Lessons Learned From Implementing a Path Computation Element (PCE) Emulator

Mohit Chamania, Marek Drogon and Admela Jukan

Technische Universität Carolo-Wilhelmina zu Braunschweig, Hans-Sommer-Str. 66, 38106 Braunschweig

chamania, drogon, jukan@ida.ing.tu-bs.de

Abstract: We developed the first open source Path Computation Element (PCE) emulator, which includes a complete PCEP protocol implementation, network I/O support and support for concurrent path computations. We share the lessons learned in its implementation.

OCIS codes: 060.4250,060.4250.

1. Introduction

The upcoming QoS sensitive services such as IPTV, VoD, etc, are making path computation a critical factor in the service provisioning. To this end, the PCE (Path Computation Element) is fast emerging as the de-facto solution for constrained path computation in carrier-grade networks [1]. The PCE is a centralized server which uses its Traffic Engineering Database (TED) to compute optimal paths in the network. In order to further develop the PCE architecture for practical deployment in carrier networks, it is imperative to understand the limitations of the PCE in terms of it's capacity to serve users requests for path information. While analytical frameworks to model the PCE [3] can help providers compute the number of PCEs required to support the network, there is currently no evaluation on the performance of the PCE. Given the diversity in deployment scenarios and complexity of path computation itself, experimental validation of a PCE is essential before deploying them in real networks.

We developed the an open-source PCE emulator in Java to facilitate experimental validation. The emulator includes a complete PCEP protocol implementation [2], network I/O support and support for concurrent path computations using different path computation algorithms. It uses a modular architecture, and each module can be changed independently without affecting the overall operation. The modules break operations into basic functions of the PCE required by each PCE implementation and each independent PCE implementation is then optimized according to the overall architectural requirements. The separation of basic functions also makes it easer to integrating new extensions such as PCEP protocol extensions, new path computation algorithms, security etc, without significant changes to the PCE emulator. In this paper, we present the software architecture of our PCE emulator and share lessons learned during the implementation along with some preliminary performance figures.

2. Architecture

The emulator was designed to adhere to the PCE architecture and PCEP protocol specifications as defined in [1] and [2] respectively. The PCE architecture consists of four major functions, i.e.,: the PCEP protocol definition, network I/O, session management, and path computation. In our architecture, each of these functions is implemented as a separate module as shown in Fig. 1, with the corresponding modules named PCE Protocol, Network Layer, Session Handler and Computation Layer. Fig. 1 indicates the interactions between the different modules in a typical PCEP session. First, an incoming connection request from a remote PCEP peer is received by the Network Layer which initializes a new state machine in the Session Handler initiating the initial PCEP exchange. After handshaking, the remote peer sends a PCEP Path Computation Request, which is forwarded by the Network Layer to the Session Handler. The Session Handler extracts the relevant path computation request parameters and forwards it to the Computation layer, which attempts to compute a path. The corresponding path computation response is forwarded by the Computation layer to the Session Handler, which encapsulates the computation response into a PCEP Path Computation Response message and sends it to the remote peer over the Network Layer. We now describe each of the modules in more detail.

PCE Protocol: The PCE Protocol package is responsible for converting incoming PCEP messages received over the network into usable Java objects, and vice versa. Fig. 2(b) shows the basic structure of the PCEP message, including a header and a body which consists of one or more PCEP objects. Each PCEP object also has a header and a body

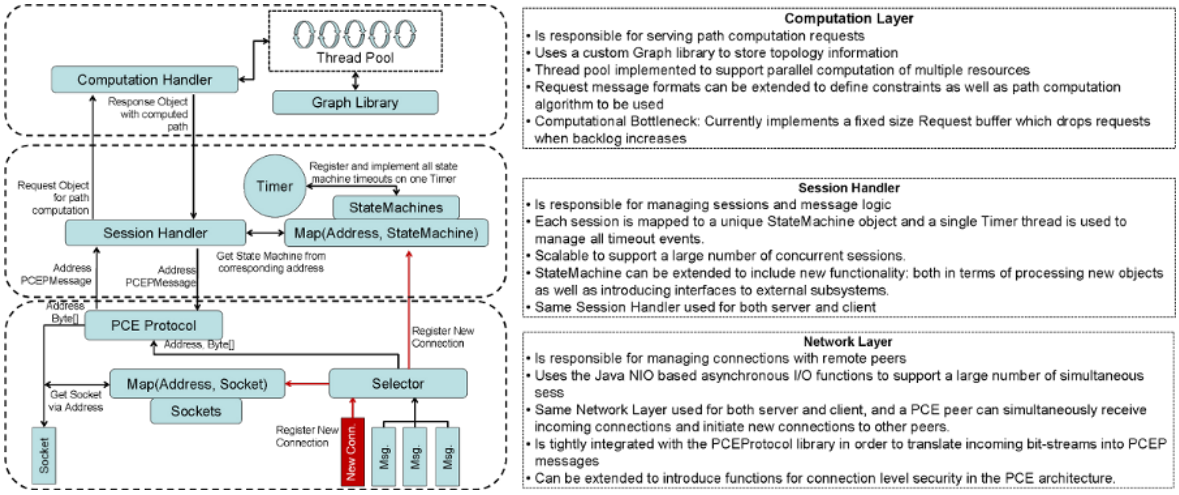
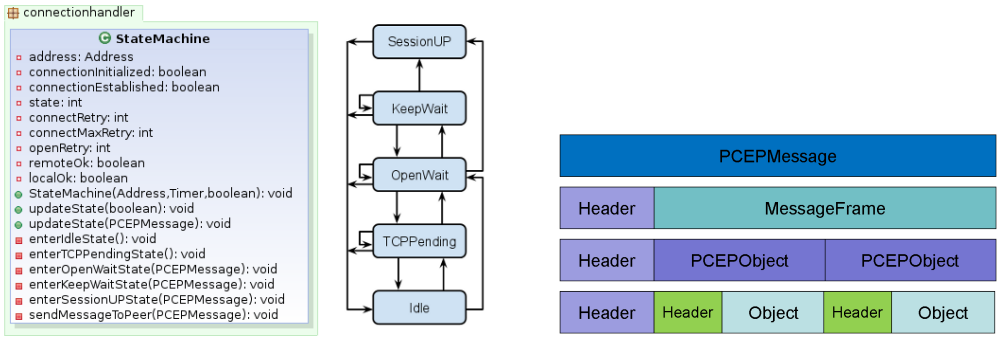


Fig. 1. The PCE Emulator Architecture

which can consist of values or other PCEP objects. The PCE Protocol package mimics the protocol structure, where all messages are defined by a PCEPMessage object, which consists of a MessageHeader object and an PCEPMessage-Frame interface implementation. Similar implementations are also used for the PCEPObject implementations. Each unique message type corresponds to a PCEPMessageFrame interface implementation which define rules for checking compulsory objects as defined in the PCEP specifications. For instance, the PCE Open Message frame implementation ensures that the message frame must contain the PCEP OPEN object. Implementing static checks not only allows the emulator to ensure conformity to the PCEP protocol, but can also be easily used to incorporate additional security functions, and conformity for PCEP extensions. We also ensure that all PCEPMessage objects are generated by a *Factory*, using either a bit stream (received from the network) or using PCEPObjects to construct a new message. The factory provides a single decision unit to incorporate new PCEP messages and logic for deciding how a PCEP message should be structured which is especially useful when incorporating protocol implementations.



(a) State Machine Implementation

(b) Message Format

Fig. 2. Implementation Details of (a) PCEP State Machine and (b) PCEP Message structure

Network Layer: The Network layer is required to communicate with remote PCEP peers and using the PCEP Protocol implementation, convert incoming bit-streams into corresponding Java objects before passing them to the Session Handler. Based on the current PCEP specifications, a remote peer can have only one active session with a PCE, and we therefore use the remote IP address as a unique identifier for synchronization between the network layer and the session handler. It was also observed that the time for path computation can be significant, and as a result of which, while I/O operations during a session may be very small, the session length itself can be very large. We

therefore use the asynchronous network I/O implementation using the Java NIO, which uses only one *selector* process where all connections are registered, instead of dedicated processes listening for data on all connections. The *selector* listens on all connection sockets and generates a trigger when data is available to be read on one or more sockets making the network layer lightweight while keeping latency low.

Session Handler: The Session handler contains the PCEP message processing logic and the PCEP state machine. Each connection is associated with a single StateMachine object (Fig. 2(a)) which implements the logic for the state machine transitions and for further processing of PCEP messages. As stated before, remote IP addresses are used as unique identifiers and a single state machine is mapped to its corresponding IP address. Again, due to the typically long session length and the low latency achievable for processing inside the Session Handler, we do not use individual processes for each state machine. As all **transitions** in the state machine are triggered by an incoming message (from the Network or the Computation Layer) or by a timeout, in this implementation, we use a single Timer thread instantiated by the Session Handler to handle all timeout events.

Extensions to the PCEP typically require additional logic to deal with incoming PCEP messages, but do not significantly disturb the operation of the state machine itself. Therefore, to provide easy extensibility, the StateMachine object provides a separate function to define additional processing functions beyond typical state machine transitions, to allow for ease of extensibility.

Computation Service: The Computation Layer is responsible for facilitating path computation inside the PCE. Seen in Fig. 1, the Computation Layer consists of a Computation Handler responsible for communication with the Session Handler and queuing path computation requests, a thread pool implementation to simultaneously serve multiple path computation requests, and a topology representation which is currently implemented using an open source data-structure library [4]. The queue implementation can be modified to incorporate priority queuing functions in case of excessive path computation load and can be extended so send PCEP Notifications to indicate overloading to remote peers. It was observed that using a large number of concurrent operations when accessing the TED (in this case the graph library) to perform path computation leads to excessive blocking of memory access requests leading to large path computation times as seen in Table 1. We also observed these issue in presence of very frequent TED updates. It is therefore necessary to design the thread-pool size and the design of the TED itself, to meet requirements on path computation latency and TED update frequency.

Computation Layer Configuration (Topology: 65 nodes, 216 links)	**Minimum**	**Maximum**	**Average**	**Standard Deviation**
Thread Pool Size = 1	0.005 ms	1.285 ms	0.110 ms	0.097 ms
Thread Pool Size = 5	0.009 ms	8.197 ms	0.440 ms	0.817 ms
Thread Pool Size = 10	0.006 ms	11.170 ms	0.859 ms	1.739 ms

Table 1. Average Request Computation Times for different sizes of Thread Pool, averaged over 10,000 measurements on a Dell Optiplex 755 Desktop PC running Ubuntu

3. Conclusion

We developed a new, fully-functional PCE Emulator in Java in accordance with the PCE architecture specifications. The modular architecture allows flexible modifications of different components without requiring significant changes in the rest of the emulator. During our implementation, we found that a few critical challenges have to be overcome, including, co-ordination of updates on TED with path computation requests, process separation and mapping between the different modules and load management in the computation layer. Our preliminary results indicate that performance of PCE can be significantly affected by TED implementations, especially when dealing with frequent read/write operations and efficient synchronization mechanisms should be designed to minimize latency for the same.

References

1. A. Farrel, et al., *"A Path Computation Element-Based Architecture"*, IETF RFC 4655, August 2006
2. JP. Vasseur, J.L. Le Roux,*"Path Computation Element Communication Protocol"*,IETF RFC 5440, March 2009
3. J. Yu, et. al. *"A Queueing Model Framework for PCE"*, Tech. Rep. UTD-EE-13-2008, UTD, Dallas, Sept. 2008.
4. Data Structure Library in Java, *http://www.cs.brown.edu/cgc/jdsl/*