

Ruby and the Java Debug Wire Protocol

What were we thinking?

DARPA UltraLog Project

- I get paid to write and maintain a Ruby framework on a DARPA project
- Scripted control of a 300 machine distributed multi-agent Java framework
- A single Ruby script controls the execution of the 100+ Java VMs per experiment run
- Interaction between Java & Ruby is limited to:
 - Log4J ‘push events’
 - Servlet ‘pull events’
 - Servlet-based reporting

Summer 2003 - Rich Meets Chad

- Chad and I started talking about what I do for DARPA
- I mentioned the Ruby projects on my plate
 - FreeRIDE, FreeBASE, Jabber4r
 - Utopia (MetaUI), Rendezvous, Ratify (testing)
- I mentioned the idea of having Ruby debug Java applications at a low level
 - It would allow my project to have better control of the JVMs
- We both thought that would be cool

Java Platform Debug Architecture (JPDA)

- Java Virtual Machine Debug Interface (JVMDI)
 - Low level native interface
 - Service JVM must provide for debugging
- Java Debug Wire Protocol (JDWP)
 - Defines format of info and requests transferred between debugging process and debugger from end
- Java Debug Interface (JDI)
 - High level programming interface for remote debugging
- Java 1.4 delivers full speed debugging!!!

What to Implement?

- JVMDI (N/A)
 - For JVM writers
- JDWP (Rich)
 - Packet-based (over TCP) binary protocol
 - 78 Packet formats
- JDI (Chad)
 - Needs to be implemented on top of JDWP
 - High level Ruby API (iterators, blocks, etc)



Implementing JDWP

Convert Protocol to Ruby “Metadata”

ClassesBySignature Command (2)

Returns reference types for all the classes loaded by the target VM which match the given signature. Multiple reference types will be returned if two or more class loaders have loaded a class of the same name. The search is confined to loaded classes only; no attempt is made to load a class of the given signature.

Out Data

string	<i>signature</i>	JNI signature of the class to find (for example, "Ljava/lang/String;").
--------	------------------	---

Reply Data

int	<i>classes</i>	Number of reference types that follow.
Repeated <i>classes</i> times:		
byte	<i>refTypeID</i>	Kind of following reference type.
referenceTypeID	<i>typeID</i>	Matching loaded reference type
int	<i>status</i>	The current class status .

Error Data

VM_DEAD	The virtual machine is not running.
-------------------------	-------------------------------------



```
set.add_command :ClassesBySignature do |cmd|
  cmd.description = "Returns reference types for all the classes loaded by the target VM which
  match the given signature. Multiple reference types will be returned if two or more
  class loaders have loaded a class of the same name. The search is confined to loaded
  classes only; no attempt is made to load a class of the given signature."
  cmd.out_data :string, :signature, "JNI signature of the class to find
  (for example, 'Ljava/lang/String;')."
  cmd.reply_data :int, :classes, "Number of reference types that follow."
  cmd.repeat(:classes) do
    cmd.reply_data :byte, :refTypeID, "Kind of following reference type."
    cmd.reply_data :referenceTypeID, :typeID, "Matching loaded reference type"
    cmd.reply_data :int, :status, "The current class status."
  end
  cmd.error_data :VM_DEAD, "The virtual machine is not running."
end
```

Matadata Structures

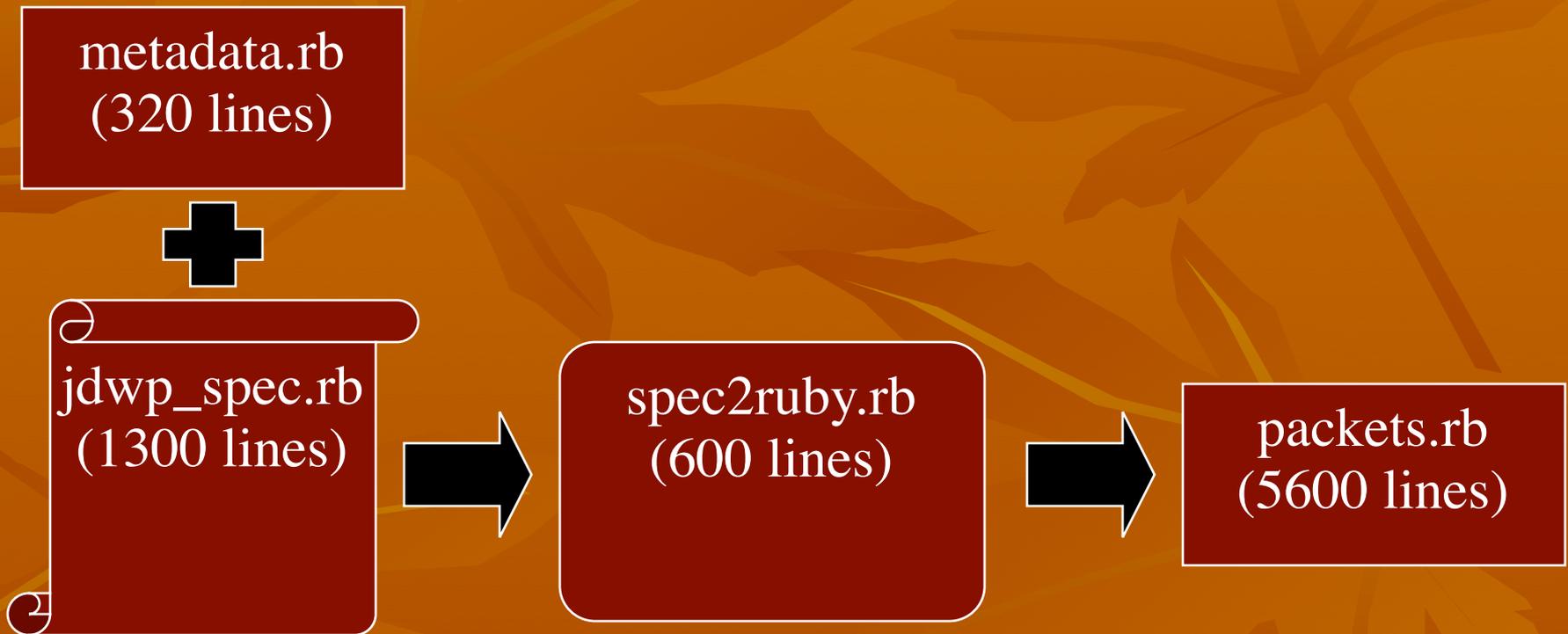
- Command sets
- Commands
- Documentation
- Repeat Structures
- Case/When Structures
- Data Elements
- Error Data Elements

```
JDWP.add_command_set :Event, 64 do |set|
  set.add_command(:Composite, 100) do |cmd|
    cmd.description = "Several events may occur
    may be more than one breakpoint request
    location as a breakpoint request. The
    uniformity, a composite event is always
```

```
cmd.reply_data :int, :events, "Events to
cmd.repeat(:events) do
  cmd.reply_data :byte, :eventKind, "Event
  cmd.case(:eventKind) do |event|
    event.when :VMStart, EventKind::VM_START
    event is received before the main thread
    executed. Before this event occurs
    of system classes have been loaded
    not explicitly requested."
    cmd.reply_data :int, :requestID, "Request ID
    automatically generated"
```

```
cmd.error_data :INVALID_OBJECT, "If this
cmd.error_data :VM_DEAD, "The virtual machine
```

Code Generate the Protocol



The spec2ruby Generator

```
module JDWP
  class Generator
    doc?
    initialize(code)
    output(text="")
    wrap(text, len=80)
    deindent
    indent
    generate_with_documentation
    generate
    output_header
    jdwp_module
    packets_module
    end_statement
    command_set(set)
    command_set_lookup
    command_lookup(set)
    command(cmd)
    command_structs(cmd)
    reply_struct(cmd)
    case_helper_methods(cmd, element=nil)
    output_when_structs(cmd, ws)
    output_when_methods(cmd, ws)
    output_rdoc_variable(name, type, description)
    command_constructor(cmd)
    command_decode(cmd)
    decode_repeat_structure(cmd, repeat, var_name)
    decode_case_statement(cmd, case_statement, varprefix="")
    decode_field(name, type)
    command_encode(cmd)
    encode_repeat_structure(cmd, repeat)
    encode_case_statement(cmd, case_statement)
    encode_field(name, type)

    def generate
      output_header
      jdwp_module
      packets_module
      command_set_lookup
      JDWP.command_sets.values.sort.each do |set|
        next if set.commands.size==0
        command_set(set)
        command_lookup(set)
        set.commands.sort.each do |cmd|
          command(cmd)
        end
      end
    end_statement
  end
end_statement
end_statement
end
```

Generator Output => packets.rb

- 5600 lines of fully RDoc'd code
- One Class per packet type
 - Subclass of JDWP::Packets::Packet
 - Struct(s) for complex constructor data
 - Struct for Reply
 - Accessors for data
 - **encode** method
 - **decode** method

ClassesBySignature Example

```
##  
# ClassesBySignature Command(2)  
# Returns reference types for all the classes loaded by the target VM  
# which match the given signature. Multiple reference types will be  
# returned if two or more class loaders have loaded a class of the same  
# name. The search is confined to loaded classes only; no attempt is made  
# to load a class of the given signature.  
#  
class ClassesBySignature < JDWP::Packets::Packet  
  SET_NUMBER=1  
  NUMBER=2  
  attr_reader :reply  
  attr_accessor :error_code
```

```
##  
# Reply Structure for ClassesBySignature  
# packet:: [ClassesBySignature] Packet that this is a reply to  
# classes:: [Array] Array of ClassesBySignature::Classes Structures  
Reply = Struct.new(:packet, :classes)
```

ClassesBySignature Example

```
##
# Classes reply substructure.
# refTypeTag:: [byte] Kind of following reference type.
# typeID:: [referenceTypeID] Matching loaded reference type
# status:: [int] The current class status.
Classes = Struct.new(:refTypeTag, :typeID, :status)

attr_accessor :signature

##
# signature:: [string] JNI signature of the class to find (for example,
#                 'Ljava/lang/String;').
#
#
def initialize(signature)
  @signature = signature
  super()
end
```

ClassesBySignature Example

```
##
# Encode this ClassesBySignature packet.
# return:: [String] The packed string
#
def encode
  encode_array = [0, @packet_id, 0, SET_NUMBER, NUMBER]
  encode_string = "NNCCC"
  encode_array << @signature.size
  encode_array << @signature
  encode_string << "NA#{@signature.size}"
  packed = encode_array.pack(encode_string)
  packed[0,4] = [packed.size].pack('N')
  packed
end
```

ClassesBySignature Example

```
##
# Decode supplied string into this ClassesBySignature packet.
# session:: [JDWP::Session] The current session (used for sizes of values)
# data:: [String] The encoded data.
# return:: [ClassesBySignature::Reply] The reply object with decoded values.
#
def decode(session, data)
  @reply = Reply.new(self)
  classes_count = data.unpack('N')[0]
  data[0,4] = ''
  @reply.classes = []
  classes_count.times do
    current_classes = (@reply.classes << Classes.new).last
    current_classes.refTypeTag = data.unpack('C')[0]
    data[0,1] = ''
    current_classes.typeID = data[0, session.reference_type_id_size]
    data[0, session.reference_type_id_size] = ''
    current_classes.status = data.unpack('N')[0]
    data[0,4] = ''
  end
  @reply
end
end
```

require “jdwp”

- packets.rb
 - Generated code
- constraints.rb
 - JDWP constants from the specification
- packet.rb
 - Encode/decode values and packet header
- session.rb
 - Manage connection, send/receive packets, events
- socket_transport.rb
 - TCP/IP based transport, send/receive data

Simple JDWP Example

```
require 'jdwpp'

include JDWP

session = Session.new
session.transport = SocketTransport.new("127.0.0.1", 8181)
session.abort_on_error = true
session.start

puts "Suspending Java..."
session.send(Packets::VirtualMachine::Suspend.new)

sleep 10
puts "Resuming Java..."
session.send(Packets::VirtualMachine::Resume.new)

session.stop

exit
```



Implementing JDI

JDI Overview

- High level Ruby classes to represent Java debug structures
 - JavaVirtualMachine
 - Suspend, resume, exit, dispose, capabilities, version, classpaths, each_class, each_thread, each_thread_group
 - Thread, ThreadGroup
 - Class, Method, Variable
 - Location, Breakpoint
- Hides JDWP packet complexity

Simple JDI Example

```
require 'jdi'  
  
jvm = JDI::JavaVirtualMachine.connect_via_tcpip('localhost', 8081)  
  
jvm.abort_on_error = true  
  
jvm.suspend  
sleep 10  
jvm.resume  
  
jvm.dispose
```



Demo

Summary

- RubyJDWP lets you debug Java applications from Ruby
 - Low level packets (jdwp)
 - Metadata driven code generation
 - High level API (jdi)
- JDWP library complete
- JDI library started
 - Moving on to Breakpoints, Locations
- Project on RubyForge
 - <http://www.rubyforge.org/projects/rubyjdwp>