Fine-grained Mobility (in Emerald)

Robert Grimm
New York University
 Altogether Now: The Three Questions

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

- Advantages of migration in general
  - Load sharing
  - Communications performance
  - Availability
  - Reconfiguration
  - Utilizing special capabilities
- Additional advantages of fine-grained object mobility
  - Data movement
  - Invocation performance
  - Garbage collection
Emerald Goals

- Provide mobility without sacrificing performance
  - Procedure calls in the local case
  - RPC in the 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 up to 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
  - They have a concrete type object (which has code)
  - They can be compared against an abstract type (interface)
  - What are the advantages/disadvantages of this instance-based object model?
Emerald Mobility: Basic Ingredients

- Five primitives
  - Locate, move, fix, unfix, refix (atomic unfix, move, fix)
    - Fix is stronger than move in what respect?
- Explicit location (through node object)
- Implicit location (through any other object)
- Attachments
  - Control what objects are moved together
  - Are transitive
  - Are not symmetric
Emerald Calling Conventions

- In general: Call-by-reference semantics
  - What is the problem in a distributed system?

- For efficiency: Automatic argument moving
  - Controlled by compiler (think, small immutable objects)
  - Controlled by programmer
    - Call-by-move
    - Call-by-visit
Emerald Processes

- Stacks of activation records
- Objects may move → What to do about activations?
  - Always return to original node
    - Leaves residual dependencies, which may limit availability
  - Move activations with objects
    - Need to be clever in implementation
Implementation
Three Types of Addressing Structures

- Global, local, and direct objects
  - What happens if 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 object descriptors
    - <timestamp, node>, OID
  - Every sent reference contains OID and forwarding address
  - Searching node follows up to two forwarding addresses
    - If this is unsuccessful, searching node broadcasts a search msg

- Why not keep a directory of nodes referencing an object?
Finding and Translating Pointers

- Problem: Emerald uses direct addresses
  - Local to a machine, need to be translated on move
- Solution: Object and activation record templates
  - Identify types (pointers, data, monitor locks) and locations

```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
```
Moving Objects

- Moving data objects
  - Messages include data area, translation information, and OID & forwarding address for global object pointers
  - Receiver allocates space, builds translation table, makes sure object descriptors exist, traverses data (templates!)

- 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
Moving Objects (cont.)

- Moving activation records (cont.)
  - Emerald solution
    - Maintain list of activation records
    - 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 need to be included in moving activation record
    - Requires scanning 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 system)
  - Implements mark-and-sweep algorithm
    - Paints objects white, gray, and black → What do colors mean?
  - Uses some clever techniques to deal with mobility and concurrency
    - Mark object black when moving it (to prevent “outrunning”)
    - Exchange information on unavailable nodes and inform them later
    - Mark process data before running (to allow concurrent operation)
      - Implemented lazily by “freezing” objects
Performance
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>
What Do You Think?