NAIS Documentation

Release 0.1.1

Attilio Donà

Jan 11, 2018

Contents:

NAIS project originate from an attempt to build a tool for multiplexing the serial port between ascii characters (printf strings) and binary encoded structures for conveying complex requests and responses.

NAIS concepts are actually implemented in python language: pynais is an asyncio based package that implements a micro-router like service between:

- tcp connected apps
- websocket connected apps
- boards using a serial line (USB)

NAIS current version targets RIOT[1] based firmware running on a cc3200 launchpad[2] development kit.

NAIS is currently developed and tested on Linux.

[1] http://riot-os.org
[2] http://www.ti.com/tool/CC3200-LAUNCHXL

CHAPTER 1

Getting Started

1.1 Requirements

1.1.1 python 3.5+

NAIS make use of async/await language constructs, so python 3.5+ is needed.

Depending on python packages installed on your machine you may need to install the venv package if you plan to configure a virtual sandbox for python:

```
sudo apt-get install python3-venv
```

1.1.2 protobuf 3.3.0+

Install protobuf from source or pick up a pre-built binary distribution.

For example:

```
PROTO_VERSION=3.3.0

curl -OL https://github.com/google/protobuf/releases/download/v${PROTO_VERSION}/
↪protoc-${PROTO_VERSION}-linux-x86_64.zip

unzip -o protoc-${PROTO_VERSION}-linux-x86_64.zip -d ${HOME}/protoc3
rm protoc-${PROTO_VERSION}-linux-x86_64.zip
```

1.2 ser2net

ser2net is used for serial communication:

```
sudo apt-get install ser2net
sudo systemctl disable ser2net
```

ser2net service has to be disabled because start/stop of *ser2net* process is managed by NAIS junction app.

1.3 Installation

Get NAIS from github:

```
git clone https://github.com/attdona/NAIS
cd NAIS
. nais.env.sh   # do it if you want a dedicated virtualenv
pip install wheel
pip install .
```

If you feel like modifying NAIS package install it in development mode. Run last step as:

```
pip install -e .[dev]
```

CHAPTER 2

NAIS packet format

SYNC_START	TYPE	SLINE	DLINE	RSV	LEN	PAYLOAD	SYNC_END
1	1	1	1	1	M	N	1

The second row contains the length in bytes of the field. `M` and `N` are variable values related by:

`N` = value contained into the `M` bytes

2.1 Fields summary

Field	Values	Description
SYNC_START	Ox1E	Mark the start of a NAIS packet
TYPE		Unique id. Map to a protobuf message type
SLINE		Not used by clients. Used by NAIS junction routing functions
DLINE		Set by clients If the outgoing packet is a response of an ingoing packet . Used by NAIS junction for routing functions
RSV	Ox00	Reserved for future uses
LEN		PAYLOAD length. The MSB bit is a continuation bit if payload len exceeds 127 bytes
PAYLOAD		Encoded protobuf message
SYNC_END	Ox17	Mark the end of a NAIS packet

CHAPTER 3

Protobuf messages

In the following sections are described the built in protobuf messages supported by NAIS.

3.1 Command

```
message Command {
  required int32 id = 1;
  optional int32 seq = 2;
  repeated string svals = 3;
  repeated int32 ivals = 4;
}
```

A Command is a message that performs some action on the board.

`id` specifies the command.

There could be some predefined commands implemented with a default behaviour, for example the cc3200 board implements (item number is the command `id`):

1. REBOOT
2. TOGGLE_WIFI_MODE
3. FACTORY_RESET
4. OTA
5. GET_CONFIG

The `id` values in the set [1,5] are reserved (for cc3200 board) to these default commands: a custom command may use a `id` starting from 6.

`seq` is a sequence number garanteed to be unique for each command message not yet acknowledged (see below).

A command returns an acknowldgement message reporting the execution status.

```
message Ack {
  required int32 id = 1;
  optional int32 seq = 2;
  optional int32 status = 3;
}
```

The pair (`id`, `seq`) identifies a command instance and it is neeeded for matching an Ack message with the correspondig Command message.

`status` values are used defined but the suggested rule is to set `status` to 0 for a success command and all other values for error reporting or abnormal execution.

3.2 Event

An `Event` is a message originating by an endpoint, typically a board.

`id` identifies the event type, for example a *Fire Alarm* has id==10 and a *Movement Detection* event has id==12.

`svals`, `fvals` and `ivals` are optional fields used for conveying specific informations related to the event.

```
message Event {
    required int32  id = 1;
    repeated string svals = 2;
    repeated float  fvals = 3;
    repeated sint32 ivals = 4;
}
```

3.3 Profile

```
message Profile {
    required string uid = 1;
    required string pwd  = 2;
}
```

A `Profile` message embeds a user id and password.

3.4 Secret

```
message Secret {
    required string key = 1;
}
```

A `Secret` is a message that can be used to transfer a super secret value.

3.5 Config

```
message Config {
    required string network = 1;
    required string board   = 2;
```

```
    required string host = 3;
    optional int32  port = 4;
    optional int32  alive_period = 5;
    optional bool   secure = 6;
}
```

`Config` message is used for communicates a well-know identity and network configuration parameters to a board/endpoint.

The pair (`board`, `network`) is a unique identifier.

When the board/endpoint support mqtt and the mqtt client is enabled:

- `network/board` is the subscribe topic
- `hb/network/board` is the publish topic

`host` is the dns or ip address of a tcp server or a mqtt broker.

`port` is the mqtt broker/server remote tcp port.

`alive_period` is specific to mqtt protocol and it is the time interval between PINGREQ packets.

`secure` define the type of connection, plain or encrypted.

CHAPTER 4

wifi provisioning demo

After flashing the cc3200 launchpad a way of setting the wlan credentials have to be accomplished for connecting to a wifi access point.

The following python script prompt the user the wlan identifier and the password and setup the board.

```
"""Persist a WLAN profile to the board
"""
import logging
import asyncio
import sys
import click
import pynais as ns

logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr
)
LOG = logging.getLogger()


async def main(port):
    sock = await ns.connect('0.0.0.0', port)

    uid = ''
    while not uid:
        uid = input("enter username: ")

    pwd = ''
    while not pwd:
        pwd = input("enter password: ")

    add_profile = ns.msg.Profile(uid, pwd)

    # send the profile message to enable wifi and autoconnect the board to AP
```

```
    # send the config message to configure the server ip address
    await ns.proto_send(sock, add_profile, wait_ack=True)

    sock.close()


@click.command()
@click.option('--port', default=3002, help='line port')
def trampoline(port):
    """Starter
    """
    loop = asyncio.get_event_loop()

    loop.run_until_complete(main(port))
    loop.close()


#pylint: disable=E1120
if __name__ == "__main__":
    trampoline()
```

The script connect to the NAIS junction using tcp port 3002. It is the work of junction to deliver the message using the serial port to the board.

Just for give a little taste of a NAIS junction use case, you could test the frontend provisioning script attaching a virtual board at the junction in case the hardware is not ready for integration.

CHAPTER 5

websocket demo

This is a simple static `index.html` page that make use of a websocket channel to send json commands to a USB connected cc3200 launchpad board using NAIS junction.

This super simple example switches the red and yellow leds of a cc3200 launchpad

```
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- The above 3 meta tags *must* come first in the head; any other head content␣
↪must come *after* these tags -->
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="../../bootstrap/favicon.ico">

    <title>Led</title>

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
↪alpha.6/css/bootstrap.min.css" integrity="sha384-rwoIResjU2yc3z8GV/
↪NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
    <script src="https://code.jquery.com/jquery-3.2.1.min.js" crossorigin="anonymous">
↪</script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js
↪" integrity="sha384-DztdAPBWPRXSA/3eYEEUWrWCy7G5KFbe8fFjk5JAIxUYHKkDx6Qin1DkWx51bBrb
↪" crossorigin="anonymous"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/js/bootstrap.
↪min.js" integrity="sha384-
↪vBWWzlZJ8ea9aCX4pEW3rVHjgjt7zpkNpZk+02D9phzyeVkE+jo0ieGizqPLForn" crossorigin=
↪"anonymous"></script>

    <!-- Custom styles for this template -->
    <link href="cstyle.css" rel="stylesheet">
```

```
</head>

<body>
  <script>

    var ws;

    function connect() {
      ws = new WebSocket("ws://127.0.0.1:3000/");

      ws.onerror = function(event) {
        $('#ws_console').show()
        $('#red_btn').text('Red ?')
        $('#yellow_btn').text('Yellow ?')
      }

      ws.onclose = function(event) {
        console.log("conn closed, retrying in 2 seconds ...")
        setTimeout(function() {
          console.log("!! connection in progress ..." + ws)
          connect()
        }, 2000)
      }

      ws.onopen = function(event) {
        $('#ws_console').hide()

        // get led states
        ws.send(JSON.stringify({'leds': 'get'}))
      }

      ws.onmessage = function (event) {
        //$('#response').show()
        console.log("recv:" + event.data)

        led = JSON.parse(event.data)
        switch (led.red) {
          case 'on':
            $('#red_btn').text('Red OFF')
            break
          case 'off':
            $('#red_btn').text('Red ON')
            break
        }
        switch (led.yellow) {
          case 'on':
            $('#yellow_btn').text('Yellow OFF')
            break
          case 'off':
            $('#yellow_btn').text('Yellow ON')
            break
        }


        var message = document.getElementById('response');
        content = document.createTextNode(event.data);
        message.innerHTML = event.data
        setTimeout(function() {
```

```
            $('#response').fadeOut(1000)
          }, 2000)
        };


      }

      connect()

      messages = document.createElement('ul');
      messages.className='list-group'

      $( document ).ready(function() {
        console.log( "ready!" );
        $('#ws_console').hide()
        $('#response').hide()

        $("#red_btn").click(function(event){

                  cmd = $('#red_btn').text()
                  switch (cmd) {
                    case 'Red ON':
                      ws.send(JSON.stringify({'leds':'set', 'red': 'on'}))
                      break
                    case 'Red OFF':
                      ws.send(JSON.stringify({'leds':'set', 'red': 'off'}))
                      break
                  }
                  event.preventDefault();
                });

        $("#yellow_btn").click(function(event){

                  cmd = $('#yellow_btn').text()
                  switch (cmd) {
                    case 'Yellow ON':
                      ws.send(JSON.stringify({'leds':'set', 'yellow': 'on'}))
                      break
                    case 'Yellow OFF':
                      ws.send(JSON.stringify({'leds':'set', 'yellow': 'off'}))
                      break
                  }
                  event.preventDefault();
                });



      });

      document.body.appendChild(messages);
    </script>


    <div class="container">

    <div class="card" style="width: 20rem;">
      <img class="card-img-top center"  src="../../../docs/images/nais-logo.png" alt=
↪"Card image cap">
```

```
      <div class="card-block">
          <h4 class="card-title">Leds reloaded</h4>
        <p class="card-text">simple UI for driving the leds on the
           <a href="http://www.ti.com/product/CC3200"> cc3200 launchpad</a>
            with JSON encoded packets using websocket protocol
          </p>

      <div class="btn-toolbar" role="toolbar">
        <button id="yellow_btn" class="btn btn-warning m-2" type="submit">Yellow ?</
↪button>
        <button id="red_btn" class="btn btn-danger m-2" type="submit">Red ?</button>
      </div>


      </div>
    </div>


    </div> <!-- /container -->

    <div class="container">
      <div id="response" class="alert alert-success" role="alert"></div>
    </div>
    <div class="container" style="margin-top:20px">
      <div id="ws_console" class="hidden alert alert-danger" role="alert">
        <strong>Whoops!</strong> connection to nais router failed, retrying ...
      </div>
    </div>


    <!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
    <!-- <script src="../../bootstrap/assets/js/ie10-viewport-bug-workaround.js"></
↪script> -->
  </body>
  </html>
```

CHAPTER 6

A simple example . . .

A board connected to a USB port and a websocket based UI running in a web browser are linked together:

```
import pynais as ns

board = ns.SerialLine(device='/dev/ttyUSB0')
web_page = ns.WSLine(port=3000)

ns.junction.route(src=web_page, dst=board
                  src_to_dst = to_protobuf, dst_to_src = to_json)

ns.junction.run()
```

`junction.route` declares the bidirectional link between the board and the web page.

The argument `src_to_dest` names a callback function that implements the custom transformation of messages originating from the `web_page` (the source) and the `board` (the destination):

```
def to_protobuf(in_packet):
    """ Transform a json packet to a protobuf equivalent

        Args:
            in_packet (str): json string
        Returns:
            a protobuf object or None if unable to transform
            the input packet.
            If return None the packet is silently discarded


    """
    LOG.debug("to_protobuf - input: %s, msg: %s",
            type(in_packet), in_packet)

    obj = json.loads(in_packet.decode())

    packet = None
```

```
    # check if json packet is a led command
    if 'leds' in obj:
        if obj['leds'] == 'get':
            packet = ns.marshall(cmd.Leds())
        elif obj['leds'] == 'set':
            print()
            packet = ns.marshall(
                cmd.Leds(obj['red'] if 'red' in obj else None,
                         obj['green'] if 'green' in obj else None,
                         obj['yellow'] if 'yellow' in obj else None))
    else:
        LOG.info("to_protobuf - unable to convert message %s",
                 in_packet)
    return packet
```

The argument `dst_to_src` transforms the messages going from the `board` (the destination) to the `wep_page` (the source):

```
def to_json(in_packet):
    """Convert a protobuf into a json message
    """
    LOG.debug("to_json - input: %s, msg: %s", type(in_packet),
              in_packet)

    # from protobuf to json is just a matter of unmarshalling
    if ns.is_protobuf(in_packet):
        obj = ns.unmarshall(in_packet)
        if ns.is_ack(obj, command_type=cmd.Leds):
            mask = obj.sts
            obj = cmd.Leds()
            obj.set_status(mask)
        return obj
    else:
        LOG.debug("to_json - |%r| > /dev/null", in_packet)
        # do not send the message
        return None
```

These snippets ends the implementation of the micro-router (see below for the complete source file).

Now a javascript web component may open a websocket client (port 3000) and send the json formatted message:

```
{'leds':'set', 'red': 'on'}
```

The NAIS junction engine receive the json string, transform to a protobuf encoded payload and send through the serial port to the cc3200 board.

The red led switches on and a protobuf encoded acknowledge message is send over the UART port.

The protobuf payload is transformed to a custom json structure and the message is finally sent to the web component that get the string:

```
{"yellow": "on", "green": "off", "red": "on"}
```

CHAPTER 7

NAIS and Protocol Buffers

cc3200 firmware and NAIS junction supports the Google Protocol Buffers[3] message encoding.

For example the Ack protobuf specification is:

```
message Ack {
    required int32 id = 1;
    optional int32 seq = 2;
    optional int32 status = 3;
}
```

`id` field value is the request message id.

The bits of the `status` field reports the board leds status, 3 bits for red, green and yellow led.

[3] https://developers.google.com/protocol-buffers

CHAPTER 8

Simple junction

The complete source for the simple junction:

```
bash> python simple_junction.py
```

simple_junction.py:

```
import logging
import sys
import json
import click
import pynais as ns
import commands as cmd

logging.basicConfig(
    level=logging.DEBUG,
    format='%(levelname)s:%(name)s:%(lineno)s: %(message)s',
    stream=sys.stderr
)
LOG = logging.getLogger()


def to_protobuf(in_packet):
    """ Transform a json packet to a protobuf equivalent

        Args:
            in_packet (str): json string
        Returns:
            a protobuf object or None if unable to transform
            the input packet.
            If return None the packet is silently discarded


    """
    LOG.debug("to_protobuf - input: %s, msg: %s",
              type(in_packet), in_packet)
```

```
    obj = json.loads(in_packet.decode())

    packet = None

    # check if json packet is a led command
    if 'leds' in obj:
        if obj['leds'] == 'get':
            packet = ns.marshall(cmd.Leds())
        elif obj['leds'] == 'set':
            print()
            packet = ns.marshall(
                cmd.Leds(obj['red'] if 'red' in obj else None,
                         obj['green'] if 'green' in obj else None,
                         obj['yellow'] if 'yellow' in obj else None))
    else:
        LOG.info("to_protobuf - unable to convert message %s",
                 in_packet)
    return packet


def to_json(in_packet):
    """Convert a protobuf into a json message
    """
    LOG.debug("to_json - input: %s, msg: %s", type(in_packet),
              in_packet)

    # from protobuf to json is just a matter of unmarshalling
    if ns.is_protobuf(in_packet):
        obj = ns.unmarshall(in_packet)
        if ns.is_ack(obj, command_type=cmd.Leds):
            mask = obj.sts
            obj = cmd.Leds()
            obj.set_status(mask)
        return obj
    else:
        LOG.debug("to_json - |%r| > /dev/null", in_packet)
        # do not send the message
        return None


@click.command()
@click.option('--serial/--no-serial', default=True,
              help='serial simulator')
def main(serial):

    web_page = ns.WSLine(port=3000)
    if (serial):
        # the real board
        board = ns.SerialLine()
    else:
        # a software simulator
        board = ns.TcpLine(port=2001)

    ns.junction.route(src=web_page, dst=board,
                      src_to_dst=to_protobuf, dst_to_src=to_json)

    ns.junction.run()
```

```
if __name__ == "__main__":
    main()
```

CHAPTER 9

References