Abstract
The ordinary features of an object-oriented distributed programming language are sufficient for describing distributed algorithms.  

LADA Category: Representation

Keywords Emerald, Timber, reactive objects

1. Introduction
In 1993, I submitted a position paper to the ECOOP workshop on Object-based Distributed Programming with the title “Objects are Enough” [1]. The claim made in that paper — and I think that I supported it fairly well — was that objects encapsulated location, and method invocation encapsulated communication, and that little else was necessary to support distributed programming. The main technical contribution of that paper was a plea not to add to the pure object model such features as multi-methods, generic functions, classes, and object identity. It may be worth reviewing those arguments again in 2012.

This paper, deliberately, shares the same title, but is more of an unsubstantiated assertion than a defensible claim. When I teach our “Scholarship Skills” class at PSU, I tell my students that they can’t make unsubstantiated assertions in scientific article, but a position paper isn’t quite a scientific article. In 1993 I could argue from ten years experience writing object-oriented, distributed programs, but I certainly can’t say the same today about describing distributed algorithms using objects.

Still, when Annie Liu challenged me to explain how to describe distributed algorithms in Emerald [3, 6], although my first reaction was that Emerald is not appropriate for this task, my second reaction was: let’s see why a programming language like Emerald is not appropriate, and look at what we can do to fix it. It is that process that I am starting here.

2. Objects for Algorithm Description
Let’s engage in a Gedankenexperiment, starting with the base of concepts from object-oriented distributed languages, and seeing how far they will take us towards algorithm description.

Objects encapsulate Location. Each mutable object — both code and data — is assumed to exist at a single location in the distributed system. In distributed programming, it is a convenient optimization to allow immutable objects to exist at multiple locations; asking where such an object is located will answer one of the current locations, but one can’t infer from that that the object is not also located somewhere else. In a distributed algorithm, the location of an actor is usually of prime importance, so this optimization is probably a bad idea.

Objects communicate by sending messages, which contain a name and some optional arguments. Departing from Emerald, but following Timber [2, 8], messages are asynchronous. A message arrives some (indeterminate amount of) time after it is sent; the sender and receiver execute concurrently. Arrival of a message triggers an action: the execution of the correspondingly-named method. (Synchronous messages can of course be represented as a pair of asynchronous messages, and I’m going to assume that for non-blocking, non-recursive methods, this can be wrapped in some convenient syntax that blurs the use-side distinction between function and synchronous method.)

Objects encapsulate data; the data (“instance variables” and “instance constants”) of an object can be accessed only by the methods of that object. At most one method can be executing at a time, so an object behaves like a monitor. If a message n arrives while a method m is executing, execution of the method n is delayed until m has completed. An object may contain a process, but that process cannot access the data of the object other than by invoking its own methods. Why, then, put the process
inside, rather than outside, the object? Because the object defines the location of the process. These semantics mean that self-invocations are bound to deadlock, since the invoked method can't start until the invoking method has finished. Thus, we need local (i.e., lexically bound) functions to provide some internal structure within methods. So a strict interpretation of “Objects are Enough” is false; still, many object-oriented languages do support lexically-bound functions, so we are not alone in making this exception.

**Method execution must be time-bounded.** Following Timber, waiting is not transparent: all methods must complete execution within a bounded time (although the bound will depend on the speed of the underlying execution platform and the load under which the system is operating). Unbounded waiting is not permitted; as a consequence, input cannot be represented as an action performed by a program or an algorithm, but is instead an action performed by the environment, to which the program or algorithm reacts [7].

**Timing** is controlled as in Timber. Every action has a baseline and a deadline; the interval between them is the timeline of the action. An action initiated by a message will inherit the baseline and deadline of its initiator, unless the message sends dictates otherwise using the before and after modifiers, both relative to the sender’s baseline.

### 3. Representing Two-phase commit

Let’s see how these tools can be used to represent two-phase commit. I’ll use the terminology of the Wikipedia description: a transaction involves a coordinator and one or more cohorts. There is one exception: Wikipedia uses an acknowledgement message from a cohort to the coordinator to acknowledge both a commit and a rollback; I use separate ack and nack messages.

The syntax of this GedankenSprache is inspired by Grace and Smalltalk rather than by Emerald. To highlight asynchronous message sends, I use the symbol $\rightarrow$. The keyword $\textit{is}$ means definitional equality. Indentation is significant, as in Python; an indented block preceded by a colon (:) is a lambda expression, and may be thought of as deferred code. So, for example, if $c$ then: $t$ else: $f$ is not a built-in syntax, but a function of three arguments: either $t$ or $f$ (but not both) will be executed, depending on the value of $c$. Similarly, in $a\text{\textit{do}}: x \rightarrow stmt$, the $x \rightarrow stmt$ after the colon is a lambda-expression binding $x$, with body $stmt$. The effect of the do method is to execute the lambda-expression once for each element of $a\text{\textit{do}}$, with $x$ bound to the appropriate element.

The sketch that follows also ignores failures, which are arguably the most interesting part of the protocol, as well as many housekeeping details, such as how the coordinator and the cohorts maintain their transaction meta-data.

```
object coordinator is:
  var xmd ← ... — a local map from transaction ids to transaction metadata
  var tid ← ... — the unique identifier of this transaction

process is:
  ... — do the work
  (xmd.for tid).cohorts.do:
    each → each⟨queryToCommit (tid, coordinator)
    self ⟨commitPhase tid after tid.queryTimeout

method abort (tid, cohort) is:
  (xmd.for tid).addToAborters cohort

method agreement (tid, cohort) is:
  (xmd.for tid).addToAgreeable cohort

method commitPhase tid is:
  if (xmd.for tid).cohorts.size = tid.agreeable.size
    then:
      (xmd.for tid).cohorts.do:
        each → each⟨commit (tid, coordinator)
      else:
        (xmd.for tid).cohorts.do:
          each → each⟨rollback (tid, coordinator)
          tid.markAsAborted

  self ⟨completePhase tid
  after (xmd.for tid).commitTimeout

method ack (tid, cohort) is:
  (xmd.for tid).recordAckFor cohort

method nack (tid, cohort) is:
  (xmd.for tid).recordNackFor cohort

method completePhase tid is:
  if (xmd.for tid).ackers = (xmd.for tid).cohorts
    then:
      (xmd.for tid).markAsCommitted
    else:
      self ⟨completePhase tid
  after (xmd.for tid).commitTimeout
  — potential infinite recursion when a cohort fails and never recovers

object cohort is:
  var xmd ← ... — a local map from transaction ids to transaction metadata

method queryToCommit (tid, coord) is:
  if (xmd.for tid).willingToCommit
    then:
      coord⟨agreement (tid, self)
    else:
      coord ⟨abort (tid, self)

method commit (tid, coord) is:
  (xmd.for tid).completeLocalOperations
  (xmd.for tid).releaseResources
  coord ⟨ack (tid, self)

method rollback (tid, coord) is:
  (xmd.for tid).undoLocalOperations
  (xmd.for tid).releaseResources
  coord ⟨nack (tid, self)
```

What happens if the deadline of a method cannot be met? In many cases, we can ascertain statically whether a particular deadline can be met on particular hardware, given a knowledge of the hardware execution speed and the code
in the method. (Recall that methods never block.) Of course, this is not always possible, so a method failing to meet its deadline is treated as a failure. Equivalently, a failure is treated as a method failing to meet its deadline.

4. Failure

Emerald provided built-in checkpoint and recovery mechanisms; checkpoints could be written to “stable storage”, or to the volatile memory of another node. Failure detection was also built-in. It seems to me now that checkpointing, recover and failure detection should all be “pluggable”: the language should support the concepts, but the mechanisms should be under programmer control. Emerald atomically checkpointed all of the state of an object; in practice, some of this may be redundant or inconsistent, and it might be better to allow the programmer direct control over exactly what is checkpointed and how it is recovered.

The lesson from Erlang is that it is generally much easier to resume correct operation from a known “good state” than by sifting through the bits of a representation that may contain inconsistencies. Thus, it may make more sense to continuously stream log entries to an active backup rather than to checkpoint one’s whole state to passive storage, even though the atomic checkpoint seems like a higher level primitive. Moreover, a conventional atomic checkpoint must be a blocking operation, while sending a log entry is asynchronous, so in our framework they are not interchangeable. Similarly, after 20 years of research on failure detectors, starting with Chandra and Toueg’s landmark 1991 paper, we now know that, even though all failure detectors are necessarily approximate, the exact character of the approximation matters in rather important ways. So, while it seems reasonable to support failure handling as a language primitive, the determination of what constitutes a failure should probably be left to the designer of the algorithm.

5. Summary

The language sketched here is not implemented, and the above algorithm for two phase-commit has neither been proved correct nor tested: it is a Gedankenexperiment conducted to see how fairly conventional distributed object-oriented programming concepts can be adopted or adapted to the task of describing distributed algorithms. If it stimulates discussion, it will have served its purpose.

Acknowledgments

I thank all of my colleagues on the Eden, Emerald, Timber and Grace projects; our joint experiences are the straw from which these ideas are moulded. The chaff that remains is my own.

References


