Redistribution in Distributed Ada

Dr. Scott James

Management Communications and Control, Inc. (MCCI)

Suite 220

2000 N. 14th Street

Arlington VA 22201

james@mcci-arl-va.com

Abstract

In this paper we will demonstrate how Ada and its Distributed Annex may be used to relocate concurrent objects in a distributed dataflow application. This relocation mechanism will provide the capability of providing both passive and active fault tolerance. Special care will be taken to demonstrate how errors are trapped and propagated across partitions containing multiple threads of execution.

1 The Model

We first summarize a model presented in a prior paper [4].

A dataflow graph is defined as a directed acyclic graph with data passing through the directed edges, or queues, and dynamics at the nodes. Each queue is connected to exactly two nodes: an upstream node attached to its *tail* and a downstream node attached to its *head*. Data flows downstream. Each node possesses a set of *input queues* entering it, and a set of *output queues* leaving it, either set possibly being empty. Each queue contains a *threshold* and a *buffer size* and receives its data in a first-in-first-out fashion from its upstream node. Queues are said to be *linked* to nodes at *ports*, thus there are both *output ports* and *input ports*.

When a queue accumulates a threshold of data, it is said to have *reached threshold*. When every upstream queue of a node has reached threshold, the node is ready to *fire*. Upon firing, a node *reads* and consumes data from each of its upstream queues, processes this data, and writes results to its downstream queues. All nodes can function concurrently, and conceptually begin processing as soon as they reach a fire condition.

A natural fit for this scenario is to implement queues as protected types (Figure 2) and nodes as tasks (Figures 3,5). To allow for maximal concurrency in a distributed system, we provide an additional protected object, a *mailbox*, (Figure 4) and divide nodes into a mailbox and *node task* so that queues may send messages asynchronously to nodes [4].
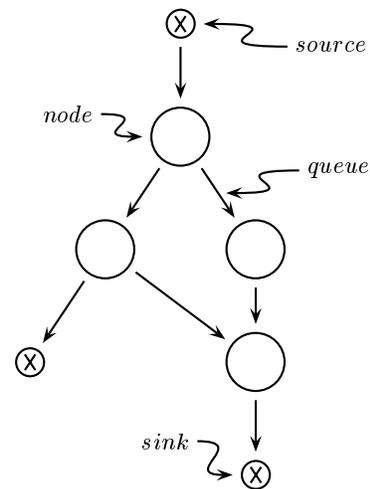
Figure 1: Dataflow Overview


```
protected type queues is
   -- data actions ...
   procedure append (token: tokens);
   entry read (amount: amounts;
               token: out tokens);
   -- link actions ...
   procedure link (at_tip: tips; to_node: nodes);
   procedure unlink (tip: tips);
   entry confirm_tip (at_tip: tips;
                      status: connection_statuses);
   --...
private
   buffer: buffers;
   -- ...
end queues;
```

Figure 2: Queue

```
package node_control is
   type nodes is limited private;
   -- ...
   procedure initialize (node: in out nodes;
                            with_state: states);
   procedure finalize (node: in out nodes;
                          returning_state: states);
   procedure link (node: in out nodes;
                      to_queue: queues;
                      through_port: ports);
   procedure unlink (node: in out nodes;
                        from_port: ports);
   procedure finish (node: in out nodes);
   -- ...
private
   type nodes is record
      node_task: node_tasks;
      mailbox: mailboxes;
      -- ...
   end record;
   -- ...
end node_control;
```

Figure 3: Node

The top level program controlling the architecture of the graph will be known as the *graph manager*. Commands to link and unlink queues are sent to the nodes by the graph manager and then to the queues by the nodes, providing a sequential mechanism by which nodes and queues become aware of their mutual connections.

```
type node_statuses is (linking, firing, finishing);
type port_arrays is private;

protected type mailboxes is
   -- ...
   procedure initialize (with_port_counter: port_counters;
                            with_distribution: distributed_nodes);
   -- graph manager access
   procedure link (queue: queues; to_port: ports);
   procedure unlink (port: ports);
   procedure finish;
   -- queue access
   procedure data_ready (on_port: ports);
   -- node task access
   entry process (status: out node_statuses;
                    port_array: out port_arrays);
   -- ...
private
   port_array: port_arrays;
   notify_the_node: boolean:=false;
   self_reference: distributed_nodes;
   -- ...
end mailboxes;
```

Figure 4: Mailbox

Nodes and queues may reside in different Ada partitions. This model's distribution mechanism involves two types of packages: a distributor package (Figures 7) and **remote call interface** packages (Figure 8). The actual node tasks and protected types are stored inside of each the remote packages and referenced by a component id. This component id, coupled with a partition id, forms a pair which the graph manager passes to the distributor; the distributor uses the partition id to call the proper remote package and passes the component id into that interface (Figure 13).

```
task type node_tasks is
   entry initialize (with_mailbox: access_to_mailboxes;
                        with_state: in states);
   entry finalize (state: out states);
   -- ...
end node_processes;

task body node_tasks is
   mailbox: access_to_mailboxes;
   state: access_to_states;
   -- ...
begin
   accept initialize (with_mailbox: access_to_mailboxes;
                        with_state: in states) do
      mailbox:=with_mailbox;
      -- ...
   end initialize;
   processing:
   declare
      port_connection: port_connections;
      status: node_statuses;
      port_array: port_arrays;
   begin
      loop
         mailbox.process (status, port_array);
         case status is
            when linking =>
               -- ...
            when firing =>
               --...
            when finishing =>
               exit;
         end case;
      end loop;
   exception
      when the_error: others =>
         alert_grm;
      -- ..
   end processing;
   accept finalize (state: out states) do
      -- ..
   end finalize;
end node_tasks;
```
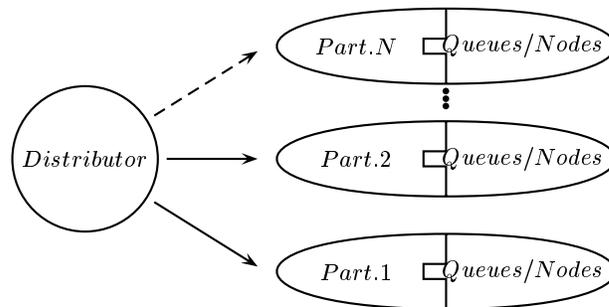
Figure 5: Node Task

Figure 6: Partitioning Architecture

4

```
package distributor is
   -- ...
   -- queues
   procedure initialize (queue: out distributed_queues;
                         on_partition: partitions;
                         for_size: indices);

   procedure append (to_queue: distributed_queues;
                     synchronous: boolean;
                     data: tokens);

   function read (from_queue: distributed_queues;
                  amount: amounts)
                  return tokens;
   -- ...
   -- nodes
   procedure data_ready (for_node: distributed_nodes;
                         on_port: ports);
   -- ...
end distributor;
```

Figure 7: Distributor

```
package sample_partition is
   pragma remote_call_interface;
   -- ..
   -- queues
   procedure initialize (queue: out partitioned_queues;
                         for_size: indices);

   function read (from_queue: partitioned_queues;
                  amount: amounts)
                  return tokens;

   procedure synch_append (to_queue: partitioned_queues;
                           data: tokens);

   procedure asynch_append (to_queue: partitioned_queues;
                            data: tokens);
   pragma asychronous (asynch_append);
   -- ...
   -- nodes
   procedure data_ready (for_node: partitioned_nodes;
                         on_port: ports);
   pragma asynchronous (data_ready);
   -- ...
end sample_partition;

package body sample_partition is
   -- ..
   queue_array is array (partitioned_queues) of queues;
   -- ...
   procedure initialize (queue: out partitioned_queues;
                         for_size: indices) is
   begin
      get_next_available_queue (queue);
      queue_array (queue).initialize (for_size);
   end initialize;
   --...
end sample_partition;
```

Figure 8: Sample Partition

2 The Relocation Mechanism

Relocating nodes and queues is essentially a matter of copying the object to another partition and finalizing the object on the original partition. It is essential, however, that this be done using a coordinated series of steps. In the following section we show how these steps are performed safely in our dataflow model.

2.1 Queues

To relocate a queue, the queue must first be unlinked from both its upstream and downstream nodes in order to prevent these concurrent entities from maintaining a link to a nonexistent queue. As mentioned in Section 1, unlink messages are sent to the nodes and then relayed to the queues; thus, to unlink a queue we must send messages to both the upstream and downstream nodes. After these commands are sent, both the head and the tail of the queue must be confirmed individually to see that the unlink has been completed. Once a queue is disconnected, its only relevant state is the buffer data it contains. This data may be placed or appended onto another queue.

The proper sequence of commands for relocating a queue is:

(1) send commands to unlink both upstream and downstream nodes from the queue

(2) confirm that the queue is unlinked

(3) read and finalize queue[1]

(4) initialize new queue and append stored data

(5) relink upstream and downstream nodes

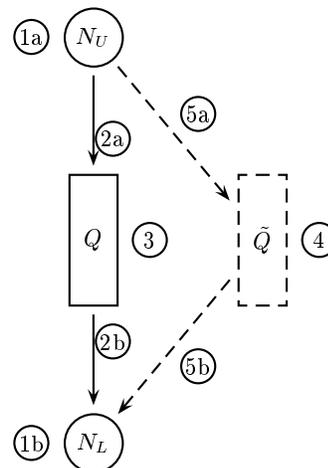See Figure 9 for a pictorial representation.

Figure 9: Relocating Queue

---

[1]Finalization for a queue might involve freeing any space involved with that queue if dynamic storage is used.

2.2   Nodes

Like queues, nodes also may have state, or information pre-served after a firing[2]. This information is placed into a node by the initialization routine and retrieved from a node by its finalization routine as indicated by the **states** parameter in Figures 5 and 3. To relocate a node, the state must be ex-tracted from the node task and placed into the new node task. To obtain this state a node task must first reach a finalization entry point shown in Figure 5.

Figure 10 displays the life cycle of a node task. Here, node (N) begins its existence waiting at entry point *Init*. It is up to the graph manager (GrM) to perform the initializa-tion, providing the node with its initial state. The node will then enter a loop waiting for commands. When the node obtains a *finish* command (N_F) it will enter a *Final* stage, the rendezvous point by which the graph manager obtains the node's final state. In all other cases such as data ready or link request (N_P), the node will process the command and then wait for the next event.
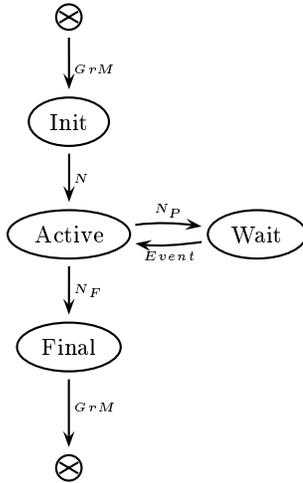

Figure 10: Node Stages

Note that the finalization command is different than the finish command, as the finish command merely prepares the node for finalization while the finalization command actually retrieves the node's state.

However, before a finish command is sent, the node must be unlinked from any input or output queues or else those queues will maintain connections to a nonexistent node. The proper sequence of commands for relocating a node is:

① send command(s) to unlink queues from the node

② confirm that each queue is unlinked

③ send finish command, finalize node, and retrieve state

④ send initialization command to new node with state

⑤ link previously unlinked queues to new node

See Figure 11 for a pictorial representation.

---

[2]It is also possible that a node has no state. In this case each firing is identical.
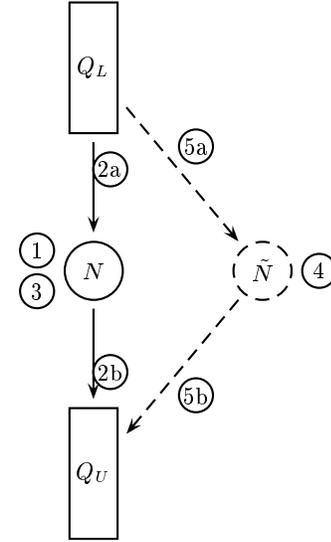

Figure 11: Relocating Node

2.3   Load Balancing

One application of node and queue relocation is load bal-ancing. In our case this will mean the graph manager has decided that it would be preferable to relocate a particular set of nodes to a different partition in order to achieve in-creased parallelism and/or has decided to relocate particular queues to different partitions to reduce queue access times[3]. See Figure 12.

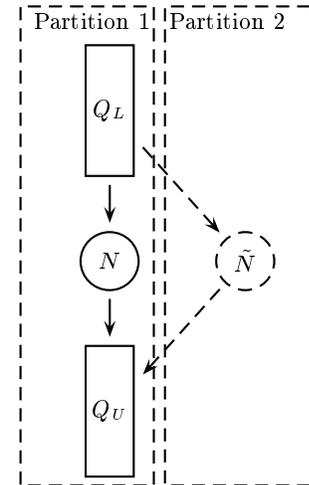

Figure 12: Load Balancing

A key feature of the relocation mechanism is the ability to

---

[3]A typical action is to move input queues of nodes to the same partition so that the node may have intrapartition access to the data when it is ready.

move nodes while the graph is processing data without affecting its functional output, allowing load balancing during execution[4]. In other words, during dynamic load balancing, data is neither lost nor duplicated. In the relocation Sections 2.1 and 2.2 we have already described the methods by which the node's state and queue's data may be retrieved. A few comments on possible data loss during unlinking are in order.

Due to the location of the entry points in the node task (Figure 3) the nodes may not be finalized during a fire. It is during firing that any data used by the node is read from and written to its queues, and during firing that any resultant state modifications are performed. Furthermore, queues may only be linked and unlinked by the nodes themselves, thus, in particular, only when the node is not in the process of transferring its data. Similarly, while a queue is having its data read, as it is a protected object, it may not perform any other operation. Consequently, the relocation mechanism cannot disrupt a data transfer.

3 Fault Tolerance

Though the problem of fault tolerance in the general scope of the Ada language is a complex one, for our specific dataflow model, the problem becomes tractable. Specifically, we are not attempting to provide transparent fault tolerance below the application level as in [1] or [9], nor are we suggesting a set of language pragmas to work in conjunction with general programs as in [10] or [7]. We will, however, demonstrate explicit recovery mechanisms for the dataflow model, in both the passive (Section 3.1) and active (Section 3.2) case.

3.1 Passive Fault Tolerance

Passive fault tolerance in our usage will mean that the recovery mechanism is *cold*, that is, recovering requires creating a new node (as in [2]). To obtain proper fault tolerance, care must be taken to assure proper recovery for all elements of concurrency, in this case for both our node tasks and Ada partitions.

3.1.1 Nodes

In passive fault tolerance, the node is able to detect its own errors, such as in an exception block. In this case, the node will not wait for a finish command but will instead alert the graph manager (Figure 5) and then proceed to its own finalization stage. After receiving the alert, the graph manager will be able to perform a recovery using the relocation mechanism described in Section 2. As a result of the node being at its finalization stage, queues attached to the node will not be able to be unlinked from the tips connected to the faulty node; they nevertheless may be read, destroyed and replaced with new queues.

In chronological order, any unexpected exceptions which a node encounters will cause the node to:

① trap the error

---
[4]Note, however, that any redistribution will potentially affect the *rate* of dataflow while the redistribution is taking place.

② alert the graph manager

③ proceed to the finalization stage

④ perform finalization when requested

Note that the consequence of allowing an exception to fall through a node task is that the node task could bypass the finalization stage and thus its state may not be recovered. Note also that this approach is of limited value if the node's state is corrupt as that corrupted state will simply be reinitialized on another node.

3.1.2 Partitions

As mentioned, the distribution mechanism involves two types of packages: a **remote call interface** package and a **distributor** package. Note in particular that neither static pointers nor class pointers are employed. One advantage of using this approach is that the distributor can isolate distribution errors (specifically, **system.rpc.communication_error**) in a single package (Figure 13).

```
package body distributor is
    -- ...
    procedure initialize (queue: out distributed_queues;
                          on_partition: partitions) is
        partitioned_queue: partitioned_queues;
    begin
        case on_partition is
            when first =>
                partition1.initialize (partitioned_queue);
            when second =>
                partition2.initialize (partitioned_queue);
            -- ...
        end case;
        queue:=to_distributed (partitioned_queue,
                               on_partition);
    exception
        when system.rpc.communication_error =>
            -- ..
        when the_error: others =>
            -- ..
    end initialize;
    -- ...
end distributor;
```

Figure 13: Distributor Body

Likewise, the **remote call interface** packages will trap all exceptions for each entry point. The consequences of not trapping an exception which percolates to the remote interface package is that the entire partition managed by the interface package could become inaccessible; in particular a single faulty node could affect all nodes on the partition. Furthermore, once an exception is trapped, errors can be passed back to the distributor although this is only relevant for synchronous calls.

3.2 Active Fault Tolerance

Active fault tolerance in our use will imply that the recovery mechanism is *hot*, that is, immediately ready for execution (as in [2]). In addition to the speed of recovery, active fault tolerance allows for the possibility of duplicate nodes voting on a correct answer and thus finding faults not detectable by the node itself. In active fault tolerance, unlike passive fault

tolerance, copies and relocations are made *before* a fault is detected.

In Figure 14 we demonstrate the replication of a node into three nodes executing in parallel. In this figure, the entry queue is passed into a *splitting* node ($\bigcap$) which duplicates the data across three queues and sends the data to the three duplicate nodes. Afterwards, the three output queues are collected into a *comparing* node ($\bigcup$), which performs a test to guarantee all three incoming values match, and then sends one set of data to the next node. Upon failure, the comparing node may send a message to the graph manager indicating a failure.

The procedure for creating this replication is outlined as follows:

(1) unlink upstream and downstream queues

(2) finalize node to be duplicated, obtaining state

(3) create duplicate nodes, attaching queues, comparer node and splitter node

(4) attach queues to respective nodes

Once this active fault tolerance is no longer desired the steps to remove this duplication are essentially the reverse of what was done to create it:

(5) unlink upstream and downstream queues attached to duplicate nodes; unlink head of upstream most queue and tail of downstream most queue

(6) finalize duplicate nodes, comparer and splitter

(7) relink upstream queue head and downstream queue tail

Again, see Figure 14.

4 Current Status and Future Work

The previous model and its applications to fault tolerance and load balancing has been successfully implemented on a network of Unix multiprocessor machines using GNAT and its Ada Distribution Annex support: GLADE. We have implemented dataflow graphs with nodes numbering on the order of 10^3, queues numbering on the order of 10^4 and passed data arrays of sizes up to 10^7 bits.

We are using this model as the foundation of our current project: an automatic load balancing, error correcting dataflow system, and anticipate implementing this on both a heterogeneous multi-site network as well as an embedded multiprocessor machine.

References

[1] BRELAND, M., ROGERS, S., NELSON, K., AND BRAT, G. Transparent fault tolerance for distributed ada applications. In *Proceedings:Tri-Ada'94 Conference* (November 1994), ACM SIGAda, pp. 446–457.

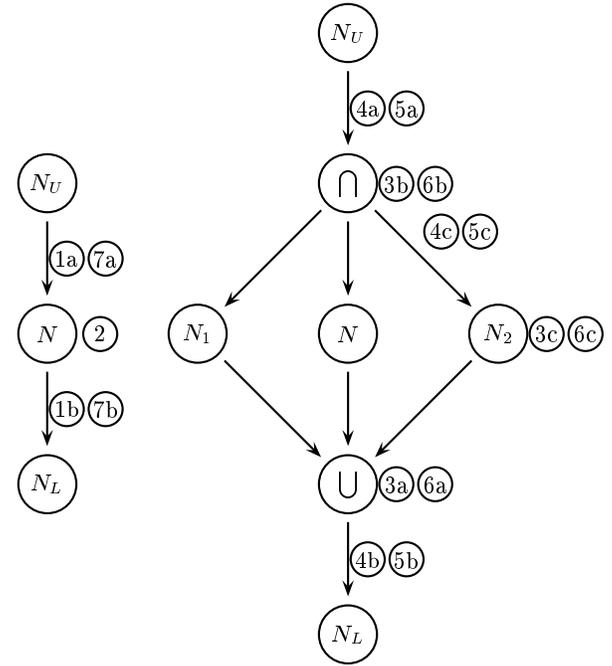[2] BURNS, A., AND WELLINGS, A. *Concurrency in Ada*. Cambridge University Press, 1995.

Figure 14: Active Fault Tolerance

[3] COHEN, N. H. *Ada as a Second Language*. McGraw-Hill, 1996.

[4] JAMES, S. The evolution of a distributed dataflow processing model using Ada. In *Proceedings: ACM SIGAda Annnual International Conference* (November 1998), ACM SIGAda, pp. 39–44.

[5] KARP, R. M., AND MILLER, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J. Appl.,Math* (1966).

[6] NIELSEN, K. *Ada in Distributed Real-Time Systems*. McGraw-Hill, 1990.

[7] PINHO, L. M., AND VASQUES, F. Multi-μ: an Ada 95 based architecture for fault tolerant support of real-time systems. In *Proceedings: ACM SIGAda Annnual International Conference* (November 1998), ACM SIGAda, pp. 52–60.

[8] SIH, G. C., AND LEE, E. A. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems 4*, 2 (1993).

[9] TARDIEU, S., AND PAUTET, L. Building fault tolerant distributed systems using IP multicast. In *Proceedings: ACM SIGAda Annnual International Conference* (November 1998), ACM SIGAda, pp. 45–51.

[10] YVON KERMARREC, L. N., AND PAUTET, L. Providing fault-tolerant services to distributed Ada95 applications. In *Proceedings:Tri-Ada'96 Conference* (December 1996), ACM SIGAda, pp. 39–47.