Redistribution in Distributed Ada

April 16, 1999

Abstract

In this paper we will provide a model using Ada and the Distributed Annex for relocating concurrent objects in a distributed dataflow application. This model will be used to provide passive and active error tolerance as well as the basis for a load balancing superstructure. Care will be taken to demonstrate how errors are trapped and propagated across partitions containing multiple threads of execution.

1 Notes to the Reviewers

The following is intended as an extended abstract. If accepted, the final paper will contain pictures, code segments, examples, and execution specifics running under a network of workstations operating Solaris using a GNAT/GLADE implementation.

2 The Model

We first summarize a model presented in a prior paper [3].

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.

To allow for maximal concurrency in a distributed system, we implement queues as protected types, nodes as tasks, and provide an additional protected object (a *mailbox*) so that queues may send asynchronously to nodes.

(figure)

Each node has three mutually exclusive **accept** states (Figure 1):

1. waiting to be initialized
2. waiting for the next command from its mailbox (processing)
3. waiting to be finalized

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

task body nodes is
   mailbox: access_to_mailboxes;
   status: statuses;
   -- ...
begin
   accept initialize (with_mailbox: access_to_mailboxes;
                      with_state: in states) do
      mailbox:=with_mailbox;
   end initialize;
   loop
      mailbox.process (status);
      -- ...
      case status is
         when sleeping =>
            null;
         when firing =>
            --...
         when linking =>
            -- ...
      end case;
   end loop;
   accept finalize (state: out states) do
      -- ..
   end finalize;
   -- ...
end nodes;
```

Figure 1: Node States

Queues and mailboxes, being protected types, can be accessed by any of their visible **entry**, **procedure** or **function** routines.

Nodes and queues may reside in different Ada partitions. This model's distribution mechanism involves two types of packages: a distributor package and **remote_call_interface** packages (figure). 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 top-level program (*graph manager*) passes to the distributor; the distributor in turn uses the partition id to call the proper remote package and passes the component id into that interface.

Finally, 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.

3 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 what follows we will show how these steps are performed on a distributed system, with messages being passed between protected objects and tasks such the relocation may be performed safely during execution.

3.1 Nodes

The parameter to node initialization and returning from node finalization is its *state* (figure). This state contains any information necessary for the node to process its current state. [1]

Before a node can be moved it must be placed into a finalization mode. However, before this can occur, the node must be unlinked from any input or output queues else those queues will maintain connections to a nonexistent object. The proper sequence of commands for relocating a node is:

1. send command to unlink all queues from the node
2. confirm that each queue is unlinked
3. send finish command to node
4. send finalization command to node retrieving its state
5. send initialization command to new node with state
6. link previously unlinked queues to new node

(figure)

Note that the finalization command is different than the finish command, as the finish command merely changes the node's state and is communicated by means of the mailbox. The finalization command, however, is sent directly to the node and retrieves the node's state as well.

[1] It is also possible that a node has no state, that is, each firing is identical.

3.2 Queues

To relocate a queue, the queue does not have to first enter a finish state, however, it does have to be unlinked else connected nodes will maintain a link to a nonexistent queue. Note that once a queue is disconnected, its only relevant state is the data it contains.

The proper sequence of commands for relocating a queue is:

1. send command to unlink both upstream and downstream nodes from the queue
2. confirm that the queue is unlinked at both its head and tail
3. read and store data from queue
4. finalize the queue
5. initialize new queue (possibly on another partition)
6. append stored data to new queue
7. relink upstream and downstream nodes

Finalization for a queue might involve, for instance, freeing any space involved with that queue if dynamic storage is used.

3.3 Load Balancing

As an example of the use of relocation we present a simple load balancing scheme. A key feature of the relocation mechanism is the ability to move nodes while the graph is processing data without affecting its functional output, allowing continual rebalancing during execution. Note, however, that any redistribution will potentially affect the *rate* of dataflow while the redistribution is taking place.

In the following simple example we demonstrate a sequence of events where an intermediate node in a graph is moved across a partition to achieve increased parallelism.

(example)

Note that, by design, data is neither lost nor duplicated. Nodes may not be interrupted during a fire, during which any data used by the queue is consumed, and any resultant state modifications are preserved. Further, queues may only be unlinked and unlinked by the nodes themselves, thus, in particular, only when the node is not in the process of firing.

4 Error tolerance

4.1 Passive Error Tolerance

Passive error tolerance in the standard usage implies that the recovery mechanism is *cold*, that is, recovering requires creating a new object. This is particularly useful when the node is able to detect its own error. In passive error tolerance, the node will perform a self finish, as described above, and possibly pass an error back which reaches the graph manager. At this point, the graph manager will be able to perform a recovery using the relocation mechanism described in Section 3.

To obtain proper error recovery, care must be made to assure proper recovery for all elements of concurrency, in this case for both tasks (nodes) and Ada partitions. For each we take the same approach: capture all exceptions. (figure)

4.1.1 Nodes

Any unexpected exceptions which a node encounters will cause the node to:

1. trap the error
2. pass an error code back to its caller (potentially across a distribution)
3. send a finish command to its own mailbox
4. read and execute the finish command
5. await finalization

Note that the node performs a self finish through its own mailbox, allowing the node and its mailbox to remain synchronized. The consequence of allowing an exception to fall through a node task is that its state cannot be recovered.

(figure)

4.1.2 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)

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 (figure) although this is only relevant for synchronous calls. We will demonstrate two type of recovery mechanisms: passive and active (see for instance [1]).

4.1.3 Example

(example)

Note that this approach will not work if the node's state is corrupt, as that corrupted state will simply be reexecuted elsewhere.

4.2 Active Error Tolerance

Active error tolerance implies that the recovery mechanism is *hot*, that is, immediately ready for execution. In addition to the speed of recovery, active error tolerance allows for the possibility of duplicate nodes/graphs voting on a correct answer, thus detecting errors not detectable by the node itself. In active error tolerance, in contrast to passive error tolerance, copies and relocations are made before an error is detected.

Though active error tolerance for the general case is difficult ([6], [1]) for our specific dataflow model the problem becomes quite tractable.

In the following example we demonstrate the replication of a node into three nodes executing in parallel. The entry queue is passed into a *splitting* node and replicated three times; the exit queue is replicated as well and the resultant entry queues are collected into a *comparing* node. The comparing node performs a match to guarantee all 3 incoming values match before resending the data. Upon failure, the comparing node may either self-finish, thus blocking the flow until correction or emit data to another queue to be handled by another process.

(example)

5 Conclusion

We have demonstrated a method of redestributing concurrent objects using the Ada Distribution annex in the application of a dataflow graph. An example case of load balancing was demonstrated. Error tolerance paradigms were discussed demonstrating the critical points at which exceptions must be handled and errors reported to enable recovery.

References

[1] Burns, A., and Wellings, A. *Concurrency in Ada.* Cambridge University Press, 1995.

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

[3] 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.

[4] Karp, R. M., and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J.Appl.,Math* (1966).

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

[6] 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.

[7] 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).