

Technical Report

CMU/SEI-89-TR-8
ESD-TR-89-89-016

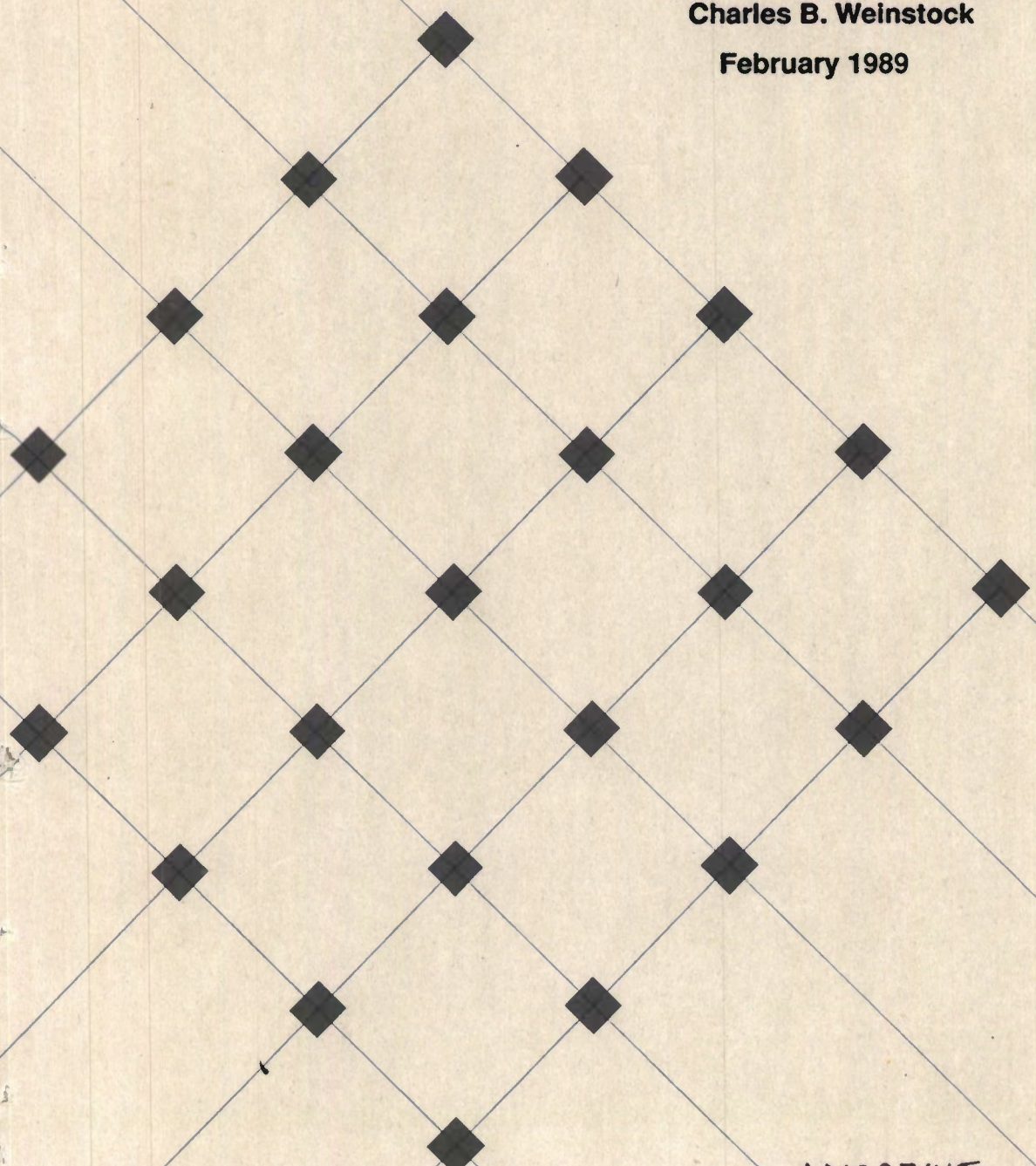


Carnegie-Mellon University
Software Engineering Institute

**Performance and Reliability Enhancement
of the Durra Runtime Environment**

Charles B. Weinstock

February 1989



ADA207415

Technical Report

CMU/SEI-89-TR-8

ESD-TR-89-016

February 1989

**Performance and Reliability Enhancement
of the Durra Runtime Environment**



Charles B. Weinstock

Software for Heterogeneous Machines Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

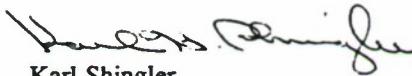
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1989 Carnegie Mellon University

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1. Introduction	1
2. Introduction to Durra	5
2.1. Task Descriptions	5
2.1.1. Interface Information	5
2.1.2. Attribute Information	7
2.1.3. Behavioral Information	7
2.1.4. Structural Information	8
2.2. Scenario	9
2.3. Runtime Components	10
2.3.1. The Scheduler	10
2.3.2. The Server	11
2.3.3. Application Tasks	11
3. Critique of the Present Runtime Environment	13
3.1. Load Balancing Problems	13
3.2. Reliability Problems	13
4. Preview	15
4.1. Load Balancing Enhancements	16
4.2. Reliability Enhancements	16
4.3. Other Possible Enhancements	17
5. Load Balancing Enhancements	19
5.1. Literature Review	19
5.2. The Load Balancing Difficulty in Durra	21
5.3. De-Centralizing Communication	21
5.4. Task Allocation	22
5.5. Cost of Communication	23
6. Reliability Enhancements	25
6.1. Literature Review	25
6.1.1. Replicated Systems	25
6.1.2. Reliable Communication Protocols	27
6.2. Reconfiguration	28
6.2.1. Reconfiguration in Durra	28
6.2.2. Problems with the Durra Reconfiguration Statement	29
6.2.3. The Global Executive	29
6.2.4. How the New Configuration Is Determined	31
6.2.5. When Is It Safe to Reconfigure?	32
6.2.6. The Reconfiguration Process	32

6.3. Adding Fault-Tolerance	33
6.3.1. Durra Level Fault-Tolerance	33
6.3.2. Application Level Fault-Tolerance	35
7. Summary	37
Appendix A. New or Redefined Remote Procedure Calls	39
A.1. Application Remote Procedure Calls	39
A.2. Global Executive Remote Procedure Calls	39
References	43

List of Figures

Figure 1-1:	The Original Durra Runtime Environment	2
Figure 1-2:	The New Durra Runtime Environment	3
Figure 2-1:	Scenario	6
Figure 2-2:	A Template for Task Descriptions	7
Figure 2-3:	Structural Information	8
Figure 4-1:	Issues in Performance and Reliability in Durra	15
Figure 5-1:	Overlapping Processor Classes	21
Figure 6-1:	A Sample Reconfiguration Statement	28
Figure 6-2:	Implementation of N-Way Voting	36

Performance and Reliability Enhancement of the Durra Runtime Environment

Abstract: Durra is a language designed to support PMS-level programming. PMS stands for Processor Memory Switch, the name of the highest level in the hierarchy of digital systems. An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

A runtime environment for Durra has been operational for some time. There are two major problems with this initial implementation: it makes no significant attempt to tune the performance of the system, and reliability has not been designed into the system. This report describes a new design for the Durra runtime environment that addresses these two issues. The new runtime environment consists of two major components: a *local executive* which runs on every processor and is responsible for process and queue management, and a *global executive* which runs replicated on several processors and is responsible for configuration management and reliability services.

1. Introduction

Durra is a language designed to support PMS-level programming. PMS stands for Processor Memory Switch, the name of the highest level in the hierarchy of digital systems. An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

Execution of Durra processes is under control of the Durra runtime environment, an initial implementation of which has been successfully running for some time. The environment consists of three active components: the application tasks, the Durra server, and the Durra scheduler. After compiling the type declarations, the component task descriptions, and the application description, the application can be executed by starting an instance of the server on each processor, starting an instance of the scheduler on one of the processors, and downloading the component *task implementations* (i.e., the programs) to the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application. Figure 1-1 shows the structure of the present runtime environment.

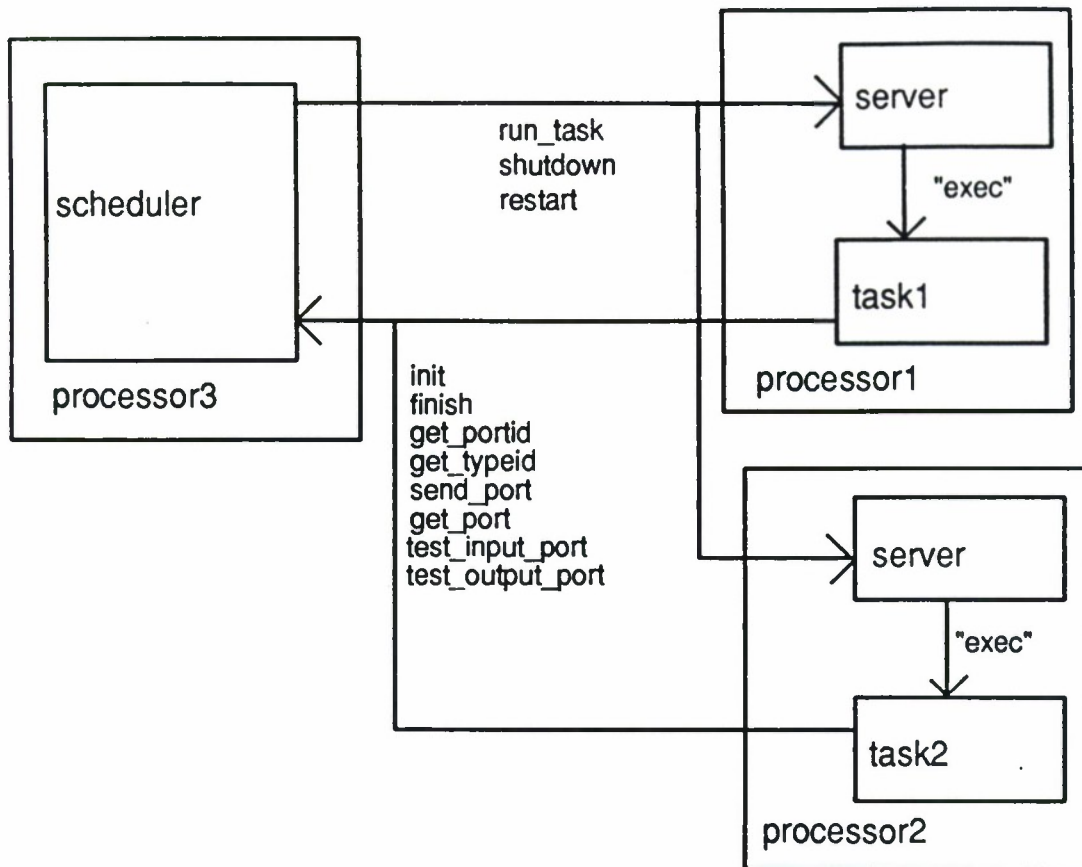


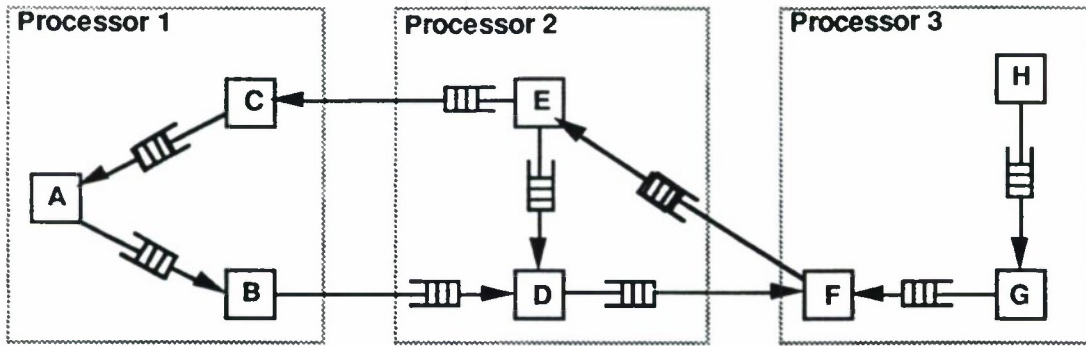
Figure 1-1: The Original Durra Runtime Environment

In the first implementation, some essential features of Durra were omitted. Deficiencies in the present Durra runtime environment include:

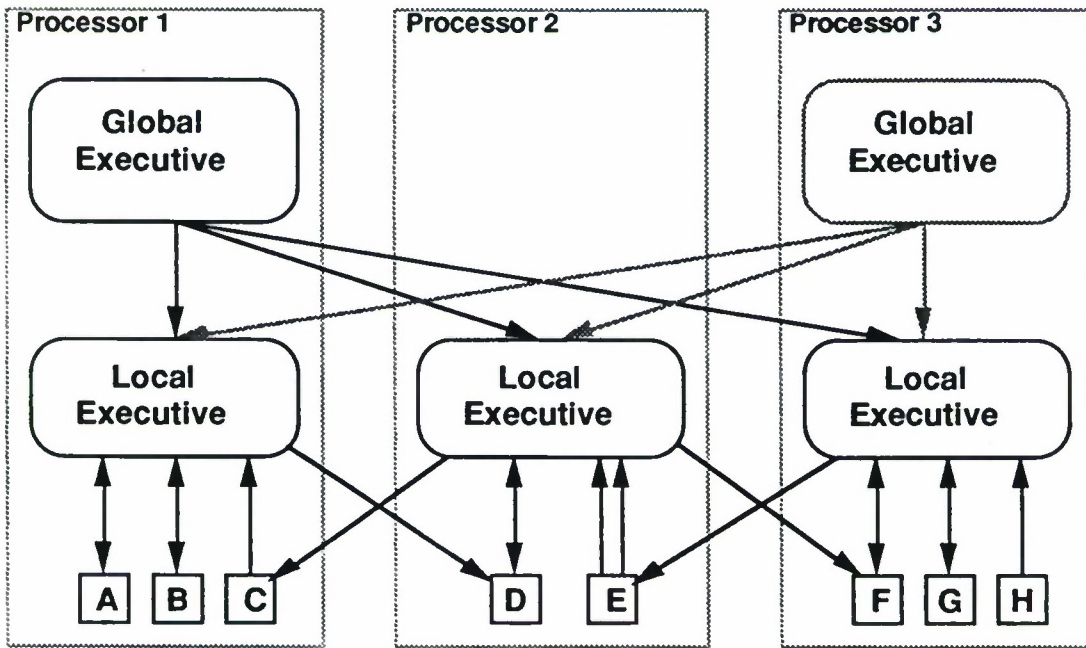
- Not all features of the Durra language are supported, most notably reconfiguration.
- No serious attempt has been made to balance processor and communication load. The scheduler simply assigns an equal number of tasks to all processors.
- Reliability has not been designed into the system. The centralized scheduler represents a single point of failure.

This report presents a design for a new implementation of the Durra runtime environment to address these concerns. The new design replaces most functions of the Scheduler and the Server with a new process called the *local executive* which executes on each processor, and introduces the *global executive* which executes on some processors and provides reliability services. The new structure is depicted in Figure 1-2.

Figure 1-2.a shows a process graph for a hypothetical application. Figure 1-2.b shows the



a -- Process Graph with Processor Allocation



b -- Actual Communication Patterns

Figure 1-2: The New Durra Runtime Environment

actual communication patterns between these processes through the local executive. There are two global executives represented in the Figure, one active and one passive as described in Section 6.3.1.1.

The remainder of this report begins with a brief description of the Durra language and its current runtime environment. Readers already familiar with Durra may want to skip ahead

to Chapter 3. Chapter 3 critiques the current Durra runtime environment. Chapter 4 is an overview of the design for the new Durra runtime environment as presented in Chapters 5 and 6. Chapter 5 considers load balancing enhancements to the Durra runtime environment. Chapter 6 describes how to make the Durra runtime environment reliable. It begins with a discussion of implementing the reconfiguration mechanism already in Durra, and concludes with a discussion of adding a degree of fault-tolerance once the reconfiguration mechanism is in place. Chapter 7 summarizes the report and suggests directions for future enhancements to the runtime environment.

2. Introduction to Durra

Durra [Barbacci 86, Barbacci 88a] is a language designed to support PMS-level programming. PMS stands for Processor Memory Switch, the name of the highest level in the hierarchy of digital systems introduced by Bell and Newell in [Bell 71]. An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term "description language" rather than "programming language" to emphasize that a Durra application is not translated into object code in some kind of executable (conventional) "machine language" (the domain of the Instruction Set Processor or ISP level introduced in [Bell 71]). Instead, a Durra application is a description of the structure and behavior of a logical machine to be synthesized into resource allocation and scheduling directives, which are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a heterogeneous machine (the domain of PMS). This is the translation process depicted in Figure 2-1.a.

2.1. Task Descriptions

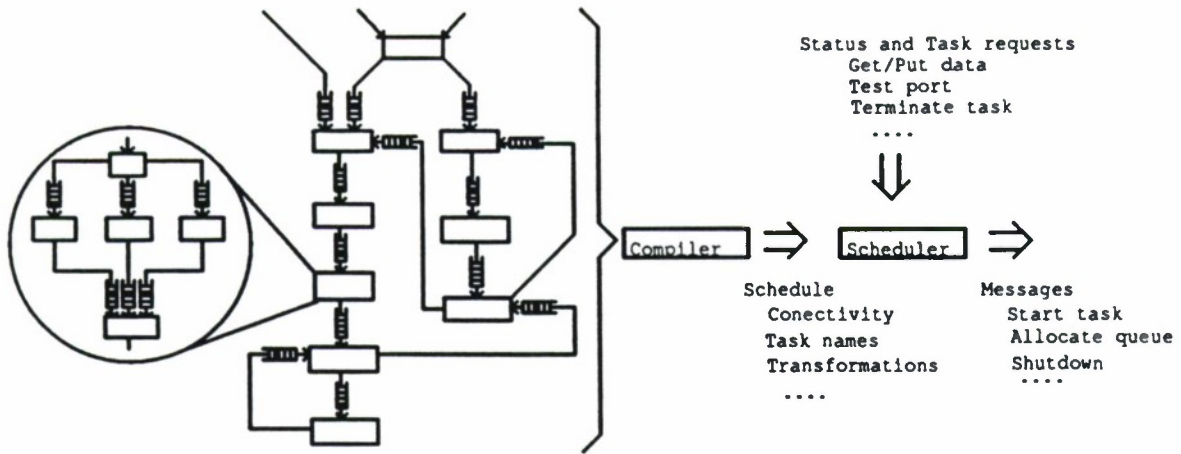
Task descriptions are the building blocks for applications. Task descriptions include the following information (Figure 2-2): (1) its interface to other tasks (**ports**) and to the scheduler (**signals**); (2) its **attributes**; (3) its functional and timing **behavior**; and (4) its internal **structure**, thereby allowing for hierarchical task descriptions.

2.1.1. Interface Information

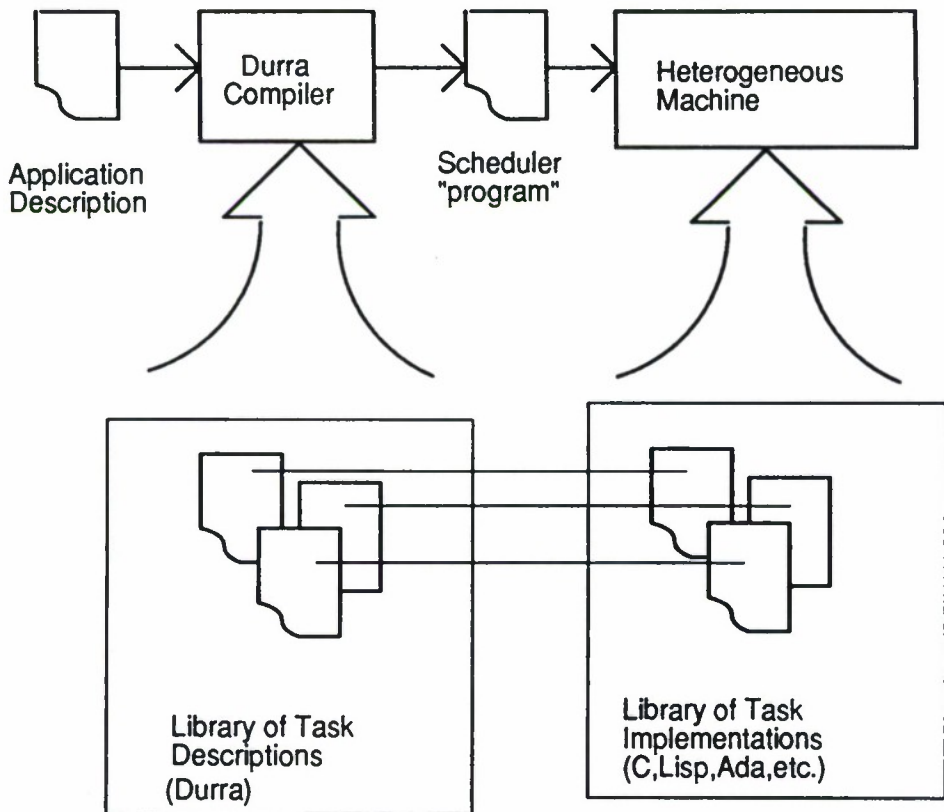
The interface information defines the ports of the processes instantiated from the task and the signals used by these processes to communicate with the scheduler:

```
ports
  in1: In heads;
  out1, out2: Out tails;
signals
  stop, start, resume: In;
  range_error, format_error: out;
```

A port declaration specifies the direction and type of data moving through the port. An **In** port takes input data from a queue; an **out** port deposits data into a queue. A signal declaration specifies only the direction of the scheduler messages. An **In** signal is a message that a process can receive from the scheduler; an **out** signal is a message that a process can send to the scheduler; an **In out** signal is used for both directions of communication.



a -- Compilation of a PMS-Level Program Graph



b -- Developing a Durra Application

Figure 2-1: Scenario

```

task task-name
  ports          -- Used for communication between a process and a queue
    port-declarations

  signals        -- Used for communication between a process and scheduler
    signal-declarations

  attributes     -- Used to specify miscellaneous properties of the task
    attribute-value-pairs

  behavior       -- Used to specify task functional and timing behavior
    requires predicate
    ensures predicate
    timing timing expression

  structure     -- A graph describing the internal structure of the task
    process-declarations  --Declaration of instances of internal subtasks

    bind-declarations     -- Mapping of internal ports to this task's ports

    queue-declarations    -- Means of communication between processes

    reconfiguration-statements  -- Dynamic modifications to the structure
end task-name

```

Figure 2-2: A Template for Task Descriptions

2.1.2. Attribute Information

The attribute information specifies miscellaneous properties of a task. Attributes are a means of indicating pragmas or hints to the compiler and/or scheduler. In a task description, the developer of the task lists the actual value of a property; in a task selection, the user of a task lists the desired value of the property. Example attributes include author, version number, programming language, file name, and processor type:

```

attributes
  author = "jmw";
  implementation = "program_name";
  Queue_Size = 25;

```

2.1.3. Behavioral Information

The behavioral information specifies functional and timing properties about the task. The functional information part of a task description consists of a pre-condition on what is required to be true of the data coming through the input ports, and a post-condition on what is guaranteed to be true of the data going out through the output ports. The timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports. For additional information about the syntax and semantics of the functional and timing behavior description, see the Durra reference manual [Barbacci 86].

2.1.4. Structural Information

The structural information defines a process-queue graph (e.g., Figure 2-1.a) and possible dynamic reconfiguration of the graph. Three kinds of declarations and one kind of statement can appear as structural information. This is illustrated in Figure 2-3, which shows the Durra (i.e., textual) version of the example in Figure 2-1.a.

```
task ALV
  ports
    in1, in2: In map_database;
    in3: In destination;
  structure
    process
      navigator:          task navigator
                          attributes author = "jmw";
                          end navigator;
      road_predictor:     task road_predictor;
      landmark_predictor: task landmark_predictor;

      . . . . .
      ct_process:        task corner_turning;
    queue
      q1: navigator.out1    >> road_predictor.in2;
      q2: navigator.out2    >> landmark_predictor.in1;

      . . . . .
      q12: position_computation.out2 >> landmark_predictor.in2;
    bind
      in1 = road_predictor.in1;
      in2 = navigator.in1;
      in3 = navigator.in2;
  end ALV;
```

Figure 2-3: Structural Information

A process declaration of the form

```
process_name : task task_selection
```

creates a process as an instance of the specified task. Since a given task (e.g., convolution) might have a number of different implementations that differ along different dimensions such as algorithm used, code version, performance, processor type, the task selection in a process declaration specifies the desirable features of a suitable implementation. The presence of task selections within task descriptions provides direct linguistic support for hierarchically structured tasks.

A queue declaration of the form

```
queue_name [queue_size]: port_name_1 > data_transformation > port_name_2
```

creates a queue through which data flow from an output port of a process (*port_name_1*) into the input port of another process (*port_name_2*). Data transformations are operations applied to data coming from a source port before they are delivered to a destination port.

A port binding of the form

```
task_port = process_port
```

maps a port on an internal process to a port defining the external interface of a compound task.

A reconfiguration statement of the form

```
if condition then  
    remove process-names  
    process process-declarations  
    queues queue-declarations  
end if;
```

is a directive to the scheduler. It is used to specify changes in the current structure of the application (i.e., process-queue graph) and the conditions under which these changes take effect. Typically, a number of existing processes and queues are replaced by new processes and queues, which are then connected to the remainder of the original graph. The reconfiguration predicate is a Boolean expression involving time values, queue sizes, and other information available to the scheduler at runtime.

2.2. Scenario

We see three distinct phases in the process of developing an application using Durra: the creation of a library of tasks, the creation of an application using library tasks, and the execution of the application. These three phases are illustrated in Figure 2-1.b.

During the first phase, the developer of the application writes descriptions of the data types (image buffers, map database queries, etc.) and of the tasks (sensor processing, feature recognition, map database management, etc.).

Type declarations are used to specify the format and properties of the data that will be produced and consumed by the tasks in the application. As we will see later in this section, tasks communicate through typed ports; and for each data type in the application, a type declaration must be written in Durra, compiled, and entered in the library.

Task descriptions are used to specify the properties of a task implementation (a program). For a given task, there may be many implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes. As in the case of type declaration, for each implementation of a task, a task description must be written in Durra, compiled, and entered in the library. A task description includes specifications of a task implementation's performance and functionality, the types of data it produces or consumes, the ports it uses to communicate with other tasks, and other miscellaneous attributes of the implementation.

During the second phase, the user writes an *application description*. Syntactically, an application description is a single task description and could be stored in the library as a new

task. This allows writing of hierarchical application descriptions. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands or instructions to be interpreted by the scheduler.

During the last phase, the scheduler loads the task implementations (i.e., programs corresponding to the component tasks) into the processors and issues the appropriate commands to execute the programs.

2.3. Runtime Components

There are three active components in the Durra runtime environment: the application tasks, the Durra server, and the Durra scheduler. Figure 1-1 shows the relationship among these components.

After compiling the type declarations, the component task descriptions, and the application description, as described previously and illustrated in Figure 2-1, the application can be executed by performing the following operations:

1. The component task implementations (Chapter 2.3.3) must be stored in a special directory in the appropriate processors. The directory name is known to the Durra servers and scheduler.
2. An instance of the Durra server (Chapter 2.3.2) must be started in each processor.
3. The scheduler (Chapter 2.3.1) must be started in one of the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

2.3.1. The Scheduler

The scheduler is the part of the Durra runtime system responsible for starting the tasks, establishing communication links, and monitoring the execution of the application. In addition, the scheduler implements the predefined tasks (**broadcast**, **merge**, and **deal**) and the data transformations described in [Barbacci 86]. The scheduler is invoked with the name of the file containing the scheduler instructions generated by the Durra compiler. A complete description of the scheduler instructions can be found in [Barbacci 88b].

After these instructions have been read and processed, the scheduler is ready to start the execution of the application. In the current UNIX implementation, this is done by performing the following steps:

1. Allocate a UNIX socket for communication with the application tasks. A UNIX socket is a special intertask communications port defined by the UNIX operating system.
2. Establish communication with each of the processors running a Durra server (Chapter 2.3.2).

3. For each of the **task_load** instructions, issue to the appropriate server a **run_task** remote procedure call (Chapter 2.3.2).
4. Listen in on the UNIX socket allocated in the first step for remote procedure calls from the application tasks (Chapter 2.3.3).
5. Process the remote procedure calls from the application tasks (Section 2.3.3).

The scheduler waits until all tasks have completed their execution before it, in turn, finishes its execution.

2.3.2. The Server

The server is responsible for starting tasks on its corresponding processor, as directed by the scheduler. One instance of the server must be running on each processor that is to (potentially) execute Durra tasks.

When a server begins execution, it listens in on a predetermined socket for messages from the scheduler. Once a communication channel is open, the scheduler communicates with the server using a set of remote procedure calls to initiate task execution (**run_task**), or to shutdown or restart the server (**shutdown**, and **restart**). Complete details of these remote procedure calls can be found in [Barbacci 88b]. The server sits in a loop responding to the requests from the scheduler, executing them as directed.

2.3.3. Application Tasks

The component task implementations making up a Durra application can be written in any language for which a Durra interface has been provided. As of this writing, there are Durra interfaces for both C and Ada. The complete interfaces appear in [Barbacci 88b].

When a task is started, the scheduler supplies it with the following information (via a server): the name of the host on which the scheduler is executing, the UNIX socket on which the scheduler is listening for communications from the task, a small integer to be used in identifying the task, and an application specific string as specified in the "source" attribute in the task description. The first three parameters are necessary to establish proper communication with the scheduler. The source parameter is provided for the convenience of the task implementation. These parameters are provided to the task by the server, which in turn obtains them, via the **run_task** instruction, from the scheduler (See Section 2.3.2).

Application tasks use the interface to communicate with other tasks. From the point of view of the task implementation, this communication is accomplished via procedure calls, which return only when the operation is completed. The following remote procedure calls (RPCs) are provided:

init	Opens a connection to the scheduler.
finish	Informs the scheduler that the task is terminating.
get_portid	Returns a descriptor for the application task to use when referring to a port.

get_typeid Returns a descriptor for the application task to use when referring to a type.

send_port Sends through an output port of the application task.

get_port Gets data from an input port of the application task.

test_input_port Tests whether data is available on an input port.

test_output_port Tests whether there is room in a queue attached to an output port (e.g., whether the process will block if doing a **send_port**).

Using this collection of scheduler calls, Durra tasks typically would exhibit the following behavior:

1. Call the **Init** function to establish communication with the scheduler.
2. Call **get_portid** for each of the task ports (these ports must correspond to the ports used in the task description).
3. Call **get_typeid** for each of the task types (these types must correspond to the data types used in the task description).
4. Call **send_port** and **get_port** as necessary to send and receive data.
5. Call **finish** to break communication with the scheduler.

3. Critique of the Present Runtime Environment

There are two main problems with the current implementation of the Durra runtime system that will be addressed in the new one. The first, load balancing, refers to ensuring that processing and communication resources are utilized effectively. The second, reliability, refers to being able to detect and adapt to problems in the execution environment. This section critiques the current Durra runtime environment with respect to these problems.

3.1. Load Balancing Problems

To effectively use a heterogeneous machine network, processes must be allocated to processors in such a manner that no processor is overloaded. This would seem to dictate maximal dispersion of the processes. However, the more dispersed the processes, the higher the potential inter-process communications load. Minimizing communication load dictates minimal dispersion of the processes. These two criteria are in fundamental conflict so the goal becomes one of minimizing the total load; applying some balance to processor load and communication load.

Because Durra is designed for heterogeneous machines, particular processes may only be allocated to particular subsets of the processors in the configuration as specified by the *processor* attribute in the task description. This both simplifies the allocation decision in the sense that it limits choices, and complicates it in the sense that it limits flexibility of process assignment.

The present implementation of the runtime environment effectively ignores all of these issues. Although it tries to equalize the count of processes running on each of the processors, it makes no attempt to minimize load, either processing or communication.

Chapter 5 discusses this problem and how it will be solved in the new Durra runtime environment.

3.2. Reliability Problems

As currently implemented, the Durra runtime environment ignores most issues of reliability. Although it uses a reliable communication protocol (TCP/IP), and is a reasonably robust piece of software, it provides no reliability functionality either at the system or application level.

The current Durra runtime environment suffers from two potential reliability problems:

1. There is a single Scheduler for the entire system. A failure of this Scheduler effectively aborts the entire application.

2. Because all communication is routed through the Scheduler, it is a potential bottleneck. While this is also an issue related to communication load balancing, the bottleneck could also potentially effect application reliability in terms of the timeliness of inter-process communication.

In addition the reconfiguration mechanism, a major method of achieving reliability has not been implemented in the current Durra runtime environment.

Chapter 6 discusses these issues in detail and presents the approach taken to achieve reliability in the new Durra runtime environment.

4. Preview

There are two major areas of improvement in the design of the new Durra runtime environment, load balancing enhancements and reliability enhancements. By necessity, the material is complex. The purpose of this Chapter is to guide the reader through the material discussed in detail in Chapters 5 and 6.

	Section	
Load Balancing		
• Processing Load	•	5.2, 5.4
• Communication Load	•	5.2-3
• System Bottlenecks	•	5.3
• Cost	•	5.5
Reliability		
• Reconfiguration:		
- Keeping the Graph Connected	•	6.3.2
- Insufficient Trigger Conditions	•	6.3.3
- Safely Relocating a Process	•	6.3.5
• Fault-Tolerance		
- Single Point Failures	•	6.4.1
- Detecting Failure	•	6.4.1-2
- Failure Recovery	•	6.4.1-2
- Reliable Communication	•	6.4.2

Figure 4-1: Issues in Performance and Reliability in Durra

Figure 4-1 shows the issues that have been considered in designing the new Durra runtime environment, and shows the specific sections of the report which discuss them.

4.1. Load Balancing Enhancements

Load balancing enhancements involve improving processor loads and communication loads. Section 5.2 discusses why existing algorithms for improving processor loads will not work without modification. Section 5.4 discusses the modifications necessary to address these problems. Removing the communication bottleneck represented by the single scheduler is discussed in Section 5.3. This is accomplished by replacing the single scheduler with a *local executive* executing on each processor. Each local executive will take responsibility for implementing some of the queues in the application and will only handle communication through those queues. Finally, a brief discussion of the costs and benefits of choosing a common protocol like TCP is discussed in Section 5.5.

4.2. Reliability Enhancements

Sections 6.2 and 6.3 discuss the proposed design for reliability in the Durra runtime environment.

Reconfiguration in Durra provides the application designer with a means of changing the way processes are interconnected at runtime. This may involve removing some processes and queues and instantiating new ones. The intent of this part of the language is to allow the application to react to events or change modes of operation. Reconfiguration may also be necessary as the result of a processor failure. In this case, the Durra runtime environment is responsible for initiating it.

Section 6.2.1 describes the reconfiguration statement in Durra and gives an example of its use. Section 6.2.2 discusses problems with the present model of reconfiguration, including the potential for dangling queues, and the highly restricted set of conditions which can trigger a reconfiguration.

Section 6.2.3 describes the *global executive* a new component of the Durra runtime which has responsibility for configuration management including reconfiguration. The global executive runs the load balancing algorithm, as already discussed in Section 5.4, and informs the local executives of which processes to run and which queues to implement. Section 6.2.5 discusses the problem of safely migrating a process, without losing state, as a part of a reconfiguration. The method used is similar to that in Conic [Kramer 88] involving making processes quiescent before a reconfiguration. However, given Durra's black-box view of application processes, the Conic model cannot be used intact. Application processes must cooperate at runtime and provide the information necessary to carry out a safe reconfiguration. Finally, Section 6.2.6 describes how the global executive carries out a reconfiguration.

The present implementation of the Durra runtime environment pays little attention to reliability. In fact, the present Scheduler represents a single point of failure in the Durra system. Section 6.3 extends the design of the new Durra runtime environment to provide a higher level of reliability.

Fault-tolerance in the proposed Durra runtime environment is achieved via replication of the global executive. Section 6.3.1.1 introduces this idea and describes the *active* and *passive* global executives. There is one active global executive; all the rest are passive. The active global executive keeps the passive updated with configuration information. The active and the passive global executives detect processor failure by repeated query of the local executives as described in Section 6.3.1.2. A fail-stop model [Schlichting 83] of failure is assumed. In the event that the active global executive fails, one of the passive global executives becomes active using an election protocol similar to that described in [Kim 88]. In the event that a passive global executive fails, a new passive global executive is started on a functioning processor. In either event a reconfiguration as described in Section 6.2.4 is performed.

Application level reliability, discussed in detail in Section 6.3.2 is achieved via several mechanisms including an extended reconfiguration statement as described in Section 6.2.3. Error detection is accomplished by implementation of n-way voting in the application. Also provided in the new Durra runtime environment is an atomic broadcast facility which guarantees that if any process receives a message they all do. Aborts are dealt with by the Durra runtime environment.

4.3. Other Possible Enhancements

This report does not deal with processor restart. Once a processor has failed it is presumed to be gone forever. Thus transient faults and therefore the problems of orphans and partitions [Liskov 87] are not discussed. These are important topics and will be considered in a later report.

5. Load Balancing Enhancements

This chapter of the report discusses adding true load balancing to the Durra runtime environment. It begins with a review of the literature, and continues with a discussion of task and communication load balancing techniques to be utilized.

5.1. Literature Review

The literature on load balancing can be roughly divided into two categories. The first category is static load balancing, which refers to making an assignment of processes to processors once, without considering the possibility of migrating tasks as processor loading characteristics change. Static load balancing also ignores the potential for new processes to be created during the execution of the application.

The second category is dynamic load balancing, which continually considers processor loading and migrates processes to keep the application running efficiently. The Durra process structure is very static. New processes start and old processes terminate only as the result of a reconfiguration (Section 6.2.1), making dynamic load balancing not applicable to the Durra runtime environment.

Chu et. al [Chu 80] present an overview of the problems inherent in load balancing on distributed systems. The obvious solution of splitting the processor load evenly among the processors does not result in the maximal usage of resources because of the relatively higher cost of inter-process communication when two processes run on different processors. Chu assumes that the cost of inter-process communication when the processes are co-resident is significantly less than when they are not.

As a rule assigning two processes that do not communicate with each other to different processors will typically speed up the overall computation. Assigning two processes that communicate heavily to different processors will slow the application down.

The first of three strategies for load balancing that Chu considers is a graph theoretic approach. In this approach, the nodes represent processes and the weighted arcs represent the communication behavior. The higher the weight on an arc, the more the communication cost between any two processes if they are not co-resident. The arc weight is zero if the processes are assigned to the same processor. There is also a processing cost associated with each processor/process pair, allowing for the possibility that a particular process will execute more efficiently on a particular processor. Then the cost of any particular task assignment is computed by summing up the arc weights along with the processing costs. The cost is minimized by performing a max-flow, min-cut algorithm on the graph. The graph theoretic approach is straightforward, but rapidly becomes computationally NP-hard as the number of processors increases beyond two. Furthermore it assumes infinite resources on each processor, and extending it to account for resource limits results in an NP-complete solution.

The second strategy considered involves integer 0-1 programming. This solves the resource constraint problem but also is computationally expensive and would probably require off-line computation for any particular application.

The final strategy presented involves the use of heuristics. Assuming homogeneous, fully connected processors, a clustering algorithm is proposed. The algorithm locates the two processes which will reduce inter-process communication cost the most by being assigned to the same processor. If the resources available allow the assignment to be made, that process pair is "fused" and considered as a single process for the next iteration of the algorithm. The system is considered load balanced if the loading of the processors is within a certain tolerance. While the heuristic strategy does not result in an optimal allocation of processes to processors, it does a reasonable job and it is computationally tractable.

Chou and Abraham [Chou 82] consider seven characteristics of processes in a distributed system. In addition to execution time and inter-process communication time, these characteristics include process failure probabilities, process start-up costs, etc. They model the execution of a set of processes in a distributed system using a semi-Markov process with rewards. A state in the model represents the execution of a process on a particular processor. The reward structure is used to model the time behavior of a program module. A policy iteration algorithm is applied to determine the optimal (reward maximizing) process-processor assignment. The resulting algorithm is more general than the graph theoretic approach previously described in that it can be used for an N-processor system, and that it considers the effects of system reliability on performance.

Lo [Lo 87] develops another heuristic which also makes use of the graph-theoretic approach described above. Choose a single processor in the distributed system, and consider all of the other processors to be another single processor. Perform the two processor max-flow, min-cut algorithm which will result in some processes being assigned to the chosen processor. Do this for each processor in the system. If all processes have been assigned to processors at completion of this part of the heuristic, the task assignment is complete. Otherwise, other heuristics are applied to complete the assignment.

Lo points out that all of the approaches that make use of total execution and communication costs make no explicit attempt to fully utilize the processors available, and often end up leaving some of them idle. To achieve higher degrees of parallelism, Lo introduces inter-process interference costs. Two CPU bound processes executing on the same processor will have high interference costs. A CPU bound task and an I/O bound task executing on the same processor might have a low interference cost. Interference cost, then, serves as a force of repulsion between processes to counterbalance the force of attraction due to high communication costs. The heuristic already presented, with minor modification, will also work when interference costs are included.

5.2. The Load Balancing Difficulty in Durra

The initial implementation of the Durra runtime environment routes all inter-process communication through a centralized scheduler. This means that there is no way to optimize the inter-process communication costs. With the exception of those tasks that happen to be allocated to the same processor as the scheduler, the communication costs will be high. Thus to better utilize resources in Durra, inter-process communication must be freed from going through a centralized switch.

Even de-centralizing inter-process communication will not allow any of the heuristics described in the literature to be directly applicable. In Durra the application designer specifies the set of processors a process is allowed to execute on. These sets are in general not disjoint as illustrated by Figure 5-1.

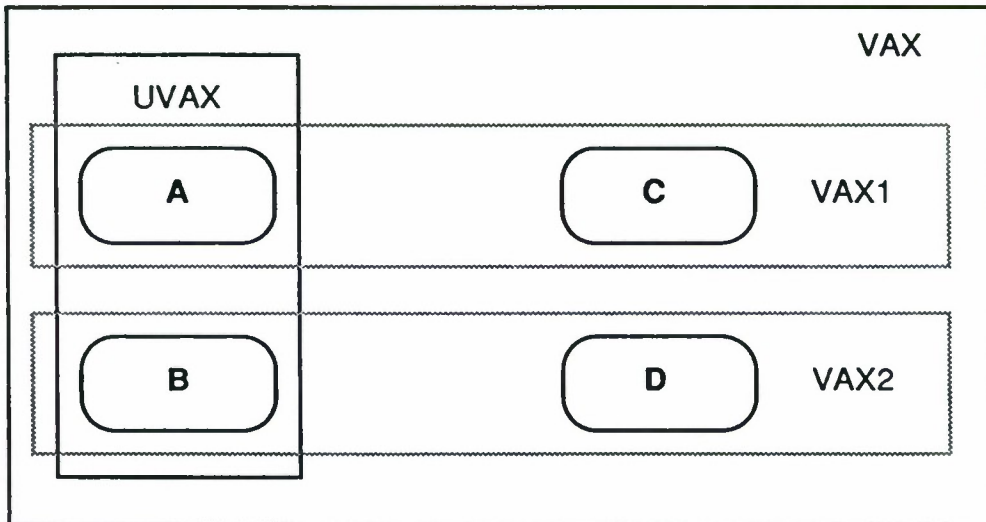


Figure 5-1: Overlapping Processor Classes

An application designer can specify that a process must run on processor A, that it can run on any MicroVAX (UVAX), that it can run on any VAX, or that it can run on VAX class one (VAX1) or VAX class two (VAX2). Since the heuristics discussed above assume a homogeneous set of processors, or at least a disjoint set of processor classes, they will have to be modified to deal with the Durra environment.

5.3. De-Centralizing Communication

To allow for more efficient inter-process communication, and to remove the bottleneck represented by the present scheduler. We propose to replace the centralized scheduler with a *local executive* which runs on all of the processors in the Durra environment. The local executive also replaces the Server on each of the machines. The local executive will have the following responsibilities:

- **process start-up and termination.** The local executive will be responsible for loading application processes on the local processor and seeing that they are started with the correct parameters. (The present Server function.)
- **queue implementation.** The local executive will implement all queues associated with the output ports of processes running on the local processor. (A portion of the current Scheduler functionality.)
- **buffer task implementation.** Broadcast and Deal buffer tasks will be implemented by the local executive on the processor where the data to be broadcast is originating. Merge buffer tasks will be implemented by the local executive on the processor where the merged data is to be used.

As an optimization, where possible (e.g., in UNIX based systems) a simplified (but reliable) communication protocol will be utilized for communication between a process and the local executive executing on the same processor. See Section 5.5.

The remainder of the functionality of the present scheduler is assumed by a new component of the Durra runtime environment, called the *global executive*. The global executive will be discussed extensively in Chapter 6.

An effect of this change is to make it reasonable to consider inter-process communication costs when allocating processes to processors.

5.4. Task Allocation

For task allocation in Durra, the heuristic described by Lo, with some modification, is appropriate. For this method to work in the Durra environment, the set of processors a process may execute on must be taken into account. This is accomplished by setting the cost of a process executing on any other processor to be infinity. Thus the heuristic will prefer to allocate those processes to the appropriate processor. The global executive will have the responsibility for carrying out this heuristic.

For the heuristic to work effectively, the inter-process communication cost, and the process interference costs must also be taken into account. This can be inferred from the behavior specification of the task descriptions when available (Section 2.1.3). In the event the application designer has not written behavior parts, a constant will be substituted for the inter-process communication cost if the processes are allocated to different processors. Similarly, a constant will be substituted for the process interference cost if the processes are allocated to the same processor. Tuning these constants will require some experimentation.

5.5. Cost of Communication

Implementing the Durra runtime environment on a heterogeneous machine network requires a common communication protocol across all of the processors of the network. Since the project was not concerned with inventing new protocols we have chosen to use TCP. This provides us with the major benefit of wide availability, which simplifies porting of the Durra runtime environment to additional host systems.

The price we pay for this simplification is one of performance. Although implementation of the TCP protocols have been optimized in some environments there are still inefficiencies caused by the protocols generality. A custom designed protocol for Durra could reduce these inefficiencies at the cost of making it harder to move the runtime environment from system to system. There is nothing in the Durra runtime environment design that precludes a change in protocol at some future time.

6. Reliability Enhancements

That a single Scheduler represents a single point of failure and is a potential communication bottleneck was effectively addressed in the previous section. But, there are two other reliability issues to be considered, reconfiguration and fault-tolerance. These issues will be discussed in the following sections.

6.1. Literature Review

For a system to be reliable it must be able to:

- avoid errors where possible,
- detect errors when they occur,
- mask errors when possible, and
- recover from errors that cannot be masked.

A major means of achieving reliability in distributed systems is via replication of program modules. Error detection and reliable communication between program modules is necessary for replication to be effective. Finally, dynamic system reconfiguration is a common means of recovering from errors that cannot be masked. This section considers some of the existing work in these areas, looking first at replicated systems, and then at reliable communication protocols.

6.1.1. Replicated Systems

Wensley et al [Wensley 78] describe the SIFT architecture. The SIFT computer is designed to operate fly-by-wire commercial aircraft and as such is ultra-reliable. A replicated task structure is used to achieve this, with n-way voting used to detect and mask errors. A summary of errors is periodically reported to a replicated global executive. The global executives use these error reports to make reconfiguration decisions. When a majority of the global executives decide on a reconfiguration, a local reconfiguration task, running on all processors (but not replicated) carries out the reconfiguration. The SIFT computational model requires all tasks to broadcast important state at the end of each iteration, and to retrieve it at the start of the next iteration. Thus restart problems are minimized. Weinstock and Green [Weinstock 78] describe the implementation of reconfiguration in the SIFT system in more detail.

Seifert [Seifert 80] discusses reconfiguration in two types of systems, those with structural redundancy (hot-standby), and those with functional redundancy. For hot-standby to work properly, the state of the back-up process must remain consistent with the state of the primary process. Upon failure of a primary process, the system activates the passive hot-standby process and attempts to repair the failed process. A repaired process becomes a new hot-standby process, ready for a subsequent failure. In a functionally redundant system, failure of a process causes the other already executing processes to assume the functionality of the failed process, possibly at the cost of some degradation in overall system

performance. Once the failed process is repaired, it is re-integrated into the system and re-assumes its old responsibilities.

Cristian [Cristian 82] compares programmed exception handling to default exception handling. The former deals with errors anticipated by the programmer in advance. In this case the programmer has inserted post-conditions in the program to detect and recover from an error. The later deals with unanticipated errors. The recovery block mechanism [Anderson 76], is used to deal with them.

Cooper [Cooper 85a] builds a reliable distributed system around replicas of modules which he calls a *troupe*. Individual members of a troupe do not communicate among themselves, and in fact don't even know of each others existence. A *replicated procedure call* mechanism, based on remote procedure call, deals with the many-to-many communication pattern between troupes. A distributed system built around troupes is reliable because the system will continue to function properly as long as one member of each troupe remains operational. A mechanism is provided for re-populating a troupe decimated by member failure.

Zicari [Zicari 86] proposes a system structure similar to that adopted by Durra. An allocation manager provides a set of "programming in the large" operations through which the user programmer can control the configuration of the system. "Programming in the small" operations, which correspond to programming the "virtual nodes" in the system are kept separate. The allocation manager allows virtual nodes to be migrated from one processor to another, complete with execution state information.

Kramer and Magee [Kramer 88] discuss the reconfiguration mechanism for the Conic system running at Imperial College. They also separate application concerns from those at the configuration level. They define the concept of a quiescent node. A node is said to be quiescent if it is not currently engaged in a transaction, will not initiate a new transaction, and if no other node will initiate a transaction that requires a response from this node. A quiescent node can pass consistent up-to-date information in the event of a reconfiguration. A passive node is one which will not initiate any transactions, but which will respond to transactions from other nodes. Both are desirable states for reconfiguration. The paper shows that to achieve the quiescent state in some target node, the configuration manager must first make it passive and also create a region of passive nodes around it. Such a node can be removed from the configuration safely. Unlinking (or linking) a node from the configuration requires that the node from which the connection is directed be in the passive state.

In many replicated systems, one process must be more equal than the others. For instance, all processes need to share consistent view of a new configuration. One means of achieving this is to require one of the processes collect the information, develop the configuration, and broadcast it to the other processes. The problem then becomes one of deciding which of the equal processes is going to take on this responsibility and become the "master." Kim and Belford [Kim 88] describe an election protocol which solves this problem. The essential idea is for each process to have a different election priority. When a system status change is detected because of a failure or recovery, an election is held wherein all processes an-

nounce their candidacy. A process seeing a candidacy from a process with higher priority abandons its candidacy and votes for the other. A two-phased commit protocol is used to ensure that all of the processes have made the same decision, even in the case of lost messages.

6.1.2. Reliable Communication Protocols

Chang and Maxemchuk [Chang 84] describe a family of reliable protocols for an unreliable broadcast network which guarantees that all of the broadcast messages are received by all operational processes in a broadcast group. It also guarantees that all processes receive messages in the same order, a feature which simplifies distributed algorithms. The different protocols trade-off the number of control messages per broadcast message, the internal storage required, and the resiliency of the system.

Cooper [Cooper 85a], already discussed above, describes replicated procedure calls which allow troupes to communicate with troupes in a reliable many-to-many fashion. When a client troupe makes a replicated procedure call to a server troupe, each member of the server troupe performs the requested procedure exactly once, and each member of the client troupe receives all the results. Performance can suffer if a server troupe member must await calls from all of the client troupe members before execution. This requirement is relaxed by allowing the server to begin operation, but to check that all requests are finally received. Similarly, a client troupe should not have to await the return of values from all members of the server troupe, but should be allowed to proceed as long as the return values are all eventually received. In [Cooper 85b] he gives a detailed implementation of replicated procedure calls.

Lin and Gannon [Lin 85] describe an atomic remote procedure call mechanism. Atomicity is a property encompassing two concepts: totality and serializability. If an atomic action completes successfully, then the action requested has taken place everywhere. If the action fails anywhere, it takes place nowhere. This is totality. Serializability requires that the effects of executing several operations concurrently be equivalent to the effects if they are executed in some sequential order. Because remote procedures can call other remote procedures, committing the operation is quite complex and may require multiple steps. For cases where atomicity is not needed, a standard remote procedure call mechanism is provided.

Birman and Joseph [Birman 87] present three communications primitives and their implementation as used in the ISIS system. These are *group broadcast*, *atomic broadcast*, and *causal broadcast*. All of these broadcast mechanisms are atomic, but they differ in other important ways. Group broadcast is intended to inform group members of changes in the global state of the group (e.g., a process joins the group, fails, etc.) Group broadcast has two additional properties of interest. First, the order in which a group broadcast is delivered relative to the delivery of all other broadcast messages (of any type) is identical at all overlapping destinations. Second, if the group broadcast is a failure message (e.g., one that is generated because of a process failure) it is required to be delivered as the last message

from that process. Once a process has failed, it will never be heard from again. Because of these properties, a process receiving a group broadcast message can act on that information without further agreement from other processes.

Atomic broadcast is provided for applications which require that the order in which data is received at a destination is same as that at all other destinations. The causal broadcast primitive is used to enforce a delivery ordering at all destinations when desired, but with minimal synchronization. It differs from atomic broadcast in that it requires a particular, predetermined ordering of delivery. These primitives can be used to implement the replicated procedure call of Cooper [Cooper 85a]. The use of the primitives to implement fault-tolerant systems (particularly ISIS) is also described.

6.2. Reconfiguration

Durra provides the application designer with a means of reacting to external stimuli (e.g., sunset or time-of-day). By making use of the reconfiguration mechanism the graph of process interconnections may be modified, by removing old processes and adding new ones, or merely by changing the way in which current processes are connected together through queues. The reconfiguration mechanism also allows an application to become more reliable by giving it a mechanism for reacting to errors detected during its operation.

The following sections consider the implementation of reconfiguration in the Durra runtime environment. First comes a discussion of the various kinds of reconfiguration that Durra must be prepared to handle. A design for a **Global Executive**, in charge of configuration management, is presented, followed by a discussion of the difficulties in accomplishing reconfiguration.

6.2.1. Reconfiguration in Durra

The Durra language includes a mechanism for reconfiguration that allows the user to specify both when to reconfigure and how to reconfigure. Figure 6-1 is an example of a reconfiguration statement in Durra.

```
if Current_Time >= 6:00:00 local and
    Current_Time < 18:00:00 local
then
    remove
        p_sonar;
    process
        p_vision: task vision attributes processor = warp;
    queues
        q_vision: p_deal.out3 >> p_vision.in1;
        q_obstacles: p_vision.out1 >> p_merge.in3;
end if;
```

Figure 6-1: A Sample Reconfiguration Statement

This example specifies that during the time between 6 am and 6 pm, a sonar process

(p_sonar) is to be removed from the application and a vision process (p_vision) is to be added. It also specifies how the new process is to be "hooked in" to the application.

In general there are three ways a reconfiguration can occur in Durra.

1. The application can require a reconfiguration as a part of its normal operations. The example above is an instance of this form of reconfiguration.
2. An external operator can direct the reconfiguration of the application in response to some external events. For instance it may be necessary to take down a processor for preventive maintenance without shutting down the application. This would require reconfiguring the application to stop using that processor.
3. A system failure can cause a reconfiguration.

In the first two cases Durra has some control over the timing of the reconfiguration, giving time to accomplish it smoothly. Reconfiguration in the third case is unpredictable and cannot be accomplished as smoothly.

6.2.2. Problems with the Durra Reconfiguration Statement

In one important way reconfiguration due to system failure can be simpler to deal with than the other two cases. If a processor fails, the graph of the configuration does not have to change, merely the allocation of processes to processors. On the other hand, the application designer may design in a graceful degradation of services in the event of a processor failure, and this might require an entirely different configuration. In this and the other two types of reconfiguration, there is a potential for the graph to become unconnected, with dangling queues being left around.

The Durra compiler checks that the graph will not become unconnected in the case in which the application designer has specified a reconfiguration statement. However, this checking is done in reference to the initial configuration, and subsequent reconfigurations may still cause an unconnected graph. It is a runtime error for the graph to be unconnected.

The present reconfiguration statement in Durra is triggered by a highly restricted set of conditions all involving time or the number of elements in a queue. Clearly this is not sufficient flexibility for being able to specify the complete range of reconfiguration conditions listed in the previous section.

6.2.3. The Global Executive

The *global executive* is a part of the Durra runtime environment responsible for Durra configuration and reconfiguration. In normal operation the global executive. functions as follows:

- **Establish Connections:** Connections are established with each of the local executives in the configuration. Once a connection is opened, each local executive returns the number of a socket on which it will listen for communication from application processes.

- **Build Graph:** After reading the interpretive instructions generated by the Durra compiler for the application, a graph representation of the configuration is built.
- **Allocate Processes:** The task allocation algorithm discussed in Section 5.4 is employed to determine an initial assignment of processes to processors.
- **Initialize the Local Executives:** Each local executive is provided the following information:
 - A list of processes to execute on its processor.
 - Descriptive information regarding the queues for which it is responsible, including process and port information. If a queue overflow condition is used in a reconfiguration statement, the local executive is given that information so that it may report the occurrence of the overflow to the global executive.
- **Process Application Requests and Reconfigurations:** Having informed the local executives of their duties, the global executive awaits messages from the application processes (via the local executive) and reconfiguration events.

The local executives must be capable of executing a set of remote procedure calls to establish a configuration. The philosophy here is that a local executive should only be given the knowledge that it needs in order to do its job. This minimizes the information that must be updated in the event of a subsequent reconfiguration. The following remote procedure calls, issued by the global executive accomplish the transfer of configuration information to a particular local executive. They closely parallel the instructions in the .SCHED file generated by the Durra compiler.

- create_buffer_task** Tells the local executive of the existence of a buffer task. A local executive only needs to know of the existence of a buffer task that will be implemented by that local executive.
- create_attribute** Is used to inform the local executive of attributes associated with a process. These attributes may include: the name of the file that implements the process, the mode of a buffer task, or the window geometry for the process to use if running under X windows. A local executive only needs to know of the attributes of a process that is executing on, or has a port connected to a queue implemented on the local processor.
- create_port** Tells the local executive of the existence of a port for a particular process. A local executive is informed of a port when it implements a queue to which that port is attached.
- create_queue** Tells the local executive to implement a particular queue (but not what ports to connect to the queue, see **attach_port** below).
- attach_port** Tells the local executive to attach a port to a queue.
- create_size_type** Tells the local executive of the existence of a simple type.
- create_array_type** Tells the local executive of the existence of an array type.
- create_union_type** Tells the local executive of the existence of an union type.
- commit_configuration** Tells the local executive to make the new configuration become effective.

abort_configuration

Tells the local executive to abort the new configuration and remain with the current one.

The remote procedure calls currently implemented by the Scheduler and issued by the application processes will be implemented by the local executive, with the exception of `get_portid` and `get_typeid` which are implemented by the global executive. When the local executive receives one of these calls from an application process it forwards it to the global executive for processing and then forwards the results back to the application process. `Get_portid` will be modified to return a host name, and a socket number for communication to that port, in addition to the information already returned. `Get_typeid` will return the same values as before.

The local executives and the global executive need to remain in communication with each other, to facilitate the reconfiguration process. A new remote procedure call, `raise_signal` is used by an application process to send a signal to the global executive (via the local executive). The conditional part of the Durra reconfiguration statement is modified to include the pre-defined boolean function:

`signal(process,signal_number)`

which is true whenever the specified process executes a `raise_signal(signal_number)`. Note that Durra places no semantics upon a particular `signal_number`. That is the responsibility of the application developer.

6.2.4. How the New Configuration Is Determined

Once the global executive determines that a reconfiguration is appropriate it performs the following analysis:

1. It determines which processes are to be removed from the application.
2. It determines which processes are to be added to the application. A migrating process is just a special case of these two steps.
3. It determines on which processor the *new* or *migrating* processes will execute. This is accomplished by running the max-flow, min-cut algorithm again, but locking existing, non-migrating processes on their original processors.
4. It determines which of the queues involved in step 1 are to be deleted, and which can be reconnected. A queue is deleted if both its input and output processes are deleted from the application. It is also deleted if the local executive which implements it terminates execution (as either the cause of, or the result of a reconfiguration.)
5. It determines which queues must be newly created, and which local executive is responsible for implementing the queue.

At this point the global executive has all of the information necessary to perform the reconfiguration.

6.2.5. When Is It Safe to Reconfigure?

The major problem with reconfiguration is preserving the state of the application. If a process is migrated at the wrong moment it is easy to lose important state information. The Conic approach of making processes quiescent before reconfiguring is that taken here. A process is quiescent if it is not currently engaged in a transaction, will not initiate a new transaction, and if no other process will initiate a transaction that requires a response from this process. The problem is that in Durra, because processes are treated as black boxes, there is currently no reliable way of determining when a process is quiescent, unless the process itself notifies the runtime system.

Although the global executive can ask the local executives to prevent a process targeted for migration from receiving data from other processes, this does not completely solve the problem. First, if the targeted process does not do many (or any) `get_ports` it will never block awaiting input, and will never be deemed quiescent. More importantly, a transaction between two processes may consist of multiple `get_ports` and `send_ports`. Thus, blocking by itself is not a reliable means of telling when a process is quiescent. The process must make this explicit.

To facilitate this, an additional remote procedure call, `safe` is provided for the application process. This call says that it is safe for the process to be migrated instead of returning from the remote procedure call. This method of determining when it is safe to reconfigure depends on the cooperation of the application programmer. Without that cooperation, it will be impossible to reconfigure a Durra application safely.

6.2.6. The Reconfiguration Process

Given the above, to accomplish a reconfiguration, the global executive issues the following remote procedure calls for execution by the local executives:

- `quiesce` Causes the local executive to make a process quiescent by not returning from a `safe` remote procedure call.
- `is_quiescent` Allows the global executive to determine if a process has successfully been made quiescent by a local executive.
- `resume` Causes the local executive to unblock a quiescent process.
- `remove_process` Causes the local executive to close all of a processes connections and to terminate its execution.
- `unattach_port` Causes a local executive to disconnect a port from a queue.
- `remove_queue` Causes the local executive to delete a queue.

Once the appropriate processes and queues have been removed from the configuration, the `create_buffer_task`, `create_port`, `create_queue`, `create_attribute`, and `attach_port` remote procedure calls defined in Section 6.2.3 are used to build the new configuration. The `commit_configuration` remote procedure call causes the new configuration to become effective.

To help the application programmer, when a process is restarted it is informed of whether it

is being run for the first time (e.g., at application start-up), is being restarted as a result of a controlled reconfiguration (e.g., the application or operator caused it to happen, and the process was made quiescent before reconfiguration), or is being restarted as the result of a system failure (e.g., the process was not made quiescent before reconfiguration).

6.3. Adding Fault-Tolerance

In order to become more reliable, the Durra runtime environment must be able to deal with errors internal to itself. This would involve, for example, starting up new Local Executives or Global Executives in the event of a failure, and the consequent restarting of application processes. Durra should also provide a mechanism to make it easier for an application designer to incorporate fault-tolerance directly into the application.

The following sections consider the addition of fault-tolerance to both the Durra runtime environment, and the applications running under it.

6.3.1. Durra Level Fault-Tolerance

In what follows, it is assumed that Durra applications and the runtime environment execute on *fail-stop processors* [Schlichting 83]. A fail-stop processor is one that halts on error instead of allowing erroneous computations to proceed. To assume otherwise would require Byzantine agreement [Pease 80, Lamport 82] among the processes at a severe performance penalty. In practice, Byzantine errors do not appear to occur frequently enough to warrant paying this penalty.

Fault-tolerance in the Durra runtime environment is achieved via replication of the global executive.

6.3.1.1. Initializing the Global Executives

A single global executive is started by the user with a parameter specifying the degree of replication desired in the global executive.

A global executive begins execution as described in Section 6.2.3 with the changes (in *italics*):

- **Establish Connections:** Connections are established with each of the local executives in the configuration. Once a connection is opened, each local executive returns the number of a socket on which it will listen for communication from application processes.
- **Start Replica Global Executives:** *The running global executive decides on which processors to run the replica global executives, and asks local executives on those processors to start them. These global executives are started in a passive mode. The original global executive is in active mode.*
- **Establish Connections:** *The passive global executives establish connection with the active global executive and the local executives.*

- **Buld Graph:** *The active global executive reads the interpretive instructions generated by the Durra compiler for the application and sends them, one at a time, to the other global executives. An atomic remote procedure call mechanism, similar to Lin's [Lin 85] is utilized to ensure that all global executives end up with the same graph representation of the configuration.*
- **Allocate Processors:** The task allocation algorithm discussed in Section 5.4 is employed *by each of the global executives* to determine an initial assignment of processes to processors.
- **Initialize the Local Executives:** Each local executive is provided *(by the active global executive)* the following information:
 - A list of processes to execute on its processor.
 - Descriptive information regarding the queues for which it is responsible, including process and port information. If a queue size is used in a reconfiguration statement, the local executive is given that information so that it may report the occurrence of the event to the *active global executive*.
- **Process Application Requests and Reconfigurations:** Having informed the local executives of their duties, the *active global executive* awaits messages from the application processes (via the local executive) and reconfiguration events.

6.3.1.2. Detecting Failure

Detection of a processor failure is accomplished by having the active global executive probe each of the local executives if there has been no communication activity from that local executive in a specified amount of time. The appropriate frequency of probing is a parameter that will be determined empirically. A failure of the active global executive is determined by having each of the passive global executives probe the local executive of the processor on which the active global executive is running (see Section 6.3.1.3.).

To accomplish the probe, a `processor_status` remote procedure call is defined on each of the local executives in the configuration.

`processor_status(time_out)`

Returns a zero if the local executive has not responded within the period of time indicated by the "time_out". Otherwise returns one.

If the local executive on a processor does not respond within the required period, the processor is presumed to have failed.

6.3.1.3. Recovering From a Failure

In order to recover from a failed processor, there are three cases to consider:

1. The failed processor was not running a global executive.
2. The failed processor was running a passive global executive.
3. The failed processor was running the active global executive.

In the first case, the active global executive informs the passive global executives of a processor failure, utilizing the atomic remote procedure call mechanism to ensure that they

all receive the same information. Then all of the global executives perform the reconfiguration process described in Sections 6.2.4 through 6.2.6. Of course, only the active global executive actually communicates information to the local executives.

In the case in which a passive global executive was running on the failed processor, before doing the above, the active global executive will select a new processor to run a replacement passive global executive, have the local executive start its execution, and initialize it.

In the case in which the active global executive ceases to operate, one of the passive global executives must become active before doing any of the above. The distributed election protocol of Kim and Belford [Kim 88] is utilized to select one of the passive global executives and make it the active global executive.

6.3.2. Application Level Fault-Tolerance

Application level fault-tolerance is left to the application developer. However, the Durra runtime environment will provide mechanisms to make the problem of building fault-tolerant applications easier.

The first of these, the ability to respond to errors by triggering a reconfiguration using the `raise_signal` remote procedure call, has already been discussed in Section 6.2.3. But, in order for the application to trigger a reconfiguration, it must be aware that an error occurred. One way of achieving error detection is via replicated application processes and n-way voting. The application designer can specify this replication when writing task descriptions and easily implement n-way voting of the results as shown in Figure 6-2.

To make programming applications with replicated tasks easier, an *atomic* mode is added to the broadcast buffer task. If any destination receives the broadcasted data, then all destinations receive the data. In the event of a failure the reconfiguration mechanism is triggered and the broadcast is retried.

```

task a;
ports
  out1:out byte;
attributes
  processor = vax;
  implementation = "a_task";
end a;

```

```

task three_way;
ports
  in1, in2, in3:byte;
attributes
  processor = vax;
  implementation = "three_task";
end three_way;

```

```

task main;
structure
  process p1: task a;
  process p2: task a;
  process p3: task a;
  process v: task three_way;
queues
  q1: p1.out1 >> v.in1;
  q2: p2.out1 >> v.in2;
  q3: p3.out1 >> v.in3;
end main;

```

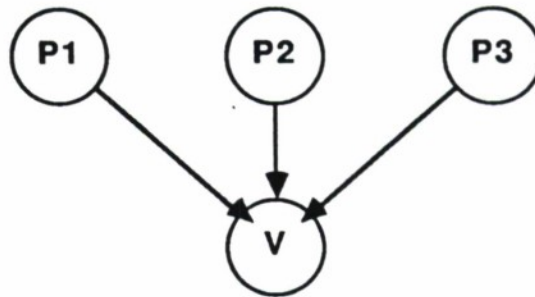


Figure 6-2: Implementation of N-Way Voting

7. Summary

This report has presented a proposed design for a new Durra runtime environment. The key feature of the new design is that it provides for both performance and reliability enhancements.

Performance enhancements are achieved in two manners. First the potential communication bottleneck represented by the Scheduler in the original Durra runtime environment is avoided by introduction of the *local executive*. The local executive executes on each processor and only handles a subset of the communication load. In addition to communication, the local executives are responsible local buffer task and queue management, some of the functionality of the current Scheduler and Server.

Configuration management is the second area of performance enhancement in the new Durra runtime environment. A new component, the *global executive*, assumes this responsibility. It uses a modified version of load balancing algorithms proposed by Lo [Lo 87] to assign processes to processors. This algorithm is designed to minimize processor loading, communication loading, and process interference costs.

Reliability in the new Durra runtime environment is achieved via enhancement and implementation of the reconfiguration mechanism in the Durra Language, and the addition of fault-tolerance. Reconfiguration is handled by the global executive which responds to events by changing the configuration as specified by the application designer. The class of events that can cause a reconfiguration is expanded from the ones in the original language design to include some that can be initiated directly by the application task. Because of the black-box view of processes in Durra, the application designer must take responsibility for helping the runtime environment to determine when it is "safe" to move a particular process as a result of a reconfiguration.

A degree of fault-tolerance is achieved by removing the single point of failure represented by the current Scheduler. A failing local executive will not bring the entire system down. Additional fault-tolerance is achieved by replicating the global executive. There is one active global executive and multiple passive global executives. Since a *fail-stop* [Schlichting 83] model is assumed, the active global executive has complete authority for configuration management unless the processor it is executing on fails. At that point a distributed election protocol [Kim 88] is used to select one of the passive global executives to become active. Failed processors are detected by probing the local executives in the event of no other communication activity. Support for fault-tolerance at the application level relies on the reconfiguration mechanism, and a reliable communication protocol.

After we have some experience running with the new Durra runtime environment, we will be able to determine further directions for enhancements. An important future enhancement will be dealing with transient failures in a fault-tolerant manner. This would allow, for example, a processor to be taken off-line for maintenance with the ability to bring it back on-line when the maintenance was done.

Appendix A: New or Redefined Remote Procedure Calls

The following is a detailed summary of the new or redefined remote procedure calls described in this report. All of the remote procedure calls described here are implemented by the local executive.

A.1. Application Remote Procedure Calls

raise_signal(In signal_number)

Notifies the global executive that the application is triggering event "signal_number". If multiple signals come in from the same process before the global executive can react, it only sees the most recent one. Durra places no semantics on "signal_number". The `signal(process,signal_number)` conditional becomes true, and the global executive will react as specified by the application designer.

safe()

Notifies the local executive that it is safe to migrate the process instead of returning from the remote procedure call. In the absence of the occurrence of a reconfiguration, this is simply a no-op.

get_portId(In name; out portid, bound, size)

Given a port name, returns a small integer port identifier to be used in referring to that port. The name of the port must correspond to one of the ports used in the task description. This call also returns the number of elements that can be stored in the queue associated with the port ("bound") and the size of the elements ("size"). If the size is variable, "size" is set to zero. Hidden from the application but utilized by the interface code, `get_portid` also returns the host and socket to be utilized when communicating with that port.

A.2. Global Executive Remote Procedure Calls

run_task(In task_id,restart_code)

Tells the local executive to actually start the process "task_id". "restart_code" indicates whether the process is being run for the first time (e.g., at application start-up), is being restarted as a result of a controlled reconfiguration (e.g., the application or operator caused it to happen, and the process was made quiescent before reconfiguration), or is being restarted as a result of a system failure (e.g., the process was not made quiescent before reconfiguration.) This will enable the application programmer to take action to restore state where possible.

create_buffer_task(In task-id,task_name,task-kind)

Tells the local executive that there is a buffer task named "task_name" with "task-id", and that its kind is "task-kind". "task-kind" can be one of broadcast, merge, or deal. A local executive is informed of a buffer task only when it has something to do with that process.

create_port(In task_id,port_id,type_id,in_flag)
Defines a port "port_id", to be associated with a process "task-id". The port's type is defined by "type_id". The port is an input port if "in_flag" is set. A local executive is informed of a port when it implements a queue to which that port is attached.

create_queue(In task_id,queue_id,type_id,bound)
Defines a queue "queue_id" having elements of "type_id", and having a bound of "bound". The queue is implemented in the process identified by "task_id". A local executive is informed of a queue when it implements that queue.

attach_port(In queue_id,task_id,port_id,in_port)
Tells the local executive to attach the port defined by "task_id", and "port_id" to the queue defined by "queue_id". If "in_port" is true, attach it to the input side of the queue, otherwise attach it to the output side of the queue. Ports are attached to queues in a separate instruction to allow for easier reconfiguration.

create_attribute(In task_id,name,value)
Associates an attribute "name" with its value "value" for the process "task_id". The local executive that will be executing a process is informed of the attribute values.

create_size_type(In type_id,name,low_bound,up_bound)
Specifies a simple type named "name", identified by "type_id", and with lower and upper size bounds of "low_bound", and "up_bound". All local executives are informed of all types.

create_array_type(In type_id,name,element,bounds)
Specifies an array type named "name", identified by "type_id" and with elements of type "element". "bounds" is a list of the array bounds.

create_union_type(In type_id,name,subtypes)
Specifies a union type named "name", identified by "type_id". "subtypes" is a list of the types that make up the union type.

remove_process(In task-id)
Closes connections to all queues, removes the processes ports, and returns when the process selected by "task-id" has terminated execution.

remove_queue(In queue-id)
Checks to see if all connections to either side of the queue identified by "queue-id" have been closed. If not it returns a failure indication, otherwise it removes the queue.

unattach_port(In task_id,port_id)
Unattaches the port identified by "task_id" and "port_id" from whatever queue it is connected to. This is used when a running process must reconnect a port to a different queue.

commit_configuration()
Make the configuration changes already sent become effective. This is used to ensure that everyone views the configuration in a consistent manner.

abort_configuration()

Abort the configuration changes already sent. This is utilized when there is a communication failure during transmission of a new configuration.

quiesce(In task_id) Cause the process selected by "task-id" to become quiescent by blocking the process after it executes the **safe** remote procedure call. The call returns immediately.

is_quiescent(In task_id)

Tests to see if the process identified by "task_id" has become quiescent.

resume(In task_id) Resumes a quiescent process "task_id". This will only happen if the process has not been migrated to another processor during a reconfiguration.

processor_status(In time_out)

Returns a zero if the local executive has not responded within the period of time indicated by the "time_out". Otherwise returns one.

References

- [Anderson 76] T. Anderson and R. Kerr.
Recovery Blocks in Action.
In *Proceedings of the 2nd International Conference on Software Engineering*, pages 447-457, 1976.
- [Barbacci 86] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3, DTIC: ADA178975, Software Engineering Institute, Carnegie Mellon University, December, 1986.
- [Barbacci 88a] M.R. Barbacci, C.B. Weinstock, and J.M. Wing.
Programming at the Processor-Memory-Switch Level.
In *Proceedings of the 10th International Conference on Software Engineering*. Singapore, April, 1988.
- [Barbacci 88b] M.R. Barbacci, Dennis L. Doubleday, and Charles B. Weinstock.
The Durra Runtime Environment.
Technical Report CMU/SEI-88-TR-18, DTIC: ADA199480, Software Engineering Institute, Carnegie Mellon University, July, 1988.
- [Bell 71] C.G. Bell and Allen Newell.
Computer Structures: Readings and Examples.
McGraw-Hill Book Company, New York, 1971.
- [Birman 87] Kenneth P. Birman and Thomas A. Joseph.
Reliable Communication in the Presence of Failures.
ACM Transactions on Computer Systems 5(1):47-76, February, 1987.
- [Chang 84] Jo-Mei Chang and N. F. Maxemchuk.
Reliable Broadcast Protocols.
ACM Transactions on Computer Systems 2(3):251-273, August, 1984.
- [Chou 82] Timothy C. K. Chou and Jacob A. Abraham.
Load Balancing in Distributed Systems.
IEEE Transactions on Software Engineering SE-8(4):401-412, April, 1982.
- [Chu 80] Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lee, and Kemal Efe.
Task Allocation in Distributed Data Processing.
IEEE Computer, November, 1980.
- [Cooper 85a] Eric C. Cooper.
Replicated Distributed Programs.
In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. pages 63-78, 1985.
- [Cooper 85b] Eric Charles Cooper.
Replicated Distributed Programs.
PhD thesis, University of California at Berkeley, 1985.

- [Cristian 82] Falviu Cristian.
Exception Handling and Software Fault Tolerance.
IEEE Transactions on Computers C-31(6), June, 1982.
- [Kim 88] Junguk L. Kim and Geneva G. Belford.
A Robust, Distributed Election Protocol.
In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*. pages 54-60, 1988.
- [Kramer 88] J. Kramer and J. Magee.
A Model for Change Management.
In *Proceedings of the IEEE Distributed Computing Systems in the '90s*. 1988.
- [Lamport 82] Leslie Lamport, Robert Shostak, and Marshall Pease.
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.
- [Lin 85] Kwei-Jay Lin and John D. Gannon.
Atomic Remote Procedure Call.
IEEE Transactions on Software Engineering SE-11(10):1126-1135, October, 1985.
- [Liskov 87] Barbara Liskov et al.
Orphan Detection.
In *Digest of Papers 17th Annual Symposium on Fault-Tolerant Computing Systems*. pages 2-7, 1987.
- [Lo 87] Virginia Mary Lo.
Heuristic Algorithms for Task Assignment in Distributed Systems.
Technical Report CIS-TR-86-13, Department of Computer and Information Science, University of Oregon, April, 1987.
- [Pease 80] Marshall Pease, Robert Shostak, and Leslie Lamport.
Reaching Agreement in the Presence of Faults.
Journal of the Association for Computing Machinery 27(2):228-234, April, 1980.
- [Schlichting 83] Richard D. Schlichting and Fred B. Schneider.
Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems.
ACM Transactions on Computing Systems 1(3):222-238, August, 1983.
- [Seifert 80] M. Seifert.
Reconfiguration and Recovery of Multiprocess Systems in Fault-Tolerant Distributed Systems.
In *Proceedings IEEE Computer Society's Fourth International Computer Software and Applications Conference*. pages 596-602, 1980.
- [Weinstock 78] Charles B. Weinstock and M. W. Green.
Reconfiguration Strategies for the SIFT Fault-Tolerant Computer.
In *Proceedings IEEE COMPSAC 78*. pages 645-650, 1978.

- [Wensley 78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
Proceedings of the IEEE 60(10):1240-1245, October, 1978.
- [Zicari 86] Roberto Zicari.
Operating System Support for Software Migration in a Distributed System.
In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*. pages 178-187, 1986.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A				
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-TR-8		5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR- 89-016		
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE	6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE		
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003		
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213		10. SOURCE OF FUNDING NOS.		
		PROGRAM ELEMENT NO.	PROJECT NO. N/A	
		TASK NO. N/A	WORK UNIT NO. N/A	
11. TITLE (Include Security Classification) PERFORMANCE AND RELIABILITY ENHANCEMENT OF THE DURRA RUNTIME ENVIRONMENT				
12. PERSONAL AUTHOR(S) CHARLES B. WEINSTOCK				
13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT 50	
16. SUPPLEMENTARY NOTATION				
17. COSATI CODES:		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP			SUB. GR.
1. Abstract: Durra is a language designed to support PMS-level programming. PMS stands for Processor Memory Switch, the name of the highest level in the hierarchy of digital systems. An application or PMS-level program is written in Durra as a set of <i>task descriptions</i> and <i>type declarations</i> that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes. <p>A runtime environment for Durra has been operational for some time. There are two major problems with this initial implementation: it makes no significant attempt to tune the performance of the system, and reliability has not been designed into the system. This report describes a new design for the Durra runtime environment that addresses these two issues. The new runtime environment consists of two major components: a <i>local executive</i> which runs on every processor and is responsible for process and queue management, and a <i>global executive</i> which runs replicated on several processors and is responsible for configuration management and reliability services.</p>				
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> OTIC USERS <input checked="" type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED		
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER		22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630	22c. OFFICE SYMBOL SEI JPO	