

Date: 2019-09-03 Author: Ian Rash Index: #4

Now that we have learned some interesting things about the communications, let's see if we can write our own suite of tools to work with the camera.

Before we start, I'll be using the wonderful programming language from the boys over at Crystal. It's a very fast, statically typed language, with very zen and down to Earth syntax. The meta-programming aspect of the language is very cool!

First thing we should do is talk about design.

From what we can see right now the camera and client go through different states, things like sending the DBP, waiting for the BPA, that sort of thing. As such, we will want to make a simple state machine. Here is the basic outline of the functionality we need.

```
class Client
  STATES = [:nothing,
    :send_dbp,
    :wait_for_dbr,
    :send_bcp,
    :handle_bp_handshake,
    :main_phase,
    :closing,
    :closed]
  getter state : Symbol = :nothing

  # Change the state of the client.
  def change_state(state)
    @state = state
    LOG.info "Changing to #{@state}"
    tick # rerun the tick since the state immediately changed and there is new stuff to do.
  end
end
```

The idea is that whenever we change the state of our client, we are moving to the next phase of communication.

Next big thing we need to talk about are Fibers. If you are unfamiliar with them, I would suggest reading [this article](#).

We have two main tasks, we need to have a **Data Loop**, which will take data in from ports (depending on the state of the client, choose which port to use), and then take that data and shovel it off into a [channel](#). Then we will have the **Tick Loop**, which will run the decisions to be made on the incoming data.

Starting the client

One of the first things we want to be able to do is start the client, and have it "idle" while waiting for connections.

```
# Sets up the client by binding the udp sockets to addresses and ports
def setup
  # Don't resetup the client if its is_running
  if !is_running?
    LOG.info("Opening ports")
    # Our socket for sending UDP data to the camera
    @data_sock.bind DATA SOCK_SRC
    # Super important to enable this or else we can't broadcast to 255.255.255.255!
    @data_sock.setsockopt LibC::SO_BROADCAST, 1
    # Our socket for sending discovery broadcasts.
    @db_sock.bind DB SOCK_SRC
    @db_sock.setsockopt LibC::SO_BROADCAST, 1

    LOG.info("Ports opened")
    return
  else
    LOG.error "CANNOT SETUP CLIENT WHILE IT IS RUNNING!"
    raise "CANNOT SETUP CLIENT WHILE IT IS RUNNING!"
  end
end

def run
  # Dont allow the client to run again!
  if !is_running?
    @is_running = true
    # Change the state so it will attempt to discover a camera on the network
    @state = :send_dbp
    #Start our fibers
    start_data_fiber
    start_tick_fiber
  else
    LOG.error "ALREADY RUNNING CLIENT!"
  end
end
```

The main idea is that we set up the client, making a variable that will track if it's running or not. It also sets up the ports we need to communicate our discovery broadcast, as well as the UDP data socket. We also have a method run which will stop the method if it is already running, change the state to the first phase, then start our fibers for data and tick.

Fiber Design

```
# Channel that will communicate data back to the tick fiber
@data_channel = Channel(Tuple(String, Socket::IPAddress)).new

# Fiber which deals with incoming packet data, holds this data temporarily and then sends the data via @data_channel
@data_fiber : Fiber = spawn {}

# Fiber which handles the decision process of handling the state of the client. Receives incoming data from data_channel
@tick_fiber : Fiber = spawn {}
```

Then we want to write our two "start fiber" methods. We reassign the fiber variable for the action, trap any exceptions and rescue them out (specifically for the case of the ports shutting down while sending data), create a while loop with `is_running?` as a condition, and then fill in the meat of the fibers.

Data fiber will block on the data socket, then transfer any data received to the channel.

Tick fiber will run the update tick continually until the client is no longer running.

Start the fiber which blocks for incoming data, then forwards it to a channel.

```
def start_data_fiber
  @data_fiber = spawn do
    begin
      # Only run this fiber while is_running, if not exit
      while is_running?
        # Will block execution
        packet = data_sock.receive
        @data_channel.send(packet)
      end
    rescue e
      LOG.info "DATA EXCEPTION #{e}"
    end
  end
end
```

Start the fiber which contains the tick logic.

```
def start_tick_fiber
  @tick_fiber = spawn do
    begin
      # Only run this fiber while is_running, if not exit
      while is_running?
        tick
      end
    rescue e
      LOG.info "TICK EXCEPTION #{e}"
    end
  end
end
```

Closing the client

When the data fiber is blocked, there is no way to "kill" the fiber when we want the program to exit. Instead, we need to simulate some data into the socket, freeing the fiber to execute and close itself.

First we turn `@is_running` to false, then send the "unblock fiber data" into the data socket. We then use a `Fiber.yield` to give control back to the data fiber. While we can't give control back directly to the data fiber, we know that the tick fiber will most likely be blocked and the data fiber will get its turn.

We then close the sockets, change `_state` to closing, and set the target camera back to 0.

```
UNBLOCK_FIBER_DATA = "e127e855-36d2-43f1-82c0-95f2ba5fe800"
def close
  LOG.info("Closing client")
  @is_running = false
  change_state(:closing)

  # This line unblocks the @data_fiber
  @data_sock.send(UNBLOCK_FIBER_DATA, Socket::IPAddress.new("127.0.0.1", DATA_SOCK_SRC.port))

  # Force a fiber change to go to the other fibers to end them
  Fiber.yield

  # Now we can close the sockets
  @data_sock.close
  @db_sock.close

  # Reset the target_camera
  new_target "0.0.0.0", 0
  change_state(:closed)
  LOG.info("Closed client")
end
```

Tick Layout

In the tick method, we want to layout a basic pattern of transitions for the client.

```
# Main decision making function
def tick
  if state == :nothing
    # Do nothing
  elsif state == :send_dbp
    send_dbp
    change_state :wait_for_dbr
  elsif state == :wait_for_dbr
    info = wait_for_dbr
    if info
      @target_info = info
      change_state :send_bcp
    else
      change_state :send_dbp
    end
  elsif state == :send_bcp
    send_bcp
    change_state :handle_bp_handshake
  elsif state == :handle_bp_handshake
    handle_bp_handshake

    if has_target?
      change_state :main_phase
    else
      change_state :send_dbp
    end
  elsif state == :main_phase
    # Do ping pong, etc in here
    main_phase
  else
    raise "THERE WAS A BAD IN TICK!"
  end
end
```

DBP and DBR

We need to send our DBP, so we can start to discover cameras. This process is fairly straight forward.

```
# Source address for the discovery packet
DB SOCK_SRC = Socket::IPAddress.new("0.0.0.0", 6801)

# Destination address for the discovery packet
DB SOCK_DST = Socket::IPAddress.new("255.255.255.255", 8600)

# Discovery packet data
DBP = "\x44\x48\x01\x01"

# Send the DBP to the camera
def send_dbp
  LOG.info("Sending DBP")
  db_sock.send(DBP, DB SOCK_DST)
  LOG.info("Sent DBP")
end
```

After sending we want to wait until the camera responds with the DBR. We also want to parse some of the data coming in, as it has some juicy info we might want to reference later.

```

# Size of the discovery packet reply
DBR_SIZE = 570

# Regex to check if a packet is a DBR
DBR_REGEX = /^DH/

# Wait for the DBR to come back from the camera.
def wait_for_dbr : Hash(Symbol, String)?
  LOG.info("Waiting for DBR")
  packet = db_sock.receive
  if packet
    if check_dbr(packet)
      info = parse_dbr(packet)
      LOG.info("DBR RECEIVED FROM #{info[:camera_ip]}, UID: #{info[:uid]}")
      return info
    else
      LOG.info("BAD/NON DBR RECEIVED! #{packet[0].bytes.map {|d| d.to_s(16).rjust(2, '0')}.join("\\x")}")
    end
  else
    LOG.info("NO DBR RECEIVED!")
  end
  return nil
end

# Check if the packet we recieved was a DBR
def check_dbr(packet) : Bool
  !! (packet[0] =~ DBR_REGEX)
end

# Parse the DBR information into a hash
def parse_dbr(packet) : Hash(Symbol, String)
  data = packet[0]
  connection = packet[1]

  result = {} of Symbol => String
  result[:camera_ip] = data[4..19].gsub("\\x00", "")
  result[:netmask] = data[20..35].gsub("\\x00", "")
  result[:gateway] = data[36..51].gsub("\\x00", "")
  result[:dns_server1] = data[52..67].gsub("\\x00", "")
  result[:dns_server2] = data[68..83].gsub("\\x00", "")
  result[:mac_address] = (data[84..88].bytes.map {|b| b.to_s(16).rjust(2, '0').upcase}).join
  result[:http_port] = ((data.bytes[91].to_i32 << 8) + data.bytes[90].to_i32).to_s
  result[:uid] = data[91..105]
  LOG.info("Parsed new target camera #{result}")
  result
end

```

BCP and BPR

We then want to send a BCP to broadcast now that the DPR has been resolved.

```
# Destination address for the f130 broadcast packet
BC SOCK_DST = Socket::IPAddress.new("255.255.255.255", 32108)
# F130 broadcast packet data
BCP = "\xf1\x30\x00\x00"
# Send the magic f130 broadcast packet
def send_bcp
  data_sock.send(BCP, BC SOCK_DST)
end
```

We now need to handle the handshake, which consists of broadcasting a BCP and waiting for a BPS, then replying back with a BPS, and then receiving a BPA.

```
# F130 broadcast packet reply header
BPR_HEADER = "\xf1\x42\x00\x14"

# Fixed BPR size
BPR_SIZE = 24

# Character sent for "SYN"
BP_SYN = 'A'

# Character sent for "ACK"
BP_ACK = 'B'

# Complete the handshake using BPS
def handle_bp_handshake
  LOG.info("Waiting for BPS")
  packet = @data_channel.receive #BPR size is always fixed
  LOG.info("Recieved a packet")
  if packet
    data = packet[0] # Contains the packet data
    camera_ip = packet[1] # Connection info to connect back into the camera
    LOG.info("Recieved a potential BPS from #{camera_ip}")
    # Check if out BPR is actually a BPR
    if data[1] == BP_SYN
      LOG.info("BPS Verified!")
    else
      LOG.info("BPS BAD! #{'\\x' + data.bytes.map {|d| d.to_s(16).rjust(2, '0')}.join('\\\\x')}")
      return
    end
    # Echo back packet data back
    LOG.info("Waiting for BPA")
    data_sock.send(data, camera_ip)
```

```
data_sock.send(data, camera_ip)
# Recv the BPR ACK packet
packet = @data_channel.receive
if packet
  LOG.info("Recieved potential BPA?")
  data = packet[0]
  camera_ip = packet[1]
  if data[1] == BP_ACK
    LOG.info("BPA Verified! Handshake successful!")
    # set the target camera to the current connection
    new_target camera_ip
  else
    LOG.info("BPA BAD! #{data.bytes.map {|d| d.to_s(16).rjust(2, '0')}.join("\\x")}")
  end
end
end
end
```

Main Phase

Now we need to handle the ping pong packets! This is super simple now that we have finished the hard part.

```

# Packet that must be sent between the camera and the client at least once every 11 packets
PING_PACKET = "\xf1\xe0\x00\x00"
# Packet that must be sent between the camera and the client at least once every 11 packets
PONG_PACKET = "\xf1\xe1\x00\x00"

def main_phase
  # Block here to receive data from the data fiber
  data = @data_channel.receive

  # Classify each packet and respond
  if data[0] == PING_PACKET
    send_pong
    LOG.info "Sent Pong"
  elsif data[0] == PONG_PACKET
    send_ping
    LOG.info "Sent Ping"
  # This is important! The data fiber will block, waiting for data to come through
  # So to exit the program, we just send the unblock data to the data socket to free it
  elsif data[0][0..3] == BPA_HEADER
    LOG.info "Receive extra BPA"
  elsif data[0] == UNBLOCK_FIBER_DATA
    LOG.info "RECEIVED UNBLOCK FIBER COMMAND!"
  else
    LOG.info "UNKNOWN PACKET RECEIVED from #{data[1]} : #{data[0].bytes.map {|d| d.to_s(16).rjust(2, '0')}.join("\x"
  end
end

def send_ping
  data_sock.send(PING_PACKET, target)
end

def send_pong
  data_sock.send(PONG_PACKET, target)
end

```

Testing

If we open up Wireshark and listen in to the connection, we should see that pings and pongs should be coming through, and there should be no errors. Also the LOG should show that there were no errors as well.

Here's the code I used to run it.

require "/.client"

client = Client.new

client.run

sleep 5

client.close

The screenshot shows a Wireshark packet capture window titled '*enx0050b6278a7f'. The packet list pane displays 49 packets. The first 47 packets are UDP packets from 192.168.1.5 to 255.255.255.255. The last two packets (48 and 49) are ICMP 'Destination unreachable' messages. The packet details pane shows the structure of the first packet: Ethernet II, Internet Protocol Version 4, User Datagram Protocol, and ASTERIX packet. The packet bytes pane shows the raw data of the first packet.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	gamingcomputer.local	255.255.255.255	ASTERIX	62	
2	0.002037821	vstarcam	gamingcomputer.local	ASTERIX	568	
3	0.002105783	vstarcam	255.255.255.255	ASTERIX	568	
4	0.002437599	gamingcomputer.local	255.255.255.255	UDP	62	11700 → 32100 Len=4
5	0.002924544	vstarcam	gamingcomputer.local	UDP	68	18951 → 11700 Len=24
6	0.003103895	gamingcomputer.local	vstarcam	UDP	68	11700 → 18951 Len=24
7	0.003767057	vstarcam	gamingcomputer.local	UDP	68	18951 → 11700 Len=24
8	0.022124252	vstarcam	gamingcomputer.local	UDP	68	18951 → 11700 Len=24
9	0.043111081	vstarcam	gamingcomputer.local	UDP	68	18951 → 11700 Len=24
10	0.061045805	vstarcam	gamingcomputer.local	UDP	68	18951 → 11700 Len=24
11	0.288684989	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
12	0.288656666	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
13	0.338319231	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
14	0.338495628	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
15	1.308969209	vstarcam	114.55.218.176	UDP	62	18950 → 32100 Len=4
16	1.301131530	vstarcam	47.91.90.222	UDP	62	18950 → 32100 Len=4
17	1.301132202	vstarcam	ec2-107-20-134-235...	UDP	62	18950 → 32100 Len=4
18	1.301274694	vstarcam	114.55.218.176	UDP	92	18950 → 32100 Len=48
19	1.301406583	vstarcam	47.91.90.222	UDP	92	18950 → 32100 Len=48
20	1.301534928	vstarcam	ec2-107-20-134-235...	UDP	92	18950 → 32100 Len=48
21	1.308982218	ec2-107-20-134-235...	vstarcam	UDP	64	32100 → 18950 Len=20
22	1.302257868	ec2-107-20-134-235...	vstarcam	UDP	62	32100 → 18950 Len=12
23	1.466114546	47.91.90.222	vstarcam	UDP	64	32100 → 18950 Len=20
24	1.468031948	47.91.90.222	vstarcam	UDP	62	32100 → 18950 Len=12
25	1.588351930	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
26	1.588494193	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
27	1.738873331	114.55.218.176	vstarcam	UDP	64	32100 → 18950 Len=20
28	1.738929077	114.55.218.176	vstarcam	UDP	62	32100 → 18950 Len=12
29	2.288771715	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
30	2.288916879	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
31	2.889411880	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
32	2.881115589	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
33	4.138883574	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
34	4.131055112	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
35	4.288914628	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
36	4.281807970	gamingcomputer.local	vstarcam	UDP	62	11700 → 18951 Len=4
37	4.608469990	vstarcam	114.55.218.176	UDP	92	18950 → 32100 Len=48
38	4.608953469	vstarcam	47.91.90.222	UDP	92	18950 → 32100 Len=48
39	4.608796416	vstarcam	ec2-107-20-134-235...	UDP	92	18950 → 32100 Len=48
40	4.602609890	ec2-107-20-134-235...	vstarcam	UDP	62	32100 → 18950 Len=12
41	4.705443178	47.91.90.222	vstarcam	UDP	62	32100 → 18950 Len=12
42	5.038275734	114.55.218.176	vstarcam	UDP	62	32100 → 18950 Len=12
43	5.064608022	gamingcomputer.local	vstarcam	ARP	62	Who has 192.168.11.140? Tell 192.168.11.5
44	5.068867879	vstarcam	gamingcomputer.local	ARP	62	192.168.11.140 is at 48:02:2a:0b:db:b4
45	5.301547383	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
46	5.31051113	gamingcomputer.local	vstarcam	ICMP	70	Destination unreachable (Port unreachable)
47	6.290483370	vstarcam	gamingcomputer.local	UDP	62	18951 → 11700 Len=4
48	6.302091241	gamingcomputer.local	vstarcam	ICMP	70	Destination unreachable (Port unreachable)
49	6.316050793	DO-WR	vstarcam	ARP	82	Who has 192.168.11.140? Tell 192.168.11.1

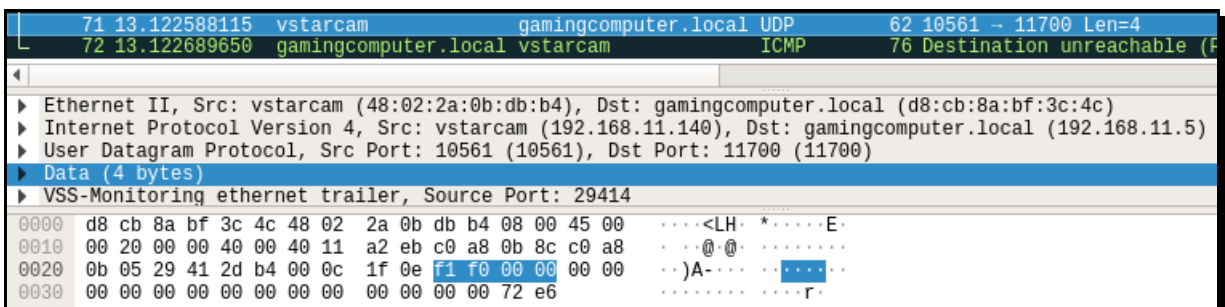
In the screenshot, we can see no errors or unexpected output. We get some destination unreachable errors at the end because that's when the server shutdown.

```

I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Opening ports
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Ports opened
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Sending DBP
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Sent DBP
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Changing to wait_for_dbr
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Waiting for DBR
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Parsed new target camera {:camera_ip => "192.168.11.140", :netmask =
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : DBR RECEIVED FROM 192.168.11.140, UID: VSTB668515UZCPK
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Changing to send_bcp
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Changing to handle_bp_handshake
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Waiting for BPS
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Recieved a packet
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Recieved a potential BPS from 192.168.11.140:10560
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : BPS Verified!
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Waiting for BPA
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Recieved potential BPA?
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : BPA Verified! Handshake successful!
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Changing to main_phase
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Receive extra BPA
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Receive extra BPA
I, [2019-03-05 06:47:36 -08:00 #12248] INFO -- : Receive extra BPA
I, [2019-03-05 06:47:37 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:37 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:38 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:39 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:40 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : Sent Pong
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : Closing client
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : Changing to closing
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : RECEIVED UNBLOCK FIBER COMMAND!
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : Changing to closed
I, [2019-03-05 06:47:41 -08:00 #12248] INFO -- : Closed client

```

From the LOG we can see everything is good!



Doing a little packet capture analysis, we can also see we have received a new packet, 0xf1f00000. This packet seems

to be sent after a certain number of pings and pongs were missed. We can assume this is some sort of disconnection packet, and we should reflect that in our code

```
# Packet sent when the camera has timed out from ping-pong
DISCONNECT_PACKET= "\xf1\xf0\x00\x00"

def send_disconnect
  data_sock.send(DISCONNECT_PACKET, target)
  LOG.info "Sent Disconnect"
end
```

After we send the disconnect packet, the camera will send one of it's own. To avoid a destination unreachable, we should sleep for 0.1 seconds just to let the packet be received by the data socket, even though we aren't going to do anything with the packet

TAGGED SKILLS:

 Crystal  Security Research  Reverse Engineering