Distributed Programming in Lua

Noemi Rodriguez
PUC-Rio
Distributed Programming

• shift to wide area
  – loose-coupling
  – asynchronism
  – highly dynamic execution conditions

• different settings require different paradigms and abstractions

how can programming language features help?
ALua - Asynchronous Lua

• asynchronism
  – wide area computing
    alua.send (dest, <string_with_chunk_of_code>)
• arrival of message is an event
• handler executes chunk of code
ALUA

lua.send(B, [[send(A, “print(‘.. c..’)”)]]])

send(A, “print(‘.. c..’)”)
ALUA PROGRAMMING MODEL

- compatible with interpreted languages
  - highly flexible but not very secure
- single-threaded
  - each event is handled to completion
Example: Job Management with ALUA

- local resource manager for Globus
- direct use of ALLua
- allocation, deallocation, and migration(?)
- system aspects
  - CPU and memory variability
- application aspects
  - bad parameters or starting points

➤ importance of interactivity
Programming Models

• ALua: low abstraction level
  – programs as state machines
  – lots of string manipulation
• many settings require more support...
Higher-level Abstractions: Classification

- libraries
  - awkward APIs
  - freely combined in applications
- specific languages
  - easier to use
  - support for specific paradigms
- reflection and extension
  - combined advantages...
ALua & abstractions

- DALua - distributed algorithms
- LuaRPC
- LuaTS - tuple space
- LuaPS - publish/subscribe
- ...

⇒ ease of integration: research & education
IMPORTANT FEATURES OF LUA

• functions as first-class values and other functional mechanisms
  – closures
• reflexive mechanisms allow us to redefine language behavior in case of exceptions
  – invocation of non-existing methods
• cooperative concurrency (coroutines)

→ high level abstractions can be easily built
DALua

• distributed algorithms library
  – very near to basic model
  – important as teaching tool

• DA classically described as a series of responses to events

example: classical Ricart&Agrawala algorithm for mutual exclusion

```plaintext
on request(ts, id) do
  ...

on oktogo do
  ...
```
EXAMPLE: MUTUAL EXCLUSION

classical Ricart & Agrawala

```lua
function mutex.enterCS (func)
    logicalclock = logicalclock + 1
    waiting = true
    local thisreq = {
        ["timestamp"] = logicalclock,
        ["proc"] = ad.self()
    }
    local procs = dalua.processes ("myapp")
    dalua.send(procs, "mutex.request", thisreq)
    thisreq.pending = table.getn(procs)
    thisreq.critical_section = func
    table.insert(requests, thisreq)
end
```
example: mutual exclusion

classical Ricart & Agrawala

function mutex.request (newreq)
  logicalclock = max(logicalclock, newreq.timestamp) + 1
  if busy then table.insert(deferred, newreq)
  elseif waiting then
    -- check if new request was issued earlier
    if haspriority(newreq, requests[1]) then
      dalua.send(newreq.proc, "mutex.oktogo", ad.self(),
      newreq.timestamp, logicalclock)
    else
      table.insert(deferred, newreq)
    end
  else -- not interested in critical region
    dalua.send(newreq.proc, "mutex.oktogo", ad.self(),
    newreq.timestamp, logicalclock)
  end
end
end
RPC

• RPC is often more comfortable than responses to events
  – critics

• LuaRPC
  – how to combine RPC view with asynchronism
  – and with "single-threadedness"

  – asynchronous invocations as a basis
function request()
  local acc, repl = 0, 0
  local peers = dalua.processes("myapp")
  local expected = table.getn(peers)
  function avrg (val)
    repl = repl + 1
    acc = acc + val
    if (repl == expected) then print("Current Value: ", acc/repl)
      end
  end
  for __,p in ipairs (peers) do
    luarpc.async(p, "currValue", avrg)()
  end
end

→ closures help deal with "unwinding the stack" problem
→ async fcts are 1st class as any other fct value
LuaRPC

- still, sometimes it is nice to work with synchronous view
  - synchronous RPC
  - futures

```lua
f = luarpc.sync(p, callback)
f(arg1, arg2)
```
Synchronous Invocations

• "blocking" semantics should allow incoming messages

• use of coroutines:
  – each new invocation is executed in a new coroutine
  – sync call invokes asynchronously and yields
possible inconsistent handling of globals but only at explicit points
  – investigation of compatible synchronization scheme
COMBINING PARADIGMS

• one same application can freely use different interaction paradigms
  – p/s, RPC, messages, ...
  – example: distributed ME algorithm can be used as part of RP implementation

• language features allow all of them to be seamlessly integrated into the language