A NEAT Way to do Network Programming

Karl-Johan Grinnemo
Anna Brunstrom
Karlstad University

Zdravko Bozakov
Per Hurtig
Naeem Khademi
Dell EMC Research Europe
Karlstad University
University of Oslo

ABSTRACT

There is a growing concern that the Internet transport layer has become ossified in the face of emerging novel applications, and that further evolution has become very difficult. The NEAT system introduces a novel and evolvable transport API that decouples applications from the underlying transport layer and network services. This paper provides an overview of the NEAT system from a programmer’s perspective and exemplifies its use through a simple client application.

KEYWORDS

NEAT, API, event-based, transport service, transport selection

1 INTRODUCTION

The Internet is often seen as having a common network layer and two widely deployed transport protocols, TCP [4] and UDP [3], with other transports, such as SCTP [5], struggling to find broad deployment. The NEAT system [1] challenges the view of an ossified transport layer by providing an API that is oblivious to specific transport protocols and instead focuses on requested transport services. Applications provide the NEAT system with traffic requirements, pre-specified policies, and measured network conditions. Based on this information, NEAT establishes and configures appropriate connections. Through its design, NEAT enables new network and transport functions and protocols to be added incrementally and transparently.

Figure 1 illustrates the NEAT system. Applications access NEAT via a user API. The API offers transport services similar to those offered by the socket API, but in an event-driven fashion. The API interfaces the NEAT User Module, which constitutes the main part of the system. The NEAT User Module is designed to be portable across different operating systems and network stacks. One of its primary responsibilities is to select the most appropriate transport solution for a requested transport service. At the core of this selection process is the Policy Manager (PM): The PM combines application requirements with available transport protocols, transport-protocol parameters, and feasible transport endpoints, i.e., IP addresses and port numbers. The PM uses this information to create a list of candidate transport solutions, sorted in order of appropriateness, to use for a requested transport service.

The PM obtains information from two sources: the Policy Information Base (PIB) and the Characteristics Information Base (CIB). The PIB is a repository that contains a collection of policies, where each policy consists of a set of rules linking a set of matching requirements to a set of preferred or mandatory transport characteristics. In contrast, the CIB is a repository storing information about available interfaces, supported protocols towards accessed destination endpoints, network properties and current/previous connections between endpoints.

This paper gives a “programmer’s view” of the NEAT system. Section 2 introduces the NEAT User API through a client application example. The paper concludes in Section 3 with a brief summary and some final remarks on ongoing and future work on the NEAT system.

2 THE NEAT API

The NEAT API is a callback-based, asynchronous, and non-blocking network API that is intended to complement and replace existing network APIs by being simpler and more flexible. It is implemented using libuv [2], which provides asynchronous I/O across multiple platforms.

The most important concept in the NEAT API is that of a flow. A flow is similar to a socket connection in the BSD API in that it provides a bidirectional link between two applications. Also, in the same way as a socket connection, a flow is tied to a transport-layer protocol. However, in contrast to a socket, the tie between a flow and a transport protocol is fairly loose. In fact, a flow may be created even without specifying a transport protocol. A flow in NEAT always belongs to a context. The context serves as a common environment for multiple related flows within an application, and comprises one libuv event loop and one DNS-resolution context.

Table 1 lists the major NEAT API functions. The neat_init_ctx function initializes a context, and neat_new_flow creates a new flow within a specified context. An application creates a flow for each connection it wants to open, and sets the requirements imposed on that flow through the function neat_set_property. It closes a flow by calling the function neat_close, and frees the context by invoking neat_free_ctx. The neat_open and neat_accept
### Table 1: The primary NEAT API functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>neat_init_ctx</td>
<td>Creates a new context.</td>
</tr>
<tr>
<td>neat_new_flow</td>
<td>Creates a new flow.</td>
</tr>
<tr>
<td>neat_set_operations</td>
<td>Sets callbacks for a flow.</td>
</tr>
<tr>
<td>neat_set_property</td>
<td>Sets the properties of a flow.</td>
</tr>
<tr>
<td>neat_open</td>
<td>Opens a flow and connects to a transport endpoint.</td>
</tr>
<tr>
<td>neat_accept</td>
<td>Listens to incoming flow requests.</td>
</tr>
<tr>
<td>neat_start_event_loop</td>
<td>Starts the NEAT event loop.</td>
</tr>
<tr>
<td>neat_stop_event_loop</td>
<td>Stops the NEAT event loop.</td>
</tr>
<tr>
<td>neat_write</td>
<td>Writes data to a flow.</td>
</tr>
<tr>
<td>neat_read</td>
<td>Reads data from a flow.</td>
</tr>
<tr>
<td>neat_close</td>
<td>Closes a flow and deallocates associated data.</td>
</tr>
<tr>
<td>neat_free_ctx</td>
<td>Deallocates a context.</td>
</tr>
</tbody>
</table>

functions roughly correspond to the socket API functions connect and listen: The neat_open function creates and opens up a flow towards a specified transport endpoint, which can be either a DNS name or an IP address and port number, and the neat_accept function listens for incoming flow requests on a given port. In the same way, the neat_write and neat_read function calls correspond with the read and write socket function calls.

As mentioned, the NEAT API promotes an event-driven programming style, i.e., an application subscribes to events from the NEAT API. Callbacks in the NEAT API are registered on a per-flow basis. The most important callbacks are on_connected, on_readable and on_writable. The on_connected callback is executed once a flow has connected to a remote endpoint. The on_readable and on_writable callbacks are executed once data may be written to or read from a flow without blocking. Other important callbacks are on_all_written and on_error. on_all_written is invoked when all data sent by a previous call to neat_write has been written out on a flow. This callback is often used by applications to change state, e.g., a client application might go from sending a request to a server to listen for the reply. The on_error callback, as the name suggests, is called whenever an error occurs during the processing of a flow.

Listing 1 provides a code excerpt of a client implemented in NEAT. In the main function, the program starts by initializing a context and a flow within that context. Next, flow requirements or desired flow properties, expressed in JSON syntax, are registered. Recall from Section 1 that NEAT combines desired flow properties, policies, and information stored in the CIB when selecting an appropriate transport solution. In this example, our client tells NEAT that it wants to establish a flow that is capable of low-latency communication. NEAT will try to satisfy this by having the PM compose a list of transport configurations, able to provide low-latency communication.

Following the flow properties, callbacks are registered. In this particular case, only one callback is registered, on_connected, which is called as soon as neat_open has initiated a flow between itself and a local server listening on port 5000. Note that the registration of callbacks is not something done once and then forgotten, but something that is continuously revised during the progress of the program. In our example, this is shown in the on_connected callback, in which callbacks for on_writable and on_all_written are registered.

```c
#include <neat.h>

static char *properties = 
"{"low_latency": true, "precedence": 1});

static neat_error_code
on_connected(struct neat_flow_operations *ops) {
    ops->on_writable = on_writable;
    ops->on_all_written = on_all_written;
    neat_set_operations(ops->ctx, ops->flow, ops);
    return NEAT_OK;
}

int main(int argc, char *argv[]) {
    ... 
    ctx = neat_init_ctx();
    flow = neat_new_flow(ctx);
    neat_set_property(ctx, flow, properties);
    memset(&ops, 0, sizeof(ops));
    ops.on_connected = on_connected;
    neat_set_operations(ops->ctx, ops->flow, &ops);
    if (neat_open(ctx, flow, "127.0.0.1", 5000, NULL, &ops)) {
        fprintf(stderr, "neat_open failed\n");
        return EXIT_FAILURE;
    }
    neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
    neat_free_ctx(ctx);
    return EXIT_SUCCESS;
}
```

Once callbacks have been registered and a flow request has been issued, the NEAT event loop is initiated by calling neat_start_event_loop; the NEAT_RUN_DEFAULT parameter in the neat_start_event_loop function call tells NEAT to run in the event loop as long as there are open flows and/or outstanding events. Our client program ends by freeing its context and the adhering flows, both of which are taken care of by the neat_free_ctx function call.

### 3 CONCLUSIONS

The NEAT system offers an simple-to-use yet powerful API for network applications that decouples them from the actual transport protocol being used. It differs from most other network APIs in that it explicitly enables applications to communicate their service requirements to the transport system in a generic, transport-protocol independent way. This paper offers a "programmer’s perspective" on the NEAT and in so doing gives a brief overview over the NEAT User API and demonstrates its use through a simple client application. The NEAT system is at the time of writing a system in flux. Ongoing work on features such as multi-path scheduling, less-than-best effort and coupled congestion control, and integration of NEAT with SDN drives a continuous extension of the capabilities of the NEAT User API.
ACKNOWLEDGMENTS
This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

REFERENCES