Eric Jul, 1986-08-02

1 Groups in Emerald

In object-oriented programming, objects are composed of other objects. When an object moves, it is usually desirable to move the object and its component objects together. For example, in a mail system such as (ref. Edmas), a mail message may have references to the text of the message, the message status, the time sent, and a mail box to which replies may be sent. The message text, status, and date should be moved with the object since they are local to the mail message object. The reply mail box should probably not be moved along since it presumably will be referenced only once while the mail box owner will reference it frequently.

Seen from a mail message sender's point of view, the mail message is a single object and one should be able to move it without having to explicitly move its component objects, or even have knowledge of their existence. This argues for a facility for grouping objects together so that they may be moved together thus avoiding explicit reference to each group member.

In general, the decision to move an object depends on the cost of moving the object (which in turn depends on the size of the object) and the usage pattern of the object. The purpose of moving an object can be to increase the locality of reference, *i.e.*, increase the percentage of invocations performed locally. Large performance gains can be achieved by avoiding remote invocations since local invocations are about 3 orders of magnitude faster than remote invocations[1]. Conversely, large performance penalties may be incurred, if frequently communicating objects are inadvertantly separated.

For some object references, it is obvious that the referenced objects should be moved along with the containing object. Objects local to another object should obviously be moved with that object since all references to the objects will then continue to be local invocations. This includes integers and other immutable object. All immutable objects are always moved along allowing them always to be invoked locally. Conceivably, large immutable objects could be left behind; see section X.X[2] for a discussion of the replication of immutable objects. For global objects, there is no similar inherently obvious choice. Using the mail example, the reply-to mail box should not move with the mail message, while the other component objects should – even if they are global.

There is a performance reason for being able to move more than one object at a time. Consider the following program which moves an array and its elements:

```
i <- 1
loop
    exit when i = n
    move A(i) to remoteNode
end loop
```

This loop performs the n moves synchronously one at a time each requiring at least one network messages. If the elements were moved all at once then the move could be performed with considerably fewer net messages.

[1]The current local invocation time for a global object is 21.3 μs while the remote invocation time is about 42 ms.

[2]A section to come some time in the future.

In summary, Emerald should include a facility for grouping objects together for the purpose of object migration.

1.1 How to compose groups

There are several alternatives for providing a grouping facility. Implicit methods include compiler decisions based on the static program text and dynamic methods where the run-time system predicts future usage patterns and uses these to make mobility decisions.

1.1.1 Implicit Methods

As mentioned above for some types of objects, the compiler may can readily decide to move certain component objects since it knows their usage patterns. However, in general, the compiler cannot derive the usage pattern of a global object merely from the static program text. The run-time system could keep track of usage patterns and use these as a prediction of future behaviour. This has several drawbacks. First, past behaviour is not necessarily a good predictor of future behaviour, for example, bimodal behaviour might lead to predictions that would cause worst case behaviour. Second, this would require extensive monitoring of not only remote invocations where the relative cost would be low, but also of local invocations where the relative cost would be significant. The costs in terms of extra storage and administration alone seems to rule out such monitoring. Since either the compiler or the run-time system can acquire enough information about global object usage to be able to make mobility decisions, we conclude that the programmer must explicitly either provide information regarding usage patterns or actually make the mobility decisions.

1.1.2 Explicit Methods

Groups could be explicitly constructed by the programmer, *e.g.*, by placing a reference to each group member in an array. The `move` statement would then take such an array as an operand and would move the objects referenced in the array. In this way, the programmer could build move groups by inserting and deleting object references from this set. Since groups actually are sets, it would make more sense to introduce a standard type `set` and use it.

Explicitly building and maintaining groups has a drawback. Using the mail message example from above, if a mail box decides to move one of its mail messages, it would do so using a reference to the mail message only – it would not have sufficient knowledge to include the component object of the mail message in the move group.

Instead of using special group objects then every object could be used as a group in the following manner. Every object would uniquely determine a group which initially would only the object itself.

```
join A to B
```

The `join` statement, when executed, would cause the object referenced by A to join the group identified by the object referenced by B. Whenever an object, X, is moved, all the members of the group identified by X follow.

```
let A leave B
```

when executed, would remove the object referenced by A from the group identified by the object referenced by B. An object could be a member of any number of groups and would be moved any time any of the groups is moved.

Another solution is for the programmer to explicitly mark the objects which should be moved. Pragmas could be attached to variables indicating that the object they reference should be moved along with the object containing them as in

```
attached var a, b:  T
```

The `attached` pragma would indicate that every time the object moved, the objects referenced by `a` and `b` should follow. Thus the programmer could, in a very simple manner, specify the objects to participate in any migration. Using the mail message example:

```
var replyToMailBox:     MailBox
attached var text:      String
attached var status:    MsgStatus
attached var sendTime:  Time
```

The `attached pragma` seems as powerful as the group idea since one can simulate the `join` method grouping by merely declaring a set where each of the set members are themselves `attached` to the set. Merely copying a reference to an object to an attached variable achieves the effect of a `join`. Nil'ing a reference in the set effects a `leave`. It is also very simple to implement since it allows for compile-time construction of groups. The compiler marks the `attached` variables in the template for each data area.

In conclusion, it seems that the simple idea of an `attached` pragma seems workable and readily implementable.

2 Atomicity of Move

How atomic should move be? The simplest (and most efficient) move protocol is not resilient in the face of crashes. A more robust protocol enables the atomic transfer of objects across the net.

2.1 Simple Move Protocol

The simplest move protocol sends the object to the destination node and immediately changes the location of the object to the new destination – despite the fact that the object has not completed the move yet. However, since the underlying message system assures reliable delivery in the absence of crashes, it will only be a matter of time before the object is successfully transferred to the destination node. The protocol is not resilient to crashes. If either node crashes during a move then the object is lost. The advantage of the protocol is its efficiency: It requires only one message per move. Furthermore, the object may be piggybacked onto, *e.g.*, an invocation message containing call-by-move parameters, in which case the move can be performed at a very low cost.

2.2 Atomic Two Phase Move Protocol

The atomic move protocol is resilient to node crashes. The protocol uses forwarding addresses[3] to commit migration transactions as described by Fowler (ref. to Fowler's thesis).

A move is initiated by sending the state of the object to the destination node marked as an object in transit. The destination rebuilds the object and sends a "ready-to-commit" message to the source. Upon reciept of this message, the source commits the move by:

- Generating a new forwarding address consisting of (destination node, new timestamp) and commiting it to stable storage.
- Deallocating the local version of the object.
- Sending the new forwarding address to the destination node.

The destiantion can abort the transaction at any time before sending the "ready to commit" by sending an "abort move" message to the source.

The forwarding address is the *commit record* in the *two phase commit.* In case of node crashes, the object is not lost if it is checkpointed.

The protocol requires three messages: one message to send the object (possibly consisting of multiple packets), one message to prepare for the commit, and finally one message to commit. Furthermore, the object must be written to stable storage at the destination and the commit record must be written to stable storage at both the source and the destination. Thus the total cost is three network messages (one of which could be multiple packet since it must contain the object data area and its checkpoint image) and three writes to stable storage.

2.3 Performance Estimate

The simple protocol can be performed at a cost of one network message.

The two phase commit protocol seems suited for objects that have checkpointed, since they will survive crashes even when both nodes involved in migrating crash.

An estimate of the performance of the protocols for small objects (¡ 500 bytes) may be obtained by plugging in the following measured figures: Network messages (estimated as 48% of remote invocation time): 20 ms each. Write to stable storage (1 disk page under UNIX using fsync): 120 ms.

Simple protocol: 1×20 ms $= 20$ ms.

Two phase commit protocol: 3×20 ms $+ 3 \times 120$ ms $= 420$ ms (or roughly $\frac{1}{2}$ second).

Protocol	Message Count	Writes to Stable storage	Estimated time (ms)
Simple Send	1	—	20 ms
Atomic	3	3	420 ms

[3] I use the term *forwarding address* in a slightly different manner than Fowler: a forwarding address is merely the address of a node and a timestamp. Fowler includes the OID in the forwarding address, *i.e.*, he calls the triple (OID, nodenumber, timestamp) a forwarding address while I leave out the OID. I will have to make this clear in my discussion of the implementation of the location protocol (which Fowler would call the Object Finding Package).

Table 1: Estimated performance of move.

2.4 Discussion

The two protocols both seem to have their place. For objects that have not checkpointed, it does not make sense to ensure that the move commits across node crashes since the object itself is usually lost anyway. Non-checkpointed objects are inherently short-lived and non-resilient to node crashes, so it seems that the protocol for moving them should have the same characteristics. Objects that have checkpointed are resilient to node crashes and it seems that this resiliency should not be lost during migration. Therefore, the atomic move protocol appears to be the most suitable.

Summarizing, the simple protocol is not resilient to crashes, but is efficient, while the two phase commit protocol is resilient to crashes, but is much more inefficient as it about 20 times slower.