Fine-Grained Mobility in the Emerald System

ERIC JUL, HENRY LEVY, NORMAN HUTCHINSON, and ANDREW BLACK
University of Washington

Emerald is an object-based language and system designed for the construction of distributed programs. An explicit goal of Emerald is support for object mobility; objects in Emerald can freely move within the system to take advantage of distribution and dynamically changing environments. We say that Emerald has fine-grained mobility because Emerald objects can be small data objects as well as process objects. Fine-grained mobility allows us to apply mobility in new ways but presents implementation problems as well. This paper discusses the benefits of fine-grained mobility, the Emerald language and run-time mechanisms that support mobility, and techniques for implementing mobility that do not degrade the performance of local operations. Performance measurements of the current implementation are included.

Categories and Subject Descriptors: C.2.4[**Computer-Communications Networks**]: Distributed Systems—*distributed applications*, *network operating systems*; D.3.3[**Programming Languages**]: Language Constructs—*abstract data types*, *control structures*; D.4.2[**Operating Systems**]: Storage Management—*distributed memories*; D.4.4[**Operating Systems**]: Communications Management—*message sending*; D.4.7[**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Design, Languages, Measurement, Performance

Additional Key Words and Phrases: Distributed languages, object-oriented languages, object-oriented systems, process mobility

1. INTRODUCTION

Process migration has been implemented or described as a goal of several distributed systems [8, 11, 16, 20, 23, 24, 28]. In these systems, entire address spaces are moved from node to node. For example, a process manager might initiate a move to share processor load more evenly, or users might initiate remote execution explicitly. In either case, the running process is typically ignorant of its location and unaffected by the move.

This work was supported in part by the National Science Foundation under grants MCS-8004111, DCR-8420945 and CCR-8700106, by Københavns Universitet (University of Copenhagen), Denmark under grant J.nr. 574-2,2, by a Digital Equipment Corporation External Research Grant, and by an IBM Graduate Fellowship.
Authors' current addresses: E. Jul, DIKU, Dept. of Computer Science, University of Copenhagen, Universitetparken 1, DK-2100 Copenhagen, Denmark; H. Levy, University of Washington, Dept. of Computer Science, FR-35, Seattle, WA 98195; N. Hutchinson, Dept. of Computer Science, University of Arizona, Tucson, AZ 85721; A. Black, Digital Equipment Corporation, 550 King St., Littleton, MA 01460.

During the last three years, we have designed and implemented Emerald [6, 7], a distributed object-based language and system. A principal goal of Emerald is to experiment with the use of mobility in distributed programming. Mobility in the Emerald system differs from existing process migration schemes in two important respects. First, Emerald is object-based, and the unit of distribution and mobility is the object. Although some Emerald objects contain processes, others contain only data: arrays, records, and single integers are all objects. Thus, the unit of mobility can be much smaller than in process migration systems. Object mobility in Emerald subsumes both process migration and data transfer. Second, Emerald has language support for mobility. Not only does the Emerald language explicitly recognize the notions of location and mobility, but the design of conventional parts of the language (e.g., parameter passing) is affected by mobility.

The advantages of process migration, which have been noted in previous work, include

(1) *Load sharing*—By moving objects around the system, one can take advantage of lightly used processors.
(2) *Communications performance*—Active objects that interact intensively can be moved to the same node to reduce the communications cost for the duration of their interaction.
(3) *Availability*—Objects can be moved to different nodes to provide better failure coverage.
(4) *Reconfiguration*—Objects can be moved following either a failure or a recovery or prior to scheduled downtime.
(5) *Utilizing special capabilities*—An object can move to take advantage of unique hardware or software capabilities on a particular node.

Along with these advantages, fine-grained mobility provides three additional benefits:

(1) *Data Movement*—Mobility provides a simple way for the programmer to move data from node to node without having to explicitly package data. No separate message-passing or file-transfer mechanism is required.
(2) *Invocation Performance*—Mobility has the potential for improving the performance of remote invocation by moving parameter objects to the remote site for the duration of the invocation.
(3) *Garbage Collection*—Mobility can help simplify distributed garbage collection by moving objects to sites where references exist [16, 29].

To our knowledge, the only other system that implements object mobility in a style similar to Emerald is a recent implementation of distributed Smalltalk [4].

In addition to mobility and distribution, we intend that Emerald provide efficient execution. We want to achieve performance competitive with standard procedural languages in the local case and standard remote procedure call (RPC) systems in the remote case. These goals are not trivial in a location-independent object-based environment. To meet them, we have relied heavily

on an appropriate choice of language semantics, a tight coupling between the compiler and run-time kernel, and careful attention to implementation.

Emerald is not intended to run in large, long-haul networks. We assume a local area network with a modest number of nodes (e.g., 100). In addition, we assume that nodes are homogeneous in the sense that they all run the same instruction set and that they are trusted.

In this paper we concentrate primarily on the language and run-time mechanisms that support fine-grained mobility while retaining efficient intranode operation. First, we present a brief overview of the Emerald language and system and its mobility and location primitives. A more detailed description of object structure in Emerald can be found in [6] and of the type system in [7]. Second, we discuss the implementation of fine-grained mobility in Emerald and new problems that arise from providing such support. Third, we present measurements of the implementation and draw implications from the measurements and our design experience.

2. OVERVIEW OF EMERALD

As previously stated, an important goal of Emerald is explicit support for mobility. From a conceptual viewpoint, a more important goal is a single-object model. Object-based systems typically lie at the ends of a spectrum: object-based languages such as Smalltalk [13] and CLU [22] provide small, local data objects; object-based operating systems like Hydra [30] and Clouds [1] provide large, active objects. Distributed systems such as Argus [21] and Eden [3] that support both kinds of objects have a separate object definition mechanism for each. Choosing the right mechanism requires that the programmer know ahead of time all uses to which an object will be put; the alternative is to accept the inefficiency and inconvenience of using the "wrong" mechanism or to reprogram the object later as needs change. For example, while programming a Collaborative Editing System in Argus, Greif, Seliger, and Weihl have observed that a designer can be forced to use a Guardian where a cluster might be more appropriate [14].

The motivation for two distinct definition mechanisms is the need for two distinct implementations. In distributed object-based systems such as Clouds and Eden, a *local* execution of the general invocation mechanism can take milliseconds or tens of milliseconds [26]. A more restrictive and efficient implementation is appropriate for objects that are known to be always local; for example, shared store can be used in preference to messages.

Although we believe in the importance of multiple implementations, we do not believe that these need to be visible to the programmer. In Emerald, programmers use a single object definition mechanism with a single semantics for defining all objects. This includes small, local data-only objects and active, mobile distributed objects. However, the Emerald compiler is capable of analyzing the needs of each object and generating an appropriate implementation. For example, an array object whose use is entirely local to another object will be implemented differently from an array that is shared globally. The compiler produces different implementations from the same piece of code, depending on the context in which it is compiled [18].