Implementation of the NEAT Policy Manager

Jan-Olof Andersson, Samuel Larsson, Tobias Sjöholm

The Faculty of Health, Science and Technology

”Computer Engineering Project” - Implementation of the NEAT Policy Manager

15 hp

Karl-John Grinnemo

Lothar Fritsch

2019-01-21

Implementation of the NEAT Policy Manager

Jan-Olof Andersson, Samuel Larsson and Tobias Sjöholm
Department of Mathematics and Computer Science

Abstract

The NEAT system was developed in 2017 to increase flexibility in the choice of network transport protocol being used. One of the most important components of the NEAT system is the Policy Manager (PM), which determines what protocol is to be utilized by the application. The PM is written in Python while the rest of the NEAT system is C-based, so a natural evolution of the PM is to perform a functional translation of it to C. While the main goal was solely to develop a fully functional C-based PM, the difference in programming languages in the end also brought a performance increase of 28 times compared with the Python-based PM. There are still a few improvements left to do in the PM, but it is already a notable improvement for the NEAT system as a whole.

Keywords: NEAT, policy manager, transport protocols, Python, C

Acknowledgements

Thanks to Zdravko Bozakov for his work on the Python-based Policy Manager and to Karl-Johan Grinnemo for his guidance.

Karlstad, January 9, 2019 Jan-Olof Andersson, Samuel Larsson and Tobias Sjöholm

Contents

1 Introduction

Network programmers of today are faced with a limited option of transport solutions for their applications. The choice stands in practice between a reliable protocol, TCP, and an unreliable protocol, UDP. This lack of choice is due to the rigidity of the standard socket API, forcing protocol specific application code, and the interference by middleboxes when they encounter new protocols. This inflexibility in the use of transport protocols is what's regarded as the "ossification" of the transport layer. The NEAT API is a proposed solution to the ossification problem of the transport layer. NEAT enables new protocols to be deployed over the internet thanks to its innovative use of features such as happy-eyeballs [11], a new socket layer API, protocol specifications, etc. One key component in the NEAT API is the Policy Manager (PM). The PM lets the developer define which services that are needed by the application. The PM, when given the desired services, will select the protocol which is most suitable for the purpose.

The current PM in NEAT is written in the programming language Python. With Python, it's possible to write clear and concise code in a fast manner. However, in system programming, it's generally preferred to write applications in a low-level language. The C language is a typical low-level language used in system programming and is the language used in the implementation of the NEAT system, except for the PM. Given the fact that C is preferred in system programming, and that the majority of the NEAT system is written in C, it's of interest to port the PM from Python to C. Porting the PM to C will also enable the PM to integrate smoothly with the rest of the system.

The porting of NEAT's PM from Python to C is the subject of this paper. Section 1 begins by reviewing background information about the NEAT API together with it's PM component. Next, Section 3 presents the planning of the project. Section 4 presents the design decisions made for the rewriting of the PM. This includes the components relating to the PM, such as the Policy Information Base (PIB) and the Characteristics Information Base (CIB). Section 5 presents the implementation, which follows the design phase, and all the technical details of the PM. The testing in Section 6 documents our test suites and the results of the test runs.

2 Background

Most communication over the internet today uses either the TCP or the UDP transport protocol. The reason is that the transport layer has become too rigid, making any attempts to extend or add new transport protocols difficult. This is what's known as the "ossification" of the transport layer [15]. The two main culprits that have contributed to the ossification are middleboxes and the standard socket API. Middleboxes are network layer devices that come in many different flavors, each putting their own limitation on the overlying transport layer. Not only do middleboxes place constraints on the protocols on networks, they themselves also ossify the infrastructure of the networks. As a result, networks end up with middleboxes that are optimized for TCP and UDP traffic with no knowledge of any other potential transport solutions. Deploying new protocols in this kind of environment is unlikely to gain any traction. The second contributor to the ossification of the transport layer is the socket API [12]. The socket API is strict about what kind of protocols that can be used and requires details that are protocol specific. New features are difficult to incorporate into the socket API and in general poses a challenge for new protocols and their configurations. A promising solution to the problem of ossification is to deploy a framework for the transport layer, composed of several components, to enable a flexible transport layer solution.

The New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT) is an implementation of such a framework, taking one step closer solving the problems with the ossification of the transport layer. NEAT will allow the transport protocol to be determined and verified at run-time instead of at design time. This eases up the connection between the application layer and the transport layer, where the developer no longer has to decide on a transport protocol but only what properties that are desired for the protocol. Then NEAT will at run-time find the most suitable protocol depending on the properties and what protocols are supported over the used path on the internet. In Figure 2, an overview of the NEAT system is presented, this figure is from Khademi et al. [12].

The PM is the "brain" of the NEAT system. The purpose of the PM is to generate a list of suitable transport candidates for a given set of requested properties, e.g., low-latency, ordered by suitability. The set of rules for how to match certain properties to candidates is defined as files that are handled by the PIB and CIB components of the PM. The policies add or update NEAT properties to each potential candidate as defined by lookup algorithms in PIB, CIB. These algorithms set the eventual score that will be used when sorting the candidate list.

The communication between the NEAT Core and the PM is through a Unix socket with libuv [1]. It's event-driven and has support for multiple platforms. Libuv provides the capability of asynchronous I/O operations based on event loops [7]. The event loops establish the information from the I/O operations and it's limited to one thread. Libuv have the capability to run multiple event loops assuming every loop run its own thread.[1] The Unix socket

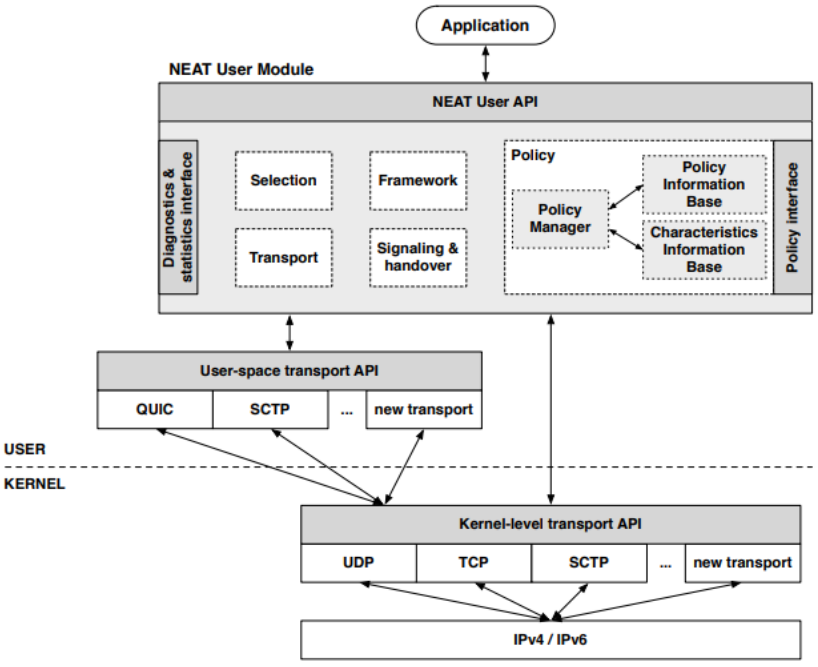

Figure 1: Overview of NEAT.

between the PM and NEAT Core is as an inter-process communication mechanism that allows the data to flow in both directions. Which is needed, the NEAT Core will send requests in the form of a JSON object (JavaScript Object Notation) and the PM will respond with a JSON object. A JSON object is a lightweight data-interchange format. The JSON content consists of a key and its value which can have multiple types.[10]

3 Project plan

The project started on the 3rd of September 2018 and ran through the second term of 2018 and finished on the 16th of January 2019. It was planned into four phases: planning, design, implementation, and conclusion. Evidently, it follows a waterfall model which is not ideal for a software development project. Because of the fact that the planning, design, and finalization phases were expected to remain quite static throughout the course of the project, we decided that this waterfall approach was not good enough and instead employed a more agile and Scrum-like [13] development model only during the implementation phase. We split up the implementation phase into five sprints à two weeks and received bi-weekly feedback from our advisor, Karl-Johan Grinnemo, after every sprint planning meeting. In this way, priorities as well as the course of the actual resulting product, could be modified much more

dynamically between sprints. We also used a Gantt-chart [8] to distribute resources across the different tasks. The initial Gantt chart of the project can be found in Appendix A.

4 Design

The NEAT documentation [11] defines the PM and its surrounding components as tools for the NEAT system to adapt to a wide range of network requirements and scenarios. It also defines the functionality of the PM and how it communicates with other components, such as the NEAT Core module. This results in a very specific requirements specification for the PM, which ultimately affects the final design.

In this design, the input that the PM receives from NEAT Core is a request that contains some requirements and the output is a list of ranked candidates for transport protocols that NEAT Core then processes. To help identify and evaluate possible candidates on the system, the PM makes use of the two separate information repositories: CIB and PIB. However, the NEAT System also needs the ability to add new information to these two repositories, which requires two extra paths of communications. The initial interface requirement is therefore to have a **three-part socket interface** towards NEAT Core.

However, the NEAT documentation also describes an **external REST API interface**. This is required so that external hosts can add and read information in the CIB and PIB, for the purpose of awareness of the external network.

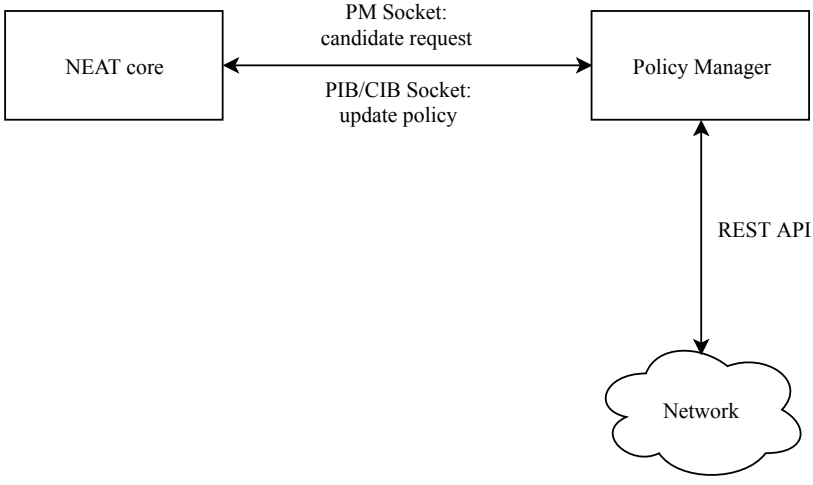

Figure 2: Diagram showing a high-level overview of the complete neat system and Policy Manager.

The PM should, as follows from Khademi et al. [11], use available information to evaluate and present possible transport candidates to the NEAT

Core. All data objects that are processed by the PM are JSON [10] objects, so the PM will need to be able to parse and deparse JSON objects.

The PIB and CIB are very similar in the way that they are designed. The PIB stores and processes NEAT profiles and NEAT policies while the CIB stores and processes CIB nodes. However, profiles are used to map high-level properties to a request, CIB nodes are used to fetch a list of possible transport candidates and policies are used to ultimately filter out or extend these candidates. Evidently, the ways policies, profiles, and CIB nodes are applied to an incoming request are very different. Profiles, policies and CIB nodes are all data objects stored locally in JSON format. Both the PIB and CIB have functional responsibilities that require them to perform lookups on their stored data objects to filter out or extend potential transport candidates for the PM. The lookups are slightly varied in functionality, but they all get a JSON object as input which they match against their stored data objects and map their properties against the input. This modified JSON object is then returned by the lookup function.

From the point-of-view of the PM, when a request arrives on the request socket, the PM should follow the algorithm specified in Khademi et al. [11], which is in short:

1. Profile lookup: A lookup is performed in the PIB to look for profiles that match the incoming request.

2. CIB lookup: The CIB lookup looks for CIB nodes that fulfil the properties of the request and returns a list of CIB node candidates.

3. Policy lookup: For each of these candidates, NEAT policies are matched against them to append more properties and possible filter away some candidates.

4. Respond to the NEAT Core with the list of ranked candidates via the request socket.

The main functions of the PM are thus to start up the CIB and PIB and then perform the algorithm above when a request arrives on the request socket. It is also responsible for delegating method calls to the CIB and PIB when data objects arrive on the two CIB and PIB sockets. The last functional requirement of the PM is that it is responsible for starting up the REST API.

Khademi et al. [11] have defined six endpoints from which the REST API[9] should be able to receive and handle requests:

- `GET: /pib`,
- `GET: /pib/{uid}`,
- `PUT: /pib/{uid}`,
- `GET: /cib`,
- `GET: /cib/{uid}`,

- `PUT: /cib/{uid}`.

When one of these endpoints are requested, the REST API directly calls the necessary methods in the CIB or PIB.

One of the main reasons behind this project is to improve efficiency of the PM. Since the PM is now developed C, which is a lower-level language compared to Python, a major benefit of the fact is the possible performance improvement compared to the Python PM. Therefore, a performance requirement is that it has to be faster than the Python PM, both in function but also in startup and initialization.

There is no requirement for the course of this project to develop the PM with multi-platform support. However, as the PM is written in C, it will be extensible to most relevant platforms. There is also a design constraint to keep the number of dependencies, especially platform dependent dependencies, to a minimum to preserve extensibility and maintainability.

4.1 Design decisions

It was decided during the initial part of the development process that the high-level design of our C version of the PM should resemble, as closely as possible, the original Python version so that the modifications around the PM i.e., the NEAT Core interface and ultimately the NEAT documentation, is kept to a minimum. The development of the PM would then be done independently from the NEAT Core, which would facilitate the development process. The idea was that the implementation of all internal components of the new PM should work the same way as in the original Python version. It would then be easy to switch between the C version of the PM and the Python version in a black box fashion in such a way that the input to the black box could be compared with the output, giving a guideline to how well certain functions work in the new PM compared to the former Python ditto.

Since it was established early on in the planning phase that there was no requirement for support for multiple platforms, the choice of platform just fell on the most accessible platforms we had at the time, and the most stable platform from a NEAT perspective. This led to the decision to implement support for Ubuntu 18.04 and Debian GNU/Linux 9. Linux is a free and open source platform, well established with a widespread usage and hence well supported by a large community. With two Linux distributions, we could expect to produce a PM that could to some extent support other different Linux distributions as well, with no expectation of actually testing it. The platform decision also led to the decision to try to keep libraries independent from Linux distributions; not only for the sake of multi-platform support but also for the sake of portability while developing on two different distros.

5 Implementation

When the PM starts up, it will go through the initialization procedure in which the PM sets up all sockets, the REST server, and starts reading in all policy file data from the `infobase` folder. `infobase` is a folder that stores all policy files for both PIB and CIB in separate directories. The first step is to read in all the PIB and CIB policy files. The PM calls the corresponding initialization functions in the CIB and PIB components which reads in the policy files and stores them as internal linked lists in memory. This is done since keeping the policies in RAM gives a performance boost when the policies need to be accessed. This step is also crucial since the CIB initialization procedure creates the CIB nodes based on available interfaces. The second step is to start the REST server. The PM starts the REST server as a separate thread via the pthread API [3]. There's no need, in this case, to use libuv to create the thread since the REST server doesn't perform any callback functions to the PM. The REST service could also run in a separate process, but since running the REST server without the PM would not be of much use, it was integrated as a thread into the PM. Finally, the PM starts to create all the sockets. The sockets created during the initialization process are `neat_pm_socket`, `neat_pib_socket`, `neat_cib_socket`. These sockets are created in the socket directory, which defaults to `$HOME/.neat/`.

5.1 Socket Handling with Libuv

Socket creation is handled by the libuv library that provides asynchronous functions for, among other things, listening on socket connections. Since these sockets are for local use, the socket type is set to a UNIX domain socket [6]. Each socket in libuv takes one callback function. The callback function executes after a new event is triggered by a socket connection. Once the bind and listen calls have been called for each socket, the main loop in libuv starts and waits for any socket events.

Using callback functions have both benefits and drawbacks. Some drawbacks include added debugging complexity and unwieldy callback chains. However, in the PM, the asynchronicity is limited to the socket communication. Thus, the mentioned drawbacks pose a negligible impact. The benefits of an event-based design, on the other hand, is simpler and intuitive code.

5.2 Format for Representing NEAT Properties

JSON [10] is the common format used for representing NEAT properties in NEAT. The policy files are written in the JSON format, the REST API depends on the requests being in JSON, and candidate requests to the PM contain an array of JSON NEAT properties. All inputs to the PM are plain JSON data. Preferably, the PM should store the JSON internally in an efficient data structure with common operations available. The library provides a suitable data structure for storing and manipulating JSON objects. The functional-

ity that libjansson provides is efficient since all keys in the JSON structure are stored in hash tables. The library also defines common operations for getting and adding JSON values, JSON array manipulation, iteration over JSON key-value pairs, and many other operations as specified by the libjansson documentation [14]. The choice to represent NEAT properties as JSON makes the PM basically independent from the core of NEAT. Since the PM and NEAT Core share a common data structure to represent NEAT properties, the JSON format became the common format between them. There is a greater potential to share data structures with the NEAT Core, since the PM is implemented in the same language. However, keeping the JSON format and using libjansson to process it is a better option instead of creating a specific data type for NEAT properties. The files in the `infobase` repository would still be written in JSON and the REST API is also dependent upon the JSON format. The JSON format and the chosen format for representing NEAT properties would need to be converted from one to the other. The trade-off between having a special internal format over a common format for NEAT properties would therefore be less rewarding. It would require a lot of refactoring of JSON dependent code in NEAT Core and the PM and require conversion steps. The JSON data or its JSON data structure are therefore not converted into any special NEAT property structure in C.

5.3 The Node Data Type

All information in the PM is saved in what is called a Node data structure. The nodes are implemented as a linked list, where every node has a pointer to another node. A linked list implementation was chosen since the nodes need variable length lists. Additionally, a linked list implementation is not a performance issue. The reason for using the node data type in the first place arises from the fact that the PM requires extra internal housekeeping information for each policy file. The complete structure definition is shown in Figure 3.

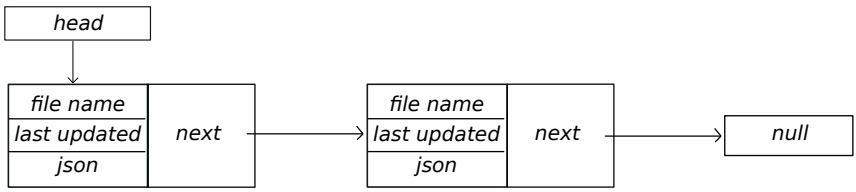

Figure 3: The linked list implementation of PIB/CIB nodes.

The identifier of a node is a filename. This means that every single PIB and CIB file will be parsed into a node structure. The content of the backend file will be stored as a JSON pointer. The filename will be saved as a char array to uniquely identify the node. The node data structure will also contain a time stamp that is set when the node is last updated. For example, when the PM receives a request and starts to perform the lookup algorithm. Updates to the backend files might make the information in the nodes become out of

date. A comparison between the last time the backend file was updated and the time stamp of the node is–therefore–made. The node is in this way ensured to always contain the newest information.

5.4 Request Path Through Policy Manager

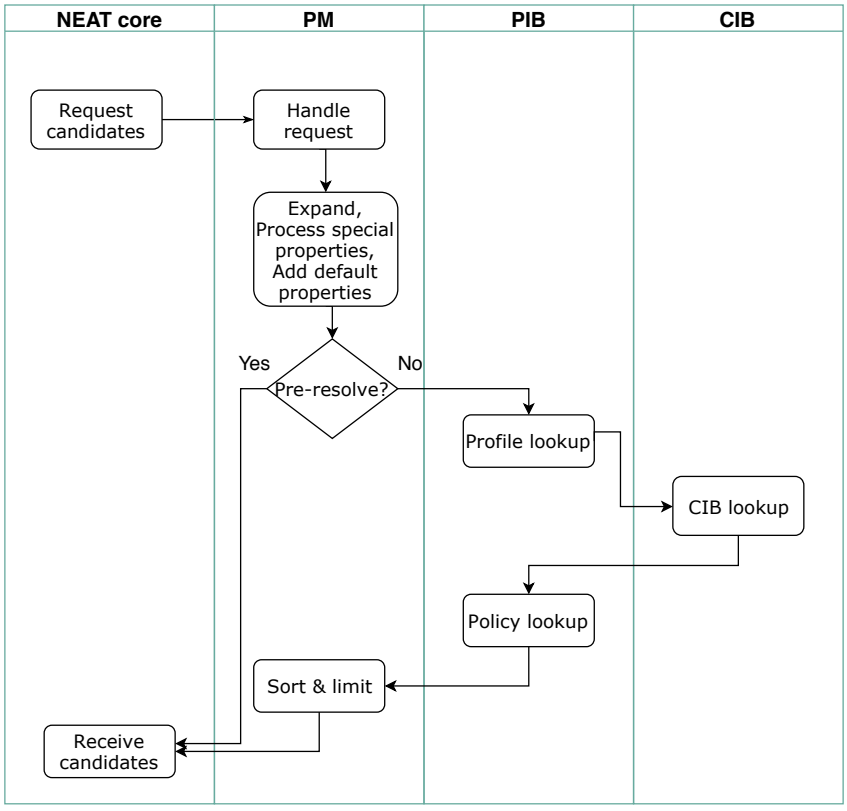

Figure 4: Diagram showing the path taken by a request from NEAT Core through the PM.

A simple illustration of how a request from the NEAT Core logic traverses through the lookup process in the PM is shown in Figure 4. First, the NEAT Core logic sends a request to the PM socket that contains a list of required properties the PM uses to generate potential candidates. Socket requests are handled by the PM, which in turn calls the lookup routine. The lookup routine has to do some pre-processing before the candidate match may take place:

1. Expand. In this step, each property array gets expanded into properties by applying the product operation.

2. Process special properties. This processes all properties that are considered special. Examples of special properties include expanding the property "interface@ip" into separate properties.

3. Add default properties. If any property attributes are missing, then this routine will add in the default values.

If the property `__request_type` is set to pre-resolve, it skips the rest of the lookup process and the current candidate list is sent back to the NEAT Core. This step is needed for DNS resolving. Otherwise, the lookup continues by first finding candidates based on PIB profiles. A lookup in CIB is performed on each candidate. Finally, a PIB policy lookup is done. The resulting candidate list is then sorted by score, limited to a maximum number which is set to a constant value of ten, and sent back to the NEAT logic.

5.5 PIB

The PIB component contains all the implementation that concerns the handling of PIB policy and profile files. These files are stored on disk in the `infobase/pib/{policy, profile}` folders. Because disk access is slow and cumbersome, the PIB files are loaded and stored into memory as a linked list of nodes with one list for policy and profile.

PIB implements functionality for adding and reading PIB nodes. Adding PIB nodes is required by the PM component and by REST PUT requests. When the PIB component adds a node it will both add it to the internal linked list and write it to a file on disk. Reading nodes is needed by the REST GET requests. Reading is provided by returning a uniquely specified node.

PIB produces a set of candidates from given properties given by the main PM component in the PIB lookup routine. The PIB lookup routine is used when performing a lookup on profiles and policies. The lookup algorithm, shown in Algorithm 1, goes through each profile or policy and tries to apply them to each of the current potential candidates. If the match property in the profile/policy matches any of the properties in a candidate, if the match property is a subset to the properties of a candidate, then the properties of the profile/policy should be added to the candidate. A subset check is conducted by the subset function defined in the parse_json component, which compares each JSON key and attribute in both JSON objects. The new properties must be expanded before being added. The expand process simply looks for attributes in the JSON object with arrays to produces a product that combines all array properties into new separate profiles/policies. The next step is to add the properties of each expanded node to the current candidate.

The function `merge_properties` adds the properties of the expanded policy to the current candidate by comparing each property. If the property is not found, it is added to the candidate. If it does exist and it's allowed to overwrite it, it is overwritten. Otherwise the property will be added to the candidate by a set of special rules applied to the property in the function `merge_update_property`.

```
for policy in pib.policies do
    updated_candidates := empty list
    for candidate in candidates do
        if policy.match ⊆ candidate then
            for expanded_prop in expand(policy) do
                updated_candidate :=
                  merge_properties(expanded_prop, candidate)

                updated_candidates.append(updated_candidate)
            end
        end
        else
            updated_candidates.append(candidate)
        end
        candidates := updated_candidates
    end
end
```

Algorithm 1: PIB

The rules for `merge_properties`:

- If the values of the two properties are equal, the first property will get the sum of both properties scores as its own score and its precedence becomes the max value of the precedence values of both properties.
- If the two properties values are unequal, and the policy's property has higher or equal precedence, then the candidate's property gets the same value and precedence of the policy's property.

A description of the `merge_properties` algorithm is given in Algorithm 2. Once the lookup is done, the lookup returns an array with candidates.

5.6 CIB

When the CIB is instantiated by the PM, the CIB nodes that are stored in the `infobase/cib` folder as `.cib` files are read into the linked list of the Node data type, called `cib_nodes`. This linked list is the actual information base where all of the CIB nodes reside and where all the operations on the CIB are performed. As CIB nodes represent network characteristics, including the local network interfaces, the CIB is responsible for generating CIB nodes from them. The CIB creates CIB nodes from the local network interfaces on the machine in the function `generate_cib_from_ifaces`. To implement this efficiently, we used the standard UNIX and C libraries shown below.

Listing 1: Libraries used to fetch local network interfaces.

```
#include <net/if.h>
```

```
for key in policy do
    if key ∉ candidate.keys OR overwrite = True then
        candidate.key.value := policy.key.value
    end
    else
        /* Decide if order should be reversed          */
        if policy.precedence = PRECEDENCE_BASE then
            tmp := policy
            policy := candidate
            candidate := tmp
        end
        /* Do update                                   */
        if policy.key.value = candidate.key.value then
            candidate.key.score := candidate.key.score +
              policy.key.score
            candidate.key.value := policy.key.value
            candidate.key.precedence :=
              max(candidate.key.precedence, policy.key.precedence)
        end
        else
            if policy.key.precedence = candidate.key.precedence =
              PRECEDENCE_IMMUTABLE then
                Error!
            end
            if policy.key.precedence ≥ candidate.key.precedence then
                candidate.key.value := policy.key.value
                candidate.key.precedence := policy.key.precedence
            end
        end
    end
end
```

Algorithm 2: Merge properties.

```
#include <ifaddrs.h>
#include <netinet/in.h>
#include <netdb.h>
```

Like the PIB implementation described in Section 5.5, the CIB adds the new nodes to both the internal linked list and to a `.cib` file on the disk. As with the PIB, the CIB has public read functions that are used by the REST API to request a CIB node with a unique ID, and write functions that are used by both the REST API and the CIB socket to add a new CIB node, provided that the sufficient properties are included in the JSON object [11].

The main function and responsibility of the CIB is the candidate lookup it performs for the PM. It receives the request that has been modified by the profile lookup. The lookup routine then continues by attempting to match every CIB node with any of the request properties. As in the PIB lookup algorithm, a subset check is done with the subset function from the `parse_json` component to determine if there is a match. If there is a match, the CIB node gets added as a candidate. The algorithm of CIB lookup is shown in Algorithm 3.

```
for node in cib.nodes do
    immutable_properties := empty list
    for property in input_properties do
        if property.precedence = IMMUTABLE then
            immutable_properties.append(property)
        end
    end
    if immutable_properties ⊆ node.properties then
        continue
    end
    candidate := input_properties
    merge_properties(node.properties, candidate)
    candidates.append(candidate);
end
```

Algorithm 3: CIB lookup

5.7 REST API

The REST API is instantiated in a pthread [3] that is created by the PM at initialization. It uses the library Ulfius [4] to implement the REST service functionality. For the sake of simplicity, it uses the same port, 45888, as the REST service in the Python PM. When one of the six endpoints is requested, they, in turn, call an individual dedicated helper function that marshals the request data and calls the proper function in the PIB or CIB. PIB/CIB nodes are updated after every PUT request to the REST API, meaning that they do not write the nodes to storage for the CIB or PIB to read or reload. Instead, they get added directly into the node structure in the CIB or PIB. It's better this way to avoid explicit reload calls and to avoid any error related to missed

reload calls. Ulfius takes care of all the callback logic and thread handling, so the REST service and its six endpoints are very lightweight.

6 Testing

To verify the functionality of the PM's own data structures and algorithms a second project was implemented. This test project contains unit tests that are executed within the Unity test framework [5]. Unity is a framework that uses assertions in order to evaluate if unit tests have succeeded. Assertions are statements that are expected to be true. This is used in our unit test: call a function and then compare it with the expected result with an assertion. For more information about Unity see ref. [5]. The design of the test project is presented in Figure 5.

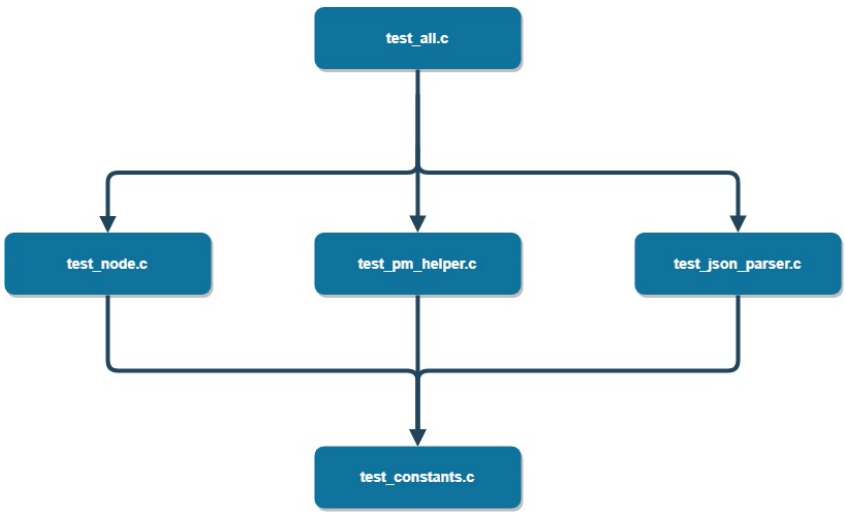

Figure 5: Block diagram of the test project.

The idea behind the design of the test project is that the file `test_all.c` will be the central part of the project that starts executing all unit tests. All constants shared among the blocks are placed in the `test_constants.c`. In this way, consistency of the constants is guaranteed, and if constants need to be changed, the change only has to be applied in a single place. The blocks in the middle part of the diagram contain the unit tests. Every single file in PM has its own test file, however, it's renamed with the word "test" in front of it, e.g., `node.c -> test_node.c` This makes the unit tests easy to locate in the test project. If you know the location of the real function in PM, you will also know where the function is in the test project. In total, the test project contains 29 unit tests. These tests verify that the functionality of the logic corresponds with the PM's own data structures. For example, the nodes used to store the information are a linked list. This linked list has some standard

operations such as add, remove, and update. The unit tests are performing a check that these operations are executed correctly. The test project also verifies the advanced expand algorithms used when the PM receives a request.

7 Discussion

There are multiple benefits with having the PM implemented in the programming language C, compared to the old PM implemented in Python. Each and every benefit will be described separately below.

One major benefit is that the whole NEAT logic is consistent. The same programming language and the same code standards are used everywhere in the NEAT system. This will be beneficial for future developers since they only need to know the programming language C and not Python. The tools and dependencies will also decrease since the PM is in the programming language C.

The biggest benefit is the performance of the new PM. As we know, C is a compiled programming language that is closer to the machine than Python, while Python is a scripting language that is interpreted at runtime which slows down the performance of the system. Python is by design much slower than C. Having powerful OOP concepts like inheritance makes it easier to code but at the cost of getting slower performance. In general, we can directly conclude that a program in C will run faster than a Python program, even though these two types of languages are often used in different contexts.

In order to find out how much faster the C version of the PM is, a Python script was implemented that sends 1000 requests to the PM and measures how long the PM on average takes to handle a request. Although the estimated mean value is strongly dependent on what computer the PM is running, it still gives us an overall feeling of how much faster our new C version of the PM is in relation to the Python version.

The test script was performed first on the old PM implemented in Python. The average time for a request was calculated to be 32.94 milliseconds. The same test script was then executed with the new PM and the average time for a request was calculated to be 1.16 milliseconds. This means that our C-based PM is executing 28 times as fast as the Python-based ditto. The summary of the time testing is presented in Figure 6. All the data from these tests can be found in the `docs/test_results` folder on the NEAT git [2].

The C-based PM is executing so fast that the time taken to communicate through the socket using libuv is the major contributor of the execution time. In order to find out how long the socket communication takes and how long the request is inside the PM, a timer clock was implemented inside the PM. The time script was executed again, and the mean value of the clock inside the PM was calculated to 0.3 milliseconds. As follows from Figure 6, the mean value from the test script was calculated to be 1.16 milliseconds. This means that the new PM only takes 0.3 milliseconds to handle a request. The rest of the 1.16 milliseconds is due to the request being passed through the socket. From these statistics, we conclude that in order for the PM to have

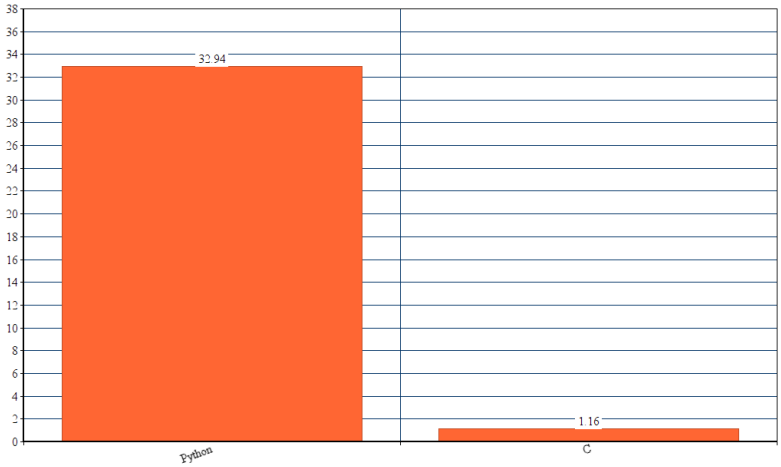

Figure 6: The mean value of the response time, Python-based PM and C-based PM.

a response time less than one millisecond, removing the socket communication is required. This is possible if the PM is integrated into NEAT Core (cf. Section 8).

Overall, our C-based PM is handling a request much faster than the Python-based one. This is as expected. The biggest factor that slows down the PM today is the use of socket communication. Reading and writing to/from a socket takes for an average NEAT Core request around a half millisecond. This, of course, depends on how much data is sent through the socket. If, in the future, there is interest to optimize the response time of the PM further, a deeper look into the socket should be done. From a performance point of view, removing the socket would be the best choice.

8 Future work

According to Khademi et al. [11], it should be possible for the PM to extend CIB nodes. There are two cases where the extend algorithm is used which has not yet been implemented: extending existing CIB nodes and extending CIB rows. In the first case the extending is based on a graph structure where all the CIB nodes are generated by traversing the graph from the root node to the connected nodes. The main purpose is for referencing existing CIB nodes to create the graph based on network characteristics. There are three criteria to be fulfilled for a CIB node A to be extended by a CIB node B.

- If A has the attribute `root` equal to `true`,
- The attribute `link` in B equals `true`,
- The properties in the match attribute in B matches the set of properties in A.

If these three criteria hold, A is extended with the properties in the properties attribute in B. In the second case, a CIB node A may extend a CIB row B if

- The attribute `link` in A is false,
- The match attribute in A matches the set of properties in B.

If this is the case, B is extended by the addition of the properties in the properties attribute in A. For more details, see [11].

At the end of the project, the developers rushed to implement a lookup algorithm that should have the capability to handle multiple requests from NEAT. In this algorithm, a few memory leaks occur when memory is dynamically allocated but not freed. In order to free these memory cells, you will need to understand how the Jansson library is allocating memory behind the scenes and when it is the PMs responsibility to free the memory.

NEAT, at the time of writing, supports Linux, FreeBSD, NetBSD, and macOS. The PM was developed on Ubuntu and Debian Linux and has only been tested in these environments. Future work is therefore needed to port the PM on different platforms.

The current PM is designed to be fully independent of the NEAT Core. However, once the above-mentioned improvements and implementations have been made, the PM will be ready to be fully integrated into NEAT. To integrate the PM into NEAT Core, the PM should preferably run as a libuv thread that is initialized concurrently with the NEAT Core. This means that the current implemented socket communication in both the PM and NEAT Core needs to be removed and replaced with asynchronous thread communication.

One possible issue with integrating the PM into the NEAT Core is the problem of executing multiple instances of the PM. Today, the PM is designed to be used by all NEAT instances running on a single machine. In order to integrate the PM with the rest of the NEAT system, a design decision needs to be made: either if NEAT should create a PM for every single NEAT instance, or if all NEAT instances should share the same PM.

To run the PM with multiple instances, two changes have to be made: the socket and the backend folder. Let's begin with the socket. NEAT is sending a JSON request to the PM through libuv. Libuv enforces an asynchronous, event-driven style of programming. Its core job is to provide an event loop and callback-based notifications of I/O and other activities. If you attempt to start multiple instances of the PM, all these instances would still utilize the same socket. This causes the instances to collide with each other. So, in order

to have multiple instances of the PM removing the socket communications on both sides are necessary.

The second change that is needed in order to have multiple instances of the PM is the backend folder. The PM creates a hidden backend folder in the user's home folder. In this folder, all the PIB and CIB files are stored during the execution of NEAT. This folder can be moved to the executing folder and receive a unique name for all instances of the PM. This would prevent different instances of the PM to impact each other. This assumes that you don't want the different instances to impact and use the same PIB and CIB files.

In order to run the PM with one instance per computer, not many changes have to be made to the PM itself. In fact, this is how it's currently designed to run. To integrate it into NEAT and prevent NEAT to start up multiple instances of the PM, a clever design is needed. For example, NEAT could create the PM in a separate process and give the process a special name. When NEAT starts up, it can explicitly check if this process is running in the system. If it does, the PM has been started by another instance of NEAT. If it does not, NEAT will initialize the PM. This will guarantee that you will never have more than one running instance of the PM on the same computer.

9 Conclusion

Rebuilding the PM in C from Python with the aim to get a more homogenous NEAT system not only resulted in bringing the PM closer to the NEAT Core from a design perspective, but also gaining a major performance improvement that will further increase the performance for the rest of the NEAT system. Since the PM is viewed as the "brain" of the NEAT system, it is a very important component of the NEAT system. There are still many improvements left for future work. However, the current C version offers significant performance gains.

References

[1] libuv documentation. `http://docs.libuv.org/en/v1.x/`. [Online; accessed 16-Dec-2018].

[2] Neat-project/neat: A new, evolutive api and transport-layer architecture for the internet. https://github.com/NEAT-project/neat. [Online; accessed 7-Jan-2019].

[3] pthreads(7) - linux manual page. `http://man7.org/linux/man-pages/man7/pthreads.7.html`. [Online; accessed 4-Jan-2019].

[4] Ulfius api documentation | ulfius. https://babelouest.github.io/ulfius/API.html. [Online; accessed 7-Jan-2019].

[5] Unity — throw the switch. http://www.throwtheswitch.org/unity/. [Online; accessed 7-Jan-2019].

[6] unix(7) - linux manual page. http://man7.org/linux/man-pages/man7/unix.7.html. [Online; accessed 4-Jan-2019].

[7] uv_loop_t - event loop - libuv documentation. http://docs.libuv.org/en/v1.x/loop.html. [Online; accessed 7-Jan-2019].

[8] What is a gantt chart? gantt chart software, information, and history. https://www.gantt.com/. [Online; accessed 7-Jan-2019].

[9] What is rest – learn to create timeless restful apis. https://restfulapi.net/. [Online; accessed 7-Jan-2019].

[10] D. Crockford. Introducing json. `https://www.json.org/`. [Online; accessed 16-Dec-2018].

[11] N. Khademi. NEAT Deliverable 2.3, Sept. 2017.

[12] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, et al. Neat: a platform- and protocol-independent internet transport api. *IEEE Communications Magazine*, 55(6):46–54, 2017.

[13] H. Kniberg. *Scrum and XP from the Trenches*. Lulu. com, 2015.

[14] P. Lehtinen. Jansson documentation. `https://jansson.readthedocs.io/en/2.11/`. [Online; accessed 16-Dec-2018].

[15] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, et al. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

A Gantt Chart

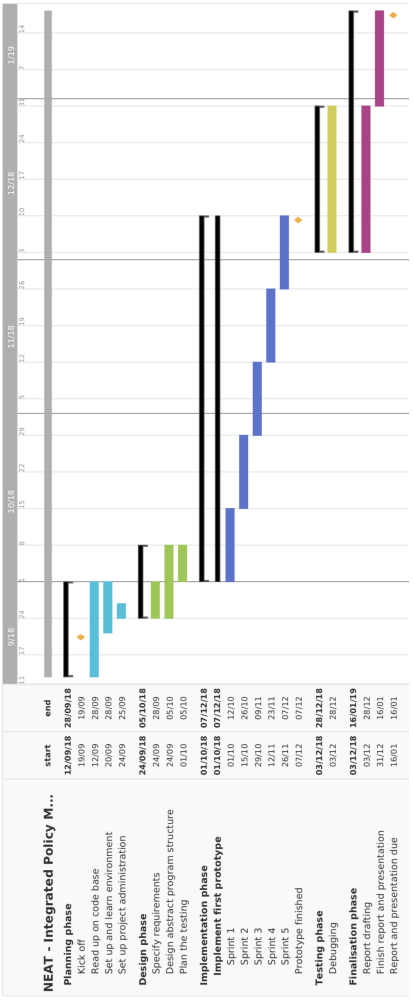

Figure 7: Gantt chart.

B Setup Instructions

Below follows instructions to setup and run the PM. Note that these instructions may not be of the latest version. For an updated set of instructions, you can refer to the official git page [2].

B.1 Install dependencies

```
apt-get install cmake libjansson-dev libuv1-dev
libmicrohttpd-dev libulfius-dev
```

B.2 Setup example policies

To begin, copy your PIB files to the backend folder. To copy the already existing PIB files to backend, go to NEAT folder and run:

```
mkdir -p ~/.neat/infobase/pib/profile
mkdir -p ~/.neat/infobase/pib/policy
cp policy_manager/json_examples/pib/*.profile \
    ~/.neat/infobase/pib/profile
cp policy_manager/json_examples/pib/*.policy \
    ~/.neat/infobase/pib/policy
```

B.3 Build the PM

To **build** the PM, go into the NEAT folder then run:

```
cd policy_manager
mkdir build
cd build
cmake ..
make
```

You can either run the PM manually or as a process, or in the background as a daemon.

B.4 Run as process

To run as a **process**, simply run the following command:

```
./pm
```

Possible **arguments** to PM:

```
-debug  (Print debug messages in the console)
-log    (Write all log messages to file: Log.txt)
```

B.5 Run as daemon

To run the PM as a **daemon**, first it needs to be installed:

```
sudo make install
```

This will install the PM in `/usr/local/bin` and create a new systemd service **neat_pm.service**. To start the PM and enable the PM to start at system boot:

```
sudo systemctl start neat_pm
sudo systemctl enable neat_pm
```

C Unit Test Instructions

Below follows instructions to setup and run the test suite for the PM. Note that these instructions may not be of the latest version. For an updated set of instructions, you can refer to the official git [2].

C.1 Install Unity

The framework used to execute the unit tests is Unity [5]. **Download** Unity by entering the NEAT folder and running:

```
cd ..
git clone https://github.com/ThrowTheSwitch/Unity
```

C.2 Setup

To **build and run** the unit tests to the PM, go into the NEAT folder and run:

```
cd policy_manager/unit_testing
mkdir build
cd build
cmake ..
make
./test_pm
```