

Fine Grained Mobility in the Emerald System

Presented by
Mayukh Saubhasik
Erica Zhang

What is Emerald ?

- Emerald is a object based language and system designed for the construction of distributed programs
- Emerald was designed to support fine grained object mobility
- The entire emerald runtime along with all the emerald processes run as single process on the BSD Operating system

What is fine grained mobility ?

- Process migration moves the entire address space from one node to another.
- Coarse grained approach
- Mobility in emerald is on a per object basis
- A emerald object can encapsulate just pure data or data + code
- Thus this approach is more light weight and fine grained
- Objects present a cleaner form of modularization than a process

Why do Object migration ?

- Load sharing
- Communication performance – move objects which interact a lot onto the same node
- Availability
- Reconfiguration
- Utilizing special capabilities
- Invocation performance – Move parameter objects to remote site for duration of the invocation

Design goals for Emerald

- Performance overhead in the local case must be minimal
- A single object definition with single semantics for defining all objects
- Language based support for mobility – in the form of a whole new language :)

Emerald Objects

- A unique network wide name
- A representation i.e the data local to the object
- A set of operations that can be invoked on the object
- An optional process
- Each object carries its own code, no concept of classes
- However identically implemented objects on the same node will share code

Type System in Emerald

- Has the notion of abstract types
- Multiple objects can implement a single abstract type
- One object can also implement multiple abstract types
- Code is stored in a immutable concrete type object

Object Migration mechanism

- Copy the data portion of the object
- Data portion also includes the process stack, but no code
- If the receiving node doesn't contain a copy of the concrete type object encapsulating the code it copies it from another node (possibly from the source node)
- The code is dynamically linked into the emerald runtime kernel

Primitives for mobility

- Locate X -> Node where X resides
- Move X to Y -> co-locate X with Y
- Fix X at Y -> Object X is fixed on node Y
- Unfix X -> Unfix object X
- Refix X at Z -> Atomic unfix move and fix
- * Move is just a hint, the kernel may or may not move the object
- One can attach objects to each other, thus when a object is moved all the attached objects are moved as one single group

Parameter Passing

- Emerald uses call-by-object-reference parameter passing semantics for all cases
- To reduce the overhead of accessing remote parameters during a remote invocation, Emerald supports
 - > call-by-move: The argument object is relocated to the destination site
 - Call-by-visit: The argument object returns to its source after the invocation ends
 - * Compiler decides mode to be used

Emerald Process

- An emerald process is a thread of control which is initiated when an object with a process is created
- A process can invoke operations on its objects or any other object that it can reference

Process stack

- On a remote invocation the process moves to the destination node and invokes the object there
- The new activation record created for this invocation moves to the destination node and becomes the base of a new segment of the process stack
- Thus the invocation stack of a single emerald process may be distributed across multiple nodes

Implementation Details.....

- Emerald prototype was implemented to run on top of the DEC's Ultrix system (Based on bsd 4.2)
- The emerald kernel and all emerald objects on a single node execute within a single Ultrix address space

Types of objects

- Global -> Can be moved independently and can be referenced globally in the network
- Local object -> completely contained within another object. Moves along with the container object
- Direct object -> A local object whose data area is allocated directly in the representation of the enclosing object

Objects continued...

- Emerald variables contain local object references. I.e. The references are location dependent
- If the target object is not resident then the code will trap into the emerald runtime, which in turn will perform a remote invocation
- Therefore if the global objects are resident the overhead is comparable to a local procedure invocation

Finding objects

- Instead of keeping every node up-to-date about the location of each global object, use concept of forwarding addresses
- Node contains hashed tables mapping OID to Object descriptors
- Object Descriptors contains the latest available forwarding address
- Forwarding address has a time stamp and a node
- On a invocation a chain of forwarding addresses is followed to locate the object

Object Migration details (For pure data objects)

- Pass the data area of the object to be moved
- Along with translation information
- Global object pointers are passed as OID, forwarding address and address of the object's descriptor in the source node
- Use data layout templates to help with the translation task
- * In emerald memory references are direct, hence they have to be translated when the object moves

Object Migration for objects containing activations

- Lazy method to link all the activation records currently executing an object
- Only tag activation record as un-linked, link them to the object descriptor when preempted
- Un-link them when process terminates, but due to the lazy method used for linking, most of the cases there is nothing to be done
- The emerald compiler tries to optimize object addressing by storing them in registers

Object migration

- So, during migration these register contents have to be translated.
- Kernel sends along a copy of the registers used in an invocation along with the activation record
- The kernel uses the template to determine which of these registers is used to store addresses and then translates them on the destination node

Performance Measurements

- Various operation costs

Operation Type	Time/ms
local invocation	0.019
kernel CPU time, remote invocation	3.4
elapsed time, remote invocation	27.9
remote invocation, local reference parameter	31.0
remote invocation, call-by-move parameter	33.0
remote invocation, call-by-visit parameter	37.4
remote invocation, remote reference parameter	61.8

- Incremental costs wrt local argument

Parameter Passing Mode	Time/ms
call-by-move	2.0
call-by-visit	6.4
call-by-remote-reference	30.8

- Call by visit and call by move help a lot in improving the performance