An Open-Source Path Computation Element (PCE) Emulator: Design, Implementation and Performance

Mohit Chamania, Marek Drogon and Admela Jukan
Technische Universität Carolo-Wilhelmina zu Braunschweig
Email: {chamania, drogon, jukan}@ida.ing.tu-bs.de

***Abstract*—In this paper, we present the first *open-source* Path Computation Element (PCE) emulator along with its key design and implementation features. The PCE is a network control and management entity which can be utilized to perform optimal path computations with multiple constraints in carrier-grade transport networks. The presented architecture incorporates all elements of the standardized PCE framework and is scalable in the number of requests and size of topologies served as well as path computation algorithm complexity. Given the diversity in current control and management practices of carrier-grade transport networks, we also identify key features that are necessary for innovation within the PCE framework, including flexible topology description and update mechanisms, extensible protocol and state machine definitions and fully programmable path computation. We incorporate all these features in our design and implementation. This works bridges an important gap between network engineering, software system design and algorithmic studies and shows that deplying a PCE system as such is not only feasible but also well performing in a range of network scenarios from IP/MPLS to WDM networks.**

I. Introduction

The emergence of dynamic connection-oriented services has led to need for increased intelligence to support on-demand service provisioning. With an increased number of location served, edge devices accessed and service attributes, new network architectures are enabling edge systems to directly control their network resources through specialized signaling. These new capabilities can be accessed through policy-based processes by independent organizations, individuals and even applications. Among many third-party control and management subsystems which have been proposed, the Path Computation Element (PCE) framework and its latest extensions stands out as the de-facto standard for constrained path computation [1]. The PCE is a third-party network control and management entity that uses its Traffic Engineering Database (TED) to compute optimal paths with QoS constraints. The PCE serves path computation requests sent by a client using the PCE protocol (PCEP) [2] and returns information about the computed path which is then used to provision connections. The ability to perform constrained path computation makes the PCE especially attractive to network operators who typically employ multiple technologies in a layered fashion, such as the IP/MPLS network over a carrier Ethernet network which is deployed over a WDM network.

In theory, the PCE architecture provides network operators a simple and scalable solution to perform path computation across various network technologies and layers. However, commercial PCE implementations are vendor-specific and tailor made for deployment in select network technologies, making it difficult to extend the same PCE implementation across various networking technologies. For example, the PCE implementations inside WDM transport equipments such as [3],[4] are often designed to address the specific path computation needs in the WDM layer such as physical impairments and are thus not adequate for path computation in other networks. Vendor-specific PCEs also differ significantly in the way topology is stored and updated which also greatly limits the ability to integrate PCE functionality from different layers inside a single PCE. As an example, whilst the Telemanagement Forum (TMF) proposes the use of the Multi-Technology Network Management (MTNM) topology specifications [5] inside carrier-grade transport network NMSs (e.g. WDM, carrier Ethernet), most IP NMSs use custom XML-based topology models to represent IP network topology, which are significantly different implementations and difficult to integrate. Most importantly, however, proprietary PCE implementations limit the network architect's capability to innovate with the PCE itself, especially with core features such as the PCE protocol and state machine, thus further limiting the operators capability to experiment as networks evolve to encompass a growing number of users and service demands.

To address this challenge, we designed and implemented *the first open-source PCE* emulator which is targeted to facilitate innovation in path computations [6]. An open-source PCE implementation is an important step towards openness and programmability of future networks as it allows software developers, operators and algorithm designers to flexibly adapt the implementation of various PCE procedures to different network technologies. At the same time, by implementing the session management, PCEP protocol and path computation (including topology update mechanisms) as separate functions, the proposed architecture is highly modular and ensures minimal effort for integration. Our implementation features the complete PCEP protocol stack implementation, asynchronous network I/O features for scalable concurrent session support and extensible state machines, which provide the basic protocol state machines, and can be extended with ease to integrate new operations. We show that the integration and modification of path computation and topology updates are not only feasible, but they largely reduce duplication of programming effort, when introducing new protocol or session

management features, such as security.

We present a performance analysis of the PCE implementation and show that it is scalable in terms of topology size, path computation complexity and connection request rate. We also analyze the effect on performance due to other often neglected design parameters such as number of concurrent path computations and topology update frequency, which can be used as guideline by implementers to *adapt* the PCE for optimal performance in specific network scenarios. Finally, we present a WDM case study to demonstrate the ease of integration of specialized computation algorithms into our PCE implementation to support (specialized) *lighpath computations*. This works bridges an important gap between network engineering, software system design and algorithmic studies and shows that deploying a PCE system is feasible and widely applicable in carrier-grade networks.

The rest of the paper is organized as follows: Section II presents a background of the PCE framework and highlights the need for an open source PCE. The system architecture is presented in Section III, while Section IV provides insights into the implementation details. Section V presents the performance study, and Section VI concludes the paper.

II. Background

The basic PCE architecture is shown in Fig. 1. In a nutshell, the PCE is a server which can perform constrained path computation using topology information stored in its Traffic Engineering Database (TED). The TED contains information of the network topology including QoS parameters such as available capacity, link delay, etc. and is typically updated using the network control or management planes. The use of a TED allows the PCE to compute optimal constrained paths which is especially useful in provisioning services in transport networks, such as WDM, which have strict QoS requirements.

In this architecture, network nodes are equipped with a Path Computation Client (PCC) which can communicate with the PCE using the PCE Protocol [2]. Using this protocol, the source node *R1* requests the PCE to compute a path to *R6* under necessary service constraints. The PCE, using the TED, computes the path and sends it to the PCC at *R1*. The computed path is stored inside an Extended Route Object (ERO) which can be then used at *R1* to initiate provisioning in a control plane (e.g. GMPLS [7]) by including the ERO into the Resource Reservation Protocol (RSVP) Path message [8]. Further extensions to the PCE allow a PCE to be a client to another PCE in order to extend the reach of the optimal path computation to the multi-layer [9] and multi-domain [10] scenarios. For example, in an inter-domain scenario, the PCE in an upstream domain acts as a client to a downstream domain PCE, requesting an optimal path to the destination and this process is repeated along a domain chain to compute the inter-domain path [10].

The PCE architecture can also be used in fully managed networks where operations such as connection provisioning are governed by centralized Network Management Systems (NMS). In this scenario, the NMS can request path information from the PCE before beginning provisioning of the connection (either via the control plane or via direct configuration of network elements). This is advantageous as typical NMSs are large proprietary software systems and simple changes such as inclusion of new path computation algorithms in the NMS can be very complex and expensive when compared to inclusion of the same in the PCE.

An important issue in deployment of the PCE are the mechanisms for management and update of the traffic engineering database, which is critical to the PCE for facilitating optimal computation. While TED update mechanisms are not part of the PCE specifications themselves, the standards propose the use of routing updates in the control plane to maintain and update the TED, while custom implementations such as [3], [4] also use the topology information stored in the NMS to update the TED. TED update frequency is also crucial as a TED update can interrupt active path computation operations inside the PCE leading to reduced performance. As a result, implementations must have the capability to integrate and support different TED update mechanisms to be ubiquitous and future-proof.

A. PCE Protocol (PCEP)

The PCE protocol (PCEP) is an integral part of the PCE which enables communication between two PCE peers. The protocol specifications in [2] define seven unique message types: The *Open* and *Close* messages are used to initialize and close the connection, the path computation requests are sent in a *Path Computation Request* message to which the response is sent in the *Path Computation Response Message* and the *Keepalive*, *Notification* and *Error* are used in order to convey additional information to remote peers.

A typical PCEP message format consists of a standard header and body and information inside the message body is encapsulated in the form of *PCEP objects* which use Type-Length-Value (TLV) type representations. Each message contains some mandatory objects and some optional objects based on the type of information exchange required: for example, a *Keepalive* message is used only to check if a remote peer is still active and does not have any mandatory objects while a complex message such as the *Path Computation Request* message must contain an *RequestParameters* object containing the Request ID and an *EndPoints* object which defines the endpoints for a path computation request and can additionally contain objects such as the *Bandwidth* and *Metric* objects which provide constraints on the path computation request.

Extensions to the PCE architecture almost always require updates to the basic PCE protocol specification defined in [2]. While the initial set of protocols are targeted to facilitate basic path computation, the PCE working group [11] is also focusing on extending the PCE architecture and protocol specifications to address administrative issues such as policy integration [12] and monitoring [13], technical issues such as point-to-multipoint path computation [14], [15], synchronized dependent path computation [16] as well as extensions to support

highly specialized networks such as Wavelength Switched Optical Networks (WSON) [17].

The PCEP specifications also propose mechanisms such as [18], [19] to secure the communication channel between PCE peers and other proposals such as [20] have also proposed security extensions to the PCEP to exchange authorization tokens for facilitating authentication and authorization between inter-domain PCEs to secure service provisioning.

Given the large body of PCE protocol extensions that are currently under standardization, it is imperative for any PCE implementation to support an extensible PCE protocol implementation to provide researchers and implementers a platform to design and evaluate these extensions in an emulated environment.

B. Importance of an Open-source Approach

Most current implementations of the PCE framework are vendor-dependent, making integration between PCEs in a multi-vendor environment difficult. While some implementations such as the [21] are independent of the vendor, the lack of openness limits the ability of users/operators to extend the PCE in order to implement new functionalities with ease. In a different approach, the path computation mechanism in the OSCARS-IDC [22] provide a web-services Application Programmer Interface (API) which can be used by developers to incorporate additional path computation algorithms into the PCE. However, the aforementioned mechanism does not conform with the PCE standards and can only be used in a network supporting the OSCARS framework.

Given the extensive research currently being carried out to extend the PCE architecture and the number of choices available for implementing features of path computation, security and topology update among others, it is necessary to have an open platform which supports easy integration of new features into the standard framework, providing researchers and implementers a platform to test their proposals in an emulated environment. Based on our work presented in [6], here we give all the details regarding the architecture, implementation and tested performance of this first open-source implementation of a PCE framework, which, as we show, can address the aforementioned drawbacks.

III. Architecture

In this section, we describe the architecture of our PCE emulator, designed to conform to the requirements of the PCE operation and the protocol as specified in [1] and [2], respectively. The presented architecture was created on the following guiding principles:

- **Flexibility through Modularity:** In our implementation, we consider the need for updating or even replacing parts of the PCE implementation without affecting the other components of the PCE. To facilitate the same, critical functions of the PCE are implemented as separate modules which can be modified independently.
- **Extensibility through Loosely Coupled Modules and Flexible Message Interface:** Our design considers extensibility as a major objective, which in our view incorporates integration of new possible modules and protocols extensions into the architecture with ease. To this end, we ensure that the inter-modular coupling in the architecture is not tightly constrained. By doing so, new modules (e.g., for policy management and security) can easily be integrated into the proposed architecture without affecting communication between the other modules.
- **Adaptability through Module-internal Processing Optimizations:** The performance requirements of different modules inside the PCE depend significantly on the deployment scenario; for instance, in optical networks, we expect the PCE to serve a relatively small number of computation requests which require execution of fairly complex path computation algorithms due to physical impairments. On the other hand, in an MPLS network, a PCE would serve a comparably larger number of requests with relatively lower path computation complexity. Our architecture therefore facilitates optimization of each individual module in the architecture to better suit specific characteristics of the network in use.

A. Overview

In designing the architecture, we identified components which are likely to be *specialized* in different commercial PCE implementations and have designed our PCE as to allow flexible modification of these functions. Each of the functions identified is designed as a *module* with fixed interfaces based on the encapsulation. At the same time, we also ensured that the introduction of modules incorporates minimal overheads into the implementation to ensure scalability of the PCE implementation. The proposed module encapsulation and the PCE architecture with the interaction between the different modules in the client and server are shown in Figs. 2 and 3 respectively. The introduction of distinct modules with fixed interfaces allows developers to individually change or optimize the operation of modules without disrupting the overall operation of the PCE. As seen in Fig. 3, both the server and the client have three primary operation modules, with the *module management* function responsible for initializing/stopping/replacing a specific module during run-time. As the names suggest, the network module is responsible for facilitating communication between the PCE peers over the TCP/IP protocol and the client module provides the user an interface to communicate with the PCE server.

The Session Management module is responsible for management of all PCEP sessions in a PCE peer. While all other modules maintain the *session ID* for active sessions, the session management module implements the *state machine* for the PCE protocol which *orchestrates* internal operation inside a peer as well as message exchange with other peers.

Finally, the computation module is responsible for facilitating path computation. Note here that we have not made a separate TED module as it is assumed to be integrated inside the computation module. In the computation module, functions related to path computation such as choice of computation

algorithms, mechanisms for prioritizing different types of requests and handling topology updates in the TED can be adjusted or modified without any change in the rest of the implementation. In order to support extensibility, the architecture does not restrict the number of modules used inside the architecture nor does it restrict duplication of modules. However, the inter-module messaging interfaces should be adapted to the specific implementation in these scenarios.

As shown in Fig. 2, each module implements standard interfaces including the internal messaging interfaces and inter-module communication interfaces, which is presented in the next subsection. We then go on to describe the working of individual modules as well as the PCE protocol mapping in our architecture.

1) Module Interfaces: Each module implements six interfaces to facilitate communication between modules and support basic management functions. Messages exchanged between the modules are identified by a unique session ID which is used to correlate a message to a particular session. In order to support flexibility inside the PCE architecture, the modules as well as the communication system between them are loosely coupled. This not only facilitates integration of intermediate modules in the architecture to incorporate specific functionality but also allows the implementer to change the mechanism used for communication between the modules.

2) Operation State Management: Each module in the PCE may be running multiple process threads and to support graceful start, stop, or restart, the module encapsulation provides two interfaces for initializing (start) and terminating (stop) all processes inside a module. Implementation of these functionns inside each module supports graceful initiation and termination of operations inside each module.

3) Session State Management: The module encapsulation provides two interfaces for individual session management, namely, register and close. These interfaces are responsible for managing the initiation and the termination of individual PCEP sessions. Note that the initial trigger for initiation or termination of a session can come from any module, and to avoid inconsistency, the implementer must ensure that a session is registered in all modules before session specific inter-module messaging is initiated.

4) Inter-Module Message Exchange: The send and receive interfaces in the module encapsulation are used to facilitate sessions specific information exchange between modules. A flexible message object, identified by a unique session ID, is used for inter-module communication. Note that the message definition used here does not directly use the PCEP messages and instead PCEP messages are encapsulated inside the message definition as individual objects. Therefore, a change in the PCEP message library does not necessarily signify large changes in the inter-module communication itself. The message interface is flexible so as to facilitate exchange of PCEP messages as well as control data for individual sessions. For example, in a client, the message interface may be used to inform the client module of the initiation of the SESSION UP state, after which the client module can send a request for path computation to the server. The send/receive functions also employ identifiers to remote modules used to define the destination/source of the message respectively. The use of this identifier enables use of a single interface pair for communication with different modules (example session to network and session to computation), and provides a single point for policy integration into inter-module communication. The feature also supports implementation of different mechanisms for communication with different modules. A possible exploitation of this design feature is shown in Fig. 4, where we attempt to distribute path computation across multiple physical computation nodes. Each node implements a computation module independently and exposes the module interfaces over web-services. The PCE session management module also implements web-service calls instead of direct function calls to communicate with the different computation modules, while direct function calls are used for communication with the network module which is at the same physical node. The web-services can then be inter-connected over an Enterprise Service Bus which routes the request from the session management to one of the computation nodes and vice versa in order to facilitate load balancing across multiple physical nodes. We also demonstrate an example of this exploitation in our results where we modify the module interfaces in the session management module to support two path computation modules on the same node simultaneously.

B. Object-Oriented PCE Protocol Mapping

The PCE protocol is a critical component in the PCE architecture as it is likely to be frequently updated to provide additional functionality in the PCE architecture or address protocol extensions. In our architecture, the TLV like objects of the PCE protocol are mapped to an object-oriented class hierarchy for internal usage. Therefore, PCEP messages are represented as objects inside the different modules making it easier to define internal logic based on them. The use of an object-oriented hierarchy makes inter-module PCEP message exchange easier when incorporating changes to the protocol, such as inclusion of new PCEP messages or objects. The use of the object-oriented hierarchy also means that all PCEP messages, regardless of message and object types are represented as a PCEPMessage object, thus ensuring that simple changes in the implementation do not lead to changes in all modules.

In the architecture, the PCE protocol is mapped into a simple object-oriented class hierarchy as shown in Fig. 5. Each PCEP message is represented as a single object, which consists of a Header and a MessageFrame. While the header remains unchanged for different message type, a unique PCEP message type is implemented as an independent class which implements the MessageFrame interface. The interface contains functions to add PCEP objects and provides an interface to check the validity of the message where implementers can incorporate conditions to check for necessary objects. For example, for a regular PCEP Path Computation Request, we incorporate checks to ensure that a *RequestParameters* object and an

EndPoints object are included in the MessageFrame. The PCEP objects themselves contain a similar structure which includes a Header object and the Object body.

Finally, all the classes in this hierarchy implement two methods: serialize and deserialize. These methods help interchange a PCEP message between an object instance (used for communication between modules) and a bit-string byte[] (used for communication with remote PCE peers). The hierarchy of the PCE protocol also means that the inclusion of a new object does not affect the serialize and deserialize functions in PCEPMessage if these functions are implemented correctly in the new PCEPObject.

C. Network Module

The network module is responsible for managing communication with remote peers for all sessions inside the PCE. Typically, the network module obtains a PCEP message object to be sent to the remote peer and uses the serialize interface to convert it into a bit-string. This bit-string is then sent over the network to the remote peer, which uses the deserialize interface to convert it back into a PCEP message object.

The network module also supports basic functions of admission control and can request or deny new connections based on policy and current operational state of the PCE. For example, constraints in the protocol which dictate that only one active PCEP session per remote IP address be allowed can be implemented in the network module. Similarly, in cases of extreme load, other modules, e.g., computation module, may instruct the network module to stop accepting more connections so that other internal modules are not overloaded.

In order to design a scalable network module, we take into consideration the rate of session arrivals, data exchanged during the session and average session durations. Together, the choice of implementation as well as the aforementioned characteristics determine the performance of the network module.

D. Session Management Module

Typically used between the network and the client or path computation modules, the session management module implements the primary logic for managing a PCEP session inside the PCE. The session management module implements the state machine for the initial PCEP message exchange [2] (shown in Fig. 6) as well as logic for session and timeout management. When a session is not active, the state machine is in *idle* state and goes into the *TCPPending* state when a TCP connection request is made. After session establishment, the state machine transition sends an Open message to the remote peer (client/server) and goes into the *OpenWait* state, where it waits for an Open message from the remote peer. Upon receiving the Open message, a peer sends a *Keepalive* message to the remote peer and goes into the *KeepWait* state, where it waits for the *Keepalive* message from the remote peer. After receiving the *Keepalive* message, the state machine goes into the *Session Up* state, after which signaling for exchanging path computation requests/responses can be initiated. Note that the initial message exchange and the corresponding state machine implementation remains static in a given system and does not change with the type of path computation request.

The state machine used in the session module dictates the operation of a specific PCEP session. In our architecture the state machine uses nested state machines with the initial state machine handling the protocol handshake, shown in Fig. 6, with the nested state machine responsible for maintaining state for extended operations such as path computation. For example, in order to implement the Backward Recursive Path Computation (BRPC) extension in a PCE [10], the inner state machine handles state changes for implementing the BRPC protocol, while the basic state machine handles the initial PCEP message exchange. The use of nested state machines allows the architect to easily integrate new protocol and state machine extensions into the session management module and manage multiple state machines with duplicating code for facilitating session initialization.

E. Computation Module

The computation module is responsible for processing path computation requests coming to the PCE server. The path computation requests inside the PCEP protocol are identified as point-to-point path requests along with a set of constraints. The computation module contains implementations of algorithms for supporting path computation and mechanisms to execute policy decisions to be imposed on path computation requests.

In our architecture, we do not restrict the choice of TED used by the computation module for path computation and the developer can integrate the computation algorithms with a TED of choice. As a consequence, we also do not inherently support any specific mechanism to handle topology updates during path computation.

The performance of the computation module is highly critical to the operation of the PCE. In specialized networks such as WDM, computation algorithms can be complex requiring large processing resources. In our architecture, developers can address this issue by using more than one computation modules to balance the implementation complexity. In addition, it is not only possible to optimize the processing resources used inside the computation modules but also across all PCE modules, by reducing the processing resources available for use by the network and the session management modules. This will be discussed in more details in the performance results sections, where we present a PCE architecture with two computation modules of use in WDM networks.

IV. Implementation Details and Considerations

In this section, we present the implementation details of the PCE emulator and its modules. The implementation was designed to support a large number of concurrent sessions, which dictated the design choices for the network and the session management modules. For implementation, we used Java to ensure operating system independence and the implementation is compatible with Java compilers (v1.6 or higher).

A. Module Encapsulation and Inter-module Communication

Each module implements an interface specifying the module encapsulation, as shown in Fig. 2. Inter-module communication was facilitated as function calls with the send () function in the source module passing arguments to the receive () function in the remote module. In order to isolate processing tasks between different modules, each module implements a queue inside the receive () function and worker threads inside the module are used to facilitate intra-module processing.

A module management function, shown in Fig. 3, was responsible for starting and stopping all modules in the PCC/PCE. The module management function also provided references to all modules in an implementation, thereby ensuring that individual modules were not required to maintain static references and that modules could be initialized/terminated during the operation of the PCE if so required.

In order to synchronize session state in the different modules, the register () function used two additional parameters apart from the *sessionID* parameter. A boolean parameter *connectionInitialized* was used to indicate if the local PCE peer was responsible for initializing the connection (indicating that it should wait for TCP connect to finish) while the boolean *connectionEstablished* was used to indicate that a connection setup was successful. Using these parameters, we ensured that the session was registered in the different modules before initializing the operations inside the session state machine or the read/write operations in the network module.

B. PCE Protocol

The PCE protocol package developed is responsible for converting incoming PCEP messages into Java objects, and vice versa. The protocol package uses an object-oriented inheritance hierarchy, as illustrated in Fig. 5. All PCEP messages are instantiated as a PCEPMessage object, which contains a MessageHeader and a an PCEPMessageFrame interface implementation. The content of the different message types in the PCEP specification are defined as an implementation of the PCEPMessageFrame interface and the implementation contains a list of objects implementing the PCEPObject interface. The PCEPObject itself contains an ObjectHeader and body content, which might include additional objects. The PCEPMessageFrame implements an interface which checks if the necessary objects are available inside the message body.

We use the *factory* method pattern for generating PCEPMessage objects from the incoming bit-strings and from inside the different modules. The factory provides a single decision unit to incorporate new PCEP messages and logic to determine the structuring and parsing of PCEP messages. A single factory method also helps eliminate complexity of creating PCEPMessage objects from the rest of the modules.

C. Network Module

The network module was designed to support asynchronous I/O for both the PCE client and server implementations. The module supports initiation of connections and accepts incoming connection requests from remote peers. The high level architecture of its implementation is shown in Fig. 7.

The network module is implemented using the Java Network I/O (NIO) library [23] which supports asynchronous network I/O with support for simultaneous read/write operations. In the Java NIO framework, all active sessions are registered with a selector process and the selector waits for events on these sessions. In our architecture, the selector waits for incoming connection requests and incoming data (read) requests.

A connection socket is registered with a) selector for reading incoming data, and b) Map structure which stores the association between a Session ID and the socket inside the network module. The data is received as a bit-string (byte[]) over the network and the PCEP Protocol package is used to generate a PCEPMessage object which is then sent up to the session management module. We also implement functions to identify if an incoming bit-string is too short or too long for a PCEP message. This is required as in asynchronous I/O, as it is possible that a read operation on the socket may recover an incomplete message or multiple concatenated messages.

Unlike the read operations, the write operations directly write onto a Java socket. As mentioned before, the receive () interface for the network module implements a queue and a worker thread reads elements in this queue. The worker converts the incoming PCEPMessage object into a bit-string, identifies the socket associated with a session ID using the Map structure and writes the bit-string onto this socket to send the message to the remote peer.

In the current implementation, we use a single selector thread with one worker thread to perform actions on the selector while one worker thread is used to send messages coming from the session management module over the network. The performance can be improved by implementing multiple workers for processing but care must be taken to ensure that a) multiple threads do not perform a simultaneous read (or write) operation on a session, and b) that the ordering of messages (incoming or outgoing) is preserved.

D. Session Management Module

As mentioned before, the session management module is responsible for managing the state machines for all connections inside the PCE. In our implementation, shown in Fig. 8, we observe small processing time and frequency required for each state transition during a session and therefore implement the state machine as an object instead of an active process. The StateMachine objects are stored in a Map structure against their session ID and are initialized or removed by the register and close interfaces, respectively. Transitions in the state machine are triggered either by a worker thread processing incoming messages, or by a timeout. The Session Management module uses a fixed number of worker threads and a consistent many-to-one mapping described in the previous section to assign messages to a worker which ensures that messages to a single state machine are always processed in sequence.

The incoming message information contains the session ID, the PCEP message and the identification of the remote module

sending the message. State machine transitions triggered by this message may lead to multiple exchanges which other modules (or remote peers) in the PCE. Timeouts inside the state machine are implemented as TimerTasks on a single Timer process. Regular transitions in the state machine are responsible for resetting the existing TimerTasks, while in case of a timeout, the timer process can initiate specific state machine transitions to deal with timeout events.

The StateMachine object can also be easily extended to support new state machine functions. A new state machine can be implemented with the same interface for performing new functions only and pass messages to these state machines from inside the original state machine. This mechanism not only allows the developer to easily extend the state but also eliminates the need for code duplication to replicate initial PCEP handshaking.

E. Computation Module

The computation module is responsible for serving path computation requests in the PCE. In our implementation, shown in Fig. 9, we use a variable size worker thread-pool to process path computation requests. All incoming requests are added to a single *receive* queue and the worker threads select requests from the head of this queue for processing. However, this mechanism can be easily extended to implement priority queuing for serving high-priority requests.

A critical feature in the computation module is the traffic engineering database integration. In order to optimize for time, we implement local topological representation inside the computation module which reduces the need for polling topological information while computing paths. Such representation also makes it easier to implement new path computation algorithms as the topology format and available interfaces remain the same. The implementation of a local representation however poses the issue of synchronization with the TED. To address this, a push-based mechanism was implemented for the external TED to send topology updates to the computation module. In the implementation, each worker thread maintains an individual copy the local topology representation to reduce deadlock overheads. An interrupt based mechanism is included to update the topology stored inside each thread when an external entity sends the new topology information.

The push mechanism allows the TED to send topology updates to the PCE when changes are made inside the topology. Policy rules can be used to dictate the maximum frequency and the threshold (of changes in network topology) at which an update should be sent to the computation module. It should be indicated here that the aforementioned design choices for TED integration are specific to our implementation and can be changed easily in other implementations without requiring changes to the rest of the module implementations.

V. Performance Evaluation

In this section, we present a performance evaluation study which demonstrates the scalability of the open-source implementation and also presents examples to demonstrate the ease of extensibility of our implementation. We utilize a ROCKS cluster [24] including the Sun-Grid engine with the PCE server running on the head-node and multiple clients running on five compute nodes to test the performance of our implementation. Each node (head, compute) in the cluster is a DELL OptiPlex760 PC (Intel®Core™2 Quad CPU Q950 (2.83GHz), 2GB Dual-Channel-DDR2-SDRAM).

In a carrier network, typical quantitative PCE quality of service parameters critical to the network designer are: 2) Maximum load (req/sec) and 2) Average session duration, while features such as extensibility and flexibility as a more qualitative measure. The Maximum load measure is critical as it helps designers evaluate the reaction of the PCE to different extreme use scenarios such as a large scale network failure or even a security attack where the PCE can temporarily receive a large number of computation requests. The Average Session Duration is also critical as it determines the location of the PCE, especially if it is used for critical applications such as path computation for dynamic restoration in transport networks. Qualitative features of the PCE can provide cost and resource savings, especially when deployed in specialized networks and in our evaluation, we present a two step performance analysis which presents both quantitative as well as qualitative advantages of our PCE architecture.

In the first part of testing, we assume the network to be a IP/MPLS network and evaluate the performance of the PCE when varying four important network design parameters, namely: path computation request rates, topology sizes, path computation algorithms used and the effect of TED update frequency. After that, we demonstrate an example deployment scenario in a WDM network where the extensible protocol and TED description and update mechanisms of our implementation are exploited.

A. Scalability Analysis

1) Path Computation Request Rate: Here, our goal is to test the scalability of the PCE in terms of rate of path computation request served. In this test, the PCE uses a simple shortest-path algorithm for a typical best-effort MPLS connection request in the Atlanta network topology [25] with 15 nodes and 22 links.

As it was difficult to emulate a large number of actual nodes with different IP addresses, we modified the network module to accept multiple connections from the same IP address and distinguished between them using the client port number. To differentiate between connections from different clients using the same IP address, we appended the internal session identifier used in our PCE implementation with the remote IP address port which can uniquely identify a client. Note that this session identifier is only used for inter-module communication inside the PCE and is not the same as the PCEP session identifier.

For the test, a small launcher was used to instantiate multiple clients on each compute node in the cluster. Each client generates connection requests to the PCE for path computation, with the inter-arrival time between subsequent requests from

Load		Nw. read	Session	Req. Wait	Comp.	Resp.
1500	avg	19329	46535	20254	56021	122379
	std	319281	499101	55901	369039	817393
2500	avg	20575	49515	21256	58345	123462
	std	330465	217162	60618	528935	587564
4500	avg	20796	41147	21714	60344	123919
	std	261759	180293	407260	332964	90200

TABLE I
TIME MEASUREMENTS (IN NANOSECONDS) FOR DIFFERENT PROCESSES INSIDE MODULES IN THE PCE SERVER (SHOWN IN FIG. 10). LOAD IS REPRESENTED AS NUMBER OF CLIENTS WITH AVERAGE REQUEST INTER-ARRIVAL TIME PER CLIENT = 1SEC.

Thread Pool Size		Queuing	Computation	Total
1 Thread	avg	5184762	50269354	55454116
	std	15121718	32560269	35479705
5 Threads	avg	226103	53045413	53271516
	std	1907186	35562894	35639152
10 Threads	avg	68466	54478004	51946471
	std	413915	32581812	32605523

TABLE II
QUEUING TIMES AND COMPUTATION TIMES (IN NANOSECONDS) IN THE COMPUTATION MODULE WHEN VARYING THREAD POOL SIZE.

a given client given by a Gaussian distribution. In the tests, each client has the same average inter-arrival time (1 sec) and we varied the total number of clients active in order to vary request rate on the PCE server. The resulting processing times for different server loading conditions are shown in Table I. It can be seen that even with a very high load (4500 path computation requests per second), the average processing time was insignificant ($< 1\ ms$) indicating that the modules in the server can scale extensively for basic path computations.

In order to measure the performance of the different modules in the network, timestamps in *nanoseconds* were taken at specific points inside the PCE server and were used to calculate the processing times for operations inside different modules in the PCE server. For a typical path computation request-response operation, we measured five different processing times as shown in Fig. 10. These measurements help us to evaluate the performance of the PCE under different request rate conditions and they include network read, session processing, computation request queuing, computation processing and response sending. It can be seen that the individual module processing times (Network, Session, Computation) are very low, and do not vary significantly with increase in network load. The large variation observed in the measurement of individual module processing times is caused largely due to scheduling of processing across multiple threads in the operating system. However, high values for multiple processes for the same session are very infrequent and as a result total processing times for a request are fairly consistent.

2) Effect of Topology Size and Path Computation Algorithm Complexity: While the network and session management modules are not affected by varying network topology or path computation complexity, the processing times and, in cases of high request rate, queuing times in the computation module are significantly affected by this increase in complexity. In the study presented in Table I, the time taken for path computation was comparable to the times taken by other operations in the PCE. However, as shown in Fig. 11, increasing the size of the topology significantly increases the computation time making it the dominant contributor in determining the capacity of the PCE in terms of the rate of path computation requests served. In this study, we used two known transport topologies: Atlanta (15 node, 22 links) and TA2 (65 nodes, 108 links) from [25] and generated two Internet-like topologies with (120 nodes, 237 links) and (200 nodes, 397 links) respectively using the BRITE random topology generator [26]. We also study the computation times for two more complex path computation algorithms inside the PCE. Our computation implementation uses a custom graph API which provides interfaces for incorporating new path computation algorithms. We implemented the Widest-Shortest-Path First [27] as well as the Optimal Link Disjoint multi-path [28] algorithms in our computation module. The worst case computation complexity in our implementations of the shortest path computation, the link disjoint path computation and the widest-shortest path computation are $O(V \cdot ln(E))$, $O(2V \cdot ln(E))$ and $O(V^2)$ respectively. The measured processing times for these algorithms is presented in Fig. 11 and it can be seen that the increase in topology size led to significant increase in the computation times especially for the widest shortest path first algorithm whose search space is significantly larger than the other algorithms. Ass the computation complexity of the algorithm increases, we can also observe an increase in the variance when processing path computation requests.

In such scenarios, sporadic bursts of path computation requests can lead to large queuing delays of requests before being processed (measured time (3) in Fig. 10). A possible solution to this scenario could be to implement algorithms to dynamically manage the size of the thread pool used in the computation module. We performed a test using the 200 node topology and the Widest-Shortest-Path First algorithm to compute incoming requests with an average request inter-arrival rate of 400 milliseconds (Gaussian distribution). We then varied the size of the thread pool to determine the computation times as well as the queuing times for each request and the results are presented in Table II. It can be seen that while there is a small increase in the processing times per thread, there is also a decrease in the queuing times and as a result we see a difference of almost 0.2 milliseconds per request in the total time taken by the computation module. Therefore, carefully designed policies may be used to deal with temporary overloading of the computation module by dynamically increasing the size of the thread pool used. It is however clear that increasing the thread-pool infinitely will not solve problems with permanent overloading and in such scenarios, it is necessary to have a distributed implementation of the computation module for load balancing across servers.

3) Managing TED updates: In our implementation, we provide mechanisms to facilitate TED update in the computation module and ensure that a computed path is processed using up to date topology. In case a TED update arrives when a path

computation request is being computed, the computation is interrupted and the request is re-computed using the updated topology information to ensure that the optimal path is used in the network. However, such a mechanism can also potentially lead to large number of re-computations of requests which in turn affects performance. The frequency of the TED update is therefore a critical parameter, the affect of which on the PCE performance needs to be evaluated.

We performed a test where we varied the TED update frequency and monitored the variation in the computation times for different traffic loading conditions. We used the 200 node network topology and used the Optimal Link Disjoint multi-path algorithm to compute incoming requests with an average request inter-arrival rate of 400 milliseconds (Gaussian distribution). The topology update time interval was assumed to be fixed and was varied between 500 ms to 2 seconds. The results of this test are shown in Fig. 12.

In cases of high topology update frequency, requests that were being processed were interrupted, leading to re-computation of these request and as a consequence the average total time for processing a request in the computation module increases. This is a critical factor, as in cases of networks with high load on the PCE (as demonstrated here) TED updates can lead to overloading inside the PCE server. Another important parameter in this study was the number of worker thread in the thread pool used for path computation. In case a large number of worker threads were used, at high topology update frequencies it was likely that more path computation requests were interrupted and we saw that the average computation times increase with increase in thread-pool size. However, the computation time taken for a request decreased significantly in cases of large thread pools with decrease in update frequency indicating that there exists a trade-off between the thread-pool size optimization when taking TED updates into consideration.

4) Affect of Network Latency: The communication delay between the PCE and PCC is an important factor in network design as it directly affects the time required by a PCC to obtain a computed path. Our results do not show a detailed presentation about the dependence of the total session time on the round-trip time, primarily as we did not have efficient mechanisms to reliably tune the network latency in our experimental setup. It should also be noted that while the network latency does not overtly affect the PCE server architecture itself, special conditions on network latency pose several implementation challenges which can be critical to the performance in the PCE. We now present a discussion on the same.

In cases of high network latency between the PCC and the PCE, the average session time of a PCEP session is very high even though the total processing time required inside the PCE is low. In such a scenario, it is likely that the number of simultaneously active sessions is very high in the network. In such a scenario, even though the processing overhead on the PCE is low, memory demands to support read/write buffers for all active sockets can lead to increased system resource usage, restricting the performance of the PCE.

On the other hand, very low network latencies ($\leq 1ms$) required addressing synchronization issues inside the PCE. A representative example was found when establishing the $\mathrm{StateMachine}$ for a new connection in the PCE: in this scenario, after a connection is established, a $\mathrm{StateMachine}$ object is created and its constructor is initialized. The constructor is responsible for sending the *Open* message to the remote peer and setting the State Machine into the *Open Wait* state. However, we observed that in cases of low network latencies, the $\mathrm{StateMachine}$ object received an *Open* message from a remote peer before the constructor execution was completed, requiring implementation of synchronization mechanisms to ensure that $\mathrm{StateMachine}$ object constructors finish execution before the corresponding sockets are registered for receiving data in the network module.

B. Adaptation of the PCE Implementation to WDM networks

In the previous section, we presented how network design parameters affect the performance of the PCE and show that our implementation is scalable for implementation in real networks. In this section, we attempt to use a representative adaptation of our PCE in a WDM network scenario to demonstrate the features of flexible TED definition and module integration to demonstrate how our PCE implementation can be useful when applied in specialized network scenarios.

As a basic representative WDM network scenario, we used the Atlanta network topology [25] as the base network topology and each link was assumed to consist of two unidirectional fibers with each fiber supporting 8 wavelengths. We assumed a network scenario where network operators could provision individual lightpath requests for dynamic network connections as well as large batch requests, which would be used to provision multiple lightpaths simultaneously. Batch processing of lightpaths is envisioned as a scenario where the network operator provisions complex Layer-1 Virtual Private Network (VPN) topologies.

Note that the demonstration is targeted towards demonstrating flexibility of the PCE architecture and therefore we do not incorporate physical layer constraints in the lightpath computation and only use 8 wavelengths (unlike 40 - 80 typical to DWDM networks). However, for deployment in real networks, these parameters can be easily incorporated into the same.

In order to compute lightpath requests we used two sample computation modules. In the first computation module example, we used the shortest path first algorithm with random wavelength selection and hop count restrictions which limited the hop-count to 4 nodes. In the second computational module, and for a demonstration of a typical on-line optimization tool, we implemented a basic Integer Linear Programming (ILP) based formulation to compute optimal the lightpath requests. The formulation of the same is given in the appendix. We implemented that computational model with a model of an ILP in the Gurobi Optimizer [29] and limited the run time to 2 minutes at which point the optimizer provides a heuristic result.

The PCE implementation was configured to use 10 worker threads and when a batch of lightpath requests was received, the individual requests inside the batch were served by different worker threads. At first, we tested the PCE while using only one of the two optimizers defined for the computation of individual as well as batch computation requests and limited the number of demands in a single batch request to a random number between 3 and 10. In this scenario, when using the ILP based computation module, we experienced very high computation times with the average times of up to 5 seconds in order to compute single lightpath requests and up to 45 seconds in order to compute batch lightpath requests. While the computed results were optimal the times taken for computation were very high, limiting its capability for dynamic provisioning of individual lightpath requests.

On the other hand, when using the shortest path first algorithm with random wavelength selection we observed that the computation times were in the order of 1 - 2 milliseconds for individual requests. As the batch size was always less than or equal to the total thread-pool size, computation of batch requests also took the same approximate time. As compared to the ILP based computation module, this module had excellent scalability in terms of provisioning a large number of requests. However, as lightpaths were processed by individual threads each with its own copy of the network topology, we observed *collisions* wherein demands processed for the same batch used the same wavelength and fiber combination in order to provision a connection.

Path computation collision in optical networks has been studied in other similar contexts: for example, [30] show how low inter-arrival times between connection requests (equivalent to batch arrivals in our scenario) significantly increase connection blocking when wavelengths are assigned before reservation to individual connections, but how wavelength selection at intermediate nodes can reduce the collision rate, even while disseminating less information. In our scenario, assuming that wavelengths were selected by the PCE directly, this scenario translates to the PCE processing all batch requests simultaneously in order to avoid collisions. In our evaluation, for the batch sizes varying between 3-10 demands per batch, we observed a collision frequency of 0.8% measured over a 1000 requests and for larger batch sizes of 15 and 25 demands per batch, the collision frequency increased to up to 4.8%.

In order to address this trade-off, we incorporated both computation modules *simultaneously* in our PCE architecture as shown in Fig. 4. In our implementation, the $\mathrm{send}\ ()$ function in the session module was modified to communicate with two different computation modules, one running the shortest-path heuristic and the other using the ILP formulation to serve lightpath requests. We then updated the $\mathrm{StateMachine}$ in the Session management module to identify if an incoming $\mathrm{PCEPRequestMessage}$ contained a single lightpath request or a batch request: in case of a single request, the session management module forwarded the request to the heuristic-based computation module while in case of a batch request, we forwarded the request to the ILP based computation module. Using this integration, we significantly improved the performance of the PCE when serving individual lightpath requests, while we ensured that collisions were eliminated when serving batch requests.

This simple, but highly effective integration of multiple computation modules demonstrates the advantages of flexible integration of computation modules, allowing network designers to use existing computation modules and optimize between them to enhance performance in the network. Similar mechanisms can also be used for other specialized computation scenarios such as integration of path computation functions for different layers (MPLS, WDM) into a single PCE, while preserving the differences in their topology representation and computation mechanisms used.

VI. Conclusion

We presented the first open-source Path Computation Element (PCE) emulator along with its key design and implementation features. The architecture incorporates all elements of the standard PCE framework and is shown to easily evolve to support extensions to the PCE protocol, path computation algorithms, topology descriptions and the PCE state machines. Furthermore, flexible inter-module messaging interfaces allow for seamless integration of new modules. We presented a simple example, where multiple computation modules are used to perform path computation for different type of requests and similar extensions can be used to require a single PCE to support multiple network layers, each with a customized computation algorithms and topology representation and update mechanisms.

The scalability of the PCE implementation was tested in terms of number of requests, size of topologies served, path computation algorithm complexity and support for TED update. We showed that for small topologies, the solution can easily scale beyond multiple thousands of requests per second on typical desktop hardware. An increase in the path computation algorithm complexity and topology size increases the computation processing times inside the PCE. We also demonstrated how dynamic updates to the thread-pool size could help eliminate temporary request bottlenecks. Finally, we provided a simple example to demonstrate the trade-off in using a large thread-pool in the case of TED updates, and similar mechanisms can be applied to other specific network scenarios to evaluate the optimal performance parameters (thread-pool size).

Developers, network engineers and path computation algorithm designers can access the open-source implementation for download from [31]. We are currently testing the interoperability of our PCE with other implementations and in real network environments under the purview of EU projects ONE [32] and GEYSERS [33]. These are examples of wider PCE deployments and innovations we expect in the near future, as part of the ongoing evolution of network control and management towards openness and programability.

REFERENCES

[1] A. Farrel, J. P. Vasseur, J. Ash, *"A Path Computation Element-Based Architecture"*, IETF RFC 4655, August 2006, http://tools.ietf.org/rfc/rfc4655.txt

[2] JP. Vasseur, J.L. Le Roux,*"Path Computation Element Communication Protocol"*,IETF RFC 5440, March 2009, http://tools.ietf.org/rfc/rfc5440.txt

[3] RAYControl, Scalable Optical Transport, Data Sheet, http://www.advaoptical.com/en/resources/ /media/Resources/Data%20Sheets/RAYcontrol.ashx

[4] Marben Inter Domain Routing, OIF DDRP based on OSPF / IETF PCE Solutions, http://www.marben-products.com/osiam/DDRP-DSH.pdf

[5] Multi-Technology Operations System Interface (MTOSI) Release 2.0, Tele MAnagement Forum Standard, Nov 2007, http://www.tmforum.org/MTOSIRelease20/6076/home.html

[6] M. Chamania, M. Drogon, A. Jukan, "*Lessons Learned From Implementing a Path Computation Element (PCE) Emulator,"* (Postdeadline paper) Technical Digest of Optical Fiber Communication Conference 2011 (OSA/OFC 2011), Los Angeles, CA, March 2011

[7] E. Mannie, *"Generalized Multi-Protocol Label Switching (GMPLS) Architecture"*, IETF RFC 3945, October 2004, http://tools.ietf.org/rfc/rfc3945.txt

[8] R. Braden, et al., *"Resource ReSerVation Protocol (RSVP)"*, IETF RFC 2205, September 1997, http://tools.ietf.org/rfc/rfc2205.txt

[9] E. Oki, T. Takeda, JL. Le Roux, A. Farrel, "*Framework for PCE-Based Inter-Layer MPLS and GMPLS Traffic Engineering,*" IETF RFC 5623, Sept. 2009, http://tools.ietf.org/rfc/rfc5623.txt

[10] JP. Vasseur, et al., *"A Backward-Recursive PCE-Based Computation (BRPC) Procedure to Compute Shortest Constrained Inter-Domain Traffic Engineering Label Switched Paths"*, IETF RFC 5441, April 2009, http://tools.ietf.org/rfc/rfc5441.txt

[11] IETF PCE Working Group, http://datatracker.ietf.org/wg/pce/charter/

[12] I. Bryskin, D. Papadimitriou, L. Berger, J. Ash, "*Policy-Enabled Path Computation Framework,*" IETF RFC 5394, Dec. 2008, http://datatracker.ietf.org/doc/rfc5394/

[13] JP. Vasseur, et al., *"A Set of Monitoring Tools for Path Computation Element (PCE)-Based Architecture"*, IETF RFC 5886, June 2010, http://tools.ietf.org/rfc/rfc5886.txt

[14] Q. Zhao, et al., *"Extensions to the Path Computation Element Communication Protocol (PCEP) for Point-to-Multipoint Traffic Engineering Label Switched Paths"*, IETF RFC 6006, September 2010, http://tools.ietf.org/rfc/rfc6006.txt

[15] Q. Zhao, et al., *"PCE-based Computation Procedure To Compute Shortest Constrained P2MP Inter-domain Traffic Engineering Label Switched Paths"*, IETF Internet Draft, January 2011, http://tools.ietf.org/id/draft-zhao-pce-pcep-inter-domain-p2mp-procedures-07.txt/

[16] I. Nishioka, et al., *"Use of the Synchronization VECtor (SVEC) List for Synchronized Dependent Path Computations"*, IETF RFC 6007, September 2010, http://tools.ietf.org/rfc/rfc6007.txt

[17] Y. Lee, et al., *"PCEP Requirements for WSON Routing and Wavelength Assignment"*, IETF Internet Draft, July 2011, http://datatracker.ietf.org/doc/draft-ietf-pce-wson-routing-wavelength/

[18] A. Heffernan, "*Protection of BGP Sessions via the TCP MD5 Signature Option"*, IETF RFC 2385, August 1998, http://tools.ietf.org/rfc/rfc2385.txt

[19] J. Touch, A. Mankin, R. Bonica, *"The TCP Authentication Option"*, IETF RFC 5925, June 2010, http://tools.ietf.org/rfc/rfc5925.txt

[20] S. Greco Polito, S. Zaghloul, M. Chamania, A. Jukan, *"Inter-Domain Path Provisioning with Security Features: Architecture and Signaling Performance,"*accepted for publication in IEEE Transactions on Network and Service Management, 2011

[21] CTTC PCE, http://wikiona.cttc.es/ona/index.php/Path_Computation_Element_(PCE)

[22] OSCARS Inter-Domain Controller Application Programmer Interface, http://code.google.com/p/oscars-idc/

[23] Java Network I/O (NIO), http://download.oracle.com/javase/1.4.2/docs/api/java/nio/package-summary.html

[24] Rocks Cluster, Open Source Linux Cluster, http://www.rocksclusters.org/wordpress/

[25] SNDLib: Survivable Fixed Telecommunication Network Design, http://sndlib.zib.de/home.action

[26] BRITE: Boston University Representative Internet Topology gEnerator, http://www.cs.bu.edu/brite/

[27] R. Guerin, A. Orda, D. Williams, "*QoS Routing mechanisms and OSPF extensions,*" 2nd Global Internet Miniconference (joint with Globecom 97), 1997.

[28] J. Suurballe, "*Disjoint paths in a network",* Networks, vol. 14, pp. 125145, 1974.

[29] Gurobi Optimizer 3.0, http://www.gurobi.com/

[30] R. Muoz, R. Casellas, R. Martinez, M. Tornatore, A. Pattavina, "*An experimental study on the effects of outdated control information in GMPLS-controlled WSON for Shared Path Protection,*" Optical Network Design and Modeling (ONDM), Feb. 2011

[31] Open Source Path Computation Element Emulator, http://pce.ida-cns-group.net/

[32] EU Project ONE, http://www.ict-one.eu

[33] EU Project GEYSERS, http://www.geysers.eu

APPENDIX A
ILP FORMULATION

In this appendix, we present the formulation used for the ILP to serve batch lightpath requests. The graph was represented as a directed graph $G(V, E)$, with nodes $v_i \in V$ and edges $e_{ij} \in E$. The set of wavelengths was given by T and the availability of a wavelength λ_t on edge e_{ij} was given by a boolean indicator c_{ij}^t. The batch of lightpath demands was defined as a set D and each demand $d_x \in D$ has a source $S(d_x)$ and a destination $D(d_x)$ node, and was assumed to use only one wavelength.

We defined an indicator variable $L(d_x)$ to indicate if a request was provisioned, and defined a boolean routing variable for demand d_x as $r_{ij}^t(d_x)$, indicating if the lightpath used for demand d_x used the wavelength λ_t on the link e_{ij}. Based on these parameters, the constraints for provisioning the lightpaths are given as:

$$\forall d_x \in D : L(d_x) \leq 1 \tag{1}$$

$$\forall d_x \in D, e_{ij} \in E, t \in T : r_{ij}^t(d_x) \leq L(d_x) \tag{2}$$

$$\forall d_x \in D, v_s = S(d_x) : \sum_{t \in T} \sum_{j:e_{sj} \in E} r_{sj}^t(d_x) = L(d_x) \tag{3}$$

$$\forall d_x \in D, v_s = D(d_x) : \sum_{t \in T} \sum_{j:e_{js} \in E} r_{js}^t(d_x) = L(d_x) \tag{4}$$

$$\forall d_x \in D, t \in T, j : v_j \neq D(d_x), S(d_x) : \sum_{i:e_{ij} \in E} r_{ij}^t(d_x) = \sum_{k:e_{jk} \in E} r_{jk}^t(d_x) \tag{5}$$

$$\forall d_x \in D, t \in T, : \sum_{e_{ij} \in E} r_{ij}^t(d_x) \leq HopLimit \tag{6}$$

$$\forall e_{ij} \in E, t \in T : \sum_{d_x \in D} r_{ij}^t(d_x) \leq c_{ij}^t \tag{7}$$

In these constraints, (1) is used to indicate if a demand is provisioned or not, and (2) ensures that routing variables are set to 0 if a demand is not provisioned. Constraints (3) and (4) ensure that at max. one wavelength is selected to provision a demand at the source and the destination of the demand in case the demand is satisfied ($L(d_x) = 1$), and (5) provides routing continuity for each demand for each wavelength in the network. (6) ensures that no demand exceeds the hop count constraint on the lightpath, which is set to four hops. Finally,

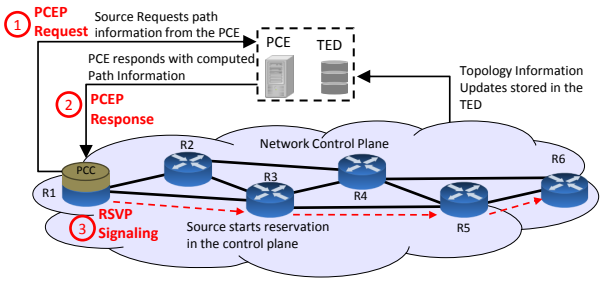


Fig. 1. Use of Path Computation Element (PCE) in transport networks

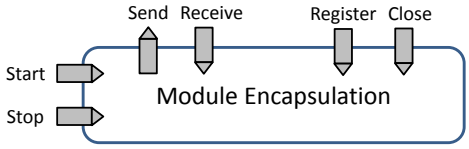


Fig. 2. Module encapsulation to support inter-module messaging interfaces inside the PCE

(7) provides the lightpath availability constraint ensuring that only available lightpaths are used.

The objective function is given as:

$$Min: \sum_{D}(1 - L(d_x)) + \beta \cdot \sum_{D}\sum_{t \in T}\sum_{e_{ij} \in E} r_{ij}^{t}(d_x) \qquad (8)$$

The first term maximizes the total number of demands provisioned, while the second term minimizes the total capacity used. We use a scaling factor β which is a small value to ensure that the ILP first maximizes the total demands served, and then minimizes the capacity used by these demands.

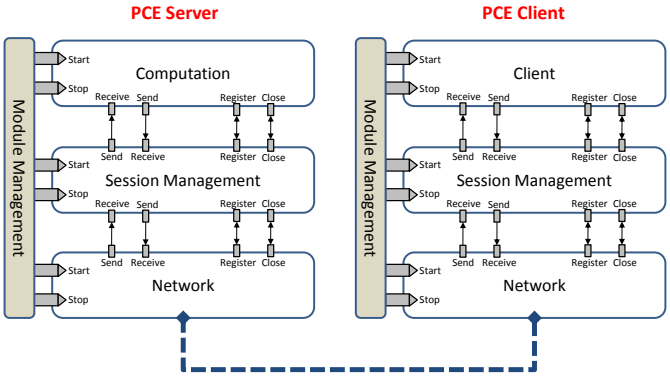


Fig. 3. Module interactions inside the the PCE server and client

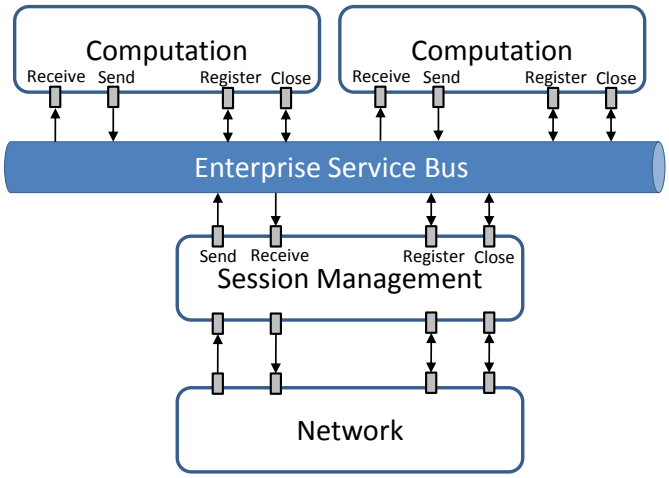


Fig. 4. Support for multiple computation modules using a web-services based load-balancing module

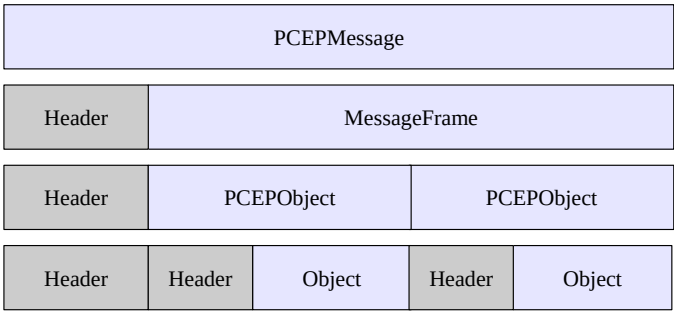


Fig. 5. The PCEP protocol represented as an object-oriented hierarchy

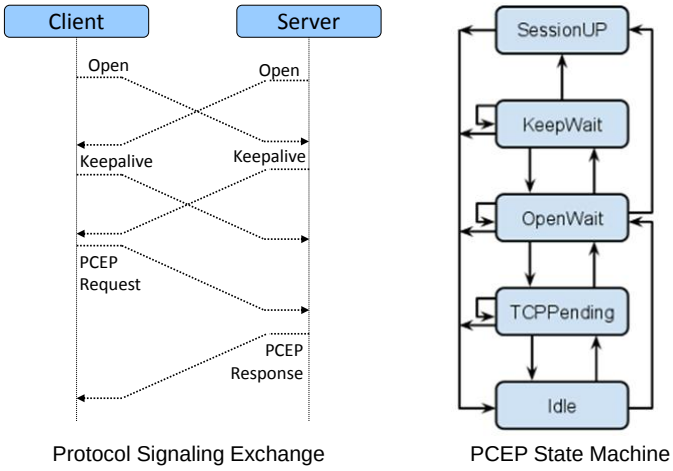


Fig. 6. The PCEP state machine, and the typical message exchange between a PCC (client) and a PCE (server)

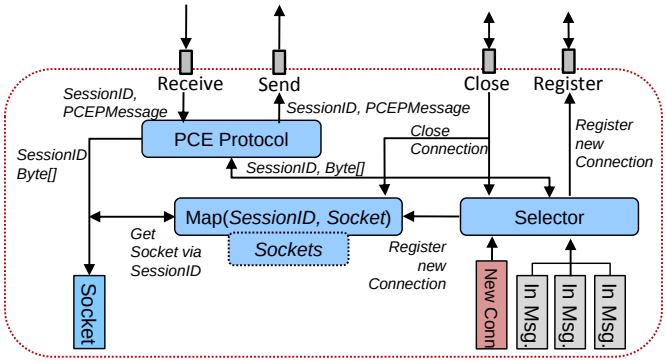


Fig. 7. Network module implementation

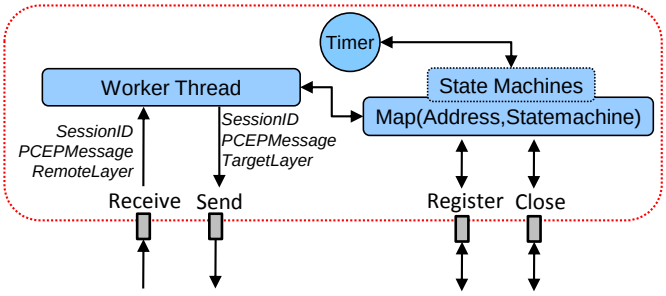


Fig. 8. Session management module implementation

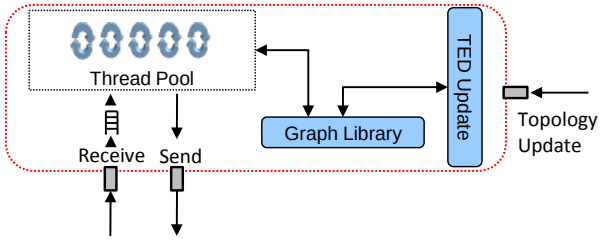
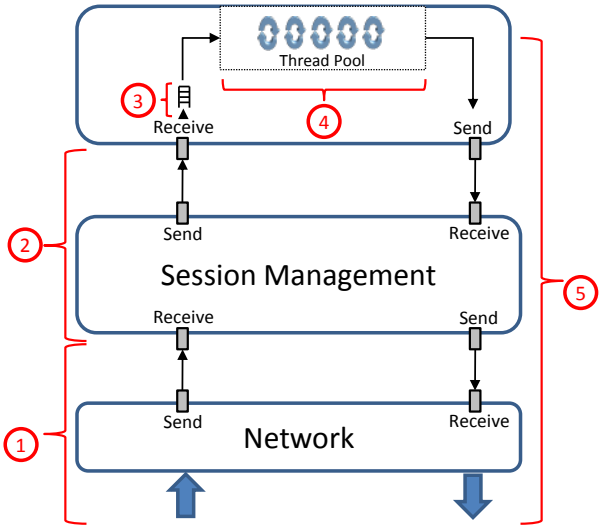


Fig. 9. Computation module implementation



Measured Times:
1) Network Read 2) Session Processing 3) Comp. Req. Queuing
4) Computation Processing 5) Response Sending

Fig. 10. Measurements for processing times in the PCE Server

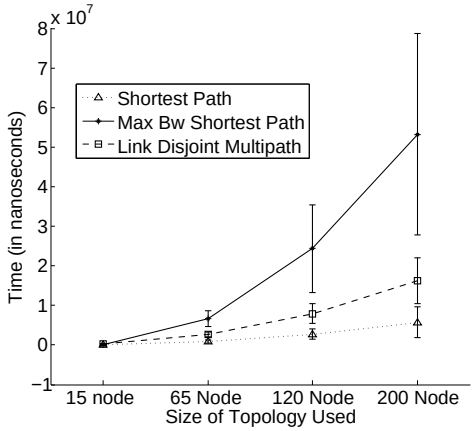


Fig. 11. Measurements for processing times in the computation module for different path computation algorithms and topology sizes

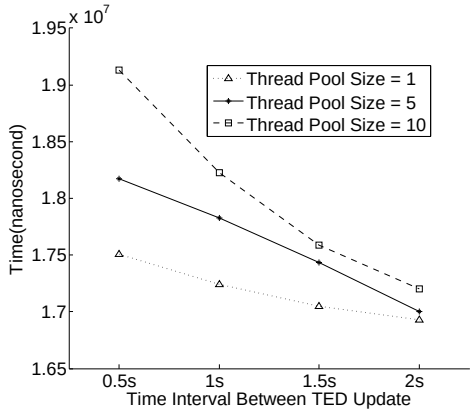


Fig. 12. Average total time taken by the computation module versus increasing time interval between TED updates under different thread pool size