December 1984

A Programming System for Distributed Real-Time Applications

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

A Programming System for Distributed Real-Time Applications

Abstract
A distributed programming system designed to support the construction and execution of a real-time distributed program is presented. The system is to facilitate the construction of a distributed program from sequential programs written in different programming languages and to simplify the loading and execution of the distributed program. The system is based on a distributed configuration language. The language is used to write the configuration of a distributed program, which includes resource requirements, process declarations, port connections, real-time constraints, process assignment constraints, and process control statements.

A PROGRAMMING SYSTEM FOR DISTRIBUTED REAL-TIME APPLICATIONS

Insup Lee

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

A distributed programming system designed to support the construction and execution of a real-time distributed program is presented. The system is to facilitate the construction of a distributed program from sequential programs written in different programming languages and to simplify the loading and execution of the distributed program. The system is based on a distributed configuration language. The language is used to write the configuration of a distributed program, which includes resource requirements, process declarations, port connections, real-time constraints, process assignment constraints, and process control statements.

Introduction

It is well known that there are many potential advantages of distributed processing. As distributed systems become readily available, we need a simple way to construct distributed programs to attain these potential advantages. In particular, there should be a simple way to combine several existing sequential programs and to execute them as a distributed program. Although message based interprocess communications support the construction of a distributed program as a collection of processes that cooperate by exchanging messages, their dependencies are hidden in the text of the source code; and therefore, make it hard to understand the overall structure of a distributed program. What we need is a clean way to glue a set of sequential processes together to form a distributed program; that is, we need a configuration language to support programming-in-the-large [4] for distributed programs.

The distributed programming system described in this paper is to provide a coherent environment for the development of a multi-sensory (vision and tactile) system to investigate "active perception" of three dimensional objects. The multi-sensory system consists of a robot arm with six joints (PUMA 560), a three finger robot hand with tactile sensors [1] and two CCD cameras (Fairchild CCD3000). Active perception means being able not only to see and feel objects but also to manipulate and probe them.

Issues in implementing real-time distributed systems for a multi-sensory system range from operating systems to application programs. At the operating systems level, we need to provide proper interprocess communication mechanisms that can deliver messages within real-time constraints so that the robot arm and hand can be controlled in real-time. The process scheduler should be general enough to support various scheduling disciplines required by different application programs without too much overhead. At the applications level, it should be easy to construct a distributed program from a set of component sequential processes, possibly written in different programming languages. The multi-sensory system will be built from a set of interacting processes written in programming languages like C, FORTRAN, LISP, Pascal, and Prolog. Its component processes range from real-time joint motor controllers written in C to knowledge-based experts written in Lisp or Prolog. Although considerable work has been done on real-time distributed systems in recent years [6, 8, 16], these systems are primarily designed to efficiently support distributed real-time control processes. What we need is a distributed system that can support not only low level real-time control processes but also high level knowledge-based systems.

The purposes of the distributed programming system, called DPS, are to facilitate the construction of a real-time distributed program from existing sequential programs written in different programming languages and to simplify the loading and execution of a real-time distributed program. The distributed programming system is based on a distributed configuration language, called DICON. DICON is used to write the

CH2105-5/84/0000/0018$01.00 © 1984 IEEE

configuration of a distributed program composed of resource requirements, process declarations, real-time constraints, process assignment constraints, and process control statements. A distributed program is constructed by compiling a configuration and is loaded and started when a compiled configuration is executed. During the execution of a distributed program, the programmer can monitor and control the execution of component processes for debugging.

Although there exist several distributed configuration languages, such as CONIC [18], PRONET/NETSLA [11], and PCL [15], they are not suitable to be used in our distributed system. Their main concern is to separate the descriptions of process hierarchies and interprocess communications from the text of component programs, presumably written in a single distributed programming language. These configuration languages do not support the specifications of real-time and process allocation constraints. Thus, finding a process assignment has to be carried out separately after a distributed program has been constructed. The process allocation problem, with and without real-time constraints, has been investigated by many researchers and many useful mathematical models (such as graph theoretical [20, 21, 22], integer programming [3, 14, 17], and heuristic [2, 7, 9] methods) have been developed to solve the problem. We believe that a configuration should include specifications of various assignment and real-time constraints so that an optimal (or near optimal) process allocation can be computed by a process allocator based on existing (or modified) assignment algorithms.

This paper is organized as follows: Section 2 describes the organization of the underlying distributed system and briefly discuss process scheduling and interprocess communications supported by the distributed programming system kernel. Section 3 illustrates the distributed programming system through an example. Section 4 contains the description of DICON. Section 5 explains the implementation structure of the distributed programming system and its current status.

DPS Kernel

The underlying distributed system consists of a loosely coupled heterogeneous mix of computers, which will eventually include three VAX 750's, ten super-microcomputers and numerous microprocessors. The three VAXes will run a UNIX system (Berkeley 4.2) extended for our distributed environment. The other computers will run the DPS kernel that provides interprocess communications, process management, and process scheduling. The DPS kernel is UNIX compatible in a sense that any UNIX user process can be executed without modifications. When the network is dedicated to the multi-sensory system, the VAXes will support knowledge-based experts. The super-microcomputers will support time-critical processes and compute-bound processes. The microprocessors are to monitor tactile sensory pads on fingers and to drive servo motors of the robot arm, hand, and two cameras. To the programmer, the distributed system is viewed as a network of UNIX systems. The distributed programming system is not to replace the UNIX system user interface for the network, but to supplement it for distributed programming.

In this paper, a distributed program means an application program (e.g., a multi-sensory system) composed of a set of communicating sequential processes. Each process in the distributed program has its own local data over which it has sole control. A process communicates with other processes by exchanging messages. The component processes of a distributed program may be written in different programming languages. To allow efficient use of the system resources, the distributed system can be partitioned to support several distributed programs at the same time. Each partition supports one distributed program. Partitions may overlap to share system resources. A resource manager handles partitioning of the network.

Process Scheduling

A process that has timing constraints and whose correctness depends on whether its timing constraints are satisfied is called a real-time process. There are two kinds of real-time processes: periodic and sporadic [19]. A periodic process becomes ready at regular intervals and a sporadic process becomes ready at any time. In our system, the programmer specifies execution time, deadline and hardware interrupt level for each real-time process. The programmer also specifies periods for periodic processes and minimum intervals between consecutive executions for sporadic processes. From the execution time and hardware interrupt level, the actual execution time is computed by adding overhead of non-disabled interrupts to the execution time [23]. The actual execution times, deadlines, periods and intervals of real-time processes are used to ensure that all real-time processes can be executed before their deadlines at compile-time.

The DPS kernel schedules real-time processes based on the earliest deadline algorithm with preemption. That is, a real-time process with the earliest deadline is executed until it blocks or another real-time process with an earlier deadline becomes ready to run. We note that the earliest deadline scheduling discipline is optimal for the case where the effects of interprocess communications are not considered [5]. Processes with no timing constraints are executed only when no real-time processes are ready. They are scheduled on a preemptive priority base with 16 priority levels. Processes with the highest priority are executed in round robin with a programmer defined quantum. It is the programmer's responsibility to assign proper priorities to processes.

Interprocess Communications

The DPS kernel supports message-based real-time interprocess communications. Message addressing schemes usually use either source object addressing or destination object addressing [8]. With source object addressing, a sending process sends a message to its local object which is connected to a destination object. On the other hand, with destination object addressing, a sending process sends a message to an object in the destination process. Since destination object addressing requires global unique object names, and therefore, hinders modularity of component processes, our system supports source addressing.

In our system, messages are sent to and received from ports. A port is an object into which messages may be placed and from which messages may be removed. The former is called in-port and the latter is called out-port. Ports are typed; that is, each port can be used to send or receive one type of messages. Each port has a unique identifier that distinguishes it within a process; however, the same port identifier can be used in different processes. Processes are written and compiled without specifying how ports are to be connected. In-ports and out-ports have to be connected before messages can be sent and received. Connections between in-ports and out-ports are defined within a configuration written in DICON.

There are six interprocess communication primitives provided in each programming language supported by DPS. The SendWait and SendNoWait system calls are to place a message into a port. The ReceiveWait and ReceiveNoWait system calls are to remove a message from a port. All messages placed into a single port are delivered in the same order as they are sent without duplication under normal condition. The MsgPending system call returns the number of outstanding messages on a port. The MsgFlush system call cancels outstanding messages on a port. The formats of these system calls are as follows:

- SendWait (PortId, Timeout, VarId)
- SendNoWait (PortId, VarId)
- ReceiveWait (PortId, Timeout, VarId)
- ReceiveNoWait (PortId, Handler, VarId)
- MsgPending (PortId)
- MsgFlush (PortId)

PortId is the name of a port. VarId is the name of a variable which contains a message to be sent or to which a message is to be received. A process executing SendWait or ReceiveWait is delayed until the message is received or until a message arrives, respectively. Here, the process may specify, in a TimeOut argument, how long it wishes to wait for the corresponding action to take a place.

Unlike SendWait or ReceiveWait, a process executing a SendNoWait or ReceiveNoWait call is never delayed. The ReceiveNoWait call requires a message-handling routine that is to be invoked when a message arrives. So, when a message arrives on a port specified in a ReceiveNoWait call, the execution of the receiving process is halted and then its message-handling routine is invoked. To support the sending and receiving of complex data structures in messages, DPS provides routines that convert a linked list into an array and vice versa. The ipc mechanism is explained and justified in [13].

Port Connections

Port connections are defined within a configuration written in DICON. Permitted port connections are one-to-one, many-to-one, one-to-many, and many-to-many. One-to-one connection means that one out-port is connected to one in-port. Many-to-one connection means that many out-ports are connected to one in-port. Here, the in-port can receive a message from any out-port. One-to-many connection means that one out-port is connected to many in-ports. Here, a message placed into the out-port is delivered to all the in-ports. Many-to-many connection combines both many-to-one and one-to-many.

Priority. To support fast propagation of unusual events (like too much pressure on a fingertip), port connections have priority. Port connections between real-time processes have the highest priority. Messages transmitted through port connections with high priority are delivered before messages transmitted through port connections with low priority. The priority based port connections and the process priorities can be used to ensure that emergency events are handled quickly (i.e., in real-time). For example, guarded move for mechanical fingers can be implemented using

ReceiveNoWait on in-ports that are connected to out-ports defined in tactile sensor processes with a high priority. It is a programmer's responsibility to assign proper port connection and process priorities when writing the configuration of a distributed program.

Type Checking. When ports are connected, it is desirable to ensure that actual arguments used to send and receive messages through the connected ports have the same type. Since type definitions among the programming languages supported by DPS are not compatible, the type checking of port connections is based on structural equivalence. That is, a pair of in-port and out-port ~an be connected if the internal representation (in terms of primitive data types such as 16 bit integer, 32 bit integer, 8 bit character, etc.) of a variable used with the out-port is the same as that used with the in-port.

The consistency of port connection can be checked either statically or dynamically. The advantage of static consistency checking is the early detection of representation mismatch. The disadvantage is that a single port can only be used to send messages with the same representation. The advantage of dynamic consistency checking is that it is possible to use one port to send messages with different representations. The disadvantages are that inconsistent port connections cannot be detected until run-time and that there is execution overhead to carry out consistency checking at run-time. Furthermore, message representation information need to be sent with data, and therefore, more bits have to be transmitted. DPS supports the static consistency checking of port connections.

Overview of DPS

In DPS, the programmer builds a distributed program and then executes it as follows: (1) construct or solicit component programs; (2) write the configuration of a distributed program in DICON; (3) compile the configuration; (4) execute the compiled configuration; (5) monitor and control the execution of processes interactively.

As an example of how to use DPS, we will implement a robot arm with three degrees of freedom, each of which is manipulated by a separate motor. The arm controller accepts the x, y, and z coordinates of the next position. From the current and next positions of the arm, it computes three joint movements in parallel. It then moves the three joints in parallel. The arm controller waits until all three joints are moved before receiving the next set of three coordinates. The same robot arm system has been used to compare three solutions for a robot arm controller using Pascal-Plus, occam, and Edison [10]. Among the three solutions, the structure of our solution is similar to that written in occam as both are based on message passing.

Our implementation of the arm controller uses seven processes written in C. The seven processes are one controller process, three joint motion calculators, and three joint motor drivers. The controller process accepts the next position from the operator. It then sends the next position to the joint motion calculators to compute three joint motions in parallel. As the controller receives computed motions, they are sent to the joint motor drivers to move the arm in parallel. The controller waits until all three joints are moved and then accepts the next position. To simplify our illustration, we have omitted error handling.

Figure 1 is the outline of a program that drives a joint motor. The arm controller will have three instances of this program to drive the three joint motors in parallel. Within the program, in-port NextMove is used to receive the next motion and out-port MoveDone is used to signal the completion of motor movement. The nil argument of the SendWait call means that no data are sent; the purpose of such a call is to synchronize with other processes.

```
/* This program drives a joint motor. */

main () {
   inport NextMove;
   outport MoveDone;
   struct { float step, direction } motion;
   for (;;) {
      ReceiveWait (NextMove, 0, &motion);
      /* Move motor according to motion.step and
         motion.direction */
      SendWait (MoveDone, nil);
   }
}
```

Figure 1: Outline of Joint Motor Driver Process

Figure 2 outlines a program that computes a joint motion for the next position of the arm. The arm controller will have three instances of this program to compute three joint motions in parallel. When each instance is started, an appropriate joint number is passed as an argument to the process to tell which joint motion it is to compute. The program receives the next position from in-port NextPosition and sends the next motion to out-port NextMotion.

Figure 3 outlines a program that accepts the new coordinates of the arm from the operator and that

```
/* This program computes next motion. */

main (jointno) int jointno; {
   inport NextPosition;
   outport NextMotion;
   struct { float x,y,z } old_position,new_position;
   struct { float step, direction } motion;
   /* Set up necessary matrices for this jointno */
   /* Initialize old_position */
   for (;;) {
      ReceiveWait (NextPosition, 0, &new_position);
      /* Compute next motor step and direction
         from old and new positions */
      SendNoWait (NextMotion, &motion);
      old_position = new_position;
   }
}
```

Figure 2: Outline of Joint Motion Compute Process

controls the joint motion calculators and motor drivers. The SendNoWait call on ComputeJointMotion starts the computations of three joint motions in parallel. The three ReceiveWait calls, each preceeded by a MsgPending call, allow messages to be received as they arrive on three in-ports JointMotion[0..2] (that is, in non-sequential fashion).

We now show how the above programs are combined and executed as a distributed arm control program in DPS. The construction and execution of the distributed arm control program is carried out in four steps. The first step is to write a configuration in DICON as shown in Figure 4. The configuration consists of four parts. The first part defines resources needed to execute the arm controller. Two node declaration statements specify that the controller is to run on one VAX/11 750, call ASP, and three super-microcomputers, called Processor0, Processor1, and Processor2. The three super-microcomputers are to be exclusively used by the arm controller, whereas ASP is to be shared with other programs. Processor0, Processor1, Processor2 and ASP are system defined names for processors in the network. They are to be identified by Joint[0..2] and Host within the configuration.

The second part describes the constituent processes of the arm controller. Three program definitions identify sequential programs that are used within the arm controller. The first program definition says that a program, called MotorController, is written in C and its object code is stored at file "driver.o". Its in-port is NextMove and its out-port is MoveDone. The other two program definitions specify the Calculator and Controller programs. The process declarations define the instances (i.e., processes) of the sequential programs needed to implement the arm controller. There are three instances of MotorController, three instances of

```
/* This process accepts new coordinates from
   the operator and moves the robot arm. */

main () {
   outport ComputeJointMotion, MoveJoint[3];
   inport JointMotion[3], MoveDone;
   struct { float x,y,z } pos;
   struct { float step, direction } motion[3];
   int i;
   for (;;) {
      printf ("Next position: ");
      scanf ("%f%f%f", pos.x, pos.y, pos.z);
      /* Compute motion for each motor in parallel */
      SendNoWait (ComputeJointMotion, &pos);
      i = 0;
      while (i < 3) {
         if (MsgPending (JointMotion[0])) {
            ReceiveWait (JointMotion[0],0,&motion[0]);
            SendNoWait (MoveJoint[0], &motion[0]);
            i++;
         };
         if (MsgPending(JointMotion[1])) {
            ReceiveWait (JointMotion[1],0,&motion[1]);
            SendNoWait (MoveJoint[1], &motion[1]);
            i++;
         };
         if (MsgPending (JointMotion[2])) {
            ReceiveWait (JointMotion[2],0,&motion[2]);
            SendNoWait (MoveJoint[2], &motion[2]);
            i++;
         };
      };
      for (i=0; i<3; i++) ReceiveWait(MoveDone,0,nil);
      printf("Arm is at %f %f %f\n",pos.x,pos.y,pos.z);
   };
}
```

Figure 3: Outline of Controller Process

Calculator, and one instance of Controller. The link definitions specify how ports used within these processes are to be connected. The first link statement connects out-port ComputeJointMotion of the Control process to in-port NextPosition of each Calculate process so that the new position sent from the Control process can be received by all three Calculate processes. The second link statement connects out-port NextMotion of process Calculate[i] to in-port JointMotion[i] of the Control process for each i from 0 to 2. The remaining two link statements should be self-explanatory.

The third part specifies process assignment constraints which should be satisfied when the arm controller is loaded into processors for execution. The two assign statements say that process Motor[i] should be allocated into processor Joint[i] for each i from 0 to 2 and that the Control process should be allocated into the Host processor. The place statement says that three Calculate processes can be allocated into any processors as long as they are assigned into different processors so that they can be executed in parallel.

```
system RobotArmController;

    node Host : shared ASP;
    node Joint[0..2]:Processor0,Processor1,Processor2;

    program MotorController in C at "driver.o";
       out port MoveDone;
       in port  NextMove;
    end MotorController;

    program Calculator in C at "calculator.o";
       out port NextMotion;
       in port  NextPosition;
    end Calculator;

    program Controller in C at "controller.o";
       out port ComputeJointMotion, MoveJoint[0..2];
       in port  JointMotion[0..2], MoveDone;
    end Controller;

    process Motor[0..2] : MotorController;
    process Calculate[0..2] : Calculator;
    process Control : Controller;

    link ComputeJointMotion to
         Calculate[0..2].NextPosition; /* one-to-many */
    link Calculate[0..2].NextMotion to
         JointMotion[0..2] one-to-one;
    link MoveJoint[0..2] to
         Motor[0..2].NextMove one-to-one;
    link Motor[0..2].MoveDone to
         Control.MoveDone; /* many-to-one */

    assign Motor[0..2] on JointProcessor[0..2];
    assign Control on Host;
    place  Calculate[0..2] separately;

 begin
    start for i := 1 to 3 do Calculate[1] (i) end for;
    start Motor[0..2], Control;
 end system;
```

Figure 4: Configuration for Arm Controller System

The last part of the configuration is start statements. The three Calulate processes are started first and then the Control and three Motor processes are started. When a Calculate process is started, a joint number is passed as an argument to the process so that matrices used to compute joint motions can be initialized properly. Since the Calculate processes need to initialize their matrices, they are started before the Motor and Control processes.

After the configuration has been constructed, the next step is to compile the configuration. Assuming that it is in file "arm.conf", it is compiled by the command

```
dcomp arm.conf
```

Its output is generated in file "arm.exec". The third step is to execute "arm.exec" to start the arm controller by the command

```
dexec arm.exec
```

The last step is to monitor and control the execution of the arm controller interactively. For example, the command

```
stop RobotArmController
```

terminates the execution of the arm controller and releases the resources used by the arm controller.

A Distributed Configuration Language

The backbone of DPS is a distributed configuration language, called DICON. A configuration written in DICON provides the centralized expression of a distributed program, which is important in presenting and understanding the abstraction to be supported by a distributed program. The design goals of DICON are that it should support modular construction of a distributed program and resource requirements, real-time constraints and process assignment constraints in addition to interprocess relationships should be part of the configuration of a distributed program. This section provides a brief overview of the language; the complete definition can be found in [12].

The syntax of DICON is described using the BNF, where syntactic constructs are denoted by words enclosed with < >. In the syntax, optional clauses are enclosed with [ ], zero or more repetitions are indicated with { }, and alternatives are separated by |.

Overall Structure

The configuration of a distributed system consists of a system interface, subsystem definitions, resource requirements, process declarations, port link definitions, real-time constraints, process assignment constraints, and process start statements as follows:

```
<system> ::= system <id>
                  <sys interface>
                  <subsys defs>
                  <resource requirements>
                  <process decls>
                  <port link defs>
                  <real-time constraints>
                  <allocation constraints>
               begin
                  <process start statements>
               end system
```

A system interface lists ports that are defined within the configuration and that are to be connected with ports defined in other configurations. Its format is as follows:

```
<sys interface> ::= <port list>
<port list> ::= [ out port <ids> ] [ in port <ids> ]
```

If a configuration defines a stand alone distributed system, its system interface should be empty. When configurations are included within a configuration, out-ports and in-ports of each configuration should be connected to in-ports and out-ports, respectively, of other configurations.

To support the modular construction of a distributed program, configurations can be nested. Nested configurations are specified by subsystem definitions, which have the following format:

```
<subsys defs> ::= { system <id> at <file> <port list> }
```

`<id>` is the name of a nested configuration and `<file>` identifies the file that contains the source code of the nested configuration. Its out-ports and in-ports are repeated to enhance readability.

Resource Requirements

Resource requirements specify processors and special-purpose devices necessary for the execution of a distributed program. The specified resources must be a subset of the underlying network. They must be acquired from the network resource manager before the distributed program start executing. The resource requirements should not include system resources used by nested configurations since they are specified within the nested configurations. The format of the resource requirements is as follows:

```
<resource requirements> ::= { <node decl> }
<node decl> ::= node <node ids> : [ shared ]
                [ <machine ids> ] [ <attributes> ]
<attributes> ::= with [ memory = <const> ]
                      [ cpu = <cpu type> ]
                      { device = <device id> }
```

We assume that there is a unique system-defined name for each processor or device. A node declaration binds `<node ids>` to processors with attached devices. If an optional `<machine ids>` is specified, each identifier specified within `<node ids>` is associated with a processor within `<machine ids>`. If an optional `<attributes>` is specified in addition to `<machine ids>`, processors included in `<machine ids>` that satisfy `<attributes>` are assigned. If `<machine ids>` is not specified, any processors that satisfy `<attributes>` are allocated. For example, a node declaration

```
node Joint[0..2]: Processor0,Processor1,Processor2
```

associates Joint[0] with Processor0, Joint[1] with Processor1, and Joint[2] with Processor2. If there are only three Joint Motors in the system, then the following node declaration is equivalent to the above declaration:

```
node Joint[0..2] : with device = JointMotor
```

Thus, to use a special-purpose device, the programmer can specify the device rather than the processor on which the device is attached. This feature will free the programmer from having to exactly know the status of each component of the network, which is going to vary occasionally due to hardware failure or expansion. The possible attributes are memory size, CPU type, and attached devices. If the keyword shared is specified, processors can be shared by other distributed programs; otherwise, they are used exclusively by the distributed program.

Process Declarations

A distributed program is defined by declaring component processes and by specifying port connections. Process declarations consist of program definitions and process (instance of a program) creations; that is,

```
<process decls> ::= { <prog def> | <proc decl> }
```

A program definition identifies a sequential program that is to be used within a distributed program. It includes in-ports, out-ports, the name of a programming language in which the program is written, and the name of a file containing the object code. Port names should be identical to the ones used within the body of a program. The format of a program definition is as follows:

```
<prog def> ::= program <prog id> in <prog lang> at <file>
                       <port list>
               end <prog id>
```

Instances of programs are created by process declaration statements. The format of a process declaration statement is as follows:

```
<proc decl> ::= process <proc ids> : <prog id>
                                   [ priority <const> ]
```

`<proc ids>` is a list of simple identifiers or arrays with constant bounds. The priority of processes can be defined using the optional priority clause.

After processes are declared, port connections are defined by link statements. The format of a link statement is as follows:

```
<port link> ::=
      link <ports1> to <ports2> [ one-to-one ]
                               [ priority <const> ]
```

The ports specified in `<ports1>` must be out-ports and the ports specified in `<ports2>` must be in-ports. As discussed before, their types should be the same. Each port listed in `<ports1>` is connected to all ports listed in `<ports2>` unless an optional clause "one-to-one" is specified. One-to-one defines pair-wise connection; therefore, there should an equal number of ports included within `<ports1>` and `<ports2>` if one-to-one is specified. Permitted port connections are one-to-one, one-to-many, many-to-one, and many-to-many. Port identifiers may be qualified by process names to resolve ambiguity. The priority of port connections can be specified using the optional priority clause.

Real-Time Constraints

When real-time processes become ready to execute, they must be executed within their deadlines. Note that it is not a process that is time-critical, but a task to be carried out by the process as a response to some event which is sent and received as a message. Since a real-time process can be blocked only if it is waiting on a SendWait or ReceiveWait system call, a real-time task is declared by associating a deadline with a port. The format of a real-time task declaration is as follows:

```
<task decl> ::= task <port ids>
                interrupt level <const>
                execution time <time>
                deadline <time>
                <interval>
<interval> ::= periodic <time> | sporadic <time>
<time> ::= <const> [ sec | msec | usec ]
```

The interrupt level is a hardware interrupt level for the task. The execution time is a maximum duration of task execution; that is, once the task is started, the process should block itself before the specified time or it may be halted by the process scheduler. The deadline defines the latest time the task can be executed after it is enabled. For a periodic process, `<interval>` defines its period. For a sporadic task, it defines a minimum interval between consecutive readiness of the task. The interrupt level, execution time, deadline and interval are used to check the possibility of task overruns at compile-time.

Process Allocation Constraints

To execute a distributed program, its component processes have to be assigned and then loaded into processors. There are two ways to specify assignment constraints. If a process should be executed on a particular processor, this constraint can be stated using a processor preference statement, which is defined as follows:

```
<processor preference> ::=
          assign <proc ids> [on <node ids>]
                            [with <attributes>]
                            [at <location>]
```

As with resource requirements, processors can be identified by its attributes. `<location>` can be specified if a process should be loaded into a particular place in memory.

Sometimes it is useful to specify where processes can be assigned with respect to each other. For example, the programmer may wish to assign several processes into the same processor to reduce ipc overhead or into different processors so that they can be executed in parallel. These constraints can be specified as follows:

```
<process grouping> ::=
        place <proc ids> together |
        place <proc ids> separately
```

The DICON compiler computes process assignments from the resource requirements, process specifications, *real-time constraints, and assignment constraints.*

Process Start Statements

The last part of a configuration written in DICON is *process start statements.* They *specify the order of* process creations at run-time and arguments to be passed when processes are started.

User Commands

A set of user commands are available to the programmer to interactively control and monitor the execution of processes for debugging. They are the pause, resume, stop, trace, whereis, and status commands. The pause command temporarily halts the execution of processes specified in the command. A port identifier may be included within a pause command to halt a process when it invokes an ipc system call on a specified port. Otherwise, processes are halted immediately. The resume command resumes the execution of halted processes. The stop command terminates some or all processes of the distributed program. The trace command traces ipc activites by printing a message to the terminal connected to the host machine whenever a specified process send or receive a message. Port names may be included to limit the tracing to particular ports. The whereis command returns a machine identifier on which a process is currently loaded. The status command returns the status of a process.

Implementation Structure

DPS consists of a set of program translaters, a DICON compiler, a loader, a resource manager, ipc servers, and run-time monitors. Each program translater consists of a preprocessor and a compiler or interpreter. The preprocessor is used to check the consistency of port uses within a sequential program. The DICON compiler processes the configuration specification of a distributed program. It ensures the consistency of port connections and generates a port connection table for each process. The process allocator takes the resource requirements, real-time constraints, and process assignment constraints of a configuration and then generates a process assignment map, which describes how processes should be assigned to processors. The process allocator also ensures that all real-time constraints can be satisfied and generates scheduling information to be used at run-time.

When a compiled configuration is executed, the resource manager is invoked to reserved necessary system resources. After the required system resources have been allocated, processes are loaded into processors according to the process assignment map. After the processes have been started, the ipc servers handle transmission and delivery of messages. The run-time monitors detect deadlock and respond to user commands such as halt a process, resume a process, stop the distributed program.

We are currently implementing the distributed programming system described in this paper on a network of three VAX 11/750s connected through Ethernet. In the prototype system, two VAX 11/750's will run Berkeley 4.2 and one VAX 750 will run the DPS kernel. We plan to port the DPS kernel into super-microcomputers (either MicroVAX or Motorola 68000 based systems).

Conclusions

We have described a distributed programming system that is designed to facilitate the construction and execution of a real-time distributed program (e.g., a multi-sensory system) from a set of sequential programs, possibly written in different programming languages. The distributed programming system is based on a distributed configuration language, called DICON, which is used to specify the resource requirements, process declarations, real-time constraints, and process assignment constraints of a distributed program. The design goals of the distributed programming system are

1. to facilitate the modular construction of a distributed program from separately constructed sequential programs;
2. to allow the programmer to specify real-time and process assignment constraints of a distributed program;
3. to help the loading and execution of a distributed program on the network; and
4. to allow the dynamic control of process execution to aid debugging.

We feel that the distributed programming system will make the development of a real-time distributed program amenable to applications (versus systems) programmers. Furthermore, the specifications of resource requirements, real-time constraints and process assignment constraints in addition to process declarations and port link definitions as part of a configuration will aid the understanding of a real-time distributed program.

Acknowledgements

The author gratefully acknowledge the contributions made by members of the DPS design and implementation group, especially N. Batchelder, Q. Chao, and G. Holder. The paper was improved by the comments of the referees. This research was supported in part by NSF MCS-8219196-CER.

References

1. Abramowitz, J., Goodnow, J. and Paul, B. PAMH - The Pennsylvania Automated Mechanical Hand. Proc. Int. Conf. on Computers in Engineering, ASME, IEEE, August 1983.

2. Balanchandran, V., McCredie, J.W. and Mikhail, V.I. Models of the Job Allocation Problem in Computer Networks. Proc. COMPCON Fall, 1976.

3. Chu, W.W. "Optimal File Allocation in a Multiple Computing System." *IEEE-Tran. on Comp. C-18*, 10 (October 1969), 885-889.

4. DeRemer, F. and Kron, H.H. "Programming-in-the-Large Versus Programming-in-the-Small." *IEEE-TSE SE-2*, 2 (June 1976), 80-86.

5. Dertouzos, M. Control Robotics: the procedural control of physical processes. Proceedings of the IFIP Congress, 1974, pp. 807-813.

6. Drummond, M., Mcmullen, C. and Vasudevan, R. Packrat - A Real Time Kernel for Distributed System. Proc. Real-Time Systems Symposium, 1982, pp. 151-154.

7. Efe, K. "Heuristic Models of Task Assignment Scheduling in Distributed Systems." *Computer 15*, 6 (June 1982), 50-56.

8. Franta, W.R., Jensen, D., Kain, R.Y. and Marshall, G.D. Real-Time Distributed Computing Systems. In *Advances in Computers*, Academic Press, 1981, pp. 40-82. Edited by M.C. Yovits

9. Gylys, V.B. and Edwards, J.A. Optimal Partitioning of Workload for Distributed Systems. COMPCON Fall, 1976, pp. 353-357.

10. Kerridge, J.M and Simpson, D. "Three Solutions for a Robot Arm Controller Using Pascal-Plus, occam and Edison." *Software--Practice and Experience 14* (1984), 3-15.

11. LeBlanc, R.J. and Maccabe, A.B. The Design of a Programming Language Based on Connectivity Network. Proc. 3rd Int. Conf. on Distributed Computing Systems, 1982, pp. 532-541.

12. Lee, I. DICON: A Distributed Configuration Language. The GRASP Group Internal Memo, Department of Computer and Information Science, University of Pennsylvania, Phila, PA

13. Lee, I. and Bachelder, N. IPC Supporting Typed Messages and Transmission of Arbitrary Data Structures. In preparation.

14. Lee, R.P. and Muntz, R.R. On the Task Assignment Problem for Computer Network. Proc. 10th Hawaii Int. Conf. Syst. Sciences, January, 1977, pp. 5-9.

15. Lesser, V., Serrain, D. and Bonar, J. PCL - A Process Oriented Job Control Language. Proc. 1st Int. Conf. on Distributed Computing Systems, 1979, pp. 315-329.

16. Light, R.A. A Real-Time Executive for Multiple Microprocessor Systems. Proc. Real-Time Systems Symposium, 1982, pp. 143-150.

17. Ma, P.R., Lee, E.Y.S. and Tsuchiya, M. "A Task Allocation Model for Distributed Computing Systems." *IEEE-Tran. on Comp. C-31*, 1 (January 1982), 41-47.

18. Magee, J. and Kramer, J. Dynamic Configuration for Distributed Real-Time Systems. Proc. Real-Time Systems Symposium, December 6-8, 1983, pp. 277-288.

19. Mok, A.K. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. Tech. Rept. MIT/LCS/TR-297, Ph.D. Dissertation, MIT, 1983.

20. Rao, G.S., Stone, H.S. and Hu, T.C. "Assignment of Tasks in a Distributed Processor System with Limited Memory." *IEEE-Tran. on Comp. C-28*, 4 (April 1979), 291-298.

21. Stone, H.S. "Critical Load Factors in Distributed Systems." *IEEE-TSE SE-4* (May 1978), 254-258.

22. Stone, H.S. and Bokhari, S.H. "Control of Distributed Processes." *Computer 11*, 7 (July 1978), 97-106.

23. Wirth, N. "Toward a Discipline of Real-Time Programming." *Comm. of ACM 20*, 8 (August 1977), 577-583.