NoPASARAN Documentation

This document contains two parts:

-UPDATED PREVIOUS PARTS: This update includes additional explanatory notes to sections that were previously published in the documentation. The main purpose is to enhance clarity and ease of understanding for all users.

-NEW PARTS: New sections have been introduced that were not covered in the initial documentation. The newly added segments aim to provide users with a more comprehensive understanding of the full range of functionalities and capabilities that NoPASARAN offers.

Feedback:

Specify the syntax when it comes to parentheses in the documentation. Some primitives need more than one parenthesis. For example: when the primitive needs one parenthesis such as: wait_packet_signal (example1 example2) and when there is an output and it needs two such as set (example) (example). Adding this detail is going to make debugging much easier as it is easy to make these mistakes.

Specify the variable format in the file to the primitives that need a variable such as get_from_file and set_DNS_resource_value. A small example of what kind of variable should be included in the file and how will the format be. There is a dedicated section in the tutorial explaining the variables, but adding this will make the user experience much easier.

Note: I addressed those feedback points in the new parts of the documentation.

--

UPDATED PREVIOUS PARTS

In the Usage section:

Create the scenario.json file and/or other necessary and add them to the machine terminal

Make sure that you are running the scenario in the dev branch:

- To check if the scenario is in which branch

```
git status
```

- To move the scenario to the dev branch

```
git checkout dev
```

In the TLS Mutual Authenticated End-to-End Tunnel: Root CA Certificate Generation and Certificate Signing:

Step 1 Generate the Root CA Certificate¶

1. Open a command prompt or terminal window.
2. Generate a private key for the root CA certificate:
3.
```
openssl genrsa -out root_ca.key 4096
```

   This command generates a 4096-bit RSA private key (root_ca.key).

4. Generate the self-signed root CA certificate:
5.
```
openssl req -new -x509 -sha256 -key root_ca.key -out root_ca.crt -days 3650
```

   This command creates a self-signed certificate (root_ca.crt) using the private key generated in the previous step.

   Provide the required information when prompted. The information typically includes the Common Name (CN), organization details, and other details related to your root CA.

   The root CA certificate (root_ca.crt) is now created and ready to be used to sign certificates for TLS endpoints.

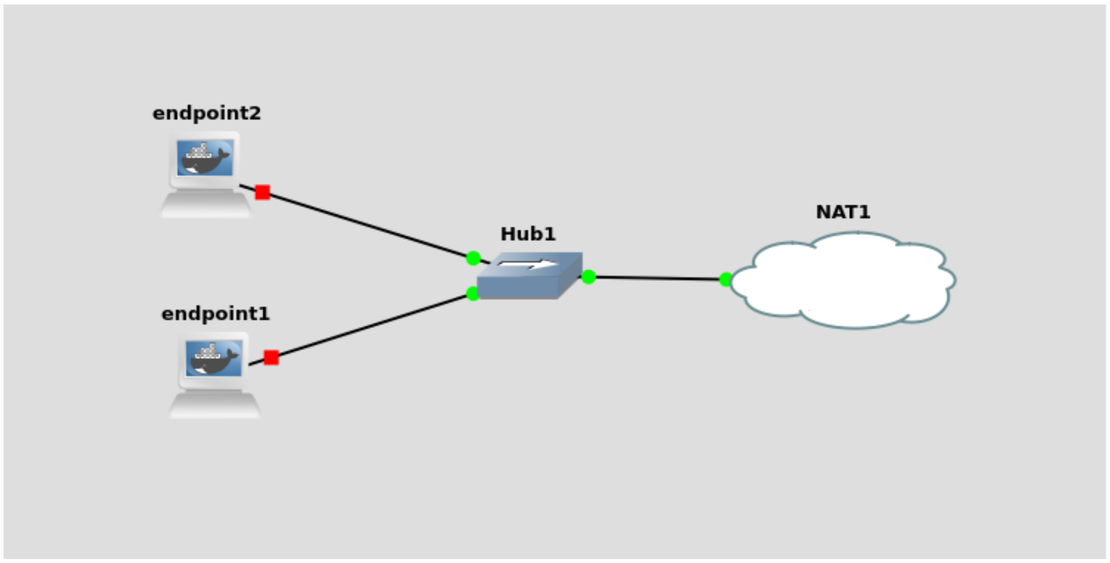
Note

Make sure you save the generated root_ca.crt file to each endpoint.

Step 2 Generate TLS Endpoint Certificates using a CSR¶

To generate certificates for the TLS endpoints signed by the root CA certificate, follow these steps for each endpoint form one of the endpoint terminals.

For instance, using endpoint1, repeat steps 1 to 4 to generate certificates for endpoint1 and endpoint2:



1. Generate a private key for the TLS endpoint:

```
2. openssl genrsa -out endpoint1.key 2048
```

This command generates a 2048-bit RSA private key (endpoint1.key).

3. Create a Certificate Signing Request (CSR) file using the private key:

```
4. openssl req -new -sha256 -key endpoint1.key -out endpoint1.csr
```

This command creates a CSR file (endpoint1.csr) using the private key generated in the previous step.

Enter the required information when prompted, including the Common Name (CN) and other details related to the TLS endpoint.

Note

It is necessary to enter the Common Name (CN) accurately when generating the Certificate Signing Request (CSR) for each endpoint. Incorrect CN entry will lead to certificate validation issues and rejection by the other endpoint.

5. Submit the CSR (endpoint1.csr) to the root CA for signing:

```
6. openssl x509 -req -sha256 -in endpoint1.csr -CA root_ca.crt -CAkey root_ca.key -CAcreateserial -out endpoint1.crt -
   days 365
```

This command signs the CSR using the root CA certificate (root_ca.crt) and private key (root_ca.key) and generates the signed certificate (endpoint1.crt) valid for 365 days.

7. Concatenate the private key and the certificate into one file:

```
8. cat endpoint1.key endpoint1.crt > endpoint1.pem
```

This command combines the private key and the certificate into a single file (endpoint1.pem). This file will be used by the program that requires both in a single file.

Repeat steps 1-4 for each TLS endpoint, generating a unique private key, CSR, certificate, and combined PEM file for each.

The root CA has now signed the CSRs and provided you with the signed certificates (endpoint1.crt, endpoint2.crt, etc.) for each TLS endpoint.

Note

Make sure you save endpoint1.pem and endpoint2.pem to each endpoint.

Step3 Configuration for Each Endpoint

With the root CA certificate and the endpoint's private certificate generated, you can configure each endpoint using a JSON configuration file, as shown below:

```
{
   "ROOT_CERTIFICATE": "root_ca.crt",
   "PRIVATE_CERTIFICATE": "endpoint1.pem",
   "DESTINATION_IP": "192.168.122.247",
   "SERVER_PORT": "443"
}
```

Replace "endpoint1.pem" with the respective filename for each endpoint, and adjust "destination_ip" and "server_port" as necessary for your network configuration.

Note

Make sure you create endpoint configurations on both endpoints. Note that the destination endpoint doesn't need a "DESTINATION_IP".

NEW PARTS

Introduction to State Machines

In nopasaran, JSON files represent Finite State Machines Files, which can be created using a software like stately.ai or on (http://127.0.0.1:4200/#). State Machines represent scenarios in nopasaran, they consist of states and events that create actions which the machine takes to performs its set scenario. Scenarios can be broken down into events and states, each state or event consists of primitives that can found in API Reference. Each state has to be followed by an event, vice versa, until the scenario ends with the final state.

Variables in State Machines

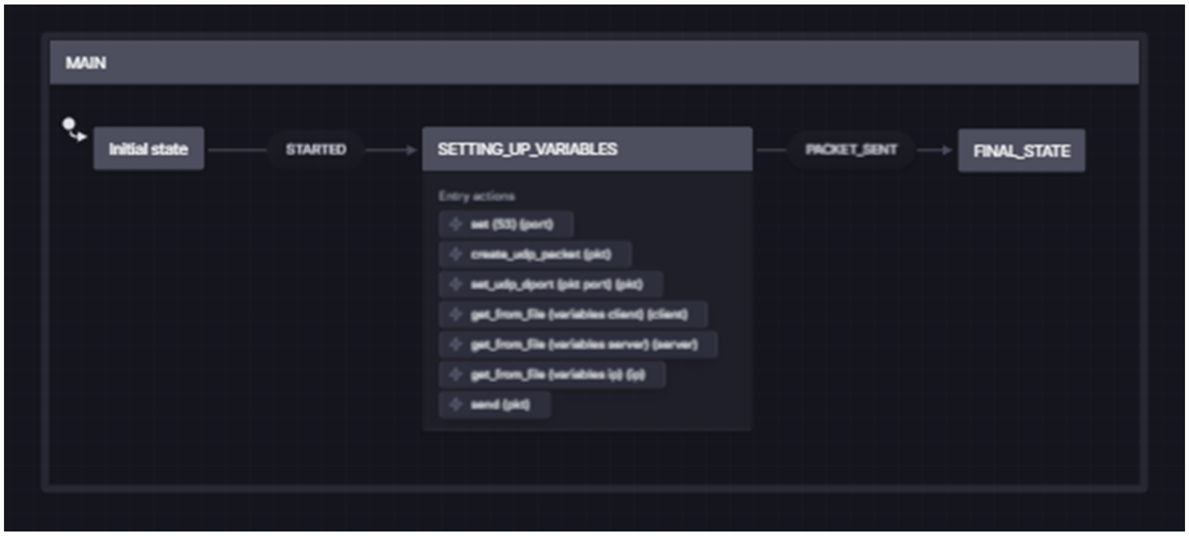
Variables play a crucial rule when designing state machine diagrams that read variable from a JSON file. Variable are especially used with the get_from_file () () primitive. More details can be found in the API Reference section. Below is an example of a variables,json file formatting:

```
{
    "role": "192.168.122.10",
    "client": "192.168.122.10",
    "server": "192.168.122.11"
}
```

State Machine Example

Below is a simple example of a State Machine Diagram, the client sends a UDP packet to a server. In this example, the scenario begins with “Initial state” before moving to event “STARTED”. Afterwards, the state goes to “SETTING_UP_VARIABLES”, in this state of the scenario, primitives were used to create a UDP packet, and set its destination port to 53. Then, the get_from_file () () fetch variables from a JSON file before finally sending the packet.

Note that some primitives, such as the send () primitive, has to be followed by a specific event accordingly, PACKET_SENT in this case, more details can be found in the API Reference section.


MAIN
Initial state
STARTED
SETTING_UP_VARIABLES
Entry actions
set (53) (port)
create_udp_packet (pkt)
set_udp_dport (pkt port) (pkt)
get_from_file (variables client) (client)
get_from_file (variables server) (server)
get_from_file (variables ip) (ip)
send (pkt)
PACKET_SENT
FINAL_STATE

Note

Depending on the primitive that was used, and whether or not it takes an input, produce an output or both, the number of parentheses should be used accordingly.

Primitives has to be chosen carefully following a logical order of states and events. A general understanding of computer networks can help with designing seamless state machine diagrams.

Please note that this is a basic representation of a state machine. In real-world applications, more states, events, and actions would likely be defined to accurately model complex behaviors.

Running Scenarios

The JSON file below represents the previous example, the JSON file can be extracted from the State Machine Diagram.

```
{
 "id": "MAIN",
 "initial": "Initial state",
 "states": {
  "Initial state": {
   "on": {
    "STARTED": {
     "target": "SETTING_UP_VARIABLES"
    }
   }
  },
  "SETTING_UP_VARIABLES": {
   "entry": [
    {
     "type": "set (53) (port)"
    },
    {
     "type": "create_udp_packet (pkt)"
    },
    {
     "type": "set_udp_dport (pkt port) (pkt)"
    },
    {
     "type": "get_from_file (variables client) (client)"
    },
    {
     "type": "get_from_file (variables server) (server)"
    },
    {
     "type": "get_from_file (variables ip) (ip)"
    },
    {
     "type": "send (pkt)"
    }
   ],
   "on": {
    "PACKET_SENT": {
     "target": "FINAL_STATE"
    }
   }
  },
  "FINAL_STATE": {}
 }
}
```

To run a scenario, create a scenario file in the client endpoint (in this case endpoint 1), and run it using the following command.

```
python main.py WORKER --scenario=<path-to-json-scenario-file>
```

Replace <path-to-json-scenario-file> with the path to your actual JSON scenario file.

If run correctly, the client will send a packet, which can be seen in a Wireshark scanning. A general understanding of Wireshark can help to analyze the state machine behavior and visualize packet traffic.

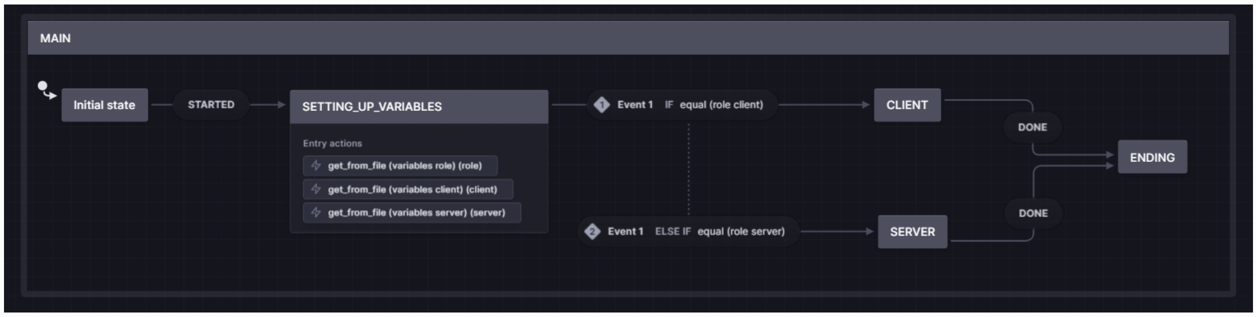
To make sure that the scenario reached its final state correctly, run the scenario using the following command:

```
python main.py WORKER --scenario=<path-to-json-scenario-file> -ll info -v
```

Using Guards in State Machines

Guards are conditional checks that determine which transition should be taken in a state based on specified event conditions. Guards can be very useful, especially when creating modular scenarios for both a client and server.

In this state machine example, a scenario starts with “Initial state” before moving to the “STARTED” event. It then goes to the “setting up variables” state, where it reads variables from a Json file, “variables.json” in this example. This scenario can be run from both the client and the server, each of these machines have their own “role” variable value. Hence, when run form the client, the guard condition will go to the CLIENT state, vice versa.


MAIN
Initial state
STARTED
SETTING_UP_VARIABLES
Entry actions
get_from_file (variables role) (role)
get_from_file (variables client) (client)
get_from_file (variables server) (server)
1 Event 1 IF equal (role client)
CLIENT
DONE
ENDING
DONE
2 Event 1 ELSE IF equal (role server)
SERVER

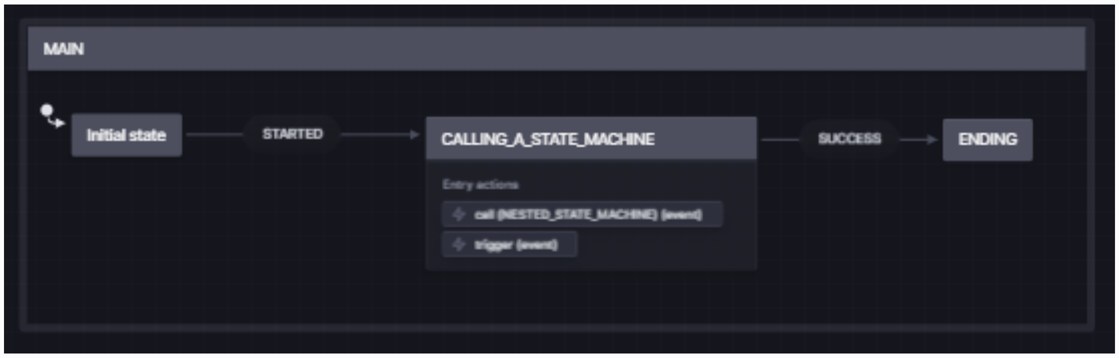
Note

When creating state machine diagrams, it’s important to keep them modular, in which the same scenario file can be used for both the client and the sever.

Using Nested State Machines

State machine are usually complex, having more states, events, and actions that result in lengthy scenarios. This can cause state machine diagrams to be unreadable or difficult to follow. Using nested state machines helps to create a much clearer, and organized scenario.

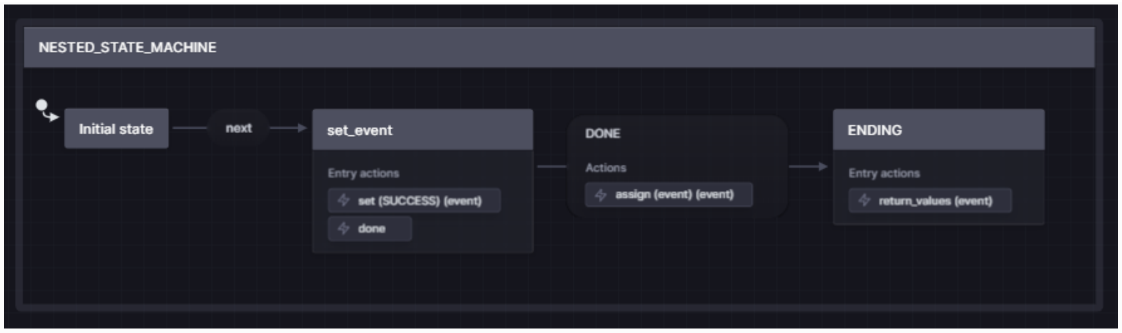
Below is an example of a nested state machine diagram:



The scenario starts with the “Initial state” state before moving to the event “STARTED”, it then goes to “CALLING_A_STATE_MACHINE” state, inside this state, the primitive call (NESTED_STATE_MACHINE) (event) calls another state machine file, which will return one output, event.

Below is the state machine diagram for NESTED_STATE_MACHINE, which was called previously:

The scenario starts with “Initial state” before going to “next”. Then in “set_event”, the primitive set (success) (event) sets the variables event to SUCCESS, after that the scenario goes to the “DONE” event. Note that assign () () is used to transition a variable from one state to another. When the scenario reaches the “ENDING” state, it returns the variable event to the main scenario.json file.



Back in the main scenario, the primitive trigger (event) is trigger, which will cause the event of the same name "SUCCESS" to be triggered. Finally, the scenario ends at "ENDING"

Note

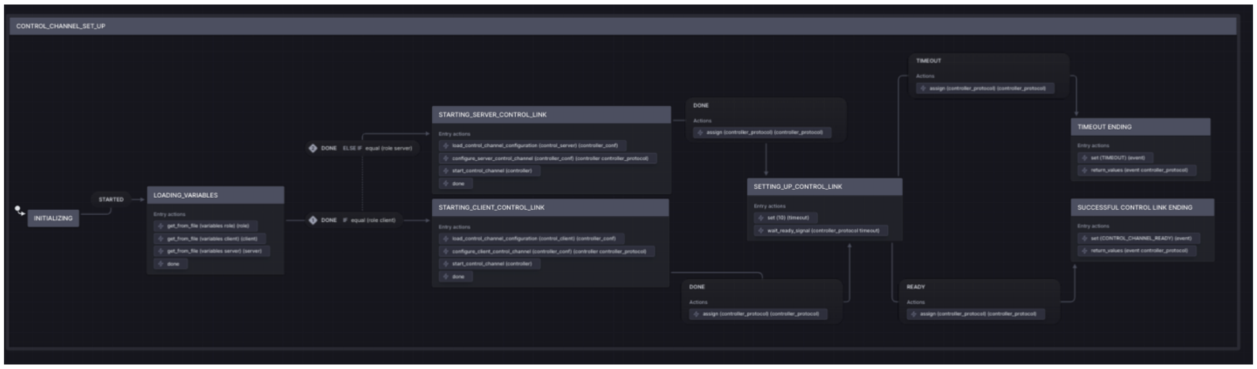
It's important to carefully use the primitives return values (), assign () (), and trigger () when designing nested state machine diagrams. Further details on primitives can be found in API References

---

Control Channels in Finite State Machine:

Setting up an end-to-end control channel and using it to sync messages is one of the key features of nopasaran. To establish an end-to-end connection through nopasaran, it is important to ensure that both endpoints have signed root_ca.crt, a signed TLS certificate, and a configurated json file. An explanation for these requirements can be found in this documentation.

Below is a state machine diagram that represents a scenario for establishing an end-to-end connection:


CONTROL_CHANNEL_SET_UP
INITIALIZING
STARTED
LOADING_VARIABLES
Entry actions
get_from_file (variables role) (role)
get_from_file (variables client) (client)
get_from_file (variables server) (server)
done
DONE ELSE IF equal (role server)
DONE IF equal (role client)
STARTING_SERVER_CONTROL_LINK
Entry actions
load_control_channel_configuration (control_server) (controller_conf)
configure_server_control_channel (controller_conf) (controller controller_protocol)
start_control_channel (controller)
done
STARTING_CLIENT_CONTROL_LINK
Entry actions
load_control_channel_configuration (control_client) (controller_conf)
configure_client_control_channel (controller_conf) (controller controller_protocol)
start_control_channel (controller)
done
DONE
Actions
assign (controller_protocol) (controller_protocol)
DONE
Actions
assign (controller_protocol) (controller_protocol)
SETTING_UP_CONTROL_LINK
Entry actions
set (10) (timeout)
wait_ready_signal (controller_protocol timeout)
TIMEOUT
Actions
assign (controller_protocol) (controller_protocol)
READY
Actions
assign (controller_protocol) (controller_protocol)
TIMEOUT ENDING
Entry actions
set (TIMEOUT) (event)
return_values (event controller_protocol)
SUCCESSFUL CONTROL LINK ENDING
Entry actions
set (CONTROL_CHANNEL_READY) (event)
return_values (event controller_protocol)

The scenario starts with the ”INITIALIZING” state before moving to the “STARTED” event and then to “LOADING_VARIABLE” to load the variables role, client, and server, from a file json file. It’s important to note that this scenario is run concurrently from the client and the server endpoints. Each of these and points has their own role variable value, which allows the guard condition to lead the endpoints to their scenario paths accordingly.

- **- - - -**

Sending and Listening to Packets in Finite State Machine:

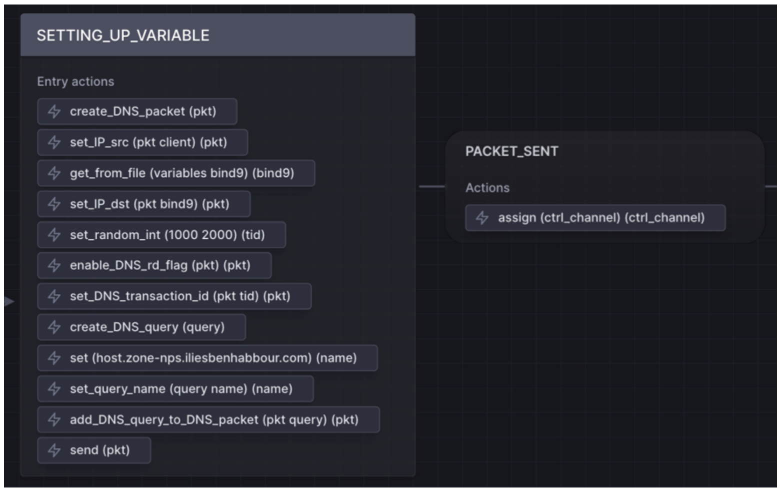
Sending and listening to packets are perhaps the most important features to use nopasaran. Different kinds of packets can be sent using the predefined primitives in nopasaran, including UDP, TCP, and DNS packets.

To illustrate the process of sending a packet from a client to a server using nopasaran, we'll walk through a step-by-step explanation, involving variables, nested state machines, and predefined primitives.

Sending a Packet

To send a packet using nopasaran, the first step is to identify the type of packet needed. In this example, we'll focus on sending a DNS packet from a client. The state machine diagram outlines the scenario, and nopasaran's predefined DNS primitives will be used in this task. Further explanation of these primitives can be found in the API Reference section.

Below is a part of a scenario that sets up the DNS packet to send it to a server:



SETTING_UP_VARIABLE

Entry actions

- create_DNS_packet (pkt)
- set_IP_src (pkt client) (pkt)
- get_from_file (variables bind9) (bind9)
- set_IP_dst (pkt bind9) (pkt)
- set_random_int (1000 2000) (tid)
- enable_DNS_rd_flag (pkt) (pkt)
- set_DNS_transaction_id (pkt tid) (pkt)
- create_DNS_query (query)
- set (host.zone-nps.iliesbenhabbour.com) (name)
- set_query_name (query name) (name)
- add_DNS_query_to_DNS_packet (pkt query) (pkt)
- send (pkt)

PACKET_SENT

Actions

- assign (ctrl_channel) (ctrl_channel)

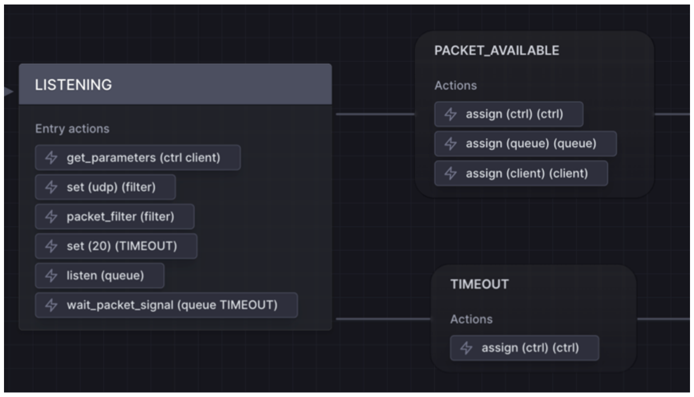
In the "SETTING_UP_VARIABLE" each of the primitives in the context of this scenario are explained below:

- create_DNS_packet (): Creates a DNS packet, which can be further specified with further primitives.
- set_IP_src () (): Sets the client's IP as the source.
- set_IP_dst () (): Sets the resolver's IP as the destination.
- set_random_int () (): Creates a variable and sets a random integer in a specified range.
- enable_DNS_rd_flag (): Enables the DNS recursion desired flag.
- set_DNS_transaction_id () (): Sets the previously created random integer as a transaction id for the packet.
- set () (): Sets a domain name to the variable query.
- set_query_name () (): Adds the domain name to the previously created DNS query.
- Add_DNS_query_to_DNS_packet: Adds the previously created DNS query to the DNS packet.
- send (): Finally sends the packet, and triggers the event PACKET_SENT

Receiving a Packet

From the server point of view, it will listen to the packets sent by the client and stores them in a queue, using special primitives in the state machine diagram. When the server is in the listening state, it awaits incoming packets from the client.

Below is a part of the scenario that explains the process of listening to client packets:



In the “LISTENING” state each of the primitives in the context of this scenario are explained below:

- get_parameters (): Gets parameters specified in a call () primitive, since this is part of a nested state machine diagram.

- set () (): Sets the variable filter to udp, since it’s a DNS packet.

- packet_filter (): Set the packet sniffer to sniff udp packets only.

- set () (): Sets a value for the variable timeout.

- listen (): Initiates packet sniffer.

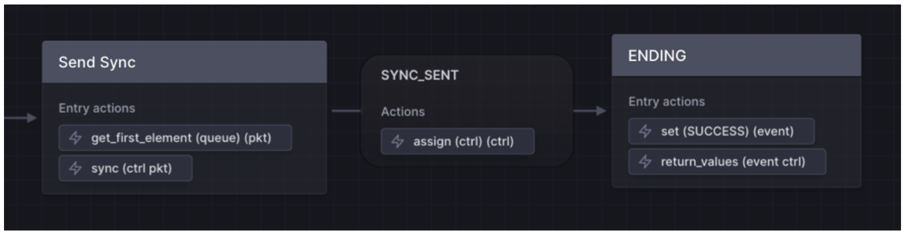
- wait_packet_signal (): The server waits for the packet signal sent by the other client. This primitive branches out into two possible events: either PACKET_AVAILABLE, indicating that the packet was received, or TIMEOUT, signifying that no signal was received within the specified time.

Syncing a Message Through the Control Channel

After securing an end-to-end connection between the endpoints, messages can be directly synced between them without any interruptions. This is especially useful for detecting abnormal behaviors in the network. When syncing messages through the control channel, one of the endpoints takes on the role of syncing the message, while the other endpoint waits for sync signals. Each of these points of view is explained in this section.

Sending a Sync

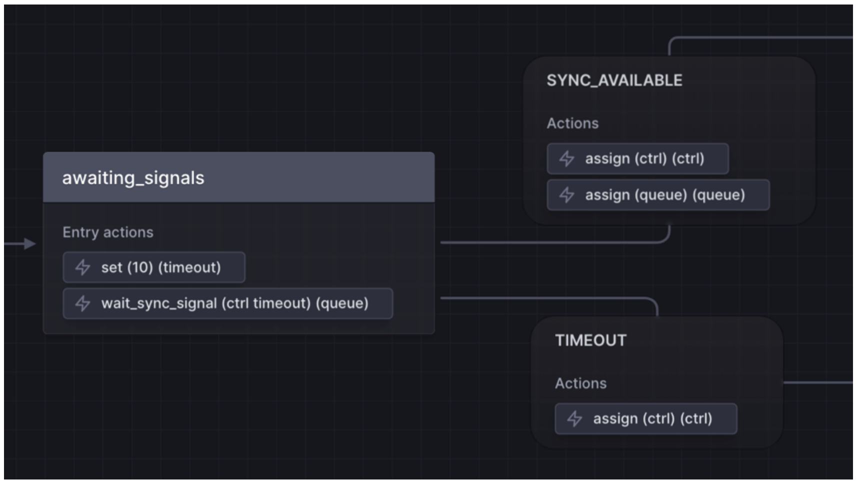
Below is a part of a scenario that syncs a message through the control channel to the other endpoint:



In the "Send Sync" state each of the primitives in the context of this scenario are explained below:

- get_first_element () (): Takes a queue as input, and gets the first element, storing it in an output variable.
- sync (): Takes an input and syncs it to the other end through the control channel, triggering the event SYNC_SENT.

Below is a part of a scenario that wait for a syncs message signal through the control channel from the other endpoint:



In the "awaiting_signals" state each of the primitives in the context of this scenario are explained below:

-set () (): Sets the integer value 10 and stores it in the variable, timeout.

-wait_sync_signal () (): The endpoint waits for the sync signal sent by the other endpoint. This primitive branches out into two possible events: either SYNC_AVAILABLE, indicating that the signal was received, or TIMEOUT, signifying that no signal was received within the specified time.