COntent Mediator architecture for content-aware nETworks

European Seventh Framework Project FP7-2010-ICT-248784-STREP

Deliverable D4.3

Prototype Implementation and System Integration Interfaces for Enhanced Network Platforms

The COMET Consortium

Telefónica Investigación y Desarrollo, TID, Spain
University College London, UCL, United Kingdom
University of Surrey, UniS, United Kingdom
PrimeTel PLC, PRIMETEL, Cyprus
Warsaw University of Technology, WUT, Poland
Intracom SA Telecom Solutions, INTRACOM TELECOM, Greece

For more information on this document or the COMET project, please contact:

Andrzej Bęben
Warsaw University of Technology, abeben@tele.pw.edu.pl

Document Control

Title: Prototype Implementation and System Integration Interfaces for Enhanced Network Platforms

Type: Public

Editor(s): Andrzej Bęben

E-mail: abeben@tele.pw.edu.pl

Author(s): Andrzej Bęben, Piotr Wiśniewski, Jarosław Śliwiński (WUT), George Kamel (UniS), Lenos Andreou, Michael Georgiades (PrimeTel)

Doc ID: d4.3_v1.1.doc

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
v0.1	26/11/11	Andrzej Bęben, Jarosław Śliwiński	First contribution about RAE
v0.2	20/01/12	George Kamel	Contribution about stateful CAFE
V0.3	03/02/12	A.Beben, P.Wiśniewski	Contribution about stateless CAFE
V0.4	13/02/12	George Kamel	Revised contribution about stateful CAFE with validation tests
V0.5	13/02/12	P.Wiśniewski	Revised contribution about RAE
V0.6	16/02/12	George Petropoulos	Review and comments
V0.7	17/02/12	Lenos Andreou	Contribution on CAFE validation
V0.8	17/02/12	George Kamel	Update of stateful CAFE description
V0.9	20/02/12	Ioannis Psaras	Review and comments
V1.0	20/02/12	Andrzej Bęben	Final version
V1.1	20/02/12	David Flórez	Final Version for submission

Legal Notices

The information in this document is subject to change without notice.
The Members of the COMET Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the COMET Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Table of Contents

1 Executive Summary

This deliverable focuses on the software design and implementation of entities developed for the Content Forwarding Plane (CFP) of the COMET system. In the COMET architecture [1], the CFP is responsible for content delivery from the content server to the content consumer. Moreover, the CFP gathers information about the network and provides it to the Content Mediation Plane (CMP) for enabling content and network mediation, and as a consequence, optimisation of content delivery. The basic entities of CFP are: Routing Awareness Entity (RAE) and Content Aware Forwarding Entity (CAFE). The software was developed following specification of RAE and CAFEs provided in COMET deliverable, D4.2 [2]. According to this specification, we implemented two versions of CAFE, called stateless and stateful, which are specially designed for the decoupled and the coupled approaches, respectively. The CFP entities will cooperate with other components of the COMET system. Implementation of other COMET components is provided in COMET deliverable D3.3 [3].

In chapter 2, we present the outline of implemented network entities and introduce their interfaces to other COMET system entities. We summarise exchanged messages and technologies used for each interface. Moreover, we present the deployment diagram related to network elements and briefly discuss how to deploy network entities in a single domain.

Chapter 4 focuses on software design and implementation of the Routing Awareness Entity (RAE). Our implementation follows the specification of the RAE provided in COMET deliverable D4.2 [2]. We present details of designed interfaces between RAE and other COMET entities, UML class diagrams related to internal components of RAE as well as sequence diagrams corresponding to prefix advertisement, withdrawal, update of provisioning information and reset/unavailability of RAE. We also present validation test plan corresponding to RAE basic operations and stress tests aimed to validate capabilities of RAE in order to handle large number of prefixes. These test will be performed in the integration testbed.

The software design and implementation of stateless CAFE is described in chapter 5. The stateless CAFE consists of two loadable Linux kernel modules, called *cafe_forward* and *cafe_intercept* and the set of configuration tools. The *cafe_forward* module forwards COMET packets containing content, while the *cafe_intercept* module is responsible for encapsulation of IP packets received at edge nodes into COMET packets. The configuration tools allow the CME entity to configure forwarding rules and packet interception filters on CAFE modules. In this chapter we present details of internal and external interfaces used by CAFE, the UML class diagrams related to developed modules showing internal components of particular module as well as message sequence diagrams explaining interactions between developed modules. Finally, we present results of basic tests that validate the developed software.

Section 6 details the implementation of content delivery in the coupled approach within the proof-of-concept emulator. The implementation is based upon the technical specifications given in COMET deliverable, D4.2 [1]. The main aspects covered in that section are state installation within CAFEs by CRMEs, the operation of content delivery, and the mechanism of route optimisation. While the focus of section 5 is on the CAFE which is the entity responsible for forwarding content, some related operations of the CRMEs are also detailed. We describe the operation of the CAFE and related content-forwarding-related CRME functionalities by means of UML state diagrams, and finally present our methodology to test our implementation.

In this deliverable, we present only unitary/standalone tests of developed entities. As the next step, the CFP entities will be integrated with other COMET entities. The integration technologies and procedures used to develop, integrate and test the COMET software, as well as information about the system-wide validation tests and system releases, will be included in forthcoming deliverable "D5.1 – Integration of COMET Prototype and Adaptation of Applications".

2 Introduction

This deliverable focuses on the software design and implementation of entities developed for the Content Forwarding Plane (CFP) of the COMET system. In the COMET architecture [1], the CFP is responsible for content delivery from the content server to the content consumer. Moreover, the CFP gathers information about network and provides it to the Content Mediation Plane (CMP) for enabling content and network mediation, and as a consequence, optimisation of content delivery. The basic entities of CFP are: Routing Awareness Entity (RAE) and Content Aware Forwarding Entity (CAFE). The objective of RAE is to create and manage content delivery paths. The routing awareness is an off-line process performed in long time scale. It reacts to changes in inter-domain network reachability and re-provisioning of domains. The CAFEs are specialised network nodes responsible for forwarding content packets based on COMET specific forwarding method. According to specification in D4.2, we implemented two versions of CAFE, called stateless and stateful, which are specially designed for the decoupled and the coupled approaches, respectively. The stateless CAFE forwards packets based on information about content delivery path stored inside the COMET header. The COMET header is attached and removed by edge CAFEs located close to the content server and the client. The content delivery path is selected during content resolution process and then it is configured in the edge CAFE during the path configuration process [4]. On the other hand, the stateful CAFE forwards content packets based on the information configured during content resolution process. According to this approach specified in D3.2 [4], once the Content Resolution and Mediation Entity (CRME) has determined to forward the content resolution request to its counterpart in the next hop domain towards the targeted source, it is responsible for configuring the corresponding ingress and egress CAFEs in its local domain for preparing for the actual delivery of the content flow back to the consumer. The CFP entities will cooperate with other components of the COMET system. Implementation of other COMET components is documented in COMET deliverable D3.3 [3].

In this deliverable, we present details of developed entities, modules and components. For each entity, we present its internal and external interfaces with other COMET entities, UML class diagrams related to internal components as well as sequence diagrams related to basic operations and interactions between components. Moreover, we defined and performed unitary/standalone tests of each entity to validate functions of implemented software.

In particular, Chapter 2 presents outline of implemented network entities, summarises interfaces with other COMET entities and briefly discusses deployment issues related to developed entities. Chapter 3 presents implementation of RAE. We present details of designed interfaces between RAE and other COMET entities, UML class diagrams related to internal components of RAE as well as sequence diagrams corresponding to basic RAE operations. In chapter 4, we present implementation of the stateless CAFE. It covers description of two CAFE modules, i.e., *cafe_forward* and *cafe_intercept* as well as a set of tools used to configure forwarding rules and packet interception filters on CAFE modules. Chapter 5 presents implementation of stateful CAFE. It covers aspects related to state installation within CAFEs by CRMEs, the operation of content delivery, and the mechanism of route optimisation. Finally, section 6 summarises this report and outlines plan for validation and integration of developed software in the testbed environment. In addition, this document includes 2 Annexes. Annex A presents exemplary configuration file for RAE and explains how to configure RAE. In Annex B, we present self-testing script for stateless CAFE.

3 Network elements overview

Figure 1 and Figure 2 present the component diagram of network elements developed in COMET. The network elements cover the Routing Awareness Entity (RAE) and two types of Content Aware Forwarding Entity (CAFE), which were designed and developed for the decoupled and the coupled approaches, respectively. The main functions of network elements are the following:

- The RAE is responsible for discovering inter-domain content delivery paths. It exchanges routing information with RAEs located in peering domains. The routing information covers network prefixes, supported COMET CoSs and long-term QoS characteristics of content delivery paths. Basd on this information, the RAE creates the set of preferred paths and provides them to CME/CRMEs. The information about content delivery paths is used in the resolution process to select the optimal source to deliver the content.
- The CAFEs are used to deliver content from the content server to the client through the content delivery path selected during the resolution phase. We have designed and implemented two versions of CAFEs, called stateful and stateless, in order to meet path configuration requirements of the coupled and decoupled approaches.
  - The stateless CAFE assumes that the content delivery path is configured at the server side once the resolution process has been carried out. The stateless CAFE is used in the decoupled approach.
  - The stateful CAFE assumes that the content delivery path is configured in a hop-by-hop manner during the content resolution process. This process is used in the coupled approach.

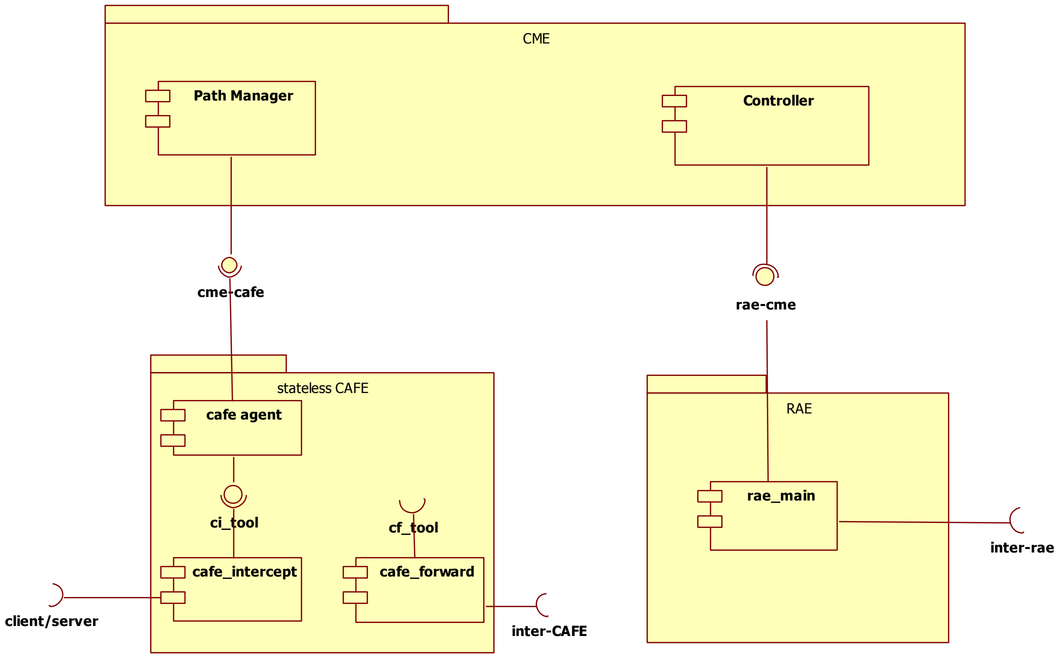


Figure 1: Network components diagram for decoupled approach and their relation with cooperating entities

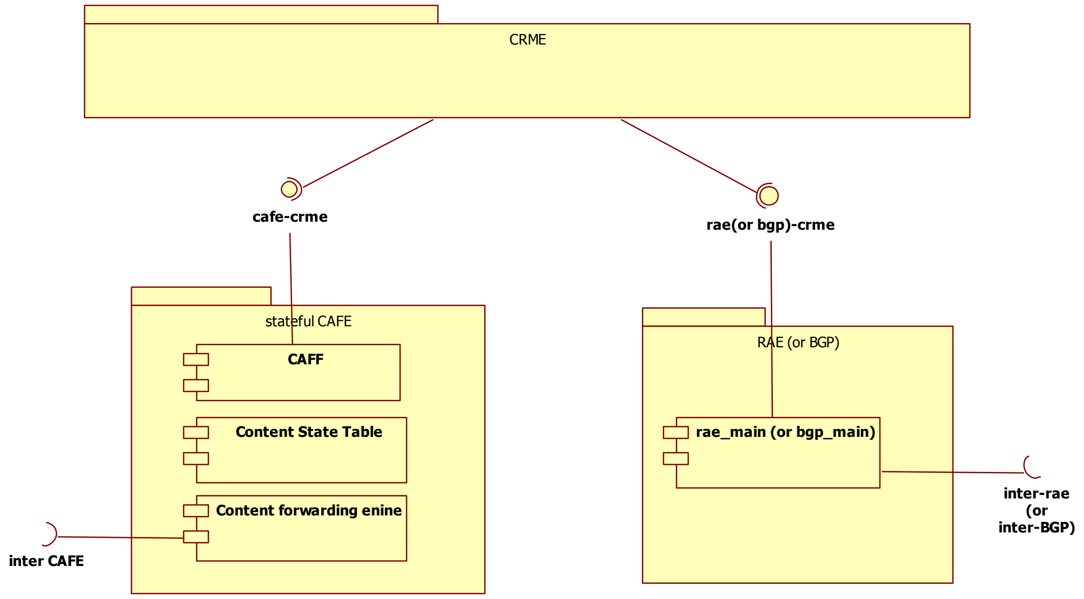


Figure 2: Network components diagram for coupled approach and their relation with cooperating entity

3.1 Interfaces

Table I presents the list of interfaces and the protocols used for network elements in decoupled approach.

Table I: The list of interfaces of network elements in the decoupled approach

Interface ID	Entity/Component providing the interface	Entity/Component using the interface	Purpose	Protocol	Reference
inter-rae	RAE	RAE	Exchange information about available/withdraw prefixes, paths and their characteristics.	Protobuf	Section 4.2.2
rae-cme	CME Controller	RAE	Path and provisioning information sent by RAE	Protobuf	Section 4.2.3
inter-cafe	CAFE	CAFE	Forward COMET packets between CAFEs	COMET over Ethernet or over VLAN Ethernet or over GRE	Section 5.2.1
CAFE-IP router or terminal	CAFE	IP router or terminal	Forward IP packets from IP router (terminal) to edge CAFE	IP over Ethernet	Section 5.2.2
cme-cafe	CAFE agent	Path Manager	CAFE configuration	Protobuf	Section 5.2.3

Table II presents the list of interfaces and the protocols used for network elements in the decoupled approach.

Table II: The list of interfaces of network elements in the decoupled approach

inter-CAFE	CAFE	CAFE	Forward content between CAFEs	UDP	Section 6.2.2
cafe-crme	CRME	CAFE	Announce and Notify messages sent by CAFE to CRME	UDP	Section 6.2.1
	CAFE	CRME	Configure messages sent by CRME to CAFE	UDP	Section 6.2.1

3.2 Network elements deployment

Figure 3 presents the deployment diagram for COMET network elements running in a single domain. The CME is deployed in the domain to perform content mediation functions in the decoupled approach. These same functions are performed by the CRMEs within the coupled approach. A single RAE server is deployed to collect information about content delivery paths and provide them to the CRE/CRMEs. On the other hand, several CAFEs are deployed to forward content from the content server to the client. In principle, the CAFEs should be located at domain edge nodes, i.e., the nodes where clients and servers are connected.

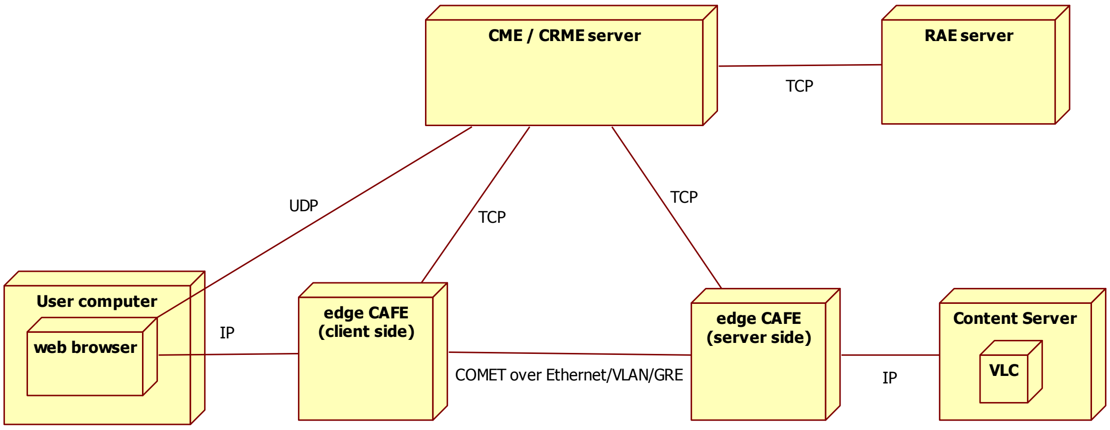


Figure 3: Deployment diagram for COMET network elements in a single domain (and its relation with CME/CRME entity).

4 Routing Awareness Entity

In this chapter, we present the implementation of the Routing Awareness Entity (RAE) and results of basic validation tests. The objective of RAE is to create and manage content delivery paths. The routing awareness is an off-line process. This process is responsible for reacting to changes in inter-domain network reachability and re-provisioning of domains. The routing awareness is performed by RAE entity, which should be located in every COMET domain as presented on Figure 4. Each RAE requires information about its domain, which among others covers: Autonomous System (AS) number, IP addresses of peering RAEs and network prefixes available inside the domain, as well as supported COMET CoSs and values of QoS parameters, such as maximum IP Packet Transfer Delay (IPTD), maximum IP Packet Loss Ratio (IPLR), maximum bandwidth (BW), which are assured between any ingress and egress points of the domain. RAEs exchange Network Layer Reachability Information (NLRI) in *update* messages to build or update content delivery paths. Once a given prefix becomes unavailable, the RAE removes it by a *withdraw* message. Each content delivery path is characterised by the list of AS numbers, supported COMET CoS and aggregated values of QoS parameters. The RAE provides information about discovered routes and their properties to the Path Storage component of CME [4]. This information is used by the Decision Maker during content resolution process to select the best path for content consumption. Detailed specification of RAE is included in D4.2. [1]

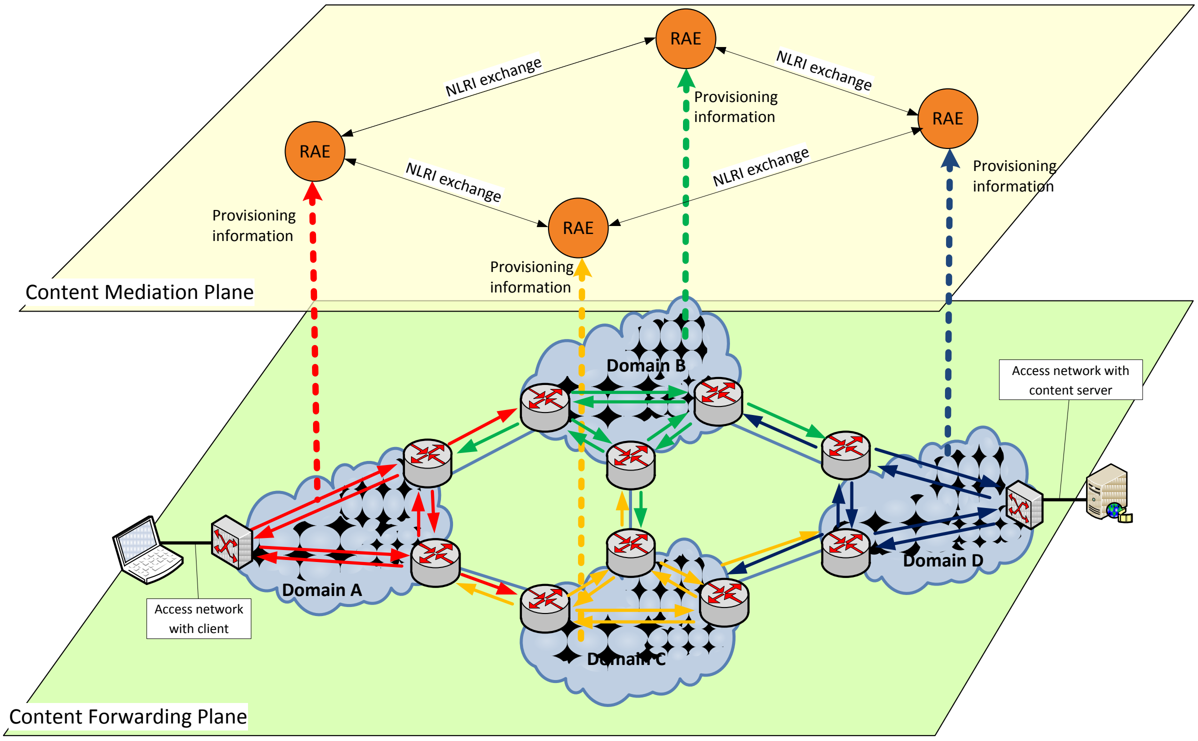


Figure 4: Routing awareness and provisioning

4.1 Description of functionality

The RAE is responsible for:

- Connecting to other RAE in peering domains,
- Propagating the information about own domain prefixes,
- Gathering the information about paths available to prefixes of other domains,
- Selection of preferred paths from available paths (to prefixes of other domains),
- Exchanging the information about preferred paths to prefixes of other domains,

- Sending the information about available prefixes and routing paths towards the prefix to the CME.

RAE is a self-contained entity and does not provide any substitutable elements.

4.2 Interfaces

In this section we describe the RAE interfaces. It uses 3 interfaces that include:

- Configuration file: the configuration file includes information about network prefixes available inside the domain(s), peering ASs, supported COMET CoSs and provisioned values of QoS characteristics between any pair of ingress and egress points.
- Inter-RAEs interface: on this interface RAE exchange messages with peers to create inter-domain content delivery paths.
- RAE-CME: on this interface RAE provides information to CME about available content delivery paths.

4.2.1 Configuration file

The configuration is stored in a binary file following *protobuf* encoding. The definition of the structure is provided in file *rae/resources/config.proto*. Figure 5 illustrates this structure. The exemplary RAE configuration file is included in Annex A.

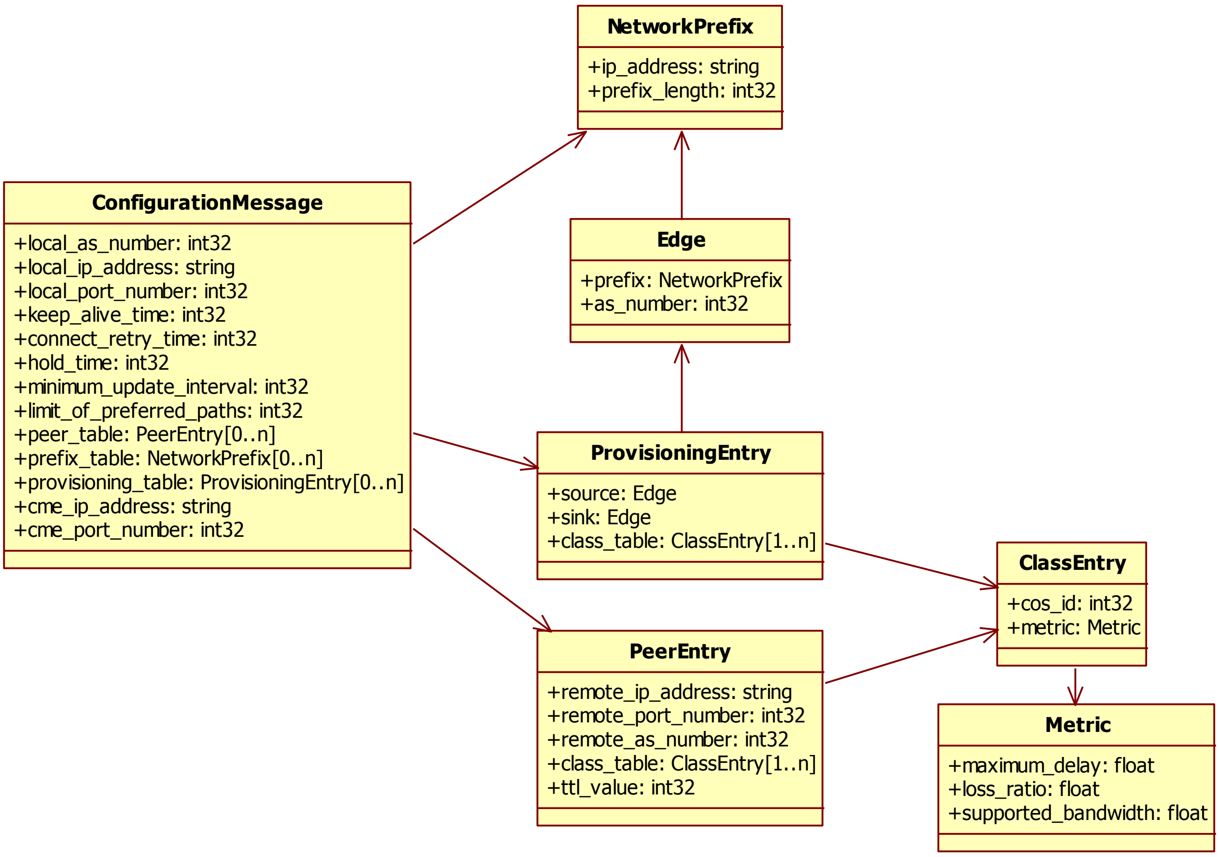


Figure 5: Data structure in configuration file of the RAE

The *ConfigurationMessage* includes all information required by RAE:

- local AS number (*local_as_number*) and local IP address (*local_ip_address*),

- parameters of decision process – number of preferred paths to prefixes of other domains (*limit_of_preferred_paths*),
- list of peering RAEs (*peer_table*),
- RAE's own domain prefixes (*prefix_table*),
- RAE's own domain provisioning information (pro*visioning_table*).

4.2.2 Inter-RAE interface

The RAEs exchange messages between themselves in an asynchronous manner. The message is encoded using *protobuf* encoding. The message structure is defined in file *rae/resources/config.proto*. Notice that the header has constant length of 5 bytes, while the length of the message body is variable. Figure 6 illustrates this structure.

This sending part of the interface consists of a single method in *tcp_connection* class:

```
void
tcp_connection::send_message(
boost::shared_ptr<MessageBody>message_ptr);
```

The receiving part of the interface consists of a single callback method in the *session* class:

```
void
session::notify_inter_message(
boost::shared_ptr<MessageBody>message_ptr);
```

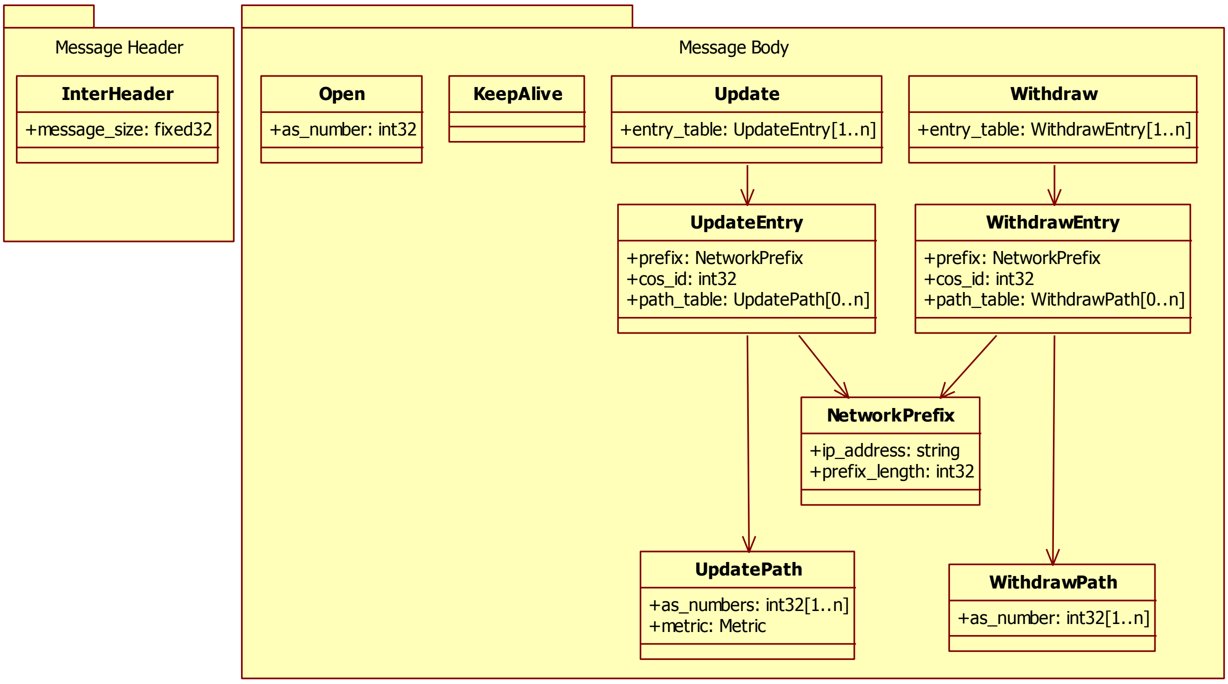


Figure 6: Inter-RAE message structure

RAEs exchange four types of messages: *open, keepalive, update* and *withdraw*. The *open* message is used to initiate a peering session. Once the session is established the corresponding RAEs exchange *update* messages in order to propagate available network prefixes and paths. RAE's send *withdraw* messages to inform that some prefixes are no longer available. Moreover RAEs exchange *keepalive* messages to inform each other that they are still in service. If the RAE stops receiving *keepalive* messages from the peering RAE, it withdraws all paths going through the “dead” domain. Next, it reselects preferred paths and propagates updated paths to available prefixes through *update* messages.

4.2.3 RAE – CME interface

The RAE reports the results (information about paths) to the CME. The interface is defined by the *protobuf* structure.

This sending part of the interface consists of a single method in *cme_connection* class:

```
void
cme_connection::send_message(
boost::shared_ptr<comet::cmerae::GenericRequest>gm);
```

The receiving part of the interface consists of a single callback method in *session* class, which indicates that the connection to the CME has been reset using the *version* field in the *Reply* message:

```
void
rae_logic::notify_cme_reset();
```

Figure 7 presents the structure of the *GenericRequest* message.

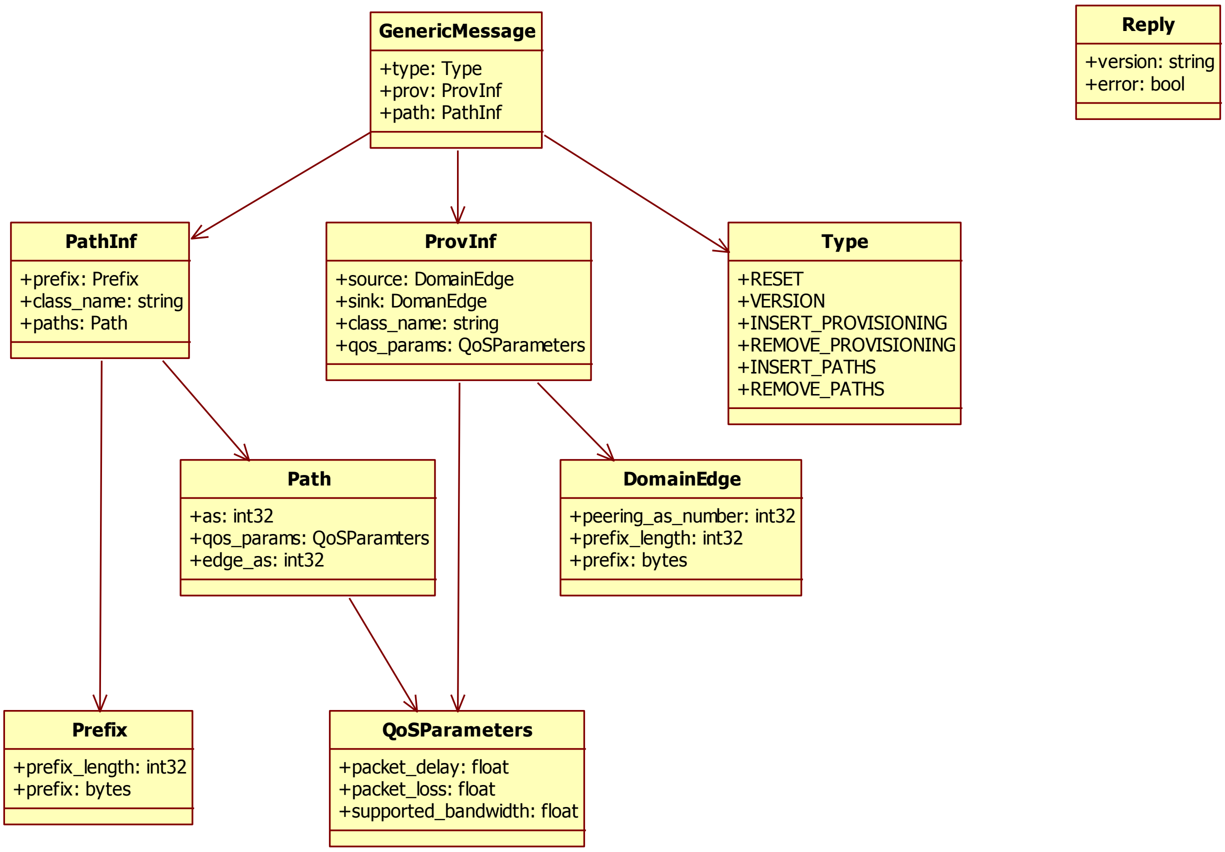


Figure 7: RAE-CME message structure

The *GenericMessage* includes information about paths towards available prefixes (*PathInf* message) and provisioning information about the local domain (*ProvInf* message).

4.3 Design

The RAE is implemented using the *boost::asio* library, which uses asynchronous network programming. Figure 8 shows the internal class structure of the RAE implementation.

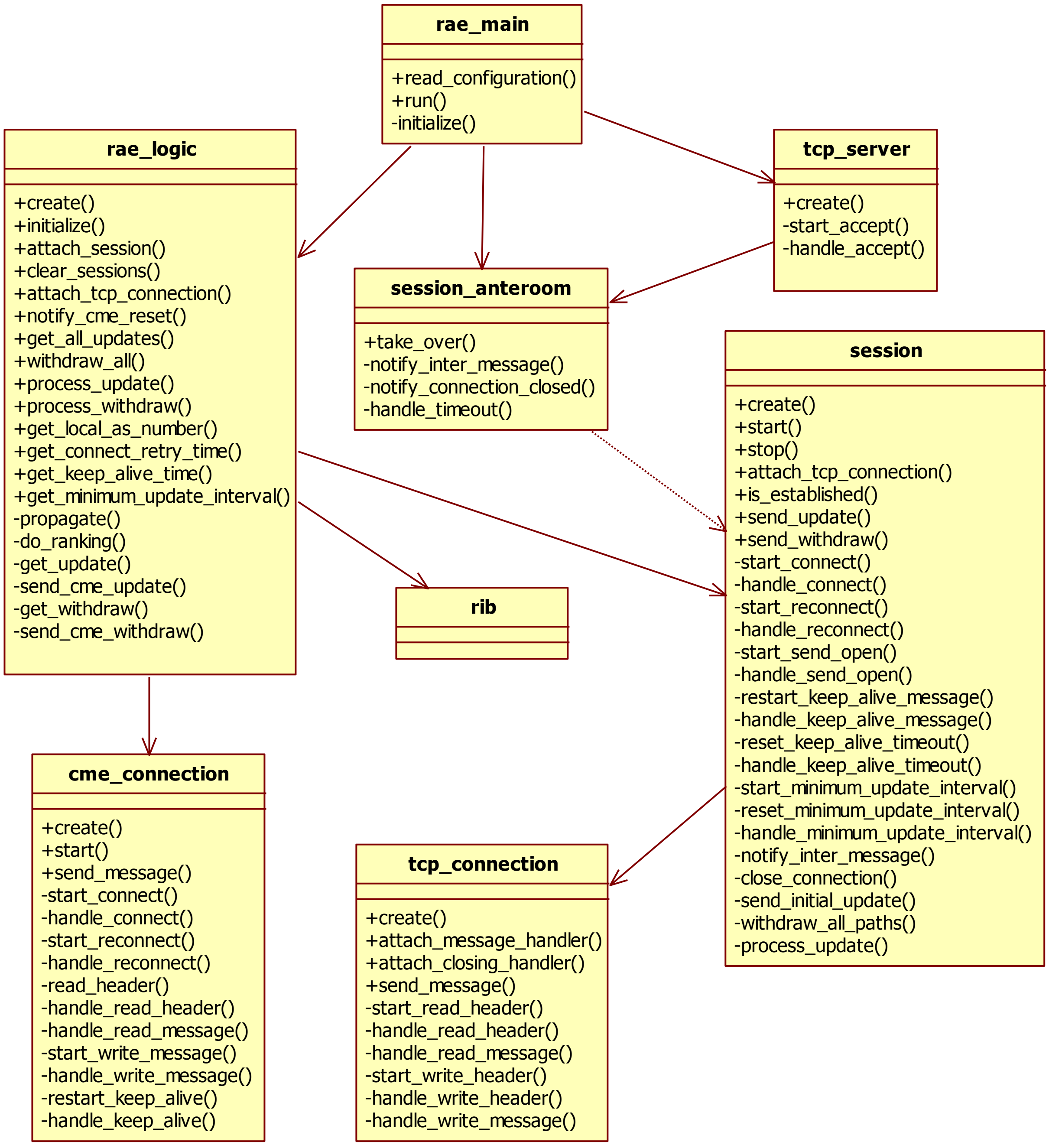


Figure 8: Internal class structure of the RAE

The *rae_main* class handles the starting and the configuration of the RAE. The *tcp_server* and *session_anteroom* classes process incoming TCP connection requests. The *tcp_connection* class handles incoming and outgoing TCP connections. The *session* class controls RAE to RAE sessions including: sending and receiving of *update/withdraw* messages and automatic management of *keepalive* messages exchange. The *rae_logic* class carries main logic of RAE covering:

- initiating of RAE to RAE sessions,

- processing of incoming *update* and *withdraw* messages,
- decision making process– selecting of preferred paths from available paths,
- propagating to peering RAEs updated information about paths and prefixes via *update* and *withdraw* messages,
- propagating to CME information about paths to available prefixes and about RAE's own domain provisioning information.

The *rib* class provides data structures for paths and prefixes. Finally, the *cme_connection* class handles the connection to the CME.

Most of the messages sent by the RAE are asynchronous, as depicted in Figure 9. Only session establishment (*open* message) requires confirmation from peering RAE.

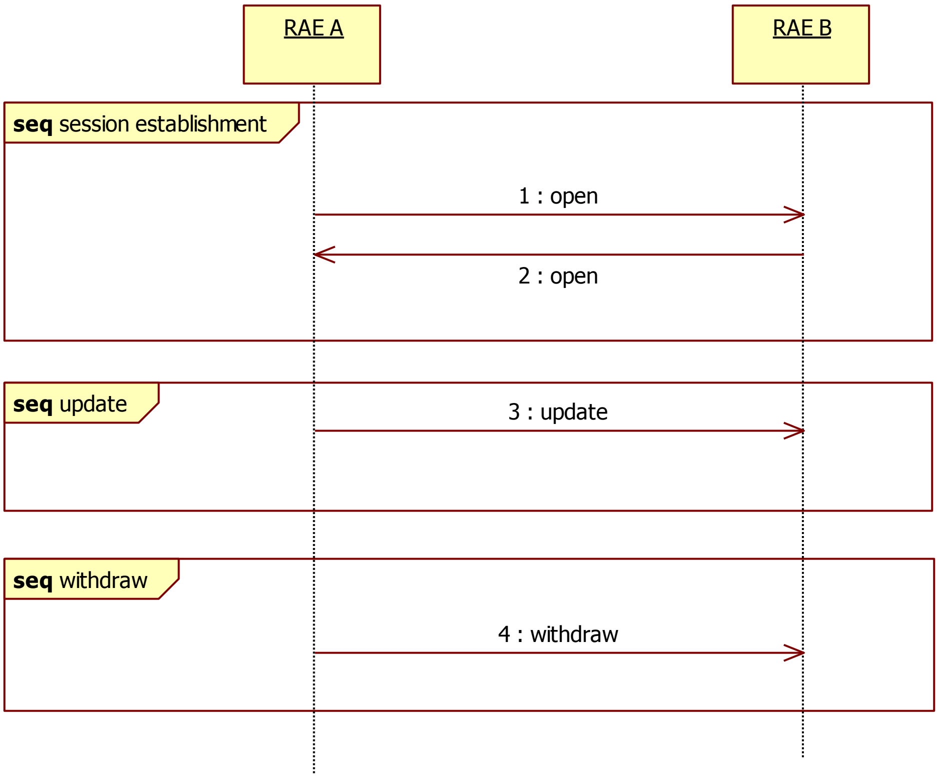


Figure 9: Message sequence diagram for inter RAE protocol.

4.4 Testing and Test scenarios

4.4.1 Validation tests

No automatic standalone tests are envisioned for the RAE. During implementation phase, we performed basic manual tests, which proved that RAE works according to expectations.

More advanced validation tests aimed to verify RAE functions corresponding to prefix advertisement, withdrawal, updating of provisioning information and reset/unavailability of RAE will be performed in the integration testbed. The objective of these tests is to verify interactions between RAEs as well as between RAE and CME. We defined three main tests cases:

Test 1: single-domain

The objective of this test is to verify whether the RAE provides correct information about network prefixes, intra-domain routing paths and CoS provisioning to the CME. The testbed network should correspond to a single domain with a single border router, a single access network and installed the RAE and the CME.

Test 2: multi-domain

The objective of this test is to verify whether RAEs located in different domains correctly advertise and withdraw information about network prefixes and routing paths. The topology consists of 7 domains creating a tiered topology.

Test 3: stress-test

This test aims to validate capabilities of RAE to handle large number of prefixes. The network should consist of two domains. The east domain hosts a variable number of prefixes, where the default count is 10000. The west domain receives the paths and propagates them to the CME.

The results of validation tests will be reported in forthcoming deliverable D5.1.

5 Stateless Content Aware Forwarding Entity

This chapter presents implementation issues of stateless Content Aware Forwarding Entity (CAFE). The stateless CAFE forwards packets based on information about the content delivery path stored inside the COMET header. The COMET header is attached and removed by edge CAFEs located close to the content server and the client, respectively. The content delivery path is selected during content resolution process and then it is configured in the edge CAFE during the path configuration process [4]. As a consequence, CAFEs maintain only the neighbourhood (local) information, i.e., how to forward a packet to the peering CAFEs.

Figure 10 presents the concept of stateless content delivery process. In this example, we assume simple network consisting of a client domain, a server domain and two transit domains. Let us assume that the path selected during the content resolution process goes through domains C-B-A, which is different than the IP routing path (C-D-A).

In order to deliver content through path C-B-A, the edge CAFE located in domain C intercepts the IP packets generated by the content server and encapsulate them with the COMET header. The CAFE includes the list of forwarding keys. Each forwarding key determines the next CAFE. Note that the forwarding key has only local meaning within a given CAFE. Consecutively, each CAFE on the content delivery path draws a successive forwarding key from the COMET header and encapsulates the packet following specific forwarding technology used between CAFEs. In our example, the forwarding key “2” enables forwarding of data packets from the edge CAFE located in domain C towards the CAFE located in domain B. Then, this CAFE uses the next forwarding key, “3”, to deliver packets to the edge CAFE located in domain A. Finally, the last CAFE removes the COMET header and sends the IP packet directly to its destination.

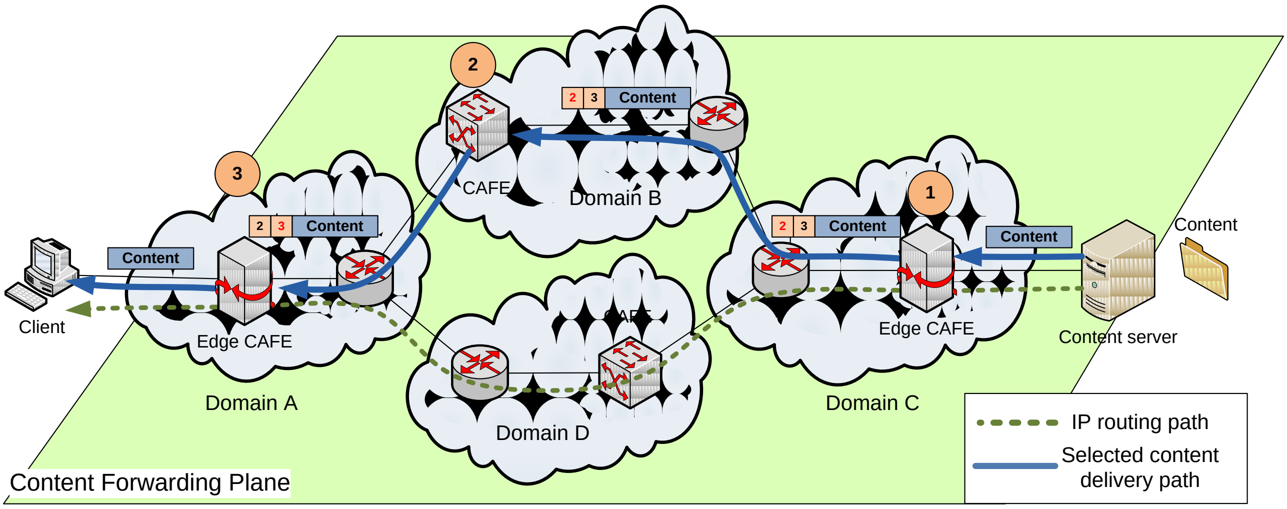


Figure 10: The concept of stateless content forwarding

The specification of stateless CAFE is included in deliverable D4.2 [1].

5.1 *Description of overall functionality*

As mentioned above, the CAFE is the specialised network node responsible for content forwarding from the content server to the client using the COMET specific forwarding method, called stateless content forwarding. The main functionalities of CAFE cover:

- Forwarding of content packets from input interface to queue at the output interface based on the information of content delivery path included in the COMET header and local forwarding rules included in CAFE forwarding table.

- Interception of IP packets received at the ingress edge node, classifying them according to configured filters and encapsulating them with appropriate COMET headers
- Decapsulation of IP packets carried in COMET packets at egress nodes.
- Measurements of traffic carried on content delivery paths.

The CAFE consists of the following modules:

- cafe_forwarder – this module is responsible for packet forwarding based on the COMET header,
- cafe_intercept – this modules is responsible for intercepting of IP packets and encapsulating them with COMET header (according to configured filters),
- cf_tool – this tool is used to manage the CAFE forwarding table and collect statistics about carried traffic,
- ci_tool – this tool manages the CAFE intercept function and allows to collect statistics about carried traffic,
- cafe_agent – this module is used as an interface between CME and CAFE. It receives configuration commands from CME and enforces them on CAFE using ci_tool or cf_tool. Moreover, it collects statistics from CAFE and provides this information to CME.

Figure 11 presents CAFE modules implemented in the Linux OS environment.

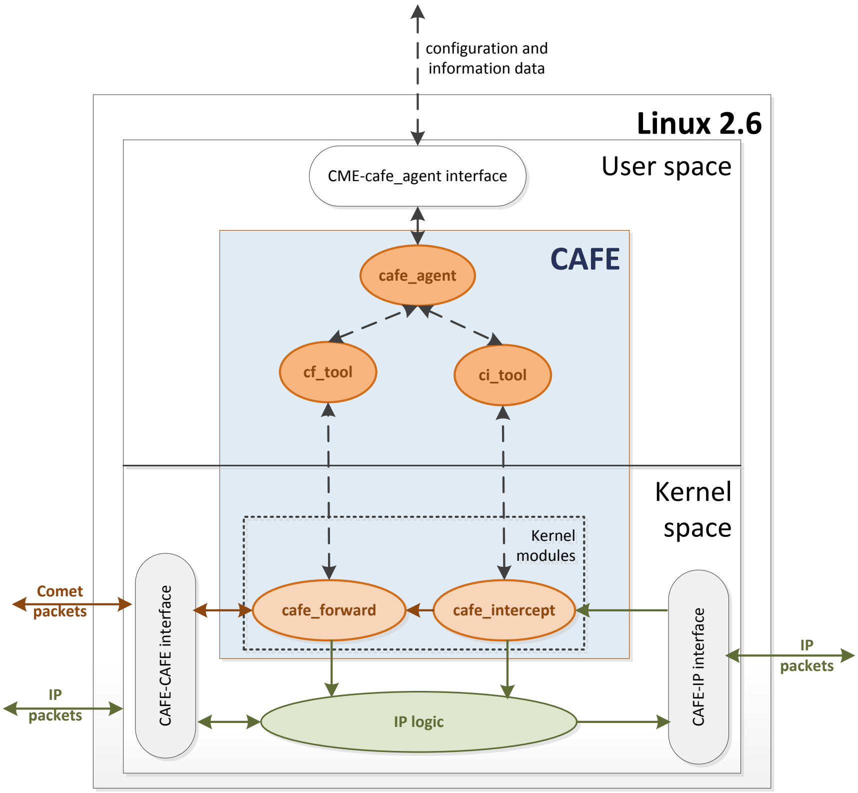


Figure 11: The CAFE modules in Linux environment

In order to assure effective content forwarding, the *cafe_forward* and *cafe_intercept* modules are implemented as loadable kernel modules for Linux kernel 2.6. The *cf_tool* and *ci_tool* are user space programs, which use *libnl* library to communicate with kernel modules. The cafe_agent was implemented in python 2.7 environment with the *protobuf* interface to CME.

5.2 Interfaces

In this section, we describe the CAFE interfaces. Basically, the stateless CAFE uses 3 interfaces that are:

- CAFE-CAFE: on this interface CAFEs exchange data packets, i.e. COMET packets and IP packets
- CAFE-IP router or access networks: on this interface edge CAFE intercepts IP packets, and encapsulate them with the COMET header
- CME-CAFE: this interface is used to receive configuration commands from the CME and configure CAFE by configuration tools. This interface is also used to report measurements performed by CAFE.

5.2.1 CAFE-CAFE

Once the Ethernet frame is received at network interface and placed in *skb buffer*, the kernel invokes the appropriate packet processing function for the protocol type encoded in EtherType filed. The CAFE uses this mechanism and defines its own protocol type with code *0xcccc*. The following packet processing functions were implemented:

- **`Cafe forward`**
  - o `intcf_skb_recv(structsk_buff *skb, structnet_device *dev, structpacket_type *ptype, structnet_device *orig_dev);` This function handles received COMET packet. It parses COMET header, finds forwarding key, which should be used for packet forwarding. Then, CAFE looks up the local forwarding table to find out the output interface and required layer 2 headers. Based on this information, the outgoing frame is prepared and placed in skb buffer. Finally, COMET packet is transferred to output queue by `dev_queue_xmit(skb).`
  - o `void cf_decapsulate(structsk_buff *skb, structcafe_header *ch);` This function is invoked at edge CAFE, when the last forwarding key is processed. In this case, the COMET header is removed and CAFE invokes IP packet processing by `netif_receive_skb(skb).`

5.2.2 CAFE-IP router or access network

This interface is used at edge CAFE to intercept IP packets and encapsulate them to the COMET header. For this purpose, we use the Linux kernel *netfilter* functions, which allow to modify packet processing. Once the IP packet is received at the interface, one of the following functions is invoked:

- **`CAFE intercept`**
  - o `unsigned int ci_nf_hook_ipv4(unsigned inthooknum, structsk_buff *skb, conststructnet_device *in, conststructnet_device *out, int (*okfn)(structsk_buff *));` This function handles IPv4 packets. If the IPv4 packet matches the preconfigured filter (also called as the interception rule), *cafe_intercept* adds the COMET header to the IP packet placed in skb buffer. Then, it invokes the *cafe_forward* process instead of standard IP processing. Finally, the intercepted packet is forwarded to appropriate output interface based on the COMET header.
  - o `static unsigned int ci_nf_hook_ipv6(unsigned inthooknum, structsk_buff *skb, conststructnet_device *in, conststructnet_device *out, int (*okfn)(structsk_buff *));` This function handles IPv6 packets. If the IPv6 packet matches the predefined filter (also called as the interception rule), *cafe_intercept* adds the COMET header to packet placed in skb buffer. Then, it invokes the *cafe_forward* process instead of

standard IP processing. Finally, the intercepted packet is forwarded to appropriate output interface based on the COMET header.

5.2.3 CME-CAFE management agent

The CME-CAFE management agent interface is used to configure edge CAFE by CME and collect measurements performed by CAFEs.

- **`CMEHandler:`** `handle(self):` This method handles configuration messages received from CME. If "CONFIGURE_STREAM" message is received, the handler invokes method `configure_stream().` On the other hand, when message "COLLECT_EXPIRED_STREAMS" is received, the handler invokes method `collect_expired_streams().`
    - `configure_stream(self, m):`This method is used to configure a stream to the respective CAFE.It returns CAFE_SUCCESS or CAFE_FAILURE depending on the operation status.
    - `collect_expired_streams(self, m):` This method is used to collect information about expired flows. It returns list of expired flows in the following form:
      <id><ip_source><ip_destination><protocol><port_source><port_destination><bandwidth><cos><as_path><transferred_bytes><duration>

5.3 *Design*

In this section, we present details of the designed software modules related to stateless CAFE. For each module, we present UML class diagram and sequence charts corresponding to basic operations.

5.3.1 cafe_forward module

Figure 12 presents UML class diagram corresponding to cafe_forward module.

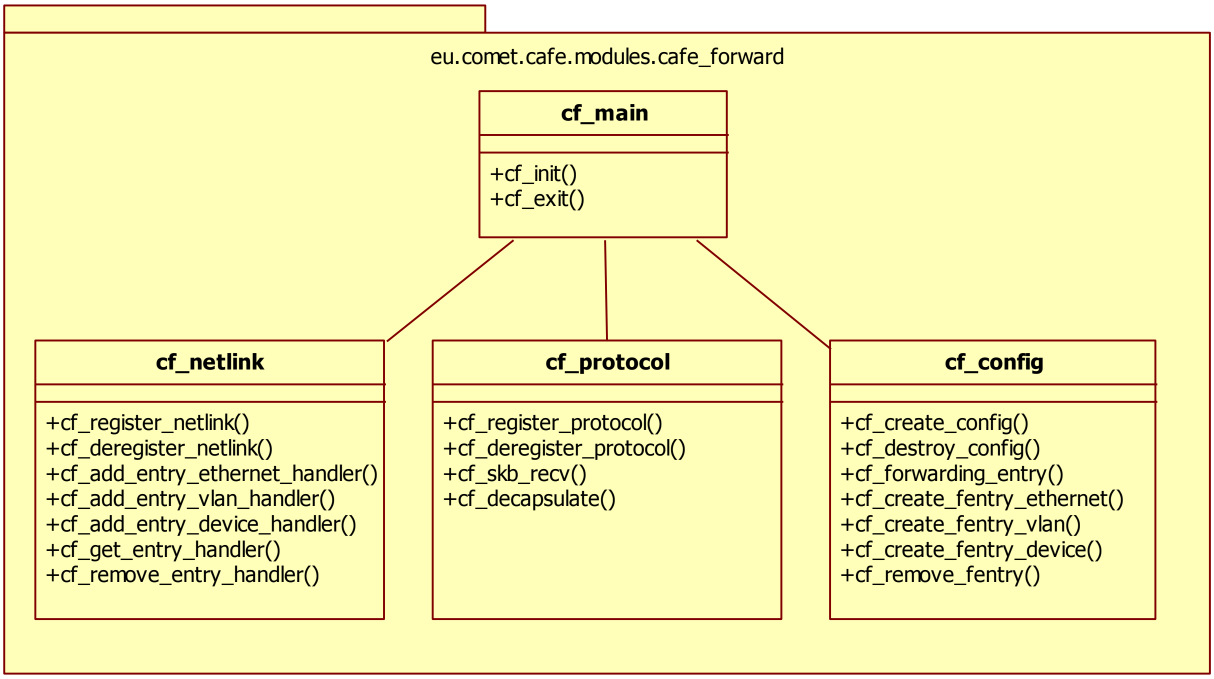


Figure 12: The class diagram of cafe_forward module

The cafe_forward module uses 4 sets of functions. The *cf_main* functions are invoked when cafe_forward module is inserted or removed from the Linux kernel. The *cf_netlink* function enables communication between the cafe_forward module and the Linux kernel network processing functions. The *cf_protocol* functions are responsible for forwarding of COMET packets based on the COMET header. The *cf_config* functions are responsible for management of CAFE forwarding table. They allow creating and removing forwarding rules. The COMET packets could be forwarded using three data link technologies; these are: Ethernet, VLAN Ethernet and GRE (*Generic Routing Encapsulation*) tunnelling.

Figure 13 presents the sequence diagram related to COMET packet forwarding in transit CAFE. Once CAFE receives the COMET packet, the *cf_skb_recv()* function is invoked. It processes the COMET header, finds current forwarding key in the list of forwarding keys, and then it looks up the CAFE forwarding table to find output interface and required layer 2 header. Finally, the cafe_forward module sends the COMET packet to the output interface.

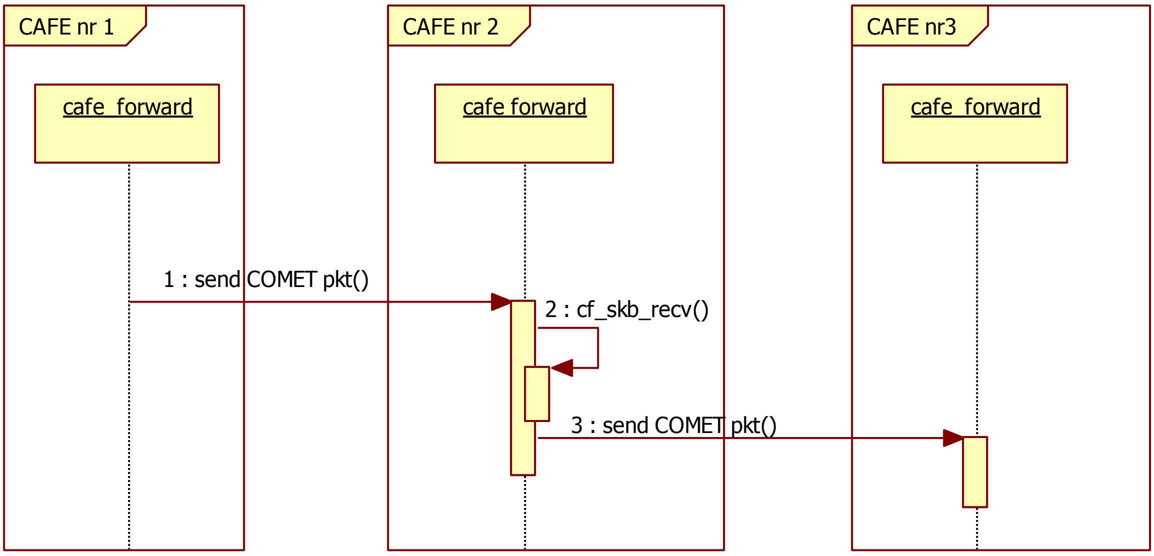


Figure 13: Sequence diagram related to packet forwarding in transit CAFE

Figure 14 presents the sequence diagram related to COMET packet forwarding in egress edge CAFE. Once edge CAFE receives the COMET packet, the *cf_skb_recv()* function is invoked. It detects that the last forwarding key has been used and it invokes *cf_decapsulate()* function to remove the COMET header. Then, the packet is handed over to standard IP processing, which forwards the packet based on IP routing table to the appropriate output interface.

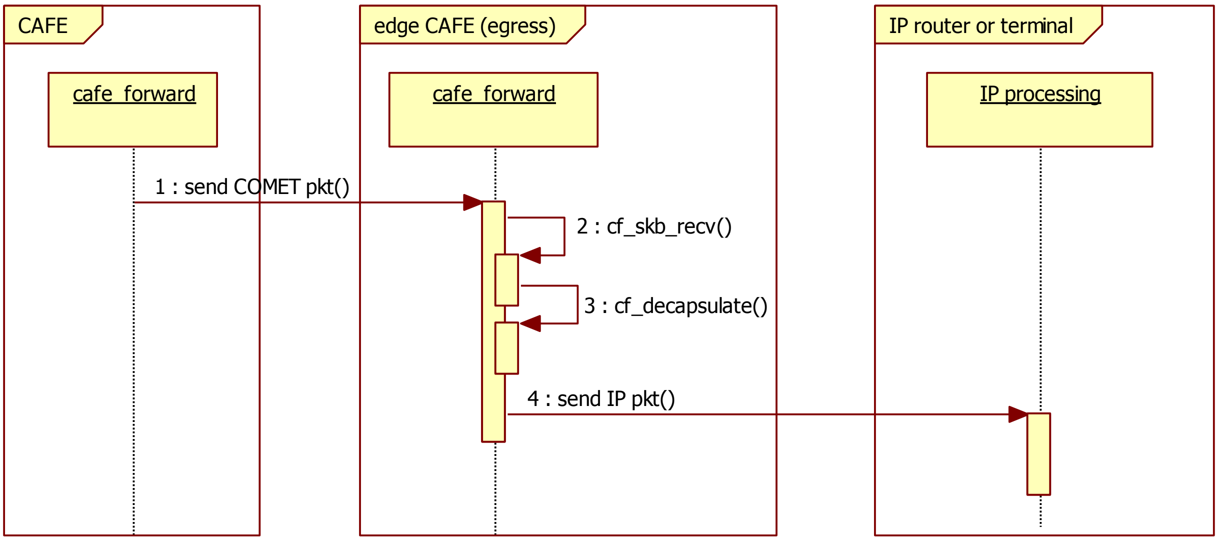


Figure 14: Sequence diagram related to packet forwarding in edge CAFE (egress point)

5.3.2 cafe_intercept module

Figure 15 presents the UML class diagram corresponding to cafe_intercept module.

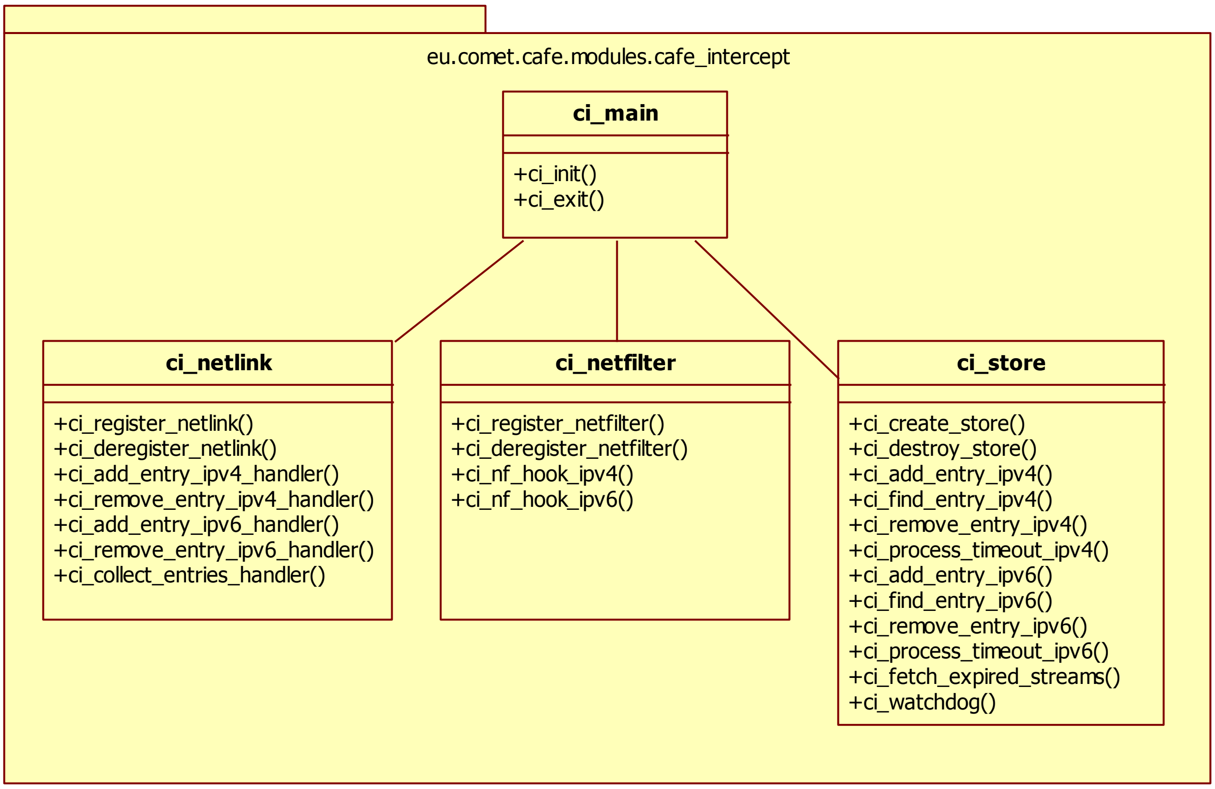


Figure 15: The class diagram of cafe_intercept module

The cafe_intercept module uses 4 sets of functions. The *ci_main* functions are invoked when cafe_intercept module is inserted or removed from the Linux kernel. The *ci_netlink* function enables communication between the cafe_intercept module and the Linux kernel network processing functions. The *ci_netfilter* functions are responsible for filtering intercepted IP packets. If a packet matches the preconfigured filter, it is enhanced with the COMET header and handed to cafe_forward module. The other packets are passed to standard IP processing. The *ci_store* functions are responsible for management of packet filters. They allow creating, lookup and removing packet filters. Moreover, these functions allow collecting packet level statistics related to expired streams.

Figure 16 presents the sequence diagram related to IP packet forwarding in ingress edge CAFE. Once it receives the IP packet, it uses the cafe_intercept module to apply packet filters. The packets matching the filter are handed to cafe_forward module, the other packets are handed to the standard IP processing.

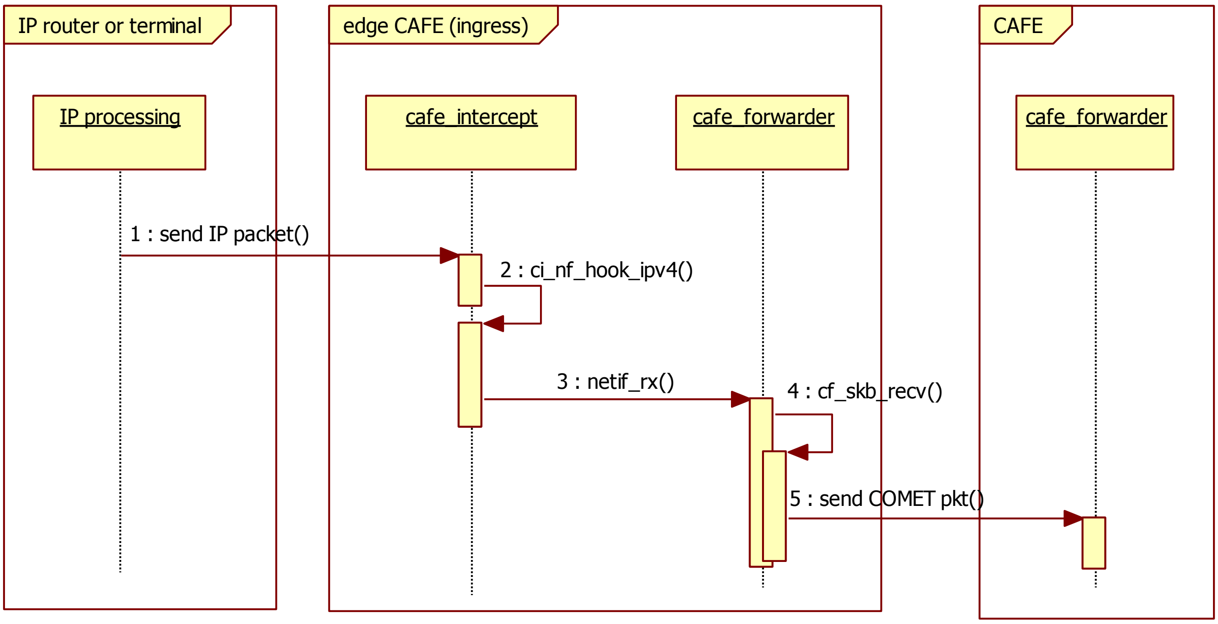


Figure 16: Sequence diagram related to packet forwarding in edge CAFE (ingress point)

5.3.3 Configuration tools

Figure 15 presents UML class diagram corresponding to CAFE configuration tools.

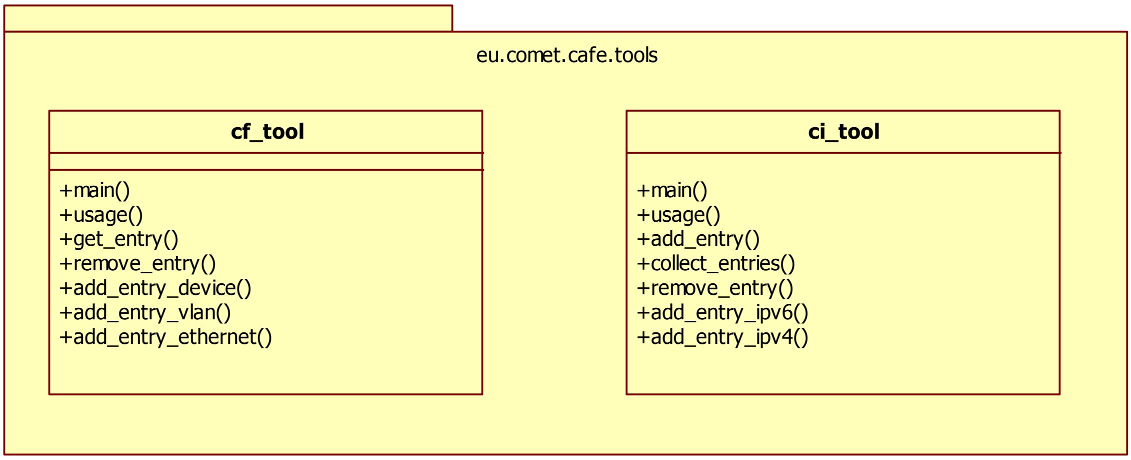


Figure 17: The class diagram for CAFE configuration tools

The CAFE requires two tools for configuration. The *cf_tool* is used to configure the CAFE forwarding table. It allows to add, remove or read configured forwarding rules. Below, we present how to use the *cf_tool*:

```
# How to add forwarding rule
     cf_tool add_ethernet <key> <device> <dst.ethernet.address>
     cf_tooladd_device <key> <device>


<key>    -    forwarding    key    (1    byte    -    unsigned    integer)
<device> - identifier of the network interface, e.g. eth0, gre0
<dst.ethernet.address> - destination MAC address of Ethernet frame


   # How to remove forwarding rule
     cf_tool remove <key>


   # How to read forwarding rule
     cf_tool get <key>
     cf_tool get
```

Some examples how to configure forwarding table for ethernet and gre interfaces:

```
     cf_tool add_ethernet a7 eth1 aa:11:b0:00:00:01
     cf_tool add_device 05 gre1
     cf_tool remove a7
     cf_tool get a1
```

The *ci_tool* is used to configure and remove packet filters at cafe_intercept module. This tool allows also to collect statistics about expired streams.

```
# How to add packet filter
     ci_tool  add  <id>  <timeout>  <src.ip.address>  <dst.ip.address>
     <protocol> <src.port> <dst.port> <list_of_forwarding_keys>


  # How to remove packet filter
     ci_tool remove <id>


  # How to collect statistics about expired flows
     ci_tool collect
```

Some examples how to configure /remove packet filer:

```
     ci_tool add 123 60 1.1.1.2 2.2.2.1 17 0 80 1234567890abcdef
     ci_tool remove 123
     ci_tool collect
```

Figure 18 presents the sequence diagram related to *cf_tool*. It presents interactions between the *cf_tool* and cafe_forwarder module related to adding, removing and reading forwarding rule from CAFE forwarding table.

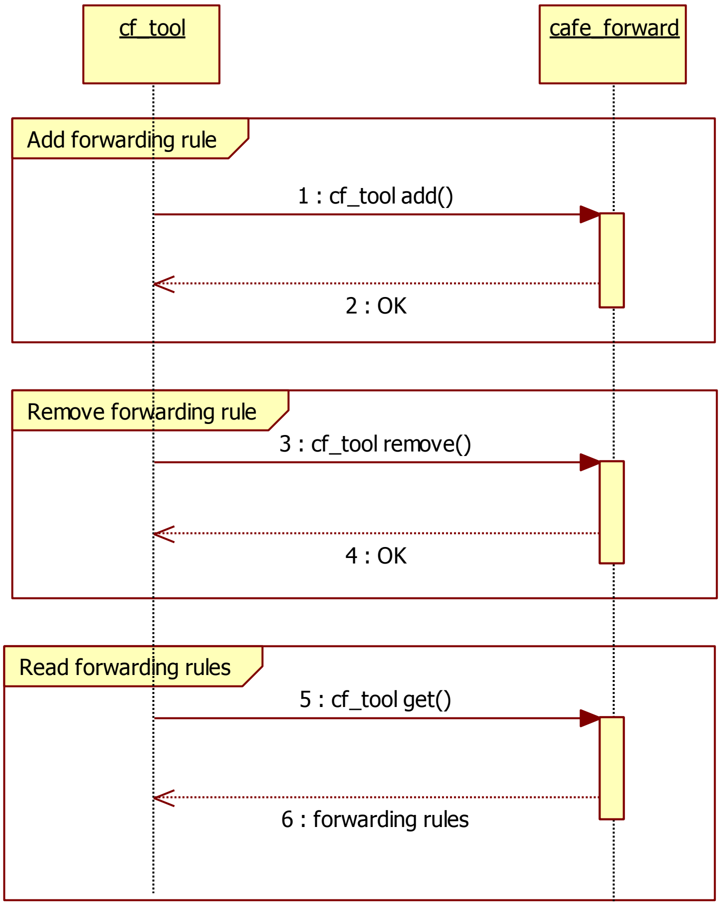


Figure 18: Sequence diagram related to *cf_tool*

Figure 19 presents the sequence diagram related to *ci_tool*. It presents interactions between the *ci_tool* and cafe_intercept module related to adding or removing packet filters. Moreover, the *ci_tool* allows for collecting statistics about expired streams.

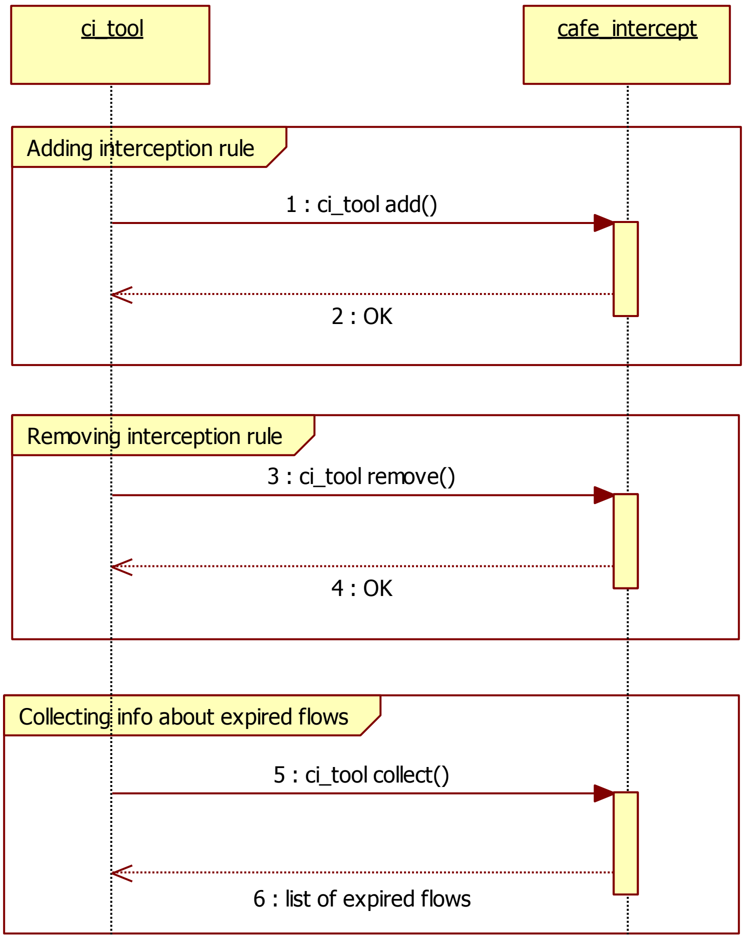


Figure 19: Sequence diagram related to *ci_tool*

5.3.4 CAFE management agent

Figure 20 presents UML class diagram corresponding to CAFE management agent.

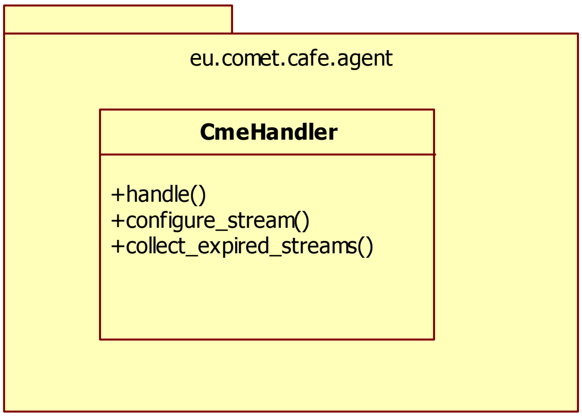


Figure 20: The class diagram for CAFE management agent

The CAFE management agent is used to configure CAFE based on the configuration commands received from CME module. The CAFE agent runs the CmeHandler::handle () method to listen for configuration commands. If "CONFIGURE_STREAM" message is received, the handler invokes the `configure_stream()` function to configure cafe_intercept module for a given packet stream. On the other hand, when message "COLLECT_EXPIRED_STREAMS" is received, the handler invokes function *collect_expired_strems ()* to read statistics from cafe_intercept module about expired streams.

Figure 21 presents the sequence diagram related to 2 operations performed by CAFE management agent that are: configuration of a new stream and collection of statistics about expired streams. The configuration commands, received via *protobuf,* are translated into invocation of ci_tool, which configures filter on cafe_intercept module.

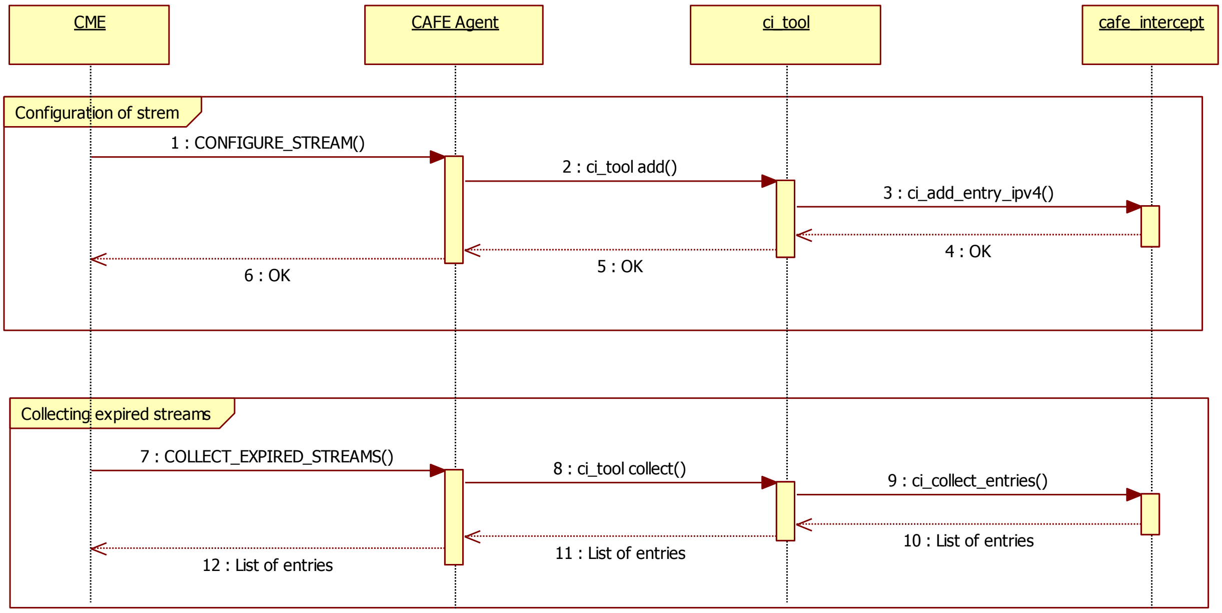


Figure 21: Sequence diagrams related to configuration of stream on edge CAFE and collecting information about expired streams

5.4 Testing and test scenarios

5.4.1 Validation tests

The validation tests aim to verify functionalities of standalone CAFE corresponding to:

- capabilities of CAFE agent to configure streams and collect statistics of expired streams,
- capabilities of cafe_intercept to encapsulate IP packets with COMET header based on preconfigured interception rule,
- capabilities of cafe_forward to handle COMET packets based on list of forwarding keys included in COMET header.

Note that automatic tests are envisioned only for CAFE agent. The rest of the validation tests were performed manually assuming the simple scenario: tester<->standalone CAFE.

5.4.2 Validation results

Below we present results of validation tests corresponding to standalone stateless CAFE.

Test Category	Test	TimeStamp/Owner	Test Description	Step	Expected	Actual	Result
1. CAFE Agent	1	17-02-2012/ptl+wut	Perform self test of CAFE configuration agent (test.py) (check configuration a 10 flows and then collection of the flows statistics)	Initialise CAFE agent. Compile and run test.py (Appendix B: CAFE agent test script) Script executes the following: • Connect to server • Collect streams • Receive response • Send configure stream • Receive response	Configuration performed. Receive success message for each configuration for all 10 flows.	Configuration performed. Receive success message for each configuration for all 10 flows.	PASS
2. CAFE intercept	2	17-02-2012/ptl+wut	Intercept based on src IP, dstIP, srcport, dstport, protocol number, with different forwarding keys, etc.	Configure CAFE with forwarding key, egress interface, and destination MAC address. Provide intercept parameters, source, destination, path keys etc. Capture frame on the source and destination edge CAFE. Analyze frame.	Frame intercepted based on necessary parameters. The COMET header contains the necessary forwarding data at the specified structure.	Frame intercepted based on necessary parameters. The COMET header contains the necessary forwarding data at the specified structure.	PASS
3. CAFE forward	3	10-02-2012/WUT	Forwarding of Comet frames based on forwarding key. Frame is forwarded via standard Ethernet interface.	Configure CAFE with forwarding key, egress interface, and destination MAC address. Send Comet frame (with proper key) to ingress interface. Capture frame on proper egress interface. Analyze egress frame.	Frame is forwarded via proper egress interface. Captured frame has modified: Comet header (index filed is incremented), source MAC address (source MAC address is set with egress interface's MAC address) and destination MAC address (destination MAC address is set as defined).	Frame is forwarded via proper egress interface. Captured frame has modified: Comet header (index filed is incremented), source MAC address (source MAC address is set with egress interface's MAC address) and destination MAC address (destination MAC address as defined).	PASS

Test Category	Test	TimeStamp/Owner	Test Description	Step	Expected	Actual	Result
	4	10-02-2012/WUT	Forwarding of Comet frames based on key. Frame is forwarded via VLAN interface.	Configure CAFE with forwarding key, egress interface, and destination MAC address. Configure egress VLAN interface. Send Comet frame (with proper key) to ingress interface. Capture frame on proper egress interface. Analyze egress frame.	Frame is forwarded via proper egress interface. Captured frame has added 802.1q header with configured VLAN Tag. Captured frame has modified: Comet header (index filed is incremented), source MAC address (source MAC address is set with egress interface’s MAC address) and destination MAC address (destination MAC address is set as defined).	Frame is forwarded via proper egress interface. Captured frame has added 802.1q header with configured VLAN Tag. Captured frame has modified: Comet header (index filed is incremented), source MAC address (source MAC address is set with egress interface’s MAC address) and destination MAC address (destination MAC address is set as defined).	**PASS**
	5	10-02-2012/WUT	Forwarding of Comet frames based on key. Frame is forwarded via GRE tunnel interface.	Configure CAFE with forwarding key. Configure egress GRE tunnel interface. Send Comet frame (with proper key) to ingress interface. Capture frame on proper egress interface. Analyze egress frame.	Frame is forwarded via proper egress interface. Captured frame has added GRE and IPV4 headers. Captured frame has modified: Comet header (index filed is incremented), source MAC address (source MAC address is set with egress interface’s MAC address) and destination MAC address (destination MAC address is resolved by ARP protocol).	Frame is forwarded via proper egress interface. Captured frame has added GRE and IPV4 headers. Captured frame has modified: Comet header (index filed is incremented), source MAC address (source MAC address is set with egress interface’s MAC address) and destination MAC address (destination MAC address is resolved by ARP protocol).	**PASS**
	6	10-02-2012/WUT	Decapsulation of Comet frames and forwarding de-capsulated IPv4 datagrams.	Configure IPv4 routing. Send IPv4 datagram encapsulated into Comet frame (with value of the *length* field equal to value of the *index* field) to ingress interface. Capture frame on proper egress interface. Analyze egress frame.	Frame is forwarded via proper egress interface (accordingly to IPv4 routing). Captured Ethernet frame encapsulates IPv4 datagram (no Comet header). Captured Ethernet frame has *ether_type* field set up with “0x0800”	Frame is forwarded via proper egress interface (accordingly to IPv4 routing). Captured Ethernet frame encapsulates IPv4 datagram (no Comet header). Captured Ethernet frame has *ether_type* field set up with “0x0800”	**PASS**
	7	10-02-2012/WUT	Decapsulation of Comet frames and forwarding de-capsulated IPv6 datagrams.	Configure IPv6 routing. Send IPv6 datagram encapsulated into Comet frame (with value of the *length* field equal to value of the *index* field) to ingress interface. Capture frame on proper egress interface. Analyze egress frame.	Frame is forwarded via proper egress interface (accordingly to IPv6 routing). Captured Ethernet frame encapsulates IPv6 datagram (no Comet header). Captured Ethernet frame has *ether_type* field set up with “0x86DD”	Frame is forwarded via proper egress interface (accordingly to IPv6 routing). Captured Ethernet frame encapsulates IPv6 datagram (no Comet header). Captured Ethernet frame has *ether_type* field set up with “0x86DD”	**PASS**

6 Stateful Content Aware Forwarding

This chapter describes the implementation of the stateful content-aware forwarding operations within the proof-of-concept emulator, which is based upon the technical specifications given in COMET deliverable, D4.2 [1]. While content forwarding is carried out by the CAFE, due to the coupled nature of the stateful approach, operations of other COMET entities related to content forwarding are also detailed in this section.

6.1 Overall functionality

Content-aware forwarding within the stateful approach consists of three distinct operations:

1. The installation of content states by CRMEs within the CAFEs on the path along which the content is being resolved. This is carried out during the content resolution phase specified in COMET deliverable, D3.2 [4].
2. The delivery of content, which follows the CAFE states installed during the content resolution phase. Content delivery is typically carried out in a hop-by-hop fashion, where 'hop' here refers to CAFE-level hops.
3. Optimisation of the route taken to deliver the content. Since content is typically resolved from the customer along its provider domains, the route taken tends to be long. Therefore, route optimisation aims to ensure content is delivered along the optimal route.

The primary entity involved in content delivery is the content-aware forwarding entity (CAFE), the architecture of which is shown in Figure 22. CAFEs contain three main logical components, namely:

Content State Table – stores state information relating to content streams that the CAFE is currently handling. The basic table structure is given in the next section.

Content Forwarding Engine – forwards content it receives from uphill CAFEs to downhill CAFEs, in accordance with the states contained in the Content State Table.

Content-Aware Forwarding Function (CAFF) – logically interfaces the CAFE with its local CRME to allow for installation of states and to send notifications of new content flows.

CAFEs also contain a number of logical interfaces, which are specified in the following subsection.

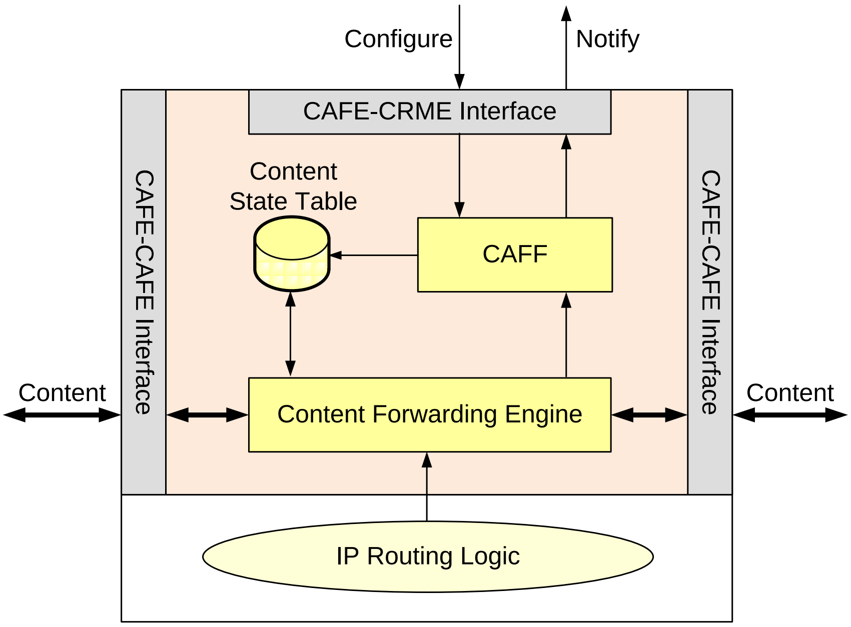


Figure 22: Internal stateful CAFE architecture

6.2 Interfaces

Communication between CAFE components and external entities is carried out over two main interface types: a CRME-CAFE interface and a CAFE-CAFE interface. TheCRME-CAFE interface connects the CAFE with its local CRME to allow it to send `Notify` messages and receive `Configure` messages. The CAFE-CAFE/Router interface connects the CAFE to its neighbouring CAFEs and routers, both within its *own* domain and within its *neighbouring* domains. In the following subsections, we provide a detailed description of both of these logical interfaces, and the communication methods used across them.

6.2.1 CAFE-CRME Interface

The CAFE-CRME interface is that over which control-plane messages are exchanged between the CRME and CAFE, namely, the `Announce`, `Configure`, and `Notify` messages, the descriptions of which are given in Table 3. Figure 23 shows the interaction between the CAFE and CRME, including the messages exchanged between them.

Table 3: CRME-CAFE interface messages

Message	Information Passed	Description
`Announce`	- `neighbourCAfeAddr` - `neighbourCafePort` - `neighbourCrmeAddr`	Sent by a CAFE to its local CRME when it is starting up, to advertise to the CRME its presence, and its connectivity with other COMET entities (other CRMEs or CAFEs).
`Configure`	- `contentID` - `nextHopCafeAddr` - `nextHopCafePort`	Sent by a CRME to its local CAFEs to install content states within it, in response to content resolution requests the CRME receives from its neighbouring CRMEs.
`Notify`	- `contentID` - `contentSrcAddr` - `prevHopCafeAddr` - `nextHopCafeAddr` - `nextHopCafePort`	Sent by a CAFE to its local CRME to notify it about new content that has begun to flow through after successfully resolving content to a content source.

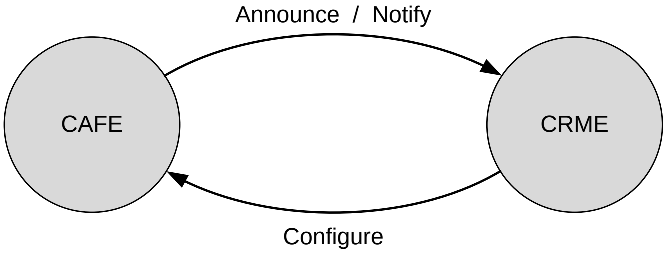


Figure 23: CRME-CAFE communication

6.2.2 CAFE-CAFE Interface

The CAFE-CAFE interface is a data-plane interface over which content is received from previous-hop CAFEs and forwarded to next-hop CAFEs along a content path. The header of each content chunk contains a number of important pieces of information, as shown in Table 4. Figure 24 shows the interaction that occurs between CAFEs, including the messages exchanged between them.

Table 4: CAFE-CAFE interface messages

Message	Information Passed	Description
`Content`	- `contentID` - `originAddr` - `prevCafeAddr` - `prevCafePort` - `<chunk of content>`	Sent either from one CAFE to the next-hop CAFE along a content delivery path, from a Content Publisher to the first CAFE on the content delivery path, or from the last-hop CAFE to the Content Consumer.

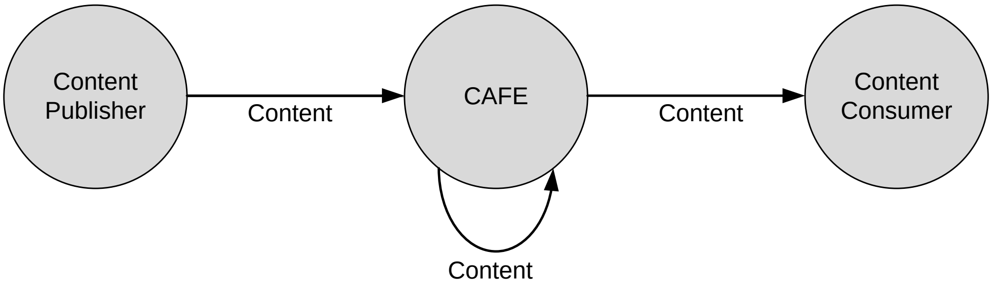


Figure 24: Content flow along CAFE interfaces

6.3 Design

6.3.1 Overall Content State Installation, Delivery and Route Optimisation

Figure 25 shows the implementation of CAFE state installation. Once a CRME receives a content consumption request from its counterpart in the previous hop domain, the `processConsume()` function is invoked, which carries out two main operations, namely to forward the `Consume` message towards the next-hop CRME, and to send `Configure` messages to both the local egress and ingress border CAFEs connecting the two neighbouring domains. Upon reception of the `Configure` messages by each CAFE, the `processConfigure()` function is invoked, which carries out the operation of installing the appropriate content state within their content state tables.

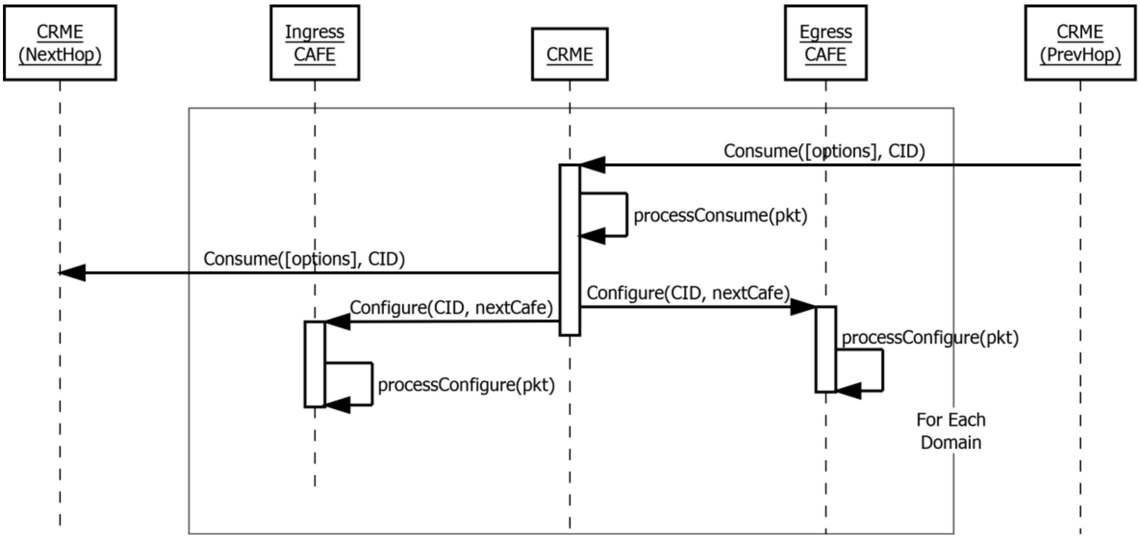


Figure 25: CAFE content state installation process in each domain

Figure 26 shows the content delivery process that takes place after the resolution phase. Once the content consumption request reaches the content server, the content server responds by forwarding the requested content to the ingress CAFE of the next-hop domain towards the client. Upon reception of this content at the ingress CAFE, the `processContent()` function is invoked, which primarily entails reading the ID of the content it has received and looking up in its content state table the next egress CAFE hop(s) within its domain to which to forward the content. When the egress CAFE(s) have received the content, it will carry out the same procedure as the ingress CAFE and forward the content to the ingress CAFE(s) of the neighbouring domain(s). The ingress and egress CAFEs in each domain will carry out the same process, until the content reaches the client.

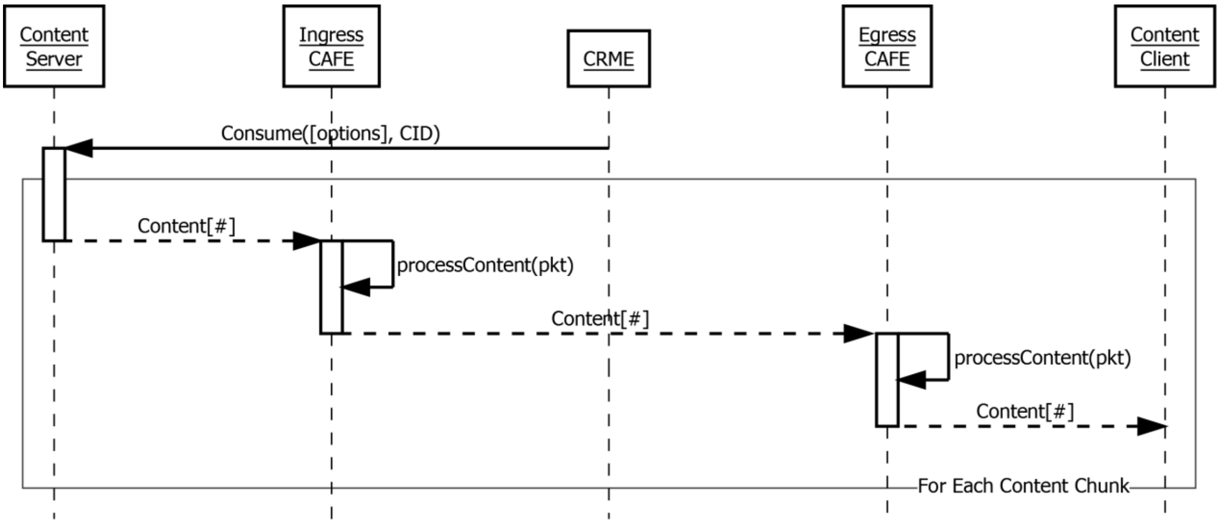


Figure 26: Content delivery process

Figure 27 shows the basic route optimisation process. For every content chunk that arrives at an ingress CAFE, the `processContent()` function that is invoked will also carry out a check of its active session table as to whether or not it needs to notify its local CRME about the arrival of a new content stream into the domain. If it does, it will send a `Notify` message to the CRME indicating the Content ID, the address of the content server, and the address of the egress CAFE of the previous domain-hop. Upon reception of this message, the CRME will in turn invoke the `processNotify()` function which will add an entry to its `ActiveSessions` table which it maintains for all Content flows it its domain, and will look up in its BGP routing table to check if the next-hop prefix towards the content server contained in the routing table differs from the address of the prefix of the previous-hop CAFE. If it does differ, the CRME will send a scoped `Consume` message to the next-hop CRME along the route-optimised (RO) path, and the resolution process will continue as before. The original CRME will also update its content ID entry in its content table with the route-optimised next-hop, so that future requests for the same content will follow the optimised path during the resolution phase.

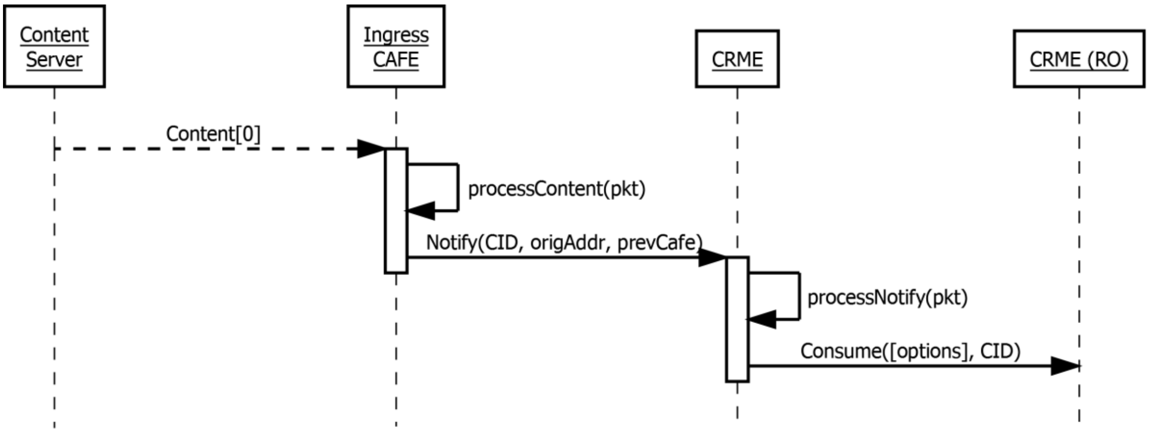


Figure 27: Route optimisation process

6.3.2 Class Diagram of Proof-of-concept Emulator

Figure 28 shows the class diagram of the proof-of-concept emulator, which consists of four main classes corresponding directly to the four main COMET entities used for the coupled approach. These classes are the `Crme`, `ContentPublisher`, `ContentConsumer`, and `Cafe`.

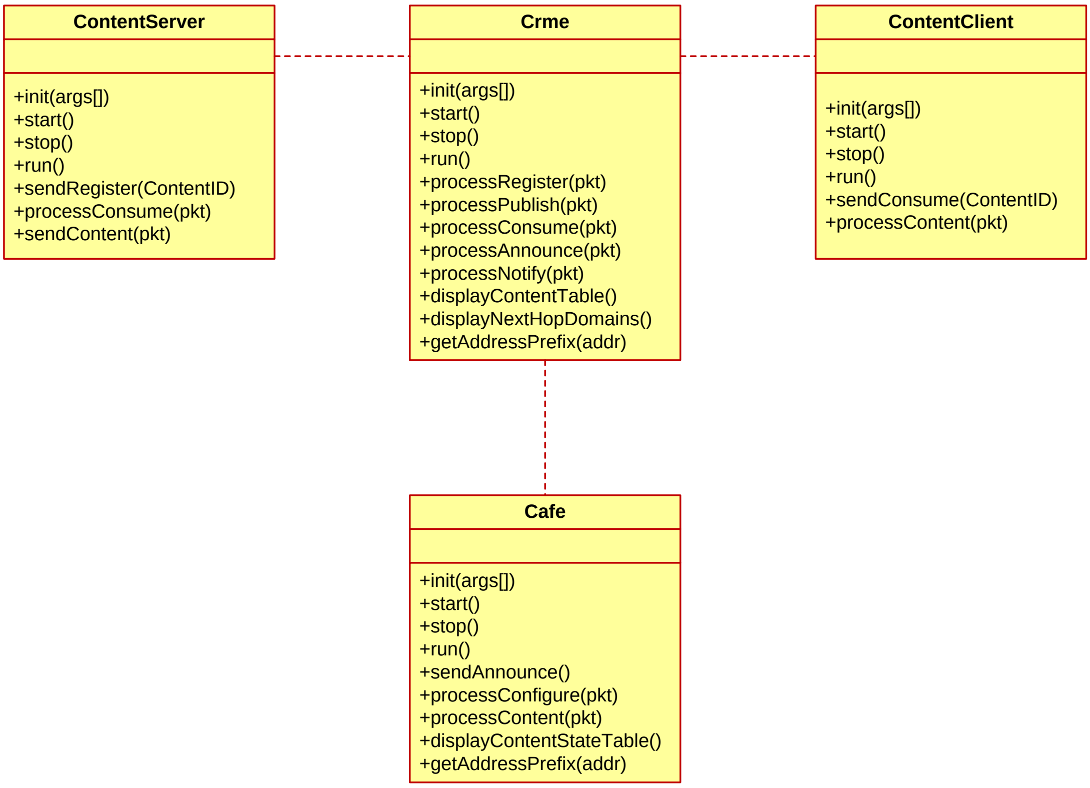


Figure 28: UML class diagram of COMET proof-of-concept emulator implementation

Common to each of these classes are four functions:

`init(args[])` – reads in the arguments from the script file to initialise the entity.

`start()` – opens the UDP sockets over which communication to other COMET entities is to take place.

stop() – closes all UDP sockets. This function is invoked when all commands in the control scrip have been exexuted.

run() – carries out the main run routine. This routine listens for messages received over its interfaces and responds to messages it receives accordingly. The function runs until the stop() function is invoked.

The following subsections will outline the functions related to the content forwarding plane operations.

6.3.2.1 Crme Functions

The `Crme` class contains three main functions loosely related to the content forwarding plane:

processConsume(pkt) – carries out content resolution and sends a `Configure` message to the ingress and egress CAFEs for the appropriate content states to be installed.

processAnnounce(pkt) – processes `Announce` messages received from its local CAFEs and updates its `locCAFE` tables accordingly.

processNotify(pkt) – processes `Notify` messages received from a local ingress CAFE and updates its `ActiveSessionsTable` with information about the new content flow. It also carries out route optimisation if a more optimal route exists.

getAddressPrefix(addr) – a helper function to extract the prefix from an IP address which is then used routing table lookups for the purpose of route optimisation.

6.3.2.2 ContentServer Functions

The `ContentServer` class contains one main function related to content forwarding:

sendContent(pkt) – sends content to the next hop CAFE towards the content client.

6.3.2.3 ContentClient Functions

The `ContentClient` class contains one main function related to content forwarding:

processContent(pkt) – processes content it receives, saving it to disk in its own content directory.

6.3.2.4 Cafe Functions

The `Cafe` class contains three main functions related to content forwarding:

sendAnnounce() – sends an `Announce` packet to its local CRME.

processConfigure(pkt) – processes `Configure` messages it receives by installing state in its `ContentStateTable`.

processContent(pkt) – processes content chunks it receives by looking up from the `ContentStateTable` the next CAFE hop towards the content client based on the `ContentID`, and transmitting the content over the correct UDP socket.

The following functions are specific to the proof-of-concept emulator for the purpose of textual display during run time:

displayContentStateTable() – prints to screen all entries in its `ContentStateTable`.

6.3.3 CRME Design

Figure 29 shows a UML state diagram illustrating the design of the CRME class, in particular the parts of that class relating to content delivery and route optimisation. The parts related to content publication and resolution are given in COMET deliverable, D3.3 [3]. When the CRME is in its idle state, it will be listening for CAFE messages, namely, an `Announce` or a `Notify` message. If the former is received, the CRME will update its internal configuration table (`locCafe` and `nextHopCrmes`) with the respective IP addresses.

If the CRME receives a `Notify` message, it will first determine whether the notification relates to a new session ('create') or a terminated session ('teardown'). In the latter case, the CRME will simply remove the corresponding entry for the corresponding Content ID from the CRME's `activeSessions` table and then return to the idle (listening) state. In the former case, it will check its BGP routing table to see if a more optimal route exists towards the content source of the given Content ID. If one does exist, it will forward a `Consume` message along it, and then add a corresponding entry to the **active sessions** table. If an optimal route does not exist, the CRME will proceed straight away to add an entry to the active sessions table, thereafter returning to the idle state.

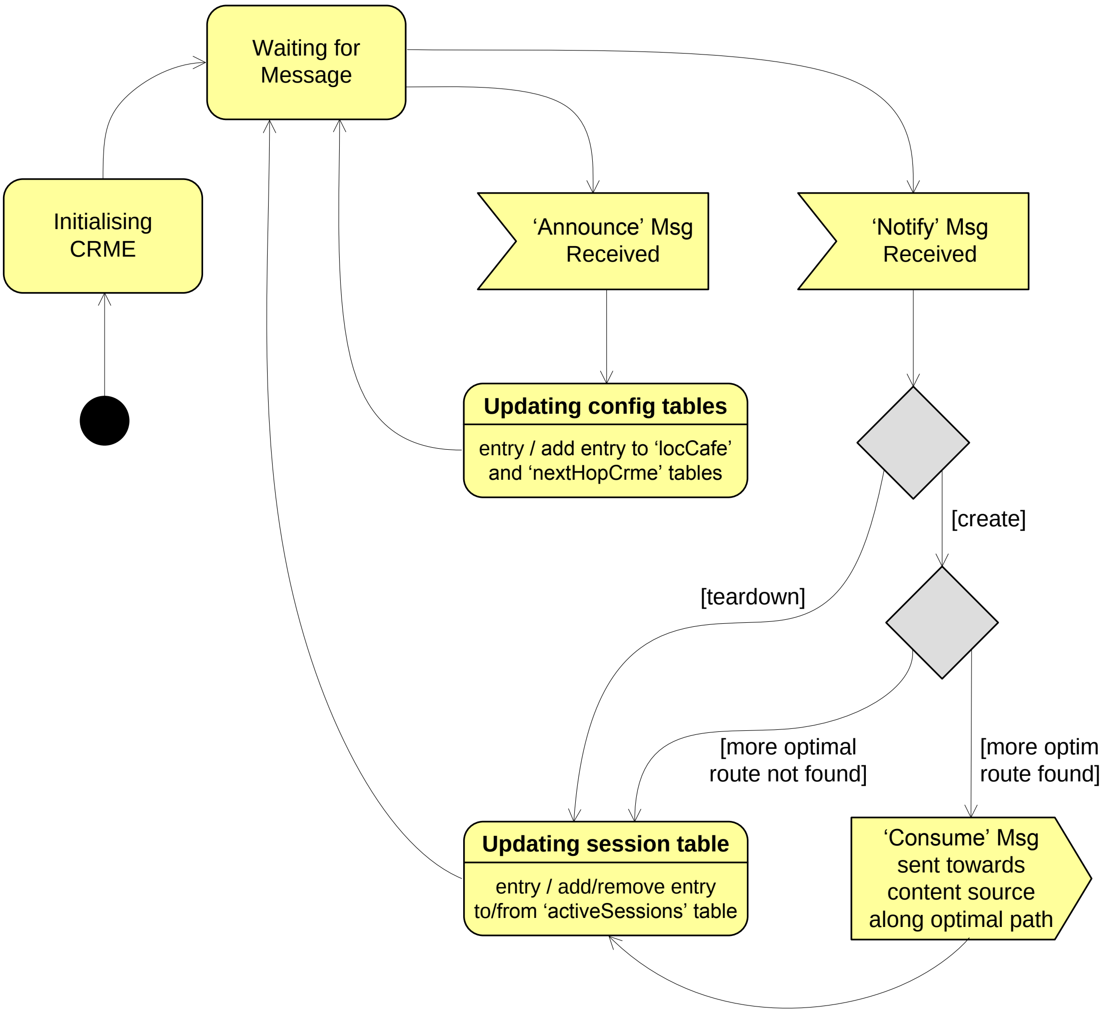

Waiting for Message
Initialising CRME
'Announce' Msg Received
'Notify' Msg Received
Updating config tables
entry / add entry to 'locCafe' and 'nextHopCrme' tables
[create]
[teardown]
[more optimal route not found]
[more optimal route found]
Updating session table
entry / add/remove entry to/from 'activeSessions' table
'Consume' Msg sent towards content source along optimal path

Figure 29: UML state diagram for content delivery and optimisation aspects of the CRME

6.3.3.1 Starting the CRME

Details on how the CRME is started and initialised in the proof-of-concept emulator are given in COMET deliverable D3.3 [3].

6.3.3.2 CRME Tables related to Content Delivery

To ensure that content requests are resolved only so far as to reach an existing a content path already established for content with the same identifier, the CRME maintains an ActiveSessions table. This table contains entries for all content flowing through each of the CAFEs it controls, and contains the following fields:

ContentID	<IngressCAFEaddr, IngressCAFEport>	numEgressCAFEs	<EgressCAFEaddr, EgressCAFEport>, ...

where:

ContentID = the ID of the content;

<IngressCAFEaddr, IngressCAFEport> = the address and port number of the ingress CAFE receiving the content with the given ContentID;

numEgressCAFEs = the number of egress CAFEs to which the content with the given ContentID is being forwarded;

<EgressCAFEaddr, EgressCAFEport> = the address and port number of the egress CAFE to which content is forwarded. The number address/port entries is indicated by the previous field (numEgressCAFEs).

6.3.4 CAFE Design

Figure 30 shows a UML state diagram illustrating the design of the CAFE class.

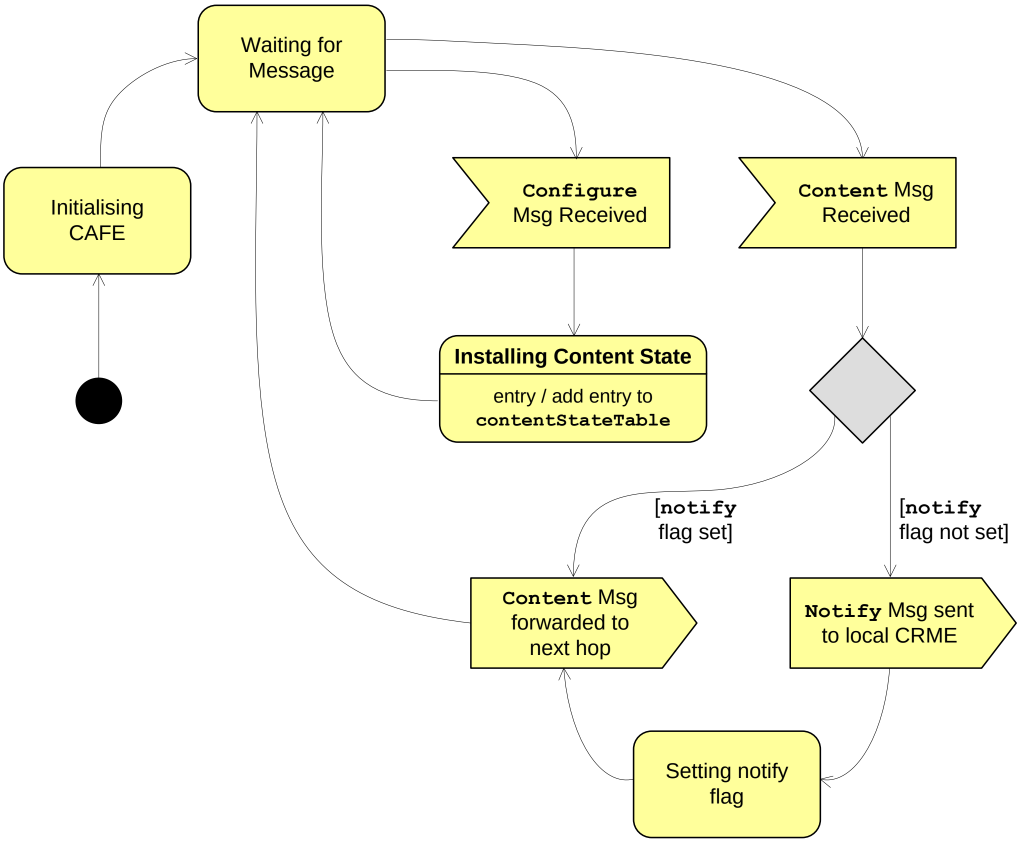


Figure 30: UML state diagram for content delivery aspects of the CAFE

When the CAFE is in its idle state, it will be listening for both control- and data-plane messages, namely, `Configure` and `Content` messages, respectively. If a `Configure` message is received, the CAFE will install states within its content state table (`ContentStateTable`) associating a content ID with a list of next-hop addresses to which to forward the content.

If the CAFE receives a `Content` message, it will first determine whether a notification related to this received content ID was previously sent to the CAFE's local CRME. Such notification state is related not only to the content ID, but also to the previous-hop CAFE address and port number from which this content was forwarded. If the notify flag is not set (i.e. a notification was not previously sent for the given content ID and previous-hop CAFE address/port) the CAFE will proceed to send a Notify message to the CRME, containing the information shown in Table 3. The notify flag is then set to ensure that future notifications are not sent for content with the same ID received from the same previous-hop CAFE address and port number. If the notify flag is already set, the CAFE will not send a Notify message. Then, whether or not the notify message was set, the CAFE will forward the content to the next-hop, as dictated by the state within the content state table.

6.3.4.1 Starting the CAFE

The CAFE entity is started using the following command format (in the startup xml script):

```
+time ON_ROUTER router_name usr.curling.Cafe myport myCRMEaddr myCRMEport
nInterDLink <neighbourCAFEaddr neighbourCAFEport neighbourCRMEaddr>
```

where:

`router_name` = the address or name of the current virtual router on which the CAFE resides;

`myport` = the port used by this CAFE;

`myCRMEaddr` = the address of the local CRME to which the CAFE is attached;

`myCRMEport` = the port of the local CRME to which the CAFE is attached;

`nInterDLink` = the number of inter-domain links stemming from the CAFE;

`<neighbourCAFEaddr neighbourCAFEport neighbourCRMEaddr>` = nInterDLink x tuple of the neighbour CAFE address, the neighbour CAFE port and the corresponding address of the neighbour CRME.

6.3.4.2 CAFE Tables

In the implementation of the CAFE entity, there exists one main CURLING-related table, which is the **content state table**, as shown below:

ContentID	PrevHopCAFEaddr	numNextHops	<NextHopCAFEaddr, NextHopCAFEport>, ...

where:

`ContentID` = the ID of the content;

`PrevHopCAFEaddr` = the address of the previous hop CAFE from which content was received;

`numNextHops` = the number of next hops to which the content represented by `ContentID` is forwarded;

`<NextHopCAFEaddr, NextHopCAFEport>` = the address and port number of a next hop CAFE to which content is forwarded.

6.4 *Testing and test scenarios*

The testing of the emulator will be carried out on each of the basic individual features of content delivery, namely, content state installation and content delivery. The following subsections detail each of the tests that will be carried out, including the validation criteria of each test. Note that the relevant validation tests of the content publication and resolution processes will be carried out separately and documented in D3.3 [3]. The validation of the overall system and advanced features such as route optimization will be carried out in the future, and documented in D5.1.

6.4.1 Validation tests

To test the installation of states within CAFEs, the inter-domain topologies shown in Figure 31 and Figure 32 will be used. These test topologies involve a content consumer attached at one point of the network issuing a request from content hosted at another point within the network which happens to be already published to the COMET system. The tests aim to validate that

- Content states are installed only in the appropriate CAFEs. So, for the tier-1 domain in the case of the topology shown in Figure 32, no content states should be installed in CAFE 1.3 or in any CAFEs under it.
- The correct handling ofConsumemessages at the content client.
- The requested content passes only through the CAFEs in which a corresponding content state is present.
- Content is delivered successfully from content server to content client, and stored correctly at the content client.

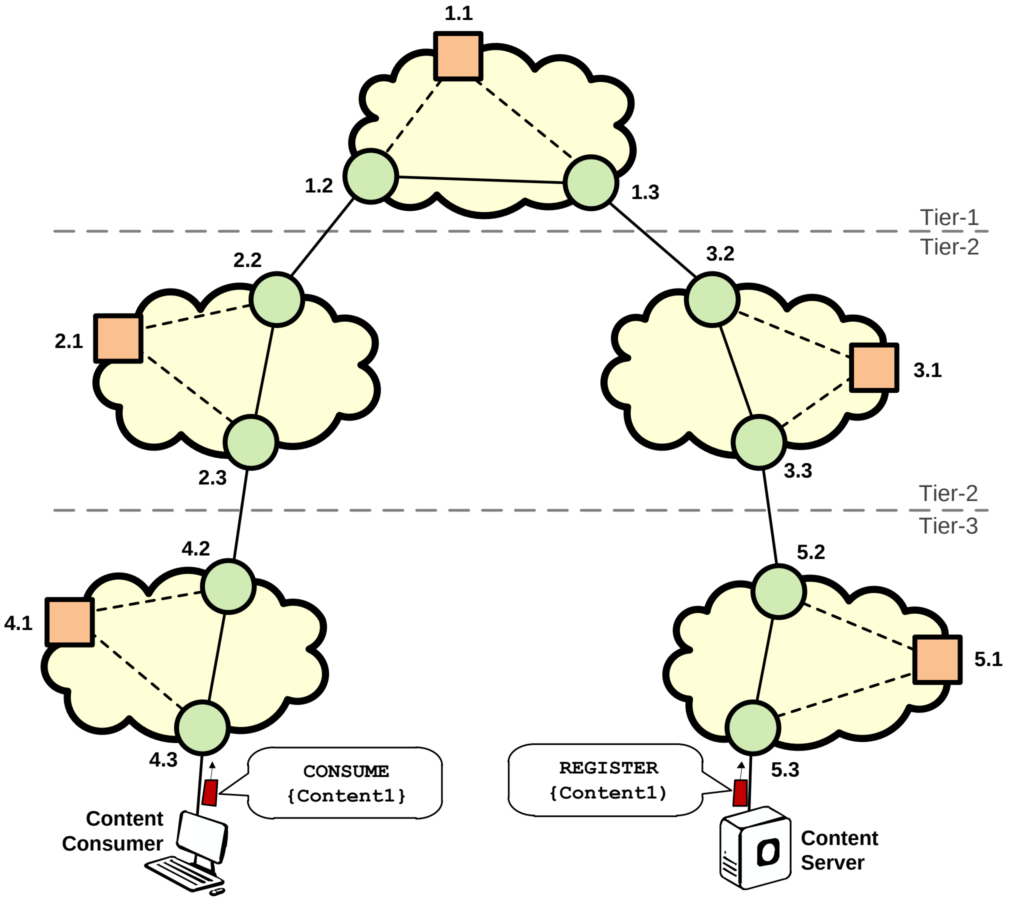


Figure 31: Topology to test basic state installation and content delivery (topology 1)

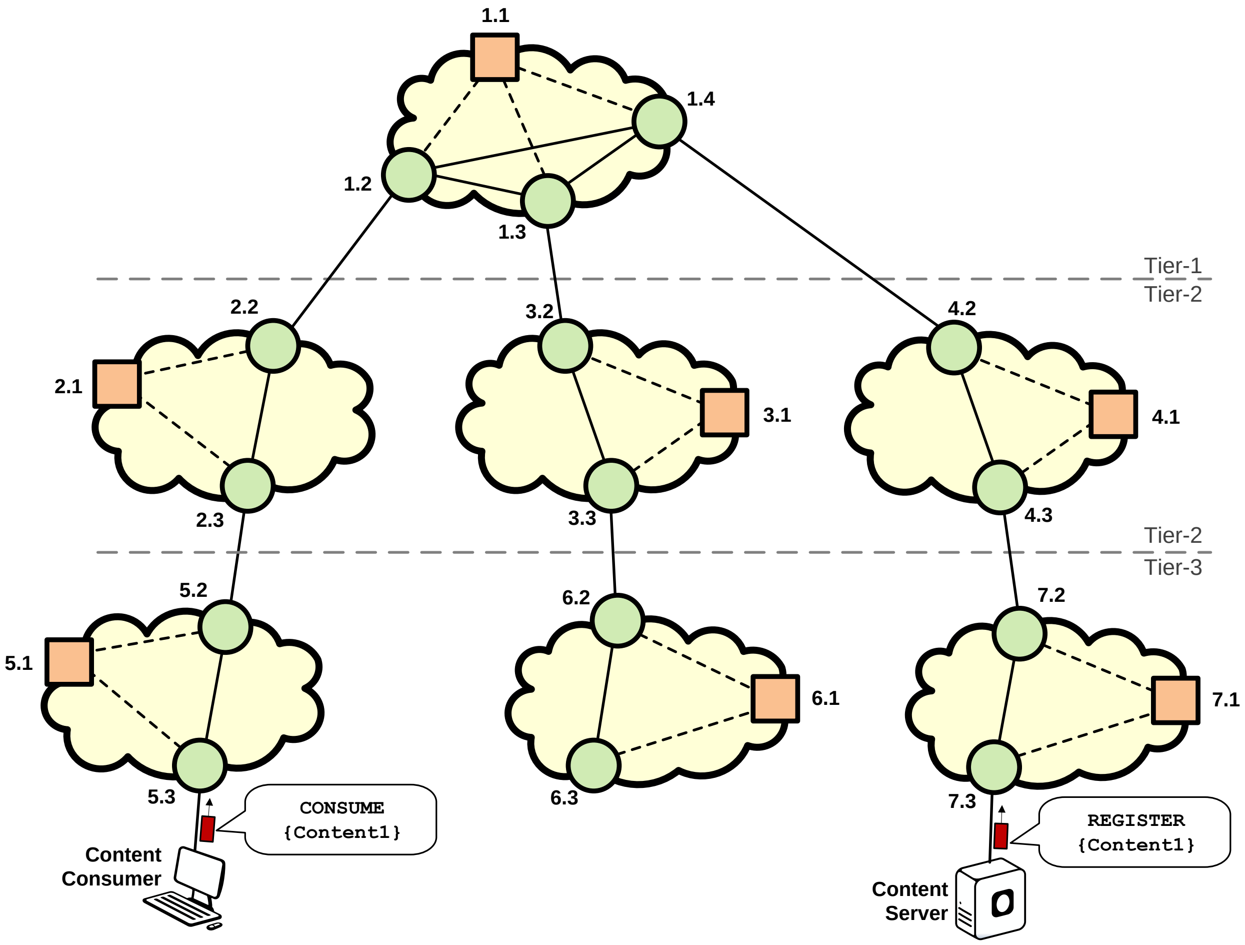


Figure 32: Topology to test basic state installation and content delivery (topology 2)

6.4.2 Validation results

This table details the validation procedures and results for basic content state installation and delivery, which was carried out using the implementation of the proof-of-concept emulator. These tests were carried out based on the two topologies shown in the previous subsection (Figure 31 and Figure 32).

Test Category	Test	TimeStamp/ Owner	Test Description	Step	Output / Result	Result
1. Content State Installation	1	07-02-2012/georgekamel	Basic state installation (topology 1)	Reception of Configure message by CAFEs from CRME	The following content state tables are installed in the domain with prefix 2: CAFE ContentStateTable at 2.2: Content ID | Next Hop Address | Next Hop Port MyContent1 2.3 2000 CAFE ContentStateTable at 2.3: Content ID | Next Hop Address | Next Hop Port MyContent1 4.2 2000 Similar content state tables are installed in domains with prefix 1, 3, 4, and 5.	PASS
				State installation in CAFE content state table		
	2	09-02-2012/georgekamel	Basic state installation when more than two CAFEs are present within a single domain (topology 2)	Reception of Configure message by CAFEs from CRME	The following content state tables are installed in the domain with prefix 1: CAFE ContentStateTable at 1.2: Content ID | Next Hop Address | Next Hop Port MyContent1 2.2 2000 CAFE ContentStateTable at 1.4: Content ID | Next Hop Address | Next Hop Port MyContent1 1.2 2000 No content state table is installed at CAFE 1.3.	PASS
				State installation in CAFE content state table		

Test Category	Test	TimeStamp/ Owner	Test Description	Step	Output / Result	Result
2. Content Delivery	3	06-02-2012/georgekamel	Basic content delivery (topology 1)	Look up next hop from CAFE content state table	CAFE 5.3: Content MyContent1 received and will be sent to 5.2:2000 CAFE 5.2: Content MyContent1 received and will be sent to 3.3:2000 ... CAFE 4.2: Content MyContent1 received and will be sent to 4.3:2000 CAFE 4.3: Content MyContent1 received and will be sent to 4.3:1000 CC: Content MyContent1 received and saved to: ./Content/4.3 1000/MyContent1	PASS
				Forward content to next hop		
	4	09-02-2012/georgekamel	Basic content delivery when more than two CAFEs are present within a single domain (topology 2)	Look up next hop from CAFE content state table	CAFE 7.3: Content MyContent1 received and will be sent to 7.2:2000 CAFE 7.2: Content MyContent1 received and will be sent to 4.3:2000 ... CAFE 5.2: Content MyContent1 received and will be sent to 5.3:2000 CAFE 5.3: Content MyContent1 received and will be sent to 5.3:1000 CC: Content MyContent1 received and saved to: ./Content/5.3 1000/MyContent1	PASS
				Forward content to next hop		

7 Summary

This deliverable presented the software design and implementation of entities developed for Content Forwarding Plane (CFP) of the COMET system. In the COMET architecture [1], the CFP is responsible for content delivery from content server to content consumer. It covers three entities that are: Routing Awareness Entity (RAE) and stateless Content Aware Forwarding Entity (CAFE) and stateful CAFE. These entities were developed following specification provided in COMET deliverable, D4.2 [2].

In this deliverable we presented details of developed entities. For each entity, we presented its internal and external interfaces with other COMET entities, UML class diagram related to internal components as well as sequence diagrams related to basic operations and interactions between components. Moreover, we defined and performed basic validation tests to verify functionalities of implemented software.

In particular in this document, we presented outline of implemented network entities, summary of interfaces with other COMET system entities and briefly discussed deployment issues related to developed entities. The implementation of RAE was presented in chapter 3. The RAE was implemented in C++ as a standalone entity with interfaces to CME, cooperating RAE and domain management system. The stateless CAFE was described in chapter 4. It was implemented as two loadable Linux kernel modules, *cafe_forward* and cafe_intercept. The *cafe_forward* module forwards COMET packets containing content, while the *cafe_intercept* module is responsible for encapsulation of IP packets received at edge nodes into COMET packets. Section 5 has described the implementation of content state maintenance and delivery, as well as route optimisation performed by stateful CAFE within the coupled approach. A number of tests have been outlined which aim to ensure correct functionality of the coupled approach's proof-of-concept emulator, providing a working framework in which to develop a graphical interface to illustrate the key features of the coupled approach.

In this deliverable, we presented validation tests focused on basic functions performed by particular entity. The next step is to validate developed modules in integrated scenario with other COMET modules. These tests will be performed in integration testbed and reported in forthcoming deliverable "D5.1 – Integration of COMET Prototype and Adaptation of Applications".

8 References

[1] COMET Deliverable, “D2.2: “High-Level Architecture of the COMET System”, January 2011

[2] COMET Deliverable, “D4.2: Final Specification of Mechanisms, Protocols and Algorithms for Enhanced Network Platforms”, December 14th, 2011.

[3] COMET Deliverable, “D3.3: Prototype Implementation and System Integration Interfaces for the Content Mediation System” January 2012

[4] COMET Deliverable, “D3.2: Final Specification of Mechanisms, Protocols and Algorithms for the Content Mediation System”, November 2011.

9 Abbreviations

AS	Autonomous System
BW	Bandwidth
CAFE	Content Aware Forwarding Entity
CC	Content Client
CME	Content Mediation Entity
CP	Content Publisher
CRE	Content Resolution Entity
CRME	Content Resolution and Mediation Entity
CS	Content Server
DB	Database
DNS	Domain Name System
HS	Handle System
IP	Internet Protocol
IPLR	IP Packet Loss Ratio
IPTD	IP Packet Transfer Delay
NLRI	Network Layer Reachability Information
RAE	Routing Awareness Entity
QoS	Quality of Service

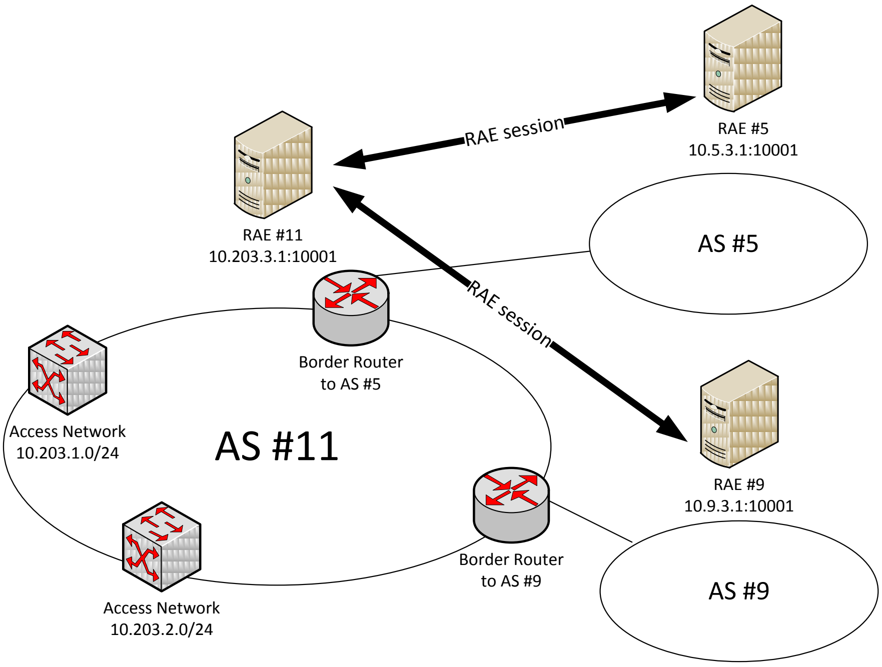
10 Acknowledgements

This deliverable was made possible due to the large and open help of the WP4 team of the COMET project within this STREP, which includes besides the deliverable authors as indicated in the document control. Many thanks to all of them.

11 Appendix A: Exemplary configuration file for RAE

11.1 Overview

Configuration for RAE is stored in binary format. In order to prepare binary configuration file one can use a script prepared in the Python language. Below we show how to prepare this script file for an exemplary domain depicted on following figure.



11.2 Script example

The script should be located in the main RAE folder, e.g., /opt/comet/rae. Otherwise, one should adapt the sys.path.append() method call.

```
#!/usr/bin/python
import sys
sys.path.append("build/resources")
import config_pb2
cm = config_pb2.ConfigurationMessage()
```

#Limit for the number of paths maintained for a single destination prefix.

```
cm.limit_of_preferred_paths = 3
```

#Configure information about local AS

```
cm.local_as_number = 11
cm.local_ip_address = "10.203.3.1"
cm.local_port_number = 10001
```

#Limit for the intensity of the UPDATE messages exchanged between RAEs

```
cm.minimum_update_interval = 15
```

```
#Provide the listening port of the CME
cm.cme_ip_address = "10.203.3.1"
cm.cme_port_number = 9090

# Define parameters of path ranking algorithm
# use 'multiplicative inverse' function for decision process
cm.decision_with_flat_function = 0

# create QoS targets for class 1 (BTBE)
dp = cm.decision_parameters.add()
dp.id = 1
dp.delay_aspiration = 0.5
dp.delay_reservation = 1.0
dp.loss_aspiration = 1e-5
dp.loss_reservation = 1e-3
dp.bandwidth_aspiration = 4e6
dp.bandwidth_reservation = 1e6

# create QoS targets for class 2 (PR)
dp = cm.decision_parameters.add()
dp.id = 2
dp.delay_aspiration = 0.15
dp.delay_reservation = 0.4
dp.loss_aspiration = 1e-6
dp.loss_reservation = 1e-4
dp.bandwidth_aspiration = 10e6
dp.bandwidth_reservation = 4e6

#Create peering to AS #5
peer = cm.peer_table.add()
peer.remote_as_number = 5
peer.remote_ip_address = "10.5.3.1"
peer.remote_port_number = 10001

#Set the TTL value that will allow to send IP packets to RAE in AS #5. It should be the minimum
#value allowing for correct operations.
peer.ttl_value = 3

#Describe QoS parameters on the link towards the AS #5 (2 classes).
c = peer.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
c = peer.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 5e-6
```

```
c.metric.maximum_delay = 0.005
c.metric.supported_bandwidth = 10e6
```

#Create peering to AS #9

```
peer = cm.peer_table.add()
peer.remote_as_number = 5
peer.remote_ip_address = "10.9.3.1"
peer.remote_port_number = 10001
```

#Set the TTL value that will allow to send IP packets to RAE in AS #9. It should be the minimum
#value allowing for correct operations.

```
peer.ttl_value = 4
```

#Describe QoS parameters on the link towards the AS #9 (2 classes).

```
c = peer.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 2e-6
c.metric.maximum_delay = 0.002
c.metric.supported_bandwidth = 10e6
c = peer.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 4e-6
c.metric.maximum_delay = 0.010
c.metric.supported_bandwidth = 10e6
```

#Create prefixes for access networks: 10.203.1.0/24 and 10.203.2.0/24.

```
prefix = cm.prefix_table.add()
prefix.ip_address = "10.203.1.0"
prefix.prefix_length = 24
prefix = cm.prefix_table.add()
prefix.ip_address = "10.203.2.0"
prefix.prefix_length = 24
```

#Provide provisioning information about the intra-domain part: from prefix 10.203.1.0/24 to
#10.203.2.0/24 prefix (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.prefix.ip_address = "10.203.1.0"
prov.source.prefix.prefix_length = 24
prov.sink.prefix.ip_address = "10.203.2.0"
prov.sink.prefix.prefix_length = 24
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
```

```
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from prefix 10.203.1.0/24 to #border router leading to AS #5 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.prefix.ip_address = "10.203.1.0"
prov.source.prefix.prefix_length = 24
prov.sink.as_number = 5
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from prefix 10.203.1.0/24 to #border router leading to AS #9 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.prefix.ip_address = "10.203.1.0"
prov.source.prefix.prefix_length = 24
prov.sink.as_number = 9
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from prefix 10.203.2.0/24 to #10.203.1.0/24 prefix (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.prefix.ip_address = "10.203.2.0"
prov.source.prefix.prefix_length = 24
prov.sink.prefix.ip_address = "10.203.1.0"
prov.sink.prefix.prefix_length = 24
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 2e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
```

```
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 2e-6
c.metric.maximum_delay = 0.002
c.metric.supported_bandwidth = 10e6

#Provide provisioning information about the intra-domain part: from prefix 10.203.2.0/24 to
#border router leading to AS #5 (2 classes).
prov = cm.provisioning_table.add()
prov.source.prefix.ip_address = "10.203.2.0"
prov.source.prefix.prefix_length = 24
prov.sink.as_number = 5
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6

#Provide provisioning information about the intra-domain part: from prefix 10.203.2.0/24 to
#border router leading to AS #9 (2 classes).
prov = cm.provisioning_table.add()
prov.source.prefix.ip_address = "10.203.2.0"
prov.source.prefix.prefix_length = 24
prov.sink.as_number = 9
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6

#Provide provisioning information about the intra-domain part: from border router connecting AS
#5 to prefix 10.203.1.0/24 (2 classes).
prov = cm.provisioning_table.add()
prov.source.as_number = 5
prov.sink.prefix.ip_address = "10.203.1.0"
prov.sink.prefix.prefix_length = 24
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
```

```
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from border router connecting AS #5 to prefix 10.203.2.0/24 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.as_number = 5
prov.sink.prefix.ip_address = "10.203.2.0"
prov.sink.prefix.prefix_length = 24
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from border router connecting AS #5 to border router leading to AS #9 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.as_number = 5
prov.sink.as_number = 9
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from border router connecting AS #9 to prefix 10.203.1.0/24 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.as_number = 9
prov.sink.prefix.ip_address = "10.203.1.0"
prov.sink.prefix.prefix_length = 24
c = prov.class_table.add()
c.cos_id = 1
```

```
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from border router connecting AS #9 to prefix 10.203.2.0/24 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.as_number = 9
prov.sink.prefix.ip_address = "10.203.2.0"
prov.sink.prefix.prefix_length = 24
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

#Provide provisioning information about the intra-domain part: from border router connecting AS #9 to border router leading to AS #5 (2 classes).

```
prov = cm.provisioning_table.add()
prov.source.as_number = 9
prov.sink.as_number = 5
c = prov.class_table.add()
c.cos_id = 1
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.0005
c.metric.supported_bandwidth = 10e6
c = prov.class_table.add()
c.cos_id = 2
c.metric.loss_ratio = 1e-6
c.metric.maximum_delay = 0.001
c.metric.supported_bandwidth = 10e6
```

Finally, save the file!

```
open("rae11.conf.pb2", "wb").write(cm.SerializeToString())
```

11.3 Prepare log4cxx.conf file

Create a file with following contents:

```
log4j.rootLogger=INFO, A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=%d %-5p [%c] %m%n
```

11.4 Starting the RAE

Type the following in the command line console to start the RAE.

```
./build/rae rae11.conf.pb2
```

12 Appendix B: CAFE agent test script

```
import socket
import sys
import cmecafe_pb2
HOST, PORT = "10.2.0.10", 9999

# Create a socket (SOCK_STREAM means a TCP socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    # Connect to server
    sock.connect((HOST, PORT))


    # Collect streams
    m = cmecafe_pb2.GenericRequest()
    m.type = cmecafe_pb2.GenericRequest.COLLECT_EXPIRED_STREAMS
    length = m.ByteSize()
    assert(length < (1 << 16))
    sock.send(''.join([chr(length >> 8), chr(length & 0xff)]))
    sock.send(m.SerializeToString())

    # Receive response
    t = sock.recv(2)
    length = ord(t[0]) << 8 | ord(t[1])
    t = sock.recv(length)
    r = cmecafe_pb2.GenericResponse.FromString(t)
    print("Received:")
    print(str(r))

    # Send configure stream
    for i in range(1,10):
        m = cmecafe_pb2.GenericRequest()
        m.type = cmecafe_pb2.GenericRequest.CONFIGURE_STREAM
        m.configure.id = i
        m.configure.filter.ip_source = "1.1.1.1"
        m.configure.filter.ip_destination = "2.2.2.2"
        m.configure.filter.protocol = 6
        m.configure.filter.port_source = 80
        m.configure.filter.port_destination = 0
        m.configure.bandwidth = 1000
        m.configure.cos = "PR"
        m.configure.key = "\x01\x02{0:c}".format(i & 0xff)
        m.configure.refresh_time = 10
        m.configure.as_path.extend([1,2,3])
        length = m.ByteSize()
        assert(length < (1 << 16))
        sock.send(''.join([chr(length >> 8), chr(length & 0xff)]))
        sock.send(m.SerializeToString())

        # receive response
        t = sock.recv(2)
        length = ord(t[0]) << 8 | ord(t[1])
        t = sock.recv(length)
        r = cmecafe_pb2.GenericResponse.FromString(t)
        print("Received:")
        print(str(r))
```