**Status:** Active

**Updated by:**

# RFC – The Shady Net Protocol (SNP)

**Abstract**

This document discusses the Shady Net Protocol (SNP), which is an anonymous request routing protocol.

The primary function of an SNP speaking system is to make http requests received from clients to web servers for them.

**Table of Contents**

- Introduction
- Conventions Used in This Document
- Commands and Responses
    - Commands
    - Responses
- The SNP Protocol Header
- Body Format
- Error Handling
- Request Queueing
- Security
- Example

## 1. Introduction

The protocol is a standardized format for sending requests to a proxy server and receiving the desired response(s). The protocol will allow a client to specify a HTTP request that they want to be made by the proxy server, an authorization token that **MAY** be used to identify them when making requests. The server **MAY** limit the number of requests that a client can make if an authorization token is not provided.

## 2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

## 4. Commands and Responses

**SEND COMMAND:**

**Purpose**:

The send command is the command that the protocol is centered around, therefore every implementation **MUST** implement this feature. This is the command that the client will send to the server that will tell the server what request they want it to make for them.

**Structure**:

This command **MUST** have a property named ‘id’ that has a value of type string. This value **SHOULD** be a randomly generated string that **MUST** be unique to the request. This means that each time you make send a command (make a request to the proxy server) you will need to generate a new random string for that request and assign it to the ‘id’ property of this command. This **COULD** be how the client makes sure that the response that it receives is for the request that it made.

The command **MUST** have a property / key named ‘type’ and this **MUST** be equal to ‘SEND’ as this is what the server will read to know what type of command the client is making and what to do with the body. This is a **REQUIRED** property as is required by every command.

This command **MUST** have a property / key named ‘body’ this property will have the value of a JSON object / map that will contain the properties that will be used to make the request. This is a **REQUIRED** property as is required by every command.

This command **MUST** define a property / key named ‘timeout’ and this **MUST** have a value of type integer. This is a **REQUIRED** property as is required by every SEND command.  This integer represents the number of milliseconds that the server will wait to get a response after making the desired HTTP request.

The body property of this command **MUST** contain a property / key named ‘path’ where the value will be of type string. This is a **REQUIRED** property as is required by every SEND command. This represents the URL/ path that the HTTP request will be made to.

The body property of this command **MUST** contain a property / key named “method” where the value will be of type string and will either have the value of “GET” or “POST”. This represents the HTTP method of the HTTP request that the server will be making. For example, if the method is “GET” then the server **MUST** make a HTTP GET request.

The body property of this command **MAY** contain a property / key named “queryParameters” where the value will be a hash-map / map of type string to a list of strings. This is an **OPTIONAL** property as it is not required for every SEND command. This represents the requests query parameters that the server **MUST** send in the request IF the method of the request id “GET”. For a “POST” request this property MAY have the value of null (that is not of type string) or **MAY** not be defined at all.

The body property of this command **MAY** contain a property / key named “body” where the value will be a hash-map / map of type string to any. This is an **OPTIONAL** property as it is not required for every SEND command. This represents the request body that the server **MUST** send in the request IF the method of the request is “POST”. For a “GET” request this property **MAY** have the value of null (this is not of type string) or **MAY** not be defined at all.

**Example SEND command for GET request:**

```
{
          “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",
          “type”: ”SEND”,
          “body”: {
                    “method”: “GET”,
                    “path”: https://www.google.com,
                    “queryParameters”: {},
                    “body”: null,
          },
          “timeout”: 1000,
}
```

**Example SEND command for a POST request:**

```
{
          “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",
          “type”: ”SEND”,
          “body”: {
                    “method”: “POST”,
                    “path”: https://www.api.myendpoint.com,
                    “body”: {“username”: “exampleUsername”},
                    “queryParameters”: null,
          },
          “timeout”: 1000,
}
```

**AUTH COMMAND:**

**Purpose:**

The AUTH command is a command that will allow a client to authenticate their client with the server. Without authenticating the client **MAY** only be able to make a certain number of requests for the client – per server running session (from when the server initializes until it is closed). When the user is authenticated the user **MAY** be able to make an unlimited number of requests. The number of requests that the user can make when authenticated or not authenticated is down to the specific implementation of the protocol.

The AUTH command will facilitate the transfer of the users auth token from the client to the server.

**Structure:**

This command **MUST** have a property named ‘id’ that has a value of type string. This value SHOULD be a randomly generated string as **MUST** be unique to the request. This means that each time you make send a command (make a request to the proxy server) you will need to generate a new random string for that request and assign it to the ‘id’ property of this command. This **COULD** be how the client makes sure that the response that it receives is for the request that it made.

This command **MUST** have a property named ‘type’ that has a value of type string. This value **MUST** be “AUTH”.  This is a **REQUIRED** property as is required by every SEND command. This represents that the command being made is this command and not any other command, and that the body of this command corresponds to an AUTH command.

This command **MUST** have a property named ‘body’ that has a value of type hash-map / map that has the type string to any. This is a **REQUIRED** property as is required by every command.

The body property of this command **MUST** have a property named ‘token’ of type AUTH. This value **MUST** not be null. This is a **REQUIRED** property as is required by every SEND command. This represents the token that the server will use to authenticate the user.

**Example AUTH command:**

```
{
        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",
        “type”: “AUTH”,
        “body”: {
                “token”: “example_authentication_token”
        }
}
```

**Responses:**

**Structure:**

The structure of all of the responses will follow a similar format, there **MUST** be property named ‘id’ with a **REQUIRED** value of type string that **MUST** be the same value as the ‘id’ property of the request that the response is for.

All responses **MUST** have a **REQUIRED** property named ‘success’ with a value of type Boolean. This represents whether the request was successfully made and interpreted. If the success if false then it must conform to one of the error structures, if true then it **MUST** conform to either the ACK response structure, Success Response structure or AUTH response structure.

All responses **MUST** have a **REQUIRED** property named ‘status’ that has a value of type int. This value is **REQUIRED** to be one of the following numbers: 200, 201, 400, 401, 403, 405, 408.

All responses must have a **REQUIRED** property named ‘content’ and have a value of hash-map or map of type string to any / dynamic. This will contain the different types of response content that the client will use as extra information about the response.

**AUTH response (successful):**

**Purpose:**

The purpose of the **AUTH** response is so that the client knows that they have successfully authenticated with the server using their token.

**Structure:**

This response **MUST** have a **REQUIRED** message property of type string in content in the body property. It MAY be set to “You have authenticated successfully.”. This message can be implementation specific as it’s the status code that **MUST** be used to identify the response type.

**Example AUTH Response:**

```
{
        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",
        “success”: true,
        “status”: 200,
        “body”: {
                “content”: {
                        “message”: “You have authenticated successfully.”
                }
        }
}
```

**ACK Response:**

**Purpose:**

The purpose of the ACK response is so that the client knows that the server has received the request and should start listening for response packets. Also, the ACK response will tell the client where in the request queue their request is.

**Structure:**

Example ACK response:

```
{
          “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",
          “status”: 201,
          “success”: true,
          “payload”: {
                    “content”: {
                              “queue”: 1,
                              “message”: “What the ACK”,
                    }
          }
}
```

**Successful SEND response**

**Purpose:**

The purpose of the successful response is after the server makes the HTTP request for the client after receiving a SEND request if making the request was successful and everything up to sending the response back was successful.  This response will not only contain the contents of the HTTP response, but **MAY** also contain a property that represents the number of requests that the user has left.

**Structure:**

The send response **MUST** contain the following properties in the content property of the payload property.

This response **MAY** have an **OPTIONAL** property named “requests” in the content property with a value of type int. This number represents the number of requests the user has left to make.

This response **MUST** have a REQUIRED property named “response” that has a value of type hash-map or map. This will have the following properties: “data” with a value of type any / dynamic, this will be the response data of the HTTP request that was made. For example, for a get request to a webpage this **MAY** be the HTML that is returned from that request. Another property of the response map is “headers” which is the response headers received after receiving the HTTP response. Another property of the response map is “status” which is the response status code that is received after receiving the HTTP response. Further **OPTIONAL** properties of the response property are “config” which is the request configuration (depends on the implementations HTTP client), and “request” which is the request that was made.

**Example SEND Response:**

```
{
        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",
        “status”: 200,
        “success”: true,
        “payload”: {
                “content”: {
                        “response”: {
                                “data”: “DOCTYPE! HTML<…",
                                “headers”: {},
                                “status”: “200”,
                        },
                        “requests”: 4,
                }
        }
}
```

**Error Responses**

**Purpose:**

In the event of an error on the server side a good implementation **MUST** send back a response to the client, so the client is aware that has been a problem processing their request.

**Structure:**

This response **MUST** have a **REQUIRED** property named ‘status’ that has a value of type int. This value is **REQUIRED** to be one of the following numbers: 400, 401, 403, 405, 408.

**Bad request response**

**Purpose:**

A response with the status code 400 represents a bad request. This means that the request did not provide the server with the correct properties that were required to perform the function / action that the client wanted. For example, if a SEND command was sent to the server without a path property, or any other property that is REQUIRED to perform the SEND functionality in the body property of the command then the server **MUST** return with this a response with the status 400 and with an error and message telling them that they have made a bad request.

**Structure:**

This variation of the error response **MUST** have a **REQUIRED** message property inside content inside payload and this **MAY** be set to “You have made a bad request”. There is the freedom here for each implementation to set their own messages.

This variation of the error response **MUST** have a **REQUIRED** error property inside content inside the payload and this **MAY** be set to “BAD_REQUEST”. There is the freedom here for each implementation to set their own error name.

**Example of bad request response**

```
{

        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",

        “status”: 400,

        “success”: false,

        “payload”: {

                “content”: {

                        “error”: “BAD_REQUEST”,

                        “message”: “You have made a bad request”

                }

        }


}
```

**Unauthorized token error**

**Purpose:**

A response with the status code 401 represents an unauthenticated request. This means the request to authenticate the user has failed due to a bad token. For example, if the user has made an AUTH command with an invalid auth token the server **MUST** return with a response with a status code of 401.

**Structure:**

This variation of the error response **MUST** have a **REQUIRED** message property inside content inside payload and this **MAY** be set to “Could not authenticate using your authentication token”. There is the freedom here for each implementation to set their own messages.

This variation of the error response **MUST** have a **REQUIRED** error property inside content inside the payload and this **MAY** be set to “UNAUTHORISED_TOKEN”. There is the freedom here for each implementation to set their own error name.

**Example of an unauthenticated token request response**

```
{

        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",

        “status”: 401,

        “success”: false,

        “payload”: {

                “content”: {

                        “error”: “UNAUTHORISED_TOKEN”,

                        “message”: “Could not authenticate using your authentication token”

                }

        }

}
```

**Unauthorized request error**

**Purpose:**

An error response with the status code 403 represents an error that is returned from the server when the user has used up their allowed number of requests and they have tried to make another.

**Structure:**

This variation of the error response **MUST** have a **REQUIRED** message property inside content inside payload and this **MAY** be set to “You have reached your request limit”. There is the freedom here for each implementation to set their own messages.

This variation of the error response **MUST** have a **REQUIRED** error property inside content inside the payload and this **MAY** be set to “UNAUTHORISED_REQUEST”. There is the freedom here for each implementation to set their own error name.

**Example of unauthorized request error:**

```
{
        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",

        “status”: 403,

        “success”:         ,

        “payload”: {

                “content”: {

                        “error”: “UNAUTHORISED_REQUEST”,

                        “message”: “You have reached your request limit”

                }

        }

}
```

**Internal Server Error Response**

**Purpose:**

This error response **MUST** be sent back to the client if the server throws an exception at any time during the processing of a request. This is different from a bad request error as that is used when the request is what caused the error, but this error response is returned to the client when the implementation does throws an exception whilst processing the request.

**Structure:**

This variation of the error response **MUST** have a **REQUIRED** message property inside content inside payload and this **MAY** be set to “There was a problem when processing your request.”. There is the freedom here for each implementation to set their own messages.

This variation of the error response **MUST** have a **REQUIRED** error property inside content inside the payload and this **MAY** be set to “INTERNAL_SEVRER_ERROR”. There is the freedom here for each implementation to set their own error name.

**Example Internal Server Error Response:**

```
{

        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",

        “status”: 405,

        “success”: false,

        “payload”: {

                “content”: {

                        “error”: “INTERNAL_SERVER_ERROR”,

                        “message”: “There was a problem when processing your request.”
                }

        }

}
```

**Timeout Error Response (SEND REQUEST)**

**Purpose:**

This error response **MUST** be sent back to the client if the server does not receive the HTTP request in the time that was specified in the SEND request. This is so that if there is a problem when making the HTTP request the server can cancel the request and the client can be notified that they will not be receiving a response to their request.

**Structure:**

This variation of the error response **MUST** have a **REQUIRED** message property inside content inside payload and this MAY be set to “Your request has timed out”. There is the freedom here for each implementation to set their own messages.

This variation of the error response **MUST** have a **REQUIRED** error property inside content inside the payload and this **MAY** be set to “TIMEOUT_ERROR”. There is the freedom here for each implementation to set their own error name.

**Example:**

```
{

        “id”: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7",

        “status”: 408,

        “success”: false,

        “payload”: {

                “content”: {

                        “error”: “TIMEOUT_ERROR”,

                        “message”: “Your request has timed out.”

                }

        }

}
```

## 7. Error Handling

Error handling client side will be as simple as checking the success of the response and the status to handle the errors as the user sees fit. The payload will be response specific (error specific).

Errors on the client side can be thrown at these points:

- When the client tries to start a socket connection between itself and the server to send a request.
- When the client makes a request to the server and the time before an ACK response meets the connection timeout.

An error payload will contain 2 properties: 'error' and 'message'. The error will contain a string representation of the status example: A server timeout payload will look like the following response.

On the server-side errors can be returned to the client at these points.

- When the server receives a request, or the authorization token is invalid.
- When the request timeout is met after the server has made the HTTP request for the user.

## 10. Packet segmentation

Due to the nature of the protocol requiring us to send large packets back from an SNP running server containing a HTTP response we need a way to turn a response into multiple packets that contain a segment of the response and some meta data about the packet (such as packet number, total number of packets etc.).

An implementation of the SNP protocol **MUST** segment **ALL** commands that are sent from client to server **AND** server to client.

Each implementation **MUST** set a required 1kb (1 kibibyte / 1024 bytes) request payload size (not including the header). For example, a response may be 2048 bytes and 1024 bytes of that response **MUST** be read into the first packet and the next 2048 bytes **MUST** be read into the second packet.

**Packet structure:**

```
{
      “id”: <PACKET_ID>,
      “packetNumber”: <PACKET_NUMBER>,
      “totalPackets”: <TOTAL_PACKET_COUNT>,
      “payloadData”: <LIST_OF_RESPONSE_BYTES>,
}
```

**Example packet:**

```
{
        “id": “snp_packet_0”,

        “packetNumber”: 1,

        “totalPackets": 15,

        “payloadData”: [123, 34, …, 111]

}
```

Each implementation **COULD** have a buffer that when they receive a command (it will be a utf8 encoded packet) it would decode the packet into a string then encode it to JSON, create a packet object from it to access it through an object-oriented approach or just to use the JSON object to add it to a buffer that keeps track of how many packets have been received and what they are. When the buffer stores the number of packets that the packet has as its ‘totalPackets’ property it will order the packets, iterate through them taking the payload data out of the objects and making one big list of bytes (when these are all added together, they will make up the original response bytes). Once the list response bytes have been put back together, utf8 decode the bytes back into a string then encode the string into a JSON object and the original response should now be formed as it was sent.

Examples for a HTTP response that was split up into packets:

```
[

{id: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7", packetNumber: 1, totalPackets: 2, payloadData: [123, 34, …, 111]},

{id: “1b3b20f0-c3f0-11ec-bb5d-510ac226a3a7", packetNumber: 2, totalPackets: 2, payloadData: [112, 101, …, 55]},

]
```