the module, we also have the option of specifying the buffer size. As discussed earlier, this can be used to support synchronous and asynchronous message passing.

At execution time both the modules will execute in parallel according to their specifications and attributes. So the instantiation for message passing is same as the instantiation command of RPC.

**Heterogeneity**

The environment supports heterogeneous computing to some degree. The hosts of different modules can be machines with different architectures and different data representations. The system performs the data conversions transparently. However, currently all hosts must be running Unix. This restriction is more for the sake of convenience. We believe that our approach can be easily extended to different operating system also, and we are currently extending our implemented system to demonstrate this point.

In addition to the different machine architectures, the environment also supports different source languages. A module can invoke a module written in any source language. However, in our current system, we only have the send/receive primitives for the C language modules (i.e., modules written in other application languages such as Ada, Lisp and Fortran can be incorporated, but access from those codes to the interconnection system is not currently transparent.) Regardless of the application language used, the binding is done in the same way as described earlier.

**Instrumentation and Traces**

The system produces a log of all the messages that pass between different processes. For each process, the log contains the source and target for all messages sent. The message order is maintained in the log, which can be extremely useful for debugging and understanding the working of the distributed application.

In addition, time information for the marked statements is also recorded. For receive commands, the time information is identified by the name used in the receive command. For a block of statements, the name of the block is used. One of the main reasons of having distinct names of blocks is to uniquely associate the timing data with the statement block.

**User Interface and Debugging Support**

Our support environment (POLYLITH, as will be described) provides a growing collection of tools to assist in construction, instrumentation and debugging of application systems. Here we briefly describe the TCP/IP-based toolbus which has a (Sun Microsystems workstation) SUntools presentation to the user.

Initially the user is shown a normal Unix shell window, from which the toolbus can be initiated, and given a compiled MIL specification as input. All input from the user through the initial shell window is passed to the user processes, and likewise all application output appears in the single window. Optionally, the toolbus can be started up in a visualization mode: an additional window will appear, showing a graph representation of the application structure (based on the MIL specification) along with a window pane for user dialogue with the toolbus. In this mode, the application will not start immediately, but instead the processes are blocked after initialization. Using the mouse, the user can select nodes in the graph for inspection (windows containing the module's source code or attributes can be popped up), and likewise can request that a separate window be opened to handle any I/O solely concerning that process.

Once the user has established all the desired windows, the application can be made to proceed with a command to the panel controlling dialogue with the toolbus itself. In this mode the user can request that the nodes in the displayed program graph be highlighted to illustrate the processes' state during execution, e.g., to illustrate graphically whether the node is blocked awaiting input on a port.

For debugging, the environment allows the user to use the mouse to indicate nodes which should be placed in a 'stepping mode', in which messages to and from the selected node are queued, and only gated through by explicit mouse action on the part of the user. In addition, certain levels of module fault localization can be partially automated [HeK88].

The environment also supports the traditional intra-module debugging capabilities: the user can point to nodes in the graph with the mouse and request that the toolbus initiate a traditional debugger (e.g., dbx or adb in Unix systems) on the appropriate host. Should a process terminate abnormally, this can be detected by the toolbus, which can start a proxy on the appropriate machine to locate the correct 'core' files left on the host. Tools are also available to assist visualization of the running application's state using a packaging technique made available in [PuCa88].
SYSTEM DESIGN AND IMPLEMENTATION

The system described in previous sections is implemented using the facilities provided in the POLYLITH software interconnection system. In this section we briefly describe the design and implementation of POLYLITH. Details can be found in [Purt85a, Purt85b]. POLYLITH has two major components - module interconnection language and a software bus.

Module Interconnection Language

An application system in POLYLITH is thought of in terms of independent modules that interact with each other through specified interconnections to form a coherent whole. A module embodies a specific functionality and defines a number of interfaces or ports through which it can communicate with other modules. Furthermore, a set of attributes can be associated with each module. The actual interconnectivity (topology) of a system is specified by combining modules in an application. The bindings between interfaces with compatible interface patterns are enumerated. The resulting system corresponds directly to a simple attributed graph, where each node represents a module and the directed arcs between them represent bindings between interfaces. Modules and interfaces are specified using the Module Interconnection Language (MIL). Besides describing the structure of a software application, the MIL is also used to specify how the application should be mapped onto host configurations, and for incorporating different implementation languages, host architectures, and interconnection media.

The important concept here is the separation of descriptive assertions (about the modules and their interconnectivity) from their implementation (in terms of specific programming languages and actual communication media). This frees the modular design from considerations of low-level, language specific details, and also frees the individual implementations from the necessity of knowing the inner workings of other modules or the interconnection media (i.e., they can handle all function calls as if they were in the same process space and written in the same programming language). This separation is possible due to the software bus. We have presented some examples of uses of MIL in the previous sections. Further details can be found in [Purt88].

The MIL specifications about modules and their interconnections are given to a compiler (called Cluster Specifications Compiler for historic reasons). The compiler generates a table which contains an encoding of all the relevant information about the modules. In particular, the table contains information about the type of all components of each interface of a module (needed for transformation between different languages and machines). It also contains information about the attributes of the module. This is needed by the run time system to execute the module according to the specs.

Software Bus

The basic support for execution in POLYLITH is provided by a software bus (or “toolbus”). The bus takes as input the table produced by the compiler. The bus is the execution-time agent which encapsulates all machine and language-specific characteristics such as how data is represented or communication between components is achieved [Purt88]. It provides various operations for run-time support.

By introducing the notion of a software bus, we improve our ability to accommodate heterogeneity in source languages, host architectures, and communication media. The system uses source-to-source translation technique to map the extended C notation described above into a regular C program that directly accesses the capabilities of the toolbus.

The toolbus has four major functions:

1. Starting the processes.
2. Supporting communication between the processes.
3. Transforming the representation of data that is passed between different source languages and machine architectures.
4. Providing information about run-time state.

The toolbus creates a process for each module imported in the application specifications. Process creation maintains the properties specified for the modules like the host machine etc. In addition, during process creation the bus automatically creates "stubs" to "catch" the function calls for intermodule communication. Also, proper "prologue" and "epilogue" routines are generated for each process (these are needed for proper communication). This approach is described in detail in [Purt88], and is different from many existing approaches for distributed systems [JoRT85, Lisk88, NoBL88, RaRo81].

The bus manages all the communication between modules at run-time. Each process, once it is started, registers itself with the bus and passes it the relevant information about its location, communication port etc. The communication between modules then proceeds through the bus. Alternatively, after the processes have registered, their interfaces can be "hard wired" together by the bus, and then any future communication between the processes does not go through the bus.

For communication, the bus can use different communication protocols. These protocols are offered as options. Our current bus uses the TCP/IP sockets. In the TCP/IP based bus, as each process is created it acquires its own IP port from the host on which it was started. It then "registers" this information with the toolbus. This information is then used later by the toolbus to provide communication between modules.
In the MIL specification, the data type of all parameters on module interfaces is given in a language-independent, POLYLITH notation. When users implement their modules, they are free to utilize the data representations natural for their application languages. Data transformation in POLYLITH is done by coercing the representation for all data transmitted through the toolbus. All data to be transferred between interconnected modules is transparently coerced into the intermediate representation, and then sent to the target module.

On the target module side, the bus transparently converts the data in the intermediate representation to the form required by the receiver machine and the language in which the target module is coded. The intermediate representation is such that both these transformations can be uniquely performed without too much overhead.

The actual code for performing the coercion is not provided by the user, but instead is implemented by a system manager. One coercion routine must be given for each application language and host architecture which will participate in POLYLITH; these routines must implement maps (and their inverses) from the host-specific data representations to the intermediate language. This coercion is a function of the source language and the machine architecture. POLYLITH has many such functions to handle the different coercion situations. Details of the coercion task are given in [Part88].

For collecting data about the messages and execution times, the bus has to be invoked with a special option. In this form the bus logs all the messages, along with the names of the source and destinations, and also logs information about the execution times. Other options are available for performing different functions.

**CONCLUSIONS**

Over the years distributed systems have proliferated in the form of local area networks and wide area networks. However, good facilities for developing distributed applications are needed for us to effectively take advantage of these distributed systems. Designing a distributed application is an extremely complex task, and there is little experience in designing such applications.

Prototyping could be a useful approach that can help in designing a distributed application. In this paper we have described an environment that can be used for prototyping distributed applications. The environment provides different communication primitives, along with primitives to aid testing and debugging. The environment allows users to control and monitor the progress of different processes in a distributed application from a single console. Our approach allows users to develop distributed applications using existing languages and packages, as opposed to restricting users to employing only specialized specification languages (as is done, for instance, in [FrVa88]). Our environment extends the type of work done in [HaMS88] by providing primitives to assist in profiling and instrumenting the applications.

For prototyping, our system distinguishes module implementations from the interconnection of those modules for a particular application. Different modules are written and compiled separately. For an application, the modules are interconnected in the desired manner using the module interconnection language of the system. We have augmented the language C to provide the basic primitives for communication, and instrumentation. The execution environment supports remote procedure call and both synchronous and asynchronous message passing. Different delays can be specified for messages. For instrumentation the environment provides the trace of all the messages between processes.

The environment provides many features for debugging. The message transmission between processes can be controlled by the user, and the user can "release" one message at a time. The user can also look at the queue of a process. The cores of different processes can be monitored on a single console in different windows. Similarly, all the input, output, of different processes can be done from a single Sun console, each process being controlled from one window.

We are continuing to work on both graphical and textual tools that assist in visualization of the run-time state of distributed programs. In the future we also plan to extend our approach to provide primitives that support fault tolerance.

**REFERENCES**


