Embedding Ruby into a Robotic Marine Laboratory

Brent Roman,
Christopher A. Scholin, Dianne I. Greenfield

Monterey Bay Aquarium Research Institute

www.mbari.org
About MBARI

Monterey Bay Aquarium Research Institute

- Part of our stated mission is to develop better instruments, systems, and methods for scientific research in the deep waters of the ocean
- Founded in 1987 by David Packard of HP
- Funded largely by the Packard Foundation
- Employs >200 in Moss Landing, CA
MBARI's 2nd Generation Environmental Sample Processor

• What it does
• How it works
• Current applications
• Future directions
• Ruby's Central Role
Machines to collect microbes are not new

- Continuous Plankton Recorder (CPR)
- First deployed on the R.R.S. Discovery 1925-27.
- Design driven by need to document “patchiness” of zooplankton
- Many modifications over the years
  - took ~10 yrs to become “operational”

Functional Requirements for Instruments that Detect Microorganisms and Gene Products

- Sample collection
- Cell concentration, preservation and disruption
- Separation of molecules based on physical properties
- Use of intermolecular reactions to reveal target molecules
  - lectin/carbohydrate
  - antibody/antigen
  - nucleic acid hybridization
  - receptor/target
  - enzyme mediated processing
- Optical and electrochemical signal transduction common
The Environmental Sample Processor (ESP)

- Rotating carousel with 100, 25mm pucks holding user-defined filters or solid phase media

- Fluid handling system permits autonomous collection of samples and timed application of multiple reagents in situ, subsurface

- Sample processing protocols written as a series of simple commands such as “Load Filter”, “Pull x ml reagent y”, “heat z °C 30 min”, etc.

- Two-way communication
Functions of the ESP

• Development of DNA probe arrays (near real-time)
  – collect sample
  – create cell homogenate
  – discharge sample collection filter > load probe array filter
  – develop probe array > image > transmit result

• Sample Archival
  – for whole cell analysis (probes, microscopy)
  – for nucleic acids (gene libraries)
  – for toxins (domoic acid)

• Whole Cell Hybridization
  (FISH -- Fluorescent In-Situ Hybridization)
  – collect, fix sample
  – label target species with probes
  – recover filter and view using epifluorescence microscopy
ESP's DNA Probe Arrays

- Array map – species/group specific DNA probes
- Sandwich hybridization assay (SHA) chemistry
- Image stored & sent to shore via radio modem

Heterosigma akashiwo

Alexandrium catenella

Pseudo-nitzschia australis

HET1

Pseudo-nitzschia multiseries

controls/orient image

16,000 cells/ml lysate *P. australis*
Multiple Taxa on Single Array

- 125 ml whole water
- 0.45 μm filter
- 2 ml homogenate
The Sandwich Hybridization Assay

 branded capture probe (streptavidin)

18S rRNA

signal probe (dioxygenin)

enzyme substrate

HRP conjugate

array spot intensity \propto absorbance (450 nm)

Verification by matching 96-well bench run

Photo courtesy of A. Haywood

HRP = Horse Radish Peroxidase

array

Imaged array

Photo courtesy of A. Haywood
Current Applications

- Develop DNA probe arrays (~2 h) for identification of:
  - Harmful Algal Blooms (plankton)
  - Bacteria
  - Invertebrate Larvae

- Sample archival to preserve cell structure
  - whole cell archival
  - (FISH) hybridization

- Domoic acid analysis (in progress)
Future Directions

• Deep ESP: Deploy on Ocean Bottom at >2000m

• Complete in situ domoic acid analysis methodology

• Increase array sensitivity

• Integrate PCR module

• The 2G ESP is a sampling tool - protocols can be tailored for user requirements
  - Modifying chemistry as appropriate
  - Development of new probes
  - Adjusting sampling procedures
  - Other uses?
NASA ASTEP: Astrobiology Science and Technology for Exploring Planets

A probe lands on Titan in this artist's concept of one of Saturn's moons. Scientists hope that someday, probes can explore the oceans of the large icy satellites orbiting the outer planets.

Study of off-world seas next big step, experts say
Acknowledgments

The David and Lucile Packard Foundation

National Oceanographic Partnership Program (NOPP) funds allocated by the National Science Foundation (NSF OCE-0314222)

Why Choose Ruby for an embedded system like ESP?

• Can be configured to run in small memory footprint
• Open Source
  – Universities have cheap labor and little $$$
  – We can fix (or break) it ourselves as we see fit
• Readable, but terse syntax
  – No white space rules and no excess punctuation
  – Make it suitable for use as an interactive shell
  – Useful subset of Ruby is approachable for non-programmers
• Everything is an Object!
  – But, no OO baggage creeps into procedural definitions.
  – Most molecular biologists don't “get” OO. They write recipes.
• User defined Exception Objects
• Multi-Threading
ESP Software Overview

Mission

User Libraries

System Libraries

Device Drivers

Device Models

Hardware

Simulation

Normal Operation

via \( I^2C \) Gateway
Real-Time Ruby?

- ESP Servo control requires hard real-time response
  - Sample rates of as high as 64hz
  - Event response in tens of microseconds
- Ruby cannot deliver this today (at low-power)
- So, true to OO, make it someone else's problem!
  - Distribute Servo Control among networked micro-controllers
    - TI's MSP 430 family consumes ~1mW/Mhz
    - Ensure each runs identical, stable 'C' firmware
    - Each configured by the Ruby application at system start up
- Decouple supervisory from servo control
  - But, this cannot be done for many real-time systems...
    - You're not going to fly a plane this way!
2\textsuperscript{nd} Generation ESP

Electrical Block Diagram

- PC/104 Host Processor
- Interconnect / Motherboard
- Analytical Modules

**External Power 10 - 16V**

- RF Modem
- Contextual Sensors

**RS-232**

**5V & 3.3V Power Supplies**

**I\textsuperscript{2}C Serial Bus**

- Sampler
- Collection
- Process
- Manipulator
- Storage

**Distributed Controllers aka “Dwarves”**

- ~2 Watts
- ~450 mW
2G ESP
PC/104 Host Processor

- Inexpensive (<$250US) COTS product
  - Technologic model TS-7200
- ARM9 CPU with 32MB DRAM + 16MB flash
  - Compact Flash storage used only for data logging
  - USB Host for Camera interface
  - 10/100Mb ethernet
  - Customized, Vendor supported Linux 2.4 kernel
- Approximately 2 Watts at idle, 4 Watts max.
- Power cut off entirely while “sleeping” between samples
  - Total power consumed while “sleeping” is <50mW
- Boots to application in <10 seconds from power on
Trimming Ruby

- Fitting Linux/Ruby in 16MB flash requires some thought
  - Use of JFFS2 filesystem compresses flash by ~40%
  - But, Linux kernel takes 2MB leaving 14MB
  - Ruby core interpreter is small (~ 500kByte)
  - Native code libraries are large (~8MB base install!)
  - “Pure” Ruby ASCII code is extremely compact!
- Avoid use of Native Libraries in favor of pure Ruby
  - Example: curses.so links with libcurses.so, libterm.so
    - Curses support brings in close to 1MB of binary
- Hack core Makefile to skip building unneeded libraries
  - Removed tcltk, tk & curses from Ruby 1.68 build
  - Reduced /usr/lib/ruby to <0.8MB, /usr/lib < 0.5MB
  - Still includes IRB, networking, readline, termios, ...
SourceRef:
Ruby Source code introspection

- Display source for specified method in a text editor
- List all source files that contribute code to specified class
- Reload (modified) methods/objects
  - `edit MyObject.method :foo; reload MyOjbect`
- Display source of any method on an exception backtrace
  - Goes to the offending source after an error
- About 10kBytes for the Ruby script “sourceref.rb”
  - Added two tiny core methods to v1.68
  - `__LINE__` & `__FILE__` may work on > Ruby versions
  - See `ruby-talk/21555` for details
- Serves as tiny IDE for our memory constrained systems
ESP seeks GUI

- We want to animate a cartoon of machine's physical state
  - Dynamically changing
  - TCP/IP Networked over a 50kbit/sec radio link
- Doesn't need to be slick, but be responsive and intuitive
- Platform independent
  - Something that works in a web browser preferred
  - No need to install anything on the client side
- Rich Kilmer's Ruby-to-Flash bridge sounds interesting
- Any other ideas very welcome!
Ruby Threads: The Good

- Non-native “Green Threads” works very well for the ESP
  - We have one instance of Ruby interpreter per process
- Thread scheduling seems to be fast and memory efficient
  - Non-native threading may even be superior in this
  - Is there a benefit of native threads on a uniprocessor?
- Core IO and Process classes handle threads well
  - Output = `myTimeConsumingCommand`
    - Other threads run while this completes
    - This was, for me, a pleasant surprise.
Ruby Threads:
Some Bad

- Standard Library Mutex class not guaranteed to be FIFO
  - Lock not held while passing to the first waiting thread
  - Third thread can then grab the lock without waiting
- Mutex.lock must hold lock after waking waiting thread
  - Not fixed in Ruby 1.8

```ruby
def lock
  while (Thread.critical = true; @locked)
    @waiting.push Thread.current
    Thread.stop
  end
  @locked = true
  Thread.critical = false
  self
end

def unlock
  return unless @locked
  Thread.critical = true
  @locked = false
  begin
    t = @waiting.shift
    t.wakeup if t
    rescue ThreadError
      retry
  end
  Thread.critical = false
  begin
    t.run if t
    rescue ThreadError
  end
  self
end
```
Ruby Threads:
More Bad

• Thread.sleep does not clear Thread.critical
• Ruby code cannot clear Thread.critical before sleeping
  – Just as one can't clear it explicitly before Thread.stop
  • Another thread may run between the clear & stop
  • This is why Thread.stop clears Thread.critical
• My solution was to add the 'C' extension: Thread.doze
  – Just like .sleep, but clears Thread.critical first
• Is there any reason Thread.sleep should not be changed?
  – Could one ever want to sleep with Thread.critical set?!
• BTW -- there is a similar issue with the select method...
• See ruby-core:4053
Ruby Threads: The Ugly

- Ruby Trap handlers are invoked even if Thread.critical
- So, it is very difficult (impossible?) to handle traps safely
- Interpreter calls rb_thread_schedule while Thread.critical
  - When this happens, Thread.critical remains set
    - Thread.pass
    - Readline extension
      - Random threads become “critical” spontaneously
- Patch in Ruby-core:4039, demoed in Ruby-core:4249
  - Signal handling deferred while Thread.critical
  - rb_thread_schedule revised to clear Thread.critical
  - Readline now reads keyboard via Ruby sysread method
Ruby Threads: Why we should fix this before 2.0

- These problems have nothing to do with whether threading implementation is native, pthreads, green, blue or yellow
  - Thread.critical is the basic Ruby threading interlock
  - All higher-level Mutexes, Queues, etc. build upon it
- This won't go away when threading finally gets implemented the “Rite” way
  - Thread.critical's app-level behaviour must change
- Has no one else run into these issues?
  - If so, how did you cope?
  - If not, how did you avoid them?