Fine-Grained Mobility in Emerald

Robert Grimm
New York University
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
Why Bother with Migration?

* Advantages in general
  * Load sharing
  * Communications performance
  * Availability
  * Reconfiguration
  * Utilizing special hardware/software features

* Fine-grained mobility in particular
  * Data movement/exchange
  * Invocation performance
  * Garbage collection
Emerald Goals

- Provide mobility without sacrificing performance
  - Procedure calls in local case
  - RPC in remote case
- Provide a single object model
  - While still allowing for different implementations
    - Small, local data objects
    - Large active objects
- Target environment: local network with \( \leq 100 \) nodes
Emerald Objects

- Objects have four components
  - Unique network-wide name
  - Representation: data & references to other objects
  - Set of operations
  - Optional process
- Objects are *not* class-based, do *not* form a hierarchy
  - Associated with concrete type object (which has code)
  - Can be compared against an abstract type (i.e., interface)
- What are advantages/disadvantages of this model?
Basic Ingredients of Mobility

- Five primitives
  - Locate, move, fix, unfix, refix (i.e., atomic unfix, move, fix)
    - How is fix is stronger than move?
- Explicit location through node object
- Implicit location through any other object
- Attachments
  - Control what objects are moved together
  - Are transitive
  - Are *not* symmetric
Calling Convention

* In general: call-by-reference semantics
  * What is the problem in a distributed system?
* For efficiency: automatic argument migration
  * Controlled by compiler (think small, immutable objects)
  * Controlled by programmer
    * Call-by-move
    * Call-by-visit
Emerald Processes

- Lightweight threads animating objects
  - Stacks of activation records
- Objects may move ➔ what about activations?
  - Always return to original node
    - Leaves *residual dependencies*, which may limit availability
  - Move activations with objects
    - Need a clever implementation
Implementation
Three Kinds of Layout

- Global, local, and direct objects
- What happens when a global object is on a different node?
Finding Objects

- Mechanism based on *forwarding addresses*
  - Each object has a global object identifier (OID)
  - Each node has an *access table* mapping OIDs to descriptors
    - OID → <timestamp, node>
  - Every sent reference contains OID and forwarding address
  - Searching node follows up to two forwarding addresses
    - If unsuccessful, resorts to broadcasts
- Why not keep a directory of nodes referencing object?
Fun with Pointers

- Problem: when possible, Emerald uses direct addresses
  - Local to a machine, need to be translated on move
- Solution: object and activation record templates
  - Identify types (pointers, data, monitors) and their slots

```plaintext
const simpleobject == object simpleobject
  monitor
    var myself : Any ← simpleobject
    var name : String ← "Emerald"
    var i : Integer ← 17
    operation GetMyName → [n : String]
      n ← name
    end GetMyName
    :
  end monitor
end simpleobject
```
The Mechanics of Moving

* Moving objects
  * Messages include data area and translation information
    * For global object pointers: OID, forwarding & local addresses
    * For local object pointers: data and local address
  * Receiver allocates space, builds translation table, makes sure descriptors exist, traverses data

* Moving activation records
  * Problem: need to locate activation records for object
  * Possible solutions
    * Record invocations → too expensive on regular invocations
    * Search invocations → too expensive on moves
More Mechanics

- Moving activation records in Emerald
  - Maintain list of activation records in object
  - On invocation, mark activation record as "not linked"
  - On preemption, traverse stack for not linked records and link them
  - Why is this cheaper than recording invocations?

- Handling processor registers
  - Emerald uses callee-saved registers — why?
  - All registers included in moving activation record — why?
    - Scan current invocation stack for callee-saved values
Garbage Collection

- Two collectors: one local and one global
- Global collector
  - Builds on object descriptors
    - Represent out edges (and are already maintained by runtime)
  - Implements mark-and-sweep
    - Paints object in white, gray, and black — what do colors mean?
  - Uses clever techniques in face of mobility and concurrency
    - Mark moving object black (to prevent "outrunning")
    - Track unavailable nodes and inform them later
    - Mark process data before running (to support concurrency)
      - "Freeze" objects to avoid high initial cost
Performance
### Table II. Remote Operation Timing

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Time/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local invocation</td>
<td>0.019</td>
</tr>
<tr>
<td>Kernel CPU time, remote invocation</td>
<td>3.4</td>
</tr>
<tr>
<td>Elapsed time, remote invocation</td>
<td>27.9</td>
</tr>
<tr>
<td>Remote invocation, local reference parameter</td>
<td>31.0</td>
</tr>
<tr>
<td>Remote invocation, call-by-move parameter</td>
<td>33.0</td>
</tr>
<tr>
<td>Remote invocation, call-by-visit parameter</td>
<td>37.4</td>
</tr>
<tr>
<td>Remote invocation, remote reference parameter</td>
<td>61.8</td>
</tr>
</tbody>
</table>

*What do we learn from this table?*
* What do we learn from this experiment?

<table>
<thead>
<tr>
<th></th>
<th>Without mobility</th>
<th>With mobility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total elapsed time (in seconds)</td>
<td>71</td>
<td>55</td>
</tr>
<tr>
<td>Remote invocations</td>
<td>1,386</td>
<td>666</td>
</tr>
<tr>
<td>Network messages sent</td>
<td>2,772</td>
<td>1,312</td>
</tr>
<tr>
<td>Network packets sent</td>
<td>2,940</td>
<td>1,954</td>
</tr>
<tr>
<td>Total bytes transferred</td>
<td>568,716</td>
<td>528,696</td>
</tr>
<tr>
<td>Total bytes moved</td>
<td>0</td>
<td>382,848</td>
</tr>
</tbody>
</table>
So, How to Evaluate?

- This paper is based on incomplete implementation, features unconvincing evaluation.
- In general, we need a well-defined hypothesis and empirical evidence to (dis)prove hypothesis.
  - What would a good hypothesis for Emerald?
  - How would you collect empirical evidence for it?
What Do You Think?