Utilizing the ZBX protocols

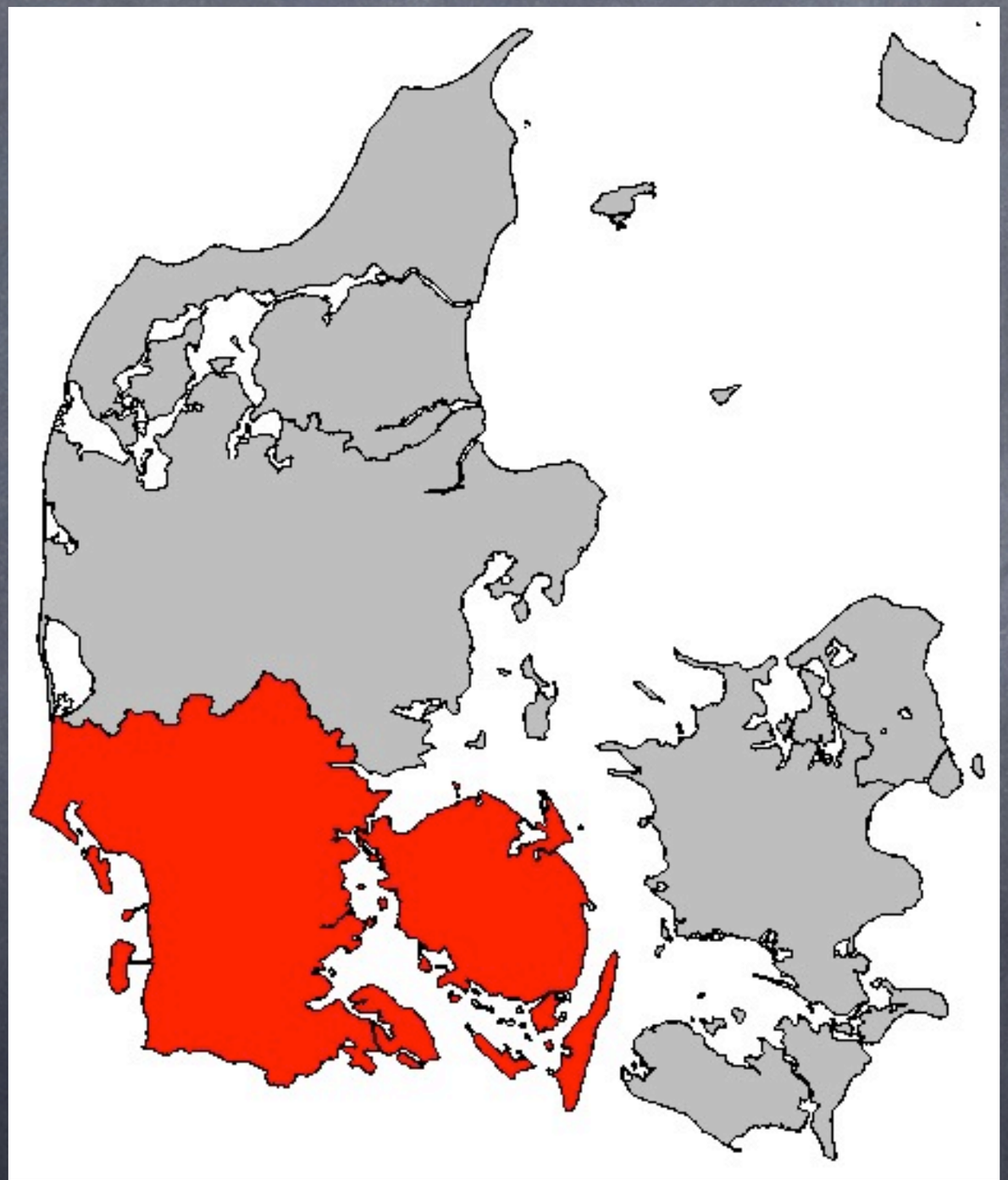
Who are we?

- Administrative region
- Prime responsibility is governing Health Care institutions (Hospitals, various Treatment Centers, etc)
- I work within the IT staff

Facts

- Population of ~1.2 million citizens
- Covering 12.200 km2
- 14 Hospitals, lots of smaller institutions
- 26.000 employes
- Using ZBX for monitoring of servers, infrastructure and applications (network monitoring is done by a different tool)

Pretty Map :)



ZBX Stats

- ~1.700 hosts / ~1500 NVPS
- 1 ZBX Server, 5 Proxies, 2 MySQL instances flying of Fusion IO
- 900+ different applications
- Win NT -> Win 2012, 4-5 different Linuxes, AIX, Solaris
- Every major storage vendor is represented
- Lots of different appliance boxes

What I want to talk about today...

- Background story
- Briefly about the ZBX protocols
- Example #1 : Windows Service restarts
- Example #2 : Near-realtime graphing
- Example #3 : Host interrogation
- Example #4 : Application monitoring API

Background Story

- We want to do stuff without having to compromise
- We need scalability
- We want architectural freedom
- We want to avoid maintenance nightmares
- YMMV

Briefly about the protocols

- Mimic the communications between ZBX components (agent, server, proxy, java gateway) *programatically*
- Multiple versions (just use the simplest invocation and be done with it)
- *Very* well documented on the Wiki
- Lots of sample code on the net in different languages (all our stuff is done in Ruby)

Example #1 - Windows Service restarts

- We use the services[] item for checking services that are set to auto start but are not in a running state.
- If multiple services are not running it returns a list -- one line per service which sc.exe cannot grok :(
- Instead of writing a script and having to maintain that on every host we implemented a workaround on the server side :)

Example #1

```ruby
#!/usr/bin/env ruby
require 'socket'
require 'rubygems'
require 'syslog'

# sending commands
def send_data(service)
        # note what we did for future reference and to stdout
        puts "#{@client}:system.run[\"c:\\windows\\system32\\sc.exe start #{service}\",nowait]\n"
        log("#{@client}:system.run[\"c:\\windows\\system32\\sc.exe start #{service}\",nowait]\n")

        # open socket connection and send request
        client = TCPSocket.open(@client, 10050)
        client.write("system.run[\"c:\\windows\\system32\\sc.exe start #{service}\",nowait]\n")
end

# syslog
def log(message)
        # $0 is the current script name
        Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning message }
end

# main execution
begin
        # fetch arguments
        @client = ARGV[0]
        @services = ARGV[1].split(/\n/).reject(&:empty?)

        # loop over services and execute one remote command per service
        @services.each do|service|
          send_data(service)
        end
```

Example #2

- We wanted high resolution graphs without destroying performance
- API call finds all groups / hosts / numerical items and caches them in Redis
- Graphing frontend issues AJAX requests to our internal ZBX REST API which fetches the data from the agent
- Multiple graphs (add / remove) on-the-fly for correlation are in my master branch :)

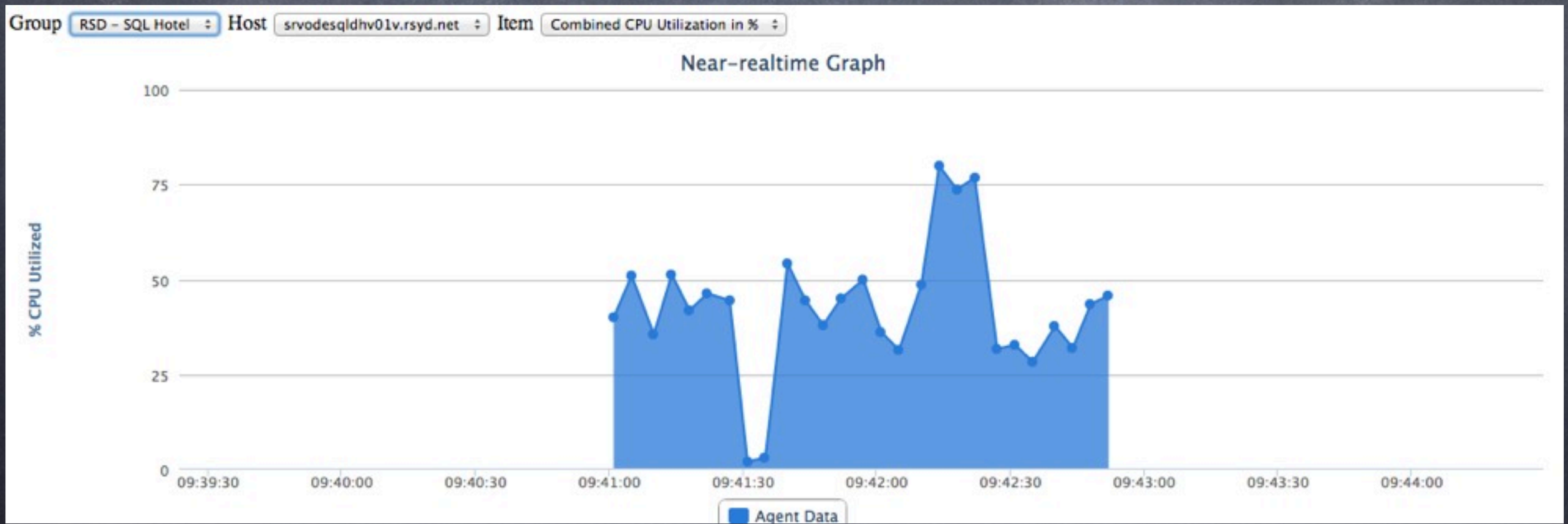
Example #2 - continued

```
get '/fetch_from_agent/:hostname/:item' do
    content_type :json
  halt 400, "Required parameter 'item' missing " unless params[:item].size >= 4
  halt 400, "Required parameter 'hostname' missing " unless params[:hostname].size >= 4

  host = params[:hostname]
  item = Base64.decode64(params[:item])
  ap item

  # fetch data from agent
  @data = collect_data(host,item)
  ap @data

  # output data
  erb :real_time_graf
end
```



Example #3 & #4 - prerequisites

- Resqueue - distributed message queue on top of Redis
- Supports multiple queues, builtin error handling (failure queue) and has lots of nifty features
- Scalable worker model
- In short : stuff gets put into a queue and later drained / processed by workers

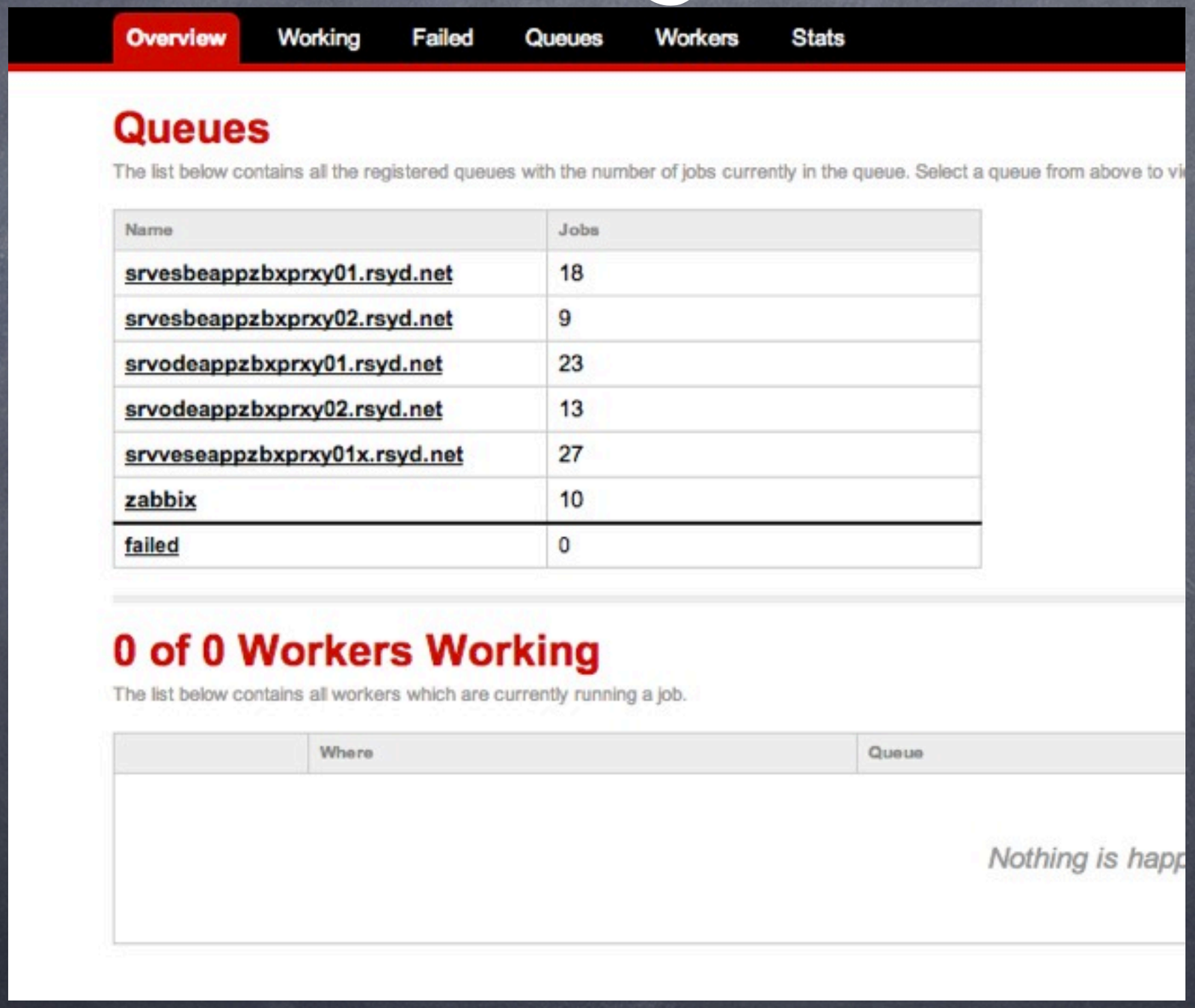
Example #3 -Host Interrogation

- We wanted to ask your infrastructure questions and get feedback within minutes
- We needed both predefined jobs and ad-hoc data queries

Example #3 - Host Interrogation

- Job gets submitted via UI or script
- Powered by a mix of items & userparams
- API looks up the hosts in question and places a job in the corresponding queue (one per proxy)
- Workers on each proxy process the queues and submit the job output into our datastore

Example #3 - Host Interrogation



Example #4 - App. monitoring API

- We wanted to collect metrics from applications (errors, response times, counters, etc)
- We wanted it to be versatile and easy to consume
- We must incur as little latency as possible to our consumers.
- We needed it to be scalable

Example #4 - continued

- We build an API that consumes JSON or XML
- Exposes 3 transport mechanisms (UDP/TCP/HTTP)
- UDP is perfect for metric collection
- TCP is perfect for important signaling
- HTTP is perfect for frontend stuff (or stuff that cannot open socket connections)

Example #4 - continued

- Reusing our existing Resqueue deployment
- Recv & send operation are disconnected (async) so that latency / overhead are kept at a minimum level
- Protecting the ZBX server from abusive / misbehaving clients (slow queue draining)
- API implemented on every proxy to limit exposure to network latency

Example #4 - continued

```ruby
require 'socket'
require 'json'

# generate data
str = { "host" => "test-host", "key" => "test-key", "value" => "1234" }
data = JSON.generate(str)

# push data
s = UDPSocket.new
s.send(data, 0, 'localhost', 3000)
s.close
```

```ruby
require'eventmachine'
require 'resque'
require './worker.rb'
require 'json'

class UDPHandler < EM::Connection
  # handle the UDP packet
  def receive_data(data)
    # parse the incoming JSON data
    body = JSON.parse(data.to_s) # now contains a valid json object of the POST body

        # TODO : log errors to stdout !

    # munge data onto queue
    Resque.enqueue(ZBXAPICall,body)
  end
end

# kick of reactor
EM.run { EM::open_datagram_socket('127.0.0.1','3000',UDPHandler) }
```

Example #4 - continued

```
zbx_str = JSON.generate(data)

# needed for ZBX sender protocol
data_length = zbx_str.bytesize
data_header = "ZBXD\1".encode("ascii") + \
              [data_length].pack("i") + \
              "\x00\x00\x00\x00"
# concat data
@data_to_send = data_header + zbx_str
ap @data_to_send

# open socket & push data
s = TCPSocket.new("zbx.rsyd.net", 10051)
s.write @data_to_send.to_s

# check response
response_header = s.recv(5)
if not response_header == "ZBXD\1"
  puts "response: #{response_header}"
  raise 'Got invalid response'
end

response_data_header = s.recv(8)
response_length = response_data_header[0,4].unpack("i")[0]
response_raw = s.recv(response_length)

# close socket
s.close

# sleep 0.5 seconds to allow the socket to close
sleep 0.5

response = JSON.load(response_raw)
ap "\n Response : #{response}\n"
end
```

My ZBX wishlist

- in-band timeout signaling
- conditional logic in the ZBX server!
- finish the !@# API :)

THANK YOU !