COntent Mediator architecture for content-aware nETworks

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

Deliverable D3.3
Prototype Implementation and System Integration Interfaces for the Content Mediation System

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

© Copyright 2012, the Members of the COMET Consortium

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

Spiros Spirou
INTRACOM TELECOM, spis@intracom.com

© Copyright 2012, the Members of the COMET Consortium

Document Control

Title: Prototype Implementation and System Integration Interfaces for the Content Mediation System

Type: Public

Editor(s): George Petropoulos

E-mail: geopet@intracom.com

Author(s): George Petropoulos, Sergios Soursos (INTRACOM TELECOM), David Flórez Rodríguez (TID), Wei Koong Chai, Ioannis Psaras, Stuart Clayman, Marinos Charalambides (UCL), Andrzej Beben, Jaroslaw Sliwinski (WUT)

Doc ID: d3.3_v7.1.doc

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
v0.1	01/06/11	George Petropoulos	First version, ToC, CME template
v0.2	08/06/11	George Petropoulos	CP, CRE contribution
v0.3	14/06/11	Jaroslaw Sliwinski	RAE contribution
v0.4	24/06/11	David Florez, George Petropoulos	CC, CS contribution, CME, CP, CRE updates
v1.1	30/06/11	George Petropoulos	Updated version for decoupled approach system release v1.1
v1.2	20/09/11	George Petropoulos	New ToC including coupled approach
v2.0	02/12/11	George Petropoulos	Updated CME, CP, CRE contributions for decoupled approach v2.0
v3.0	06/02/12	Wei Koong Chai, George Petropoulos	Summary, Coupled approach contribution, Decoupled approach overview, integration procedures and technologies contribution
v3.2	15/02/12	Wei Koong Chai, David Florez	Updated contributions in decoupled and coupled approach
v4.0	20/02/12	George Petropoulos	Final version
v5.0	20/02/12	David Flórez	Final Version for submission
v6.0	23/04/12	George Petropoulos	Updated ToC
v6.1	08/05/12	Wei Koong Chai, David Florez, Andrzej Beben	Contributed to implementation choices and Ipv6 deployment
v6.4	10/05/12	George Petropoulos	Integrated contributions, updated Conclusions section
v6.5	15/05/12	Sergios Soursos, George Petropoulos	Updated contribution and comments
v6.6	22/05/12	David Flórez, Ioannis Psarras	Review and comments
v7.0	24/05/12	George Petropoulos	Final version
V7.1	01/06/2012	David Flórez	Final Version ready 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.

© Copyright 2012, the Members of the COMET Consortium

Table of Contents

1 Executive Summary 6

2 Introduction 7

3 Decoupled Approach Overview 8

3.1 Entities 8

3.2 Interfaces 11

3.2.1 IPv4/IPv6 deployment 12

3.3 System Deployment 12

4 Coupled Approach Overview 14

4.1 Entities 14

4.2 Interfaces 15

4.3 System Deployment 15

5 Decoupled Approach Entities 19

5.1 Content Mediation Entity 19

5.1.1 Description of overall functionality 19

5.1.2 Controller 20

5.1.3 Resolver 26

5.1.4 Decision Maker 28

5.1.5 DB 30

5.1.6 Path Manager 35

5.1.7 Server Manager 42

5.1.8 Admin 44

5.2 Content Resolution Entity 49

5.2.1 Description of overall functionality 49

5.2.2 Interfaces 49

5.2.3 Design 50

5.2.4 Testing and Test scenarios 50

5.3 Content Publisher 50

5.3.1 Description of overall functionality 50

5.3.2 Interfaces 51

5.3.3 Design 53

5.3.4 Testing and Test scenarios 56

5.4 Content Client 56

5.4.1 Description of overall functionality 56

5.4.2 Interfaces 57

5.4.3 Design 57

© Copyright 2012, the Members of the COMET

5.4.4 Testing and Test scenarios *59*
5.5 Server Network Monitoring Entity 59

6 Coupled Approach Entities **71**
6.1 Content Resolution and Mediation Entity 71
6.1.1 Description of overall functionality *71*
6.1.2 Interfaces *71*
6.1.3 Design *73*
6.1.4 Testing and test scenarios *76*
6.2 Content Publisher 78
6.2.1 Description of overall functionality *78*
6.2.2 Interface *79*
6.2.3 Design *79*
6.2.4 Testing and test scenarios *80*
6.3 Content Client 80
6.3.1 Description of overall functionality *80*
6.3.2 Interface *80*
6.3.3 Design *80*
6.3.4 Testing and test scenarios *81*
7 Conclusions **82**
8 References **83**
9 Abbreviations **84**
10 Acknowledgements **85**
11 Appendix **86**
11.1 Application and Transport Protocol codification 86
11.1.1 Application Protocol *86*
11.1.2 Transport Protocol *86*
11.2 Interfaces specification 87
11.2.1 CME-CC *87*
11.2.2 CME-RAE *88*

© Copyright 2012, the Members of the COMET Consortium

11.2.3 [redacted] *89*
11.2.4 CME-CAFE *90*
11.2.5 inter-CME *91*
11.2.6 [redacted] *92*
11.3 Databases Specification 93
11.3.1 [redacted] *93*

© Copyright 2012, the Members of the COMET

1 Executive Summary

This deliverable documents the final and detailed engineering and implementation work of the COMET project with respect to the development of the Content Mediation Plane (CMP) of the COMET prototype. These activities have been carried out following the architecture design provided in D2.2 [1], as well as the final description and specification of CMP mechanisms described in D3.2 [3]. The deliverable presents the final CMP entities, the comprising components and interfaces, as implemented in the final releases for the Decoupled and Coupled approaches.

More specifically, in Sections 3 and 4, the overall architectural design for the prototype implementation of Decoupled and Coupled approaches is presented respectively. The CMP entities are identified, along with their components and interfaces. A component diagram depicts their interconnection, accompanied by a list of the available interfaces and a deployment diagram which illustrates how the designed architectures are materialized. Both sections briefly describe the technologies used to develop CMP entities, as well as the motivation behind each implementation choices and interfaces' design, while specifying whether there are any theoretical constraints for deployment of COMET to system to IPv4, IPv6 or mixed network environments.

In Sections 5 and 6, we proceed with a more detailed analysis of the architecture, respectively for the two approaches. For each entity present in the CMP, we first provide a high-level description of the functionality in place and the entity's break down in components. Then, on a per-component basis, we provide a detailed description of the functionality, the interfaces, the structure (class diagram), the behaviour (sequence diagram) and the validation tests conducted is included, acting like a high-level documentation of the prototype's source code.

Section 7 summarizes this report, providing the high-level reasoning behind all implementation choices. In addition, it is highlighted that the COMET prototype would require certain optimizations in order to be deployed in production environments, as well as that it is compatible with IPv4/IPv6, but not mixed, environments. Finally, in the Appendix, additional information about interfaces' specification is included.

Details about the implementation work performed on the CFP entities are not included in this deliverable, but in D4.3 [5]. In addition, 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".

© Copyright 2012, the Members of the COMET Consortium

2 Introduction

The overall objective of the COMET project is to define and deliver a novel content-aware Internet architecture aiming to simplify content access, distribution and delivery in a network-aware manner. This architecture was described in D2.2 [1] and follows a 2-plane approach for mediating content requests and delivering content with increased Quality of Service (QoS) and experience (QoE). It consists of the Content Mediation Plane (CMP) that offers a unified interface for content access and publication, while it's responsible to gather and disseminate all required server and network resources for content consumption, and the Content Forwarding Plane (CFP) which is in charge of the delivery of content. Both planes collaborate to achieve network and server awareness to ensure optimal content delivery across the Internet.

During the first 2 years of the project, 2 approaches for content naming, resolution and delivery were defined and implemented, the Decoupled one, following the current Internet paradigm where the aim is to build a content-based resolution framework that is readily deployable in the current Internet with minimal disruption and the Coupled one, which proposes a revolutionary content-aware Internet architecture, aiming to change the way Internet works today from the root, breaking some of the design principles of the original Internet.

This deliverable focuses on the CMP elements and mechanisms of the COMET architecture, which were initially defined in D3.1 [2] and finalized in D3.2 [3], and were implemented during the second year by the WP3 team of the COMET project. It aims to provide a detailed documentation of the implementation work, describing all implemented CMP entities, components, interfaces and mechanisms for both approaches of the defined COMET architecture.

In the first 2 chapters of the present deliverable (chapters 3 and 4, respectively), a justification of the technologies used is provided, so as to make clear the decisions taken when implementing the system. Moreover, an architectural overview of each approach is respectively provided, presenting the implemented CMP entities, components and interfaces, using UML component diagrams, while providing information on how each approach is going to be deployed, using UML deployment diagrams.

In chapters 5 and 6, all implemented entities, components, interfaces and functionalities are presented in more details. Each sub-section presents an overview for each entity or component, describes its respective classes, methods and structure and provides the detailed description of its functionality, using sequence diagrams, as well as the implemented self-contained tests, used to validate its functionality.

Finally, conclusions section provides a brief summary of current deliverable and provides conclusions on COMET prototype's IPv4/IPv6 compatibility and deployment in production environments.

© Copyright 2012, the Members of the COMET

3 Decoupled Approach Overview

3.1 Entities

Figure 1 presents the component diagram for all entities of the Decoupled approach at the CMP level, namely:

- the Content Mediation Entity (CME),
- the Content Resolution Entity (CRE),
- the Content Publisher (CP),
- the Content Client (CC),
- the Server and Network Monitoring Entity (SNME), and
- the Content Server (CS).

In addition, the 2 interfaces of CME with the CFP entities (RAE and CAFE) are presented, without providing the internal structure of each entity, which is instead defined and presented in D4.3 [5].

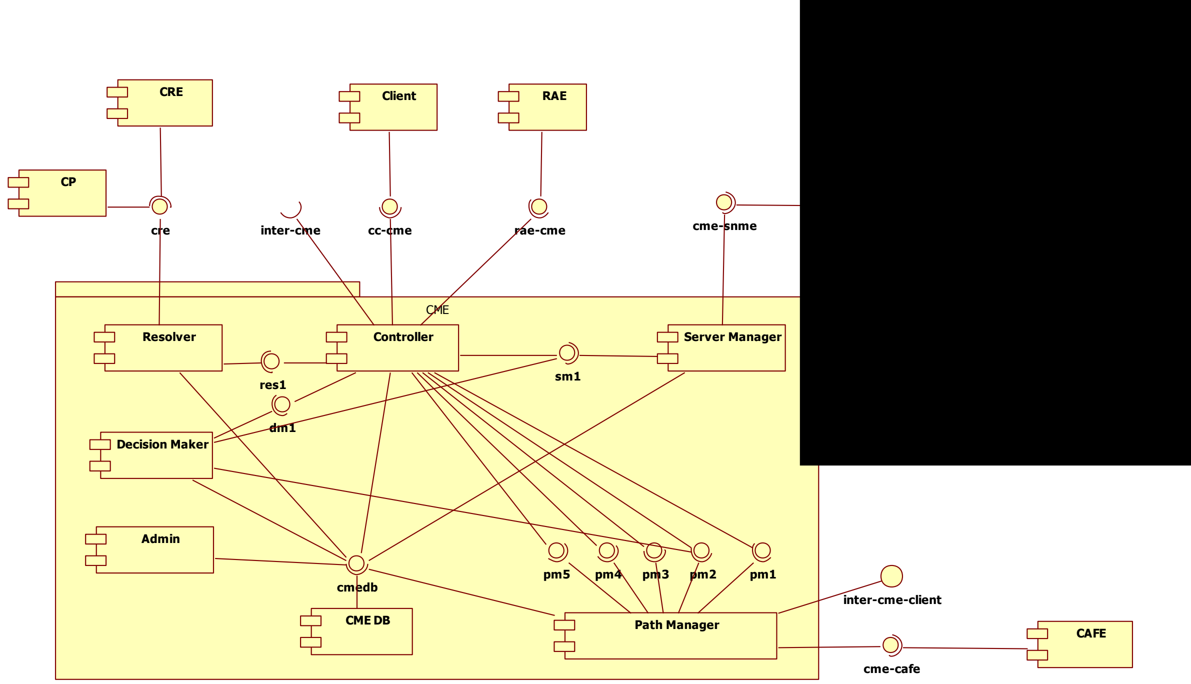


Figure 1: Decoupled approach component diagram

Each entity's and component's functionality and internal structure, as well as their interfaces are also presented in Section 3.2 and described in Chapter 5.

The CME is the key entity of the CMP, responsible for efficient and rapid content request mediation, coordination of and interaction with almost all other COMET entities. The CME was required to be as light-weight as possible, in order to handle high loads without facing any performance issues. Java was chosen as the programming language for CME prototype, due to its lack of limitations, OS independence, ease of prototyping and developers' familiarity. However, C++ could be selected in the future if there is a requirement for developing and deploying CME for production environment, due to its higher performance. There were a few options for the framework on which CME would be based on (application servers such as Jboss AS [19] and Glassfish [20], or network frameworks, such as Jboss Netty [8] and Apache MINA [18]). JBoss Netty was selected, as it is a stable, flexible and lightweight Java New I/O (NIO) Client Server Socket framework, with quick startup (comparing to application servers) and high performance, enabling a big number of connections. Approaches referring to application servers such as Java

© Copyright 2012, the Members of the COMET Consortium

Web Services (i.e., used by JBoss AS and Glassfish) were avoided for two reasons: i) CME is a network entity and not an application that runs on top of the network and ii) Java Web Services technology uses long and “heavy” messages for the transfer of application-specific and not network-level information.

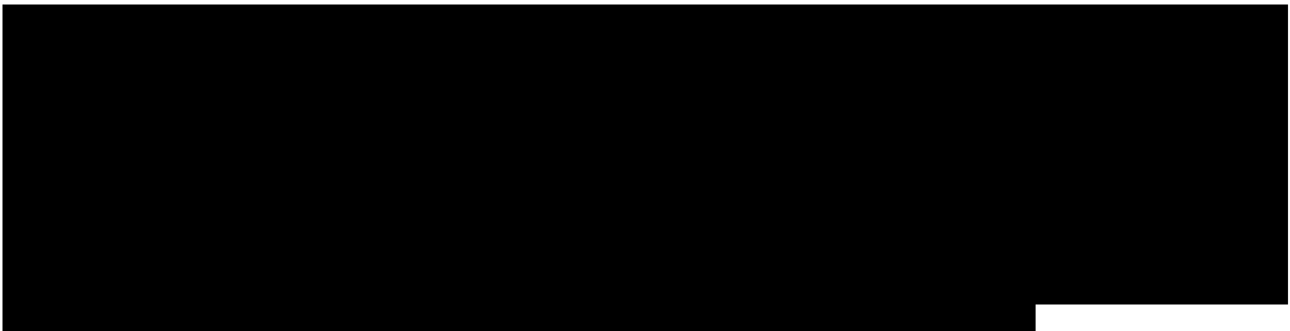
For the web interface of the Admin component, an embedded Jetty web server [9] was used; this was based on the fact that it is lightweight and easily deployable. Other lightweight (embedded) web servers could also be used instead, as long as they can be easily integrated in the Netty platform. Regarding the database, there were a few choices, but open-source MySQL [10] was preferred, since it’s highly stable and flexible, and deployable to any OS. However, any other database can be used in the future, as an ORM framework (Hibernate [21]) was chosen to be the middleware between the business logic of CME and the database.

For the name resolution system, the Handle protocol was selected, as it is an established content-centric and secure framework appropriate for the resolution and management of digital objects. Following its IETF standards, the Handle System provides the server software, implemented in Java, as well as the required client libraries available in Java and C. CRE is actually a Handle System name server, implemented in Java, with a Berkeley database for storing, updating and removing content records. Berkeley database is the default option when deploying a Handle System name server; however it can be easily integrated with MySQL if higher performance is required in the future. Alternative options for the resolution system could involve the design and implementation of proprietary systems and protocols, which would not however strengthen the exploitation capabilities of the resulting prototype, due to cross-platform compatibility issues.

The CP was implemented as a web application in order to be easily accessed by any web browser. A standalone Jetty web server was selected to host the CP web app, because it is lightweight and has comparable performance to Tomcat. JSF and RichFaces [22] were used to develop the web pages, because of their wide range of AJAX components. However, in the future, the consortium might choose to integrate CP into existing applications, by using the CP API for automatic content record creation, update, deletion and other CP-related functions. There are no real limitations for the selection of the technologies used in the design and construction of the web pages, as long as the rendering time of the resulting pages are acceptable by the end users. RichFaces is considered to be a wide-spread JSF framework with good overall performance.

The basic idea behind the implementation of CC was the development of a lightweight piece of software, targeted to the OS where it is intended to run, so that it can take advantage of the system native resources without the installation of extra pieces of software other than the CC. In other words, the CC uses the available by the system software to consume the delivered content, based on the type of the content and the default application specified by the system. For prototyping purposes, the selected OS was Microsoft’s Windows, owing to its ubiquity; however, the entire development could be easily ported to other operating systems. Because of the intended simplicity, and in order to avoid the performance problems of interpreted languages like Java, the language used in the CC implementation was C++, which is high level, multiplatform and OOP, and enables the creation of binary executables for the target OS than can be directly launched from command line.

© Copyright 2012, the Members of the COMET



The RAE is implemented in C++ as standalone self-contained entity with protobuf [12] interfaces to CME and cooperating RAEs. RAEs are responsible for creation and management of content delivery paths thus they process large amount of data describing these paths. The C++ language was chosen because it is OOP and provides efficient resource management. A more in-depth rationale behind the choice of C++ for RAE deployment is presented in D4.3 [5].

CAFE is basically responsible for forwarding data packets carrying content, thus it needs to be very efficient. To achieve this goal, basic elements of CAFE (i.e. cafe_forward, cafe_intercept) are implemented as loadable Linux kernel modules. A more in-depth rationale behind the choice of used technologies for CAFE deployment is presented in D4.3 [5].

Table 3-1 summarizes the functionalities and technologies used for each entity.

Table 3-1: CMP Entities and Technologies

Entity	**Functionality**	**Technologies**	**Reference**
CME	Content request mediation, name resolution, decision process, server awareness, path storage, discovery, provisioning and configuration and CAFE configuration.	Java JSF, RichFaces, Netty, Hibernate Jetty, MySQL	Section 5.1
CRE	Content record storage and retrieval	Java Handle System, Berkeley DB	Section 5.2
CP	Content record creation, update and deletion	Java JSF, RichFaces Jetty	Section5.3
CC	Content request, content consumption.	C++ Web Browser Windows	Section 5.4
CS	Providing content, server status monitoring	Java Sigar VLC and Apache	Section 5.6

© Copyright 2012, the Members of the COMET Consortium

RAE	NLRI exchange	C++ Boost SQLite	Deliverable 4.3 [5]
CAFE	Content forwarding	Python and C Linux	Deliverable 4.3 [5]

3.2 Interfaces

Table 3-2 details the interfaces shown in the component diagram of Figure 1. 3 types of protocols were used: Proprietary, Protobuf [12] and Handle protocol [6]. The motivation behind each protocol choice is presented below.

CC-CME interfaces are based on the exchange of binary messages over UDP. The rationale behind this decision was the implementation of a lightweight protocol, free from the packet overhead usually associated to connection oriented protocols. Besides, the messages were structured in fields of predefined length and position in order to reduce to a minimum the size of the commands exchanged in the both protocols. Lastly, both protocols were designed to carry both IPv6 and IPv4 IP addresses, enabling the related entities to be deployed in both environments.

Inter-CME, RAE-CME, and CAFE-CME interfaces are realised using Google's Protocol Buffers (protobuf). protobuf was chosen since it is a lightweight, simple and efficient mechanism for serialization of structured data. In addition, protobuf can be easily integrated with JBoss Netty, with higher performance and more robust implementation for most programming languages, over similar protocol-definition platforms for network communication (such as Apache Thrift, JSON, etc.). A more in-depth rationale behind the choice of protobuf for RAE-CME and CAFE-CME interfaces is also presented in D4.3 [5].

For CP-CRE and CME-CRE interfaces, the Handle protocol was chosen, since it is content centric, supporting security and access control, and both TCP and UDP, without any limitations on records' size.

Table 3-2: List of interfaces

Interface ID	Entity/Component providing the interface	Entity/Component using the interface	Purpose	Protocol	Reference
Inter-CME	CME	CME	Path discovery, path configuration, path provisioning, server awareness	Protobuf	Section 5.1.2.2.3
CC- CME	Controller	CC	Content request and response	Proprietary	Section 5.1.2.2.1
RAE- CME	Controller	RAE	Path and provisioning information sent by RAE	Protobuf	Section 5.1.2.2.2
CRE	CRE	Resolver, CP	Content Publication and name resolution	Handle Protocol	Section 5.2.2

© Copyright 2012, the Members of the COMET

CME -CAFE	CAFE	Path Manager	CAFE configuration	Protobuf	Section 5.1.6

3.2.1 IPv4/IPv6 deployment

CME was designed and developed to operate both in IPv4 and IPv6 environments. The JBoss Netty framework supports and operates smoothly in both environments without any significant reconfiguration. In addition, there are no specific design limitations for IPv6 support on both the CP and CRE entities.

Both RAE and stateless CAFEs were designed and developed to operate in dual stack mode [RFC 4213], where either IPv4 or IPv6 addresses could be used. In case of RAE, we used the *boost:asio* library for C++, which provides appropriate classes and methods to recognise the version of IP addresses and adequately handle them. Thanks to this library most of the functionalities required for dual stack were available in a straightforward way. In case of stateless CAFE, specific filtering rules were developed in the *cafe_intercept* module to recognise the IP protocol version and intercept the IPv4 or IPv6 packets. Moreover, as stateless CAFE also plays the role of IP router, we configure Linux to support dual stack routing operations (including ospf, ospf3, bgp-4 and mp-bpg). More details are presented in D4.3 [5].

From the point of view of the CC, and the interfaces they expose to each other or third parties, there are not any design constraints that would preclude the proper working with IPv4 or IPv6. The messages of the defined protocols have been provided with fields able to carry and identify IP addresses written in both IPv6 and IPv4 formats, which are properly stored in the databases used by the entities, namely .

The only limitation currently affecting COMET is that it can work either on IPv4 or on IPv6 pure environments but not on mixed ones. That would require "6to4" or similar tunneling techniques to be in place, which are, however, outside the scope of this project and have been investigated in the past (e.g., [23]).

3.3 System Deployment

Figure 2 presents the deployment diagram for the decoupled approach in CMP. CME server is a JBoss Netty server [8], exposing interfaces to Content Client, RAE server deployed in another machine, and remote CME servers, while an embedded Jetty web server [9] is used by the Admin component to provide a web interface for the CME administrator to configure CME. In addition, a MySQL server [10] is used as CME's database.

On the other hand, a CRE server is a standalone Handle System server [11], launched in another machine, exposing interfaces to both CME and CP services for content publication and name resolution.

The CP is a web server providing a graphical user interface to allow the Content Owner/Creator to publish, update and delete content records, through his computer's web browser.

The CC is a C/C++ standalone application that is responsible for interacting with the CMP for the name/content resolution and launches the appropriate client application for the content consumption.

The CS is a standalone media server offering the desired content, enhanced with a monitoring agent that provides feedback to the SNME.

© Copyright 2012, the Members of the COMET Consortium

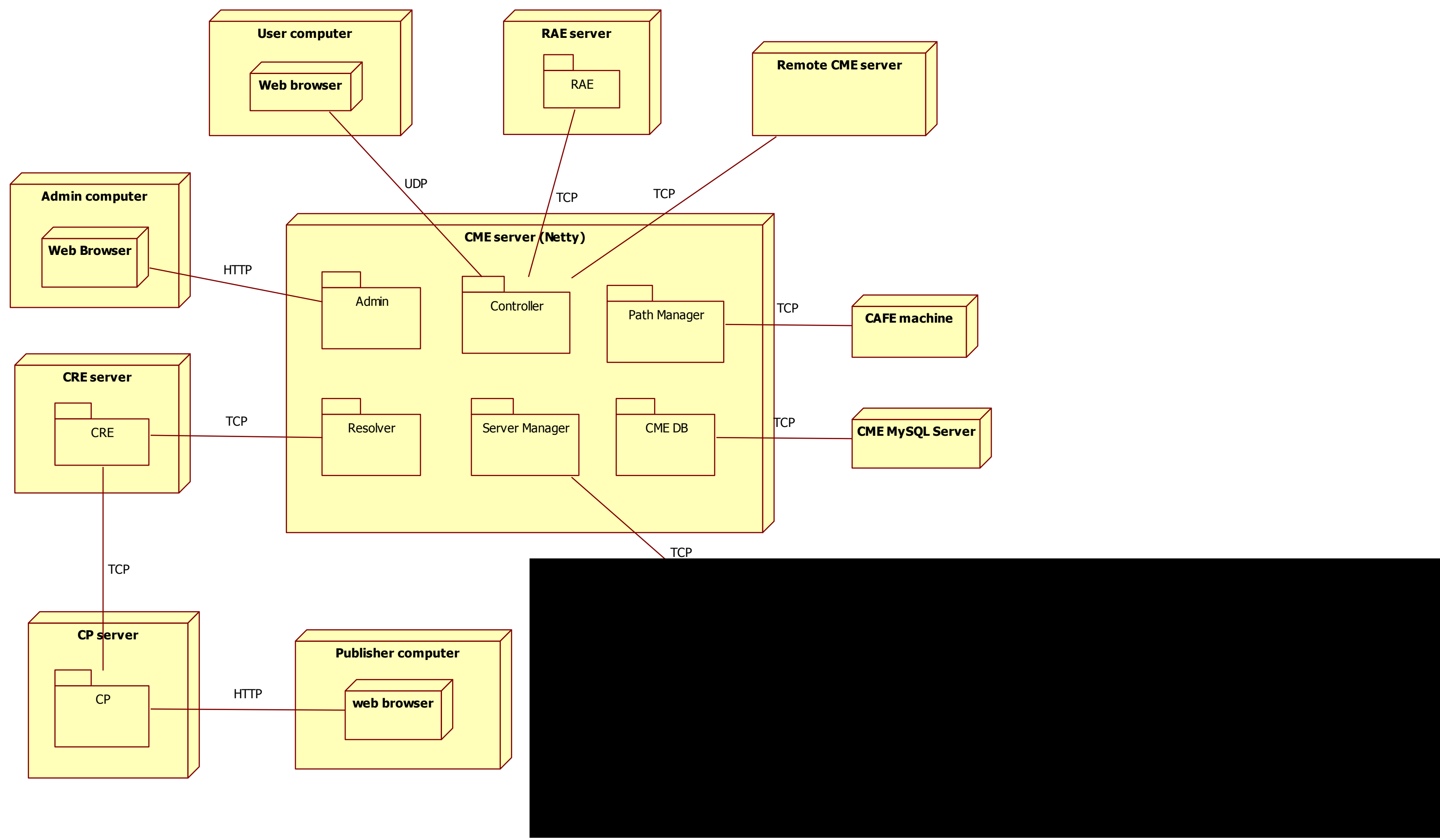


Figure 2: Decoupled approach deployment diagram

© Copyright 2012, the Members of the COMET Consortium
Version 7.1

4 Coupled Approach Overview

4.1 Entities

The specifications of the coupled approach can be found in deliverables D3.1 [2] and D3.2 [3] and this document mainly focuses on the implementation of a proof-of-concept (PoC) prototype of the proposed approach. The PoC aims at demonstrating various aspects of the approach at both the domain and router level.

At the domain level, the implementation focuses on the following:

- To show the coupled approach in resolving content consumption requests across a set of autonomous domains
- To show the approach to establish the domain-level content delivery paths according to business relationships and other factors such as ISP local policies
- To show the working of the route optimisation mechanism

At the router level, the implementation focuses on the following:

- To show the interaction between the Content Resolution and Mediation Entity (CRME) at the CMP level and the CAFE at the CFP level, including both routing awareness and path configuration
- To show the interworking between CAFEs at the network edge and conventional IP routers at the network core

In the domain level context, each node will represent a logical domain with *integrated* CMP and CFP functionalities. In contrast, at the router level, each node represents a CAFE with the CAFEs being centrally configured by a dedicated CRME belonging to the same domain. Figure 3 illustrates the relationships between the domain and router level perspective to the PoC.

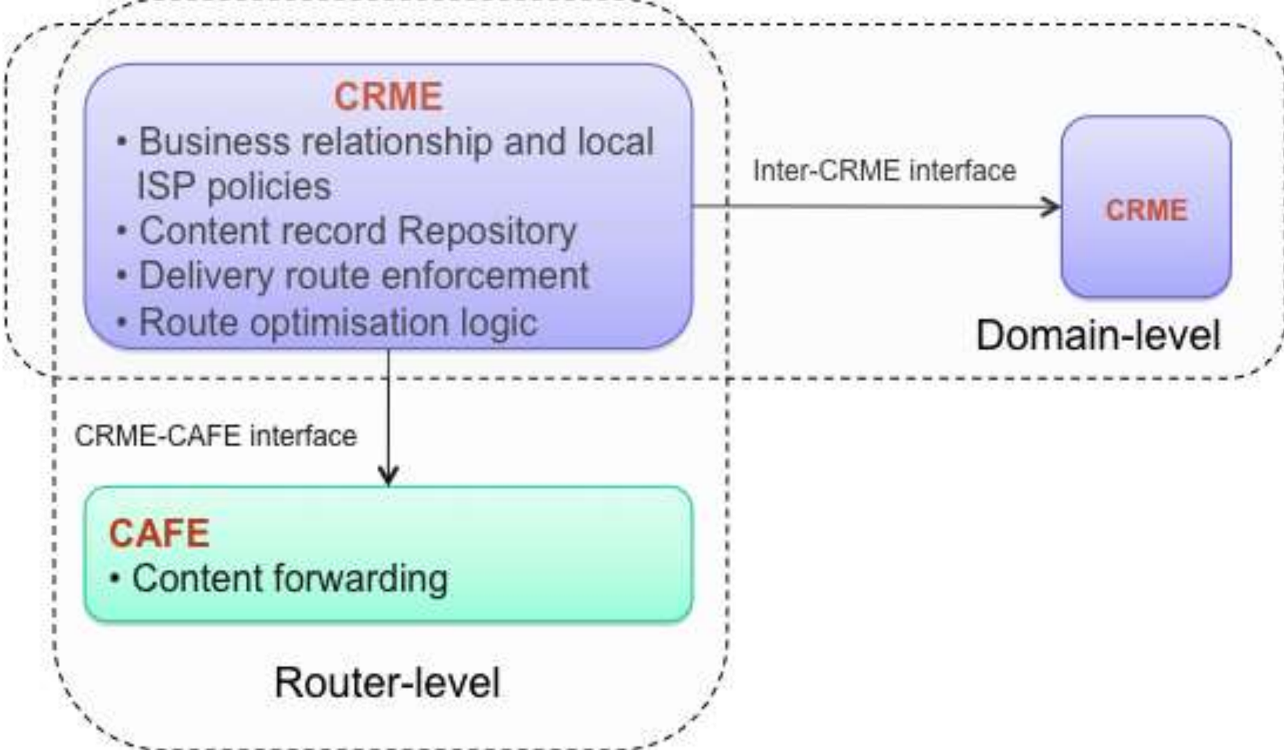


Figure 3: Demonstrating the coupled approach at different levels

© Copyright 2012, the Members of the COMET Consortium

The coupled approach PoC implementation focuses on the functionalities of content manipulation. The actual coding of the approach is at the level of entity without further separating each of the entity into further smaller components. As such, the UML component diagram of the coupled approach simply resembles the entity diagram as shown in Figure 4.

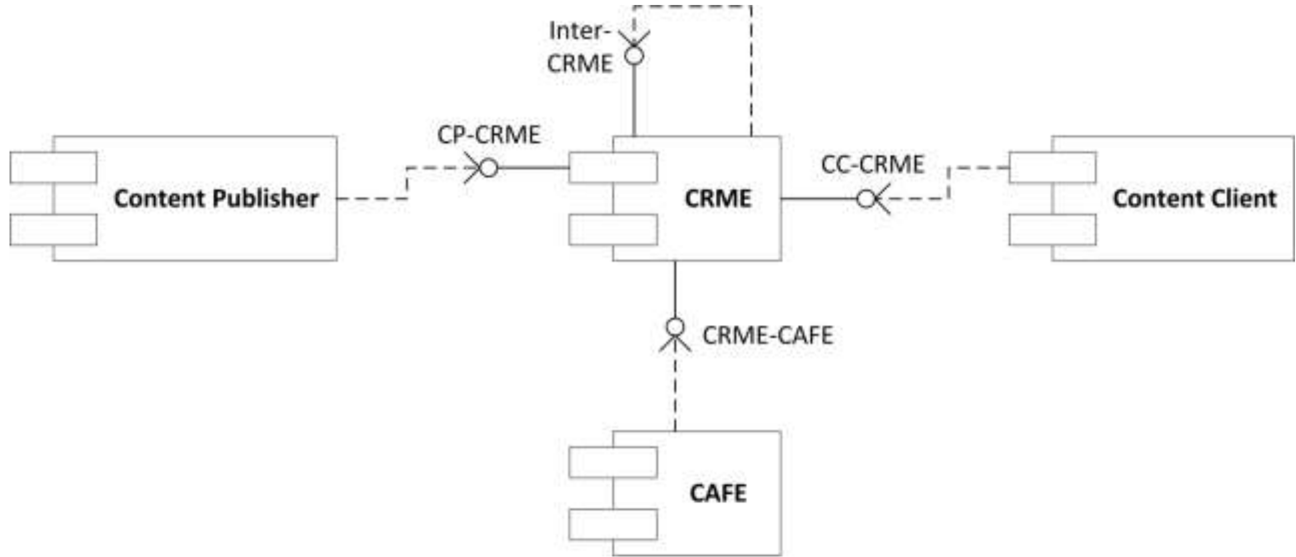


Figure 4: Coupled approach component diagram

4.2 Interfaces

Table 4-1 presents the list of interfaces in the coupled approach implementation. The choice of the protocol used is dependent on the platform the coupled approach built on (Section 4.3).

Table 4-1: List of interfaces

Interface ID	Entity/Component providing the interface	Entity/Component using the interface	Purpose	Protocol	Reference
Inter-CRME	CRME	CRME	Content publication, content resolution, content delivery path configuration	Proprietary	Section 6.1.2.1
CRME-CC	CRME	CRME, content client	Content request	Proprietary	Section 6.1.2.2
CRME-CS	CRME	CRME, content server	Content publication	Proprietary	Section 6.1.2.3
CRME-CAFE	CRME	CRME, CAFE	Content delivery path configuration	Proprietary	Section 6.1.2.4

4.3 System Deployment

The realisation of the coupled approach PoC is at a high-level and the ultimate objective is to demonstrate the basic content publication, resolution and delivery along with its routing optimisation functions through a graphical interface. Since it is designed as a *disruptive* approach, it is not within our goal to perform protocol-specific implementations.

© Copyright 2012, the Members of the COMET

The PoC is built on top of a platform called *Very Lightweight Network and Service Platform (VLNSP)* that is architecturally similar to the common virtual networks using hypervisors (e.g., Xen) but developed with the basic rationale that most of the features in virtual networks are unused and thus wasting precious resources. It is kept lightweight while maintaining the simple router to be simplistic retaining the basic capabilities of a service component. Specifically, the use of VLNSP is motivated with the following:

- Lower resource utilization – the number of virtual machines that can run on a host is limited due to the actual resources of the physical machine that need to be shared (such as the number of cores and the amount of memory available), together with the switching capabilities of the hypervisor.
- Better scalability – As a direct consequence of the high resource consumption in using full-scale virtual machines, the scale of the experiment is also very limited.
- Fast Startup speed– the speed of startup of a virtual machine can be quite slow. Although virtual machines boot up in the same order of magnitude as a physical host, there are extra layers and inefficiencies that slow them down. Also, if many virtual machines are started concurrently, then we observe that the physical machines and the hypervisor thrash trying to resolve resource utilisation.
- Reduced heaviness – the size of a virtual machine image is quite large. A virtual machine has to have a disc image which contains a full operating system and the applications needed for the relevant tasks. To start a virtual machine, the operating system needs to be booted and then the applications started. So every virtualised application needs the overhead of a full OS.
- Eliminate the issue where 98% of the router functionality not needed– in terms of virtual networks, and virtualised routers in particular, we observed that 98% of the router functionality were never utilised in any of the experiments that were run. Although software routers such as XORP and Quagga allow anyone to evaluate *soft* networks, the overhead of a virtual machine, with a full OS, and an application where only 2% is used, seems to be an ineffective approach for many experiments.
- More networking flexibility – when trying to configure the IP networking of virtual machines and virtual routers, there are some serious hurdles. The virtual machines do not talk directly to the network but via the hypervisor. The hypervisor has various schemes for connecting virtual machines to the underlying network, each of which has different behaviour. We found that the IP networking configuration and virtual machine to virtual machine interoperability a hindrance to network topology and network flexibility.

In the light of the above reasons, we implement the coupled approach PoC over VLNSP.

The key idea is to use Java Virtual Machines (JVMs) as virtual machines (VMs) with the applications and services being small Java apps. Specifically, each JVM is modelled as a network router. The JVMs will be directly linked to each other according to the topology setup and thus, real packets will not have to pass through the central hypervisor (as the bottleneck). A physical machine can now host hundreds of routers as compared to using full-scale VMs where only tens of instances can be simultaneously running in a machine. This greatly enhances the scalability of the platform.

The platform consists of three major components. The ***global controller*** supervises the overall experiment from the setup provided (in xml scripts), while the ***local controller***, which resides in each physical machine, passes out the instructions to the routers sent from the global controller (e.g., instruction to start up or shut down routers on the local machine, initiate or tear down connections with other machines etc.). The ***routers*** (i.e., the JVMs) are logically independent software entities which can only communicate with each other via network interfaces. The system can be run in a completely decentralised manner without both the local and global controllers as they are implemented simply for the conveniences of the management and experimental control. Figure 5 shows the conceptual view of the platform.

© Copyright 2012, the Members of the COMET Consortium

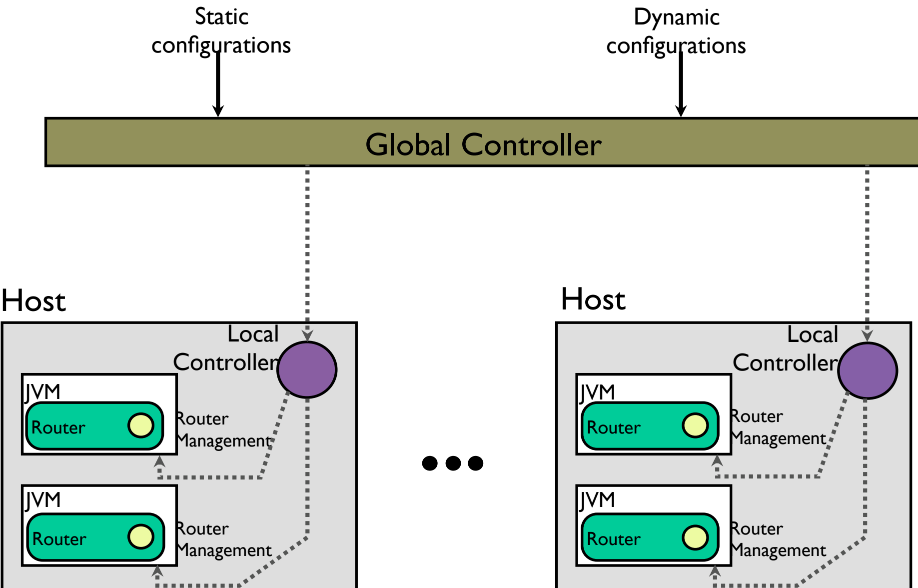


Figure 5: Conceptual illustration of the platform

The virtual routers hold network selections to the other virtual routers. Like in the real networks, they are aware of and exchange routing tables to determine the shortest path to each other. Data packets are sent between routers and queued at input and output interfaces. A system of virtual ports (similar to the current transport layer ports) is exposed with an interface much alike to standard sockets. Virtual applications are run on them by listening and sending on these virtual sockets. Datagrams have typical headers (including information such as source address, destination address, protocol type, source port, destination port, length, checksum and time to live (TTL)) which replicated many features of the real IP packets.

Figure 6 shows how the coupled approach entities are linked into the virtual routers. Basically, the entities can be started on top of any virtual routers and more than one entity can sit on top of one another. The main entities for the coupled approach are:

- Content resolution and mediation entity (CRME)
- Content-aware forwarding entity (CAFE)
- Content publisher (for this implementation, we assume that the content publisher is also hosting the content without further intermediaries, i.e. the Content Server)
- Content client

The implementation of these entities will be detailed in chapter6. In the platform, they are implemented as applications that are managed by the Application Manager (AppM) component. An Application Socket Multiplexer (ASM) component is in charge of the entities' datagram socket(s).

To coordinate experiments, a management console (MC) communicates with the controllers (i.e., the global and local controllers aforementioned) via a Management Console Request Protocol (MCRP). Conceptually, this can be seen as forming a management plane in the platform.

At the networking level, two main components on managing the routing functions are the routing controller (RC) and routing fabric (RF). Finally, the router-to-router (R2R) component enables communications between virtual routers.

© Copyright 2012, the Members of the COMET

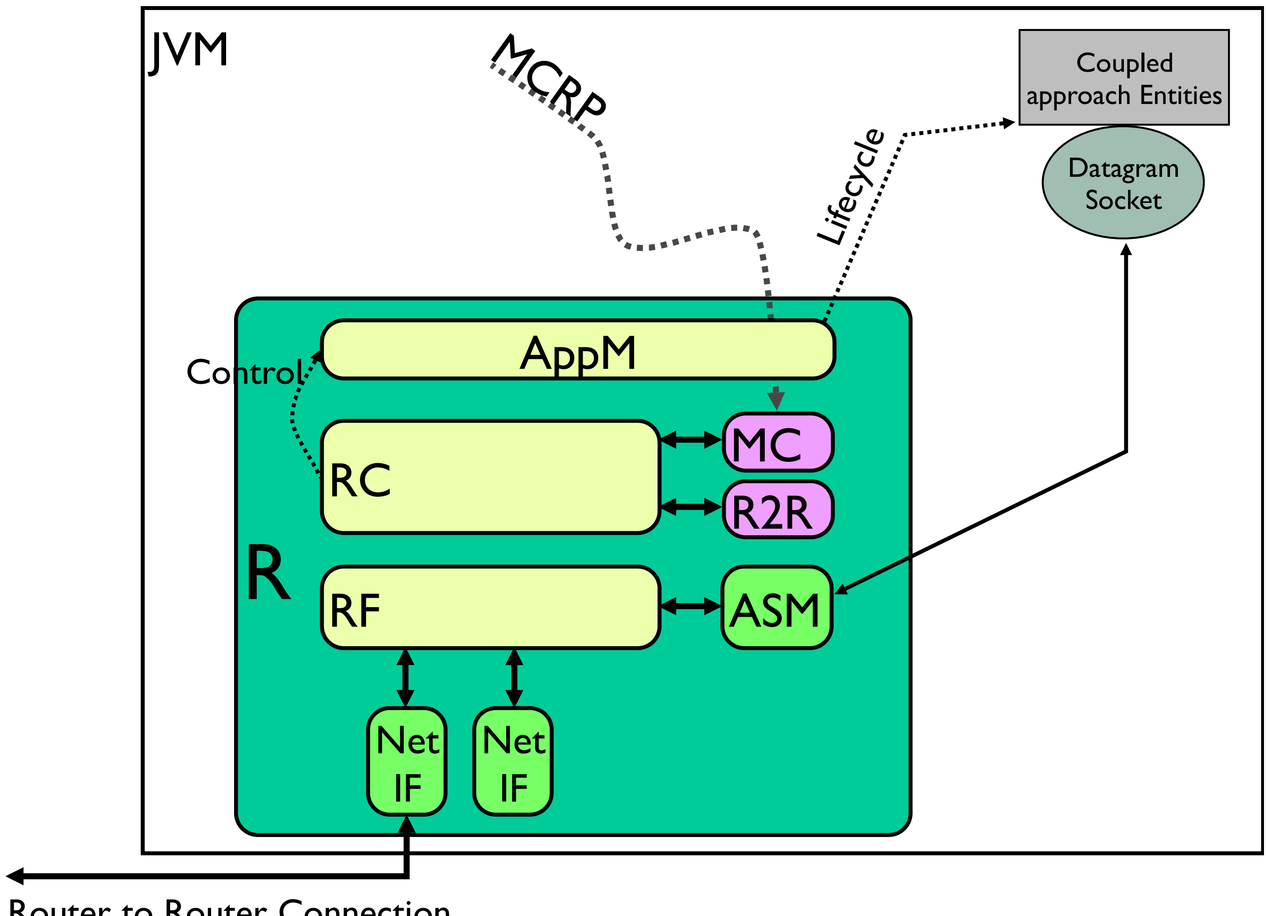


Figure 6: Coupled approach entities in the virtual router

The coupled approach is implemented as a `usr.curling` package in Java SE which in turn is implemented using packages provided by both the VLNSP and the standard Java SE libraries. The `usr.curling` package consists of four main classes: `Crme`, `Cafe`, `ContentServer` and `ContentClient`, which each communicates with its underlying router using standard UDP sockets.

© Copyright 2012, the Members of the COMET Consortium

5 Decoupled Approach Entities

In this section we go into more detail about the CMP entities of the Decoupled approach. A high level description of each entity is provided, along with the following list of information, on a per-component basis: i) a description of the functionality implemented, ii) the interfaces and the signatures of the implementing functions, iii) a number of class diagrams depicting the internal structure of the component, iv) a sequence diagram and v) the validation tests carried out for the specific component.

5.1 Content Mediation Entity

5.1.1 Description of overall functionality

The Content Mediation Entity (CME) is the leading entity of the COMET system, interacting with almost all COMET entities and making decisions on content, server and path selection. More specifically, CME has interfaces to:

- Content Client (CC)
- Content Resolution Entity (CRE)
- Routing Awareness Entity (RAE)
- Server Network Monitoring Entity (SNME)
- Content Aware Forwarding Entity (CAFE)
- other Content Mediation Entities (CMEs)

CME's main functionality includes:

- Receiving and handling content requests from CC
- Resolving received content name to content record, through interacting with CRE
- Selecting the best content server and delivery path, based on pre-configured parameters and information gathered from other CMEs
- Requesting content server load from attached SNME
- Configuring and provisioning (when necessary) the selected path
- Configuring selected CAFE for content consumption
- Responding to CC with the selected content record parameters
- Receiving network lever routing information (NLRI) from RAE and storing them to its DB
- Web interface for configuration purposes
- DB for storing required configuration and dynamic parameters

Currently, CME consists of the following components which are described further in the following sections of the document:

- Controller
- Resolver
- Decision Maker
- DB
- Path Manager
- Server Manager
- Admin

CME is implemented as a network entity, and the JBoss Netty Framework [8] was used for its development, along with all required libraries. Each component is under its respective package and exposes one or more interface to be called when necessary.

© Copyright 2012, the Members of the COMET

5.1.2 Controller

The controller is the controlling component of CME, exposing interfaces to CC, RAE and other CMEs, responsible for mediating content requests and controlling all CME components during content resolution and consumption.

5.1.2.1 Description of functionality

The Controller is the central CME component, containing the following classes under `controller` package:

- The Mediator class, which is responsible for mediating and controlling all other components during content requests.
- CC-CME, RAE-CME and inter-CME interfaces helper classes, which implement the `ChannelPipelineFactory` and extend `SimpleChannelUpstreamHandler`, `OneToOneDecoder` and `OneToOneEncoder` classes of Netty API [8], for parsing and handling incoming requests from CC, RAE and other CMEs:
    - o Regarding CME-CC interface, under `ccif` package exist all encoding (`DNSEncoder`), decoding(`DNSDecoder`) and supporting classes (`DNSHeader`, `DNSObject`, `DNSRequest`) for parsing incoming messages from CC and forwarding them to the respective handler (`DNSHandler`) class, which interacts with the resolver component to resolve received content name, and the decision maker to choose the best server. All these classes are included in `DNSPipelineFactory` class, implementing the `ChannelPipelineFactory` provided by Netty.
    - o On the other hand, in the case of CME-RAE and inter-CME interfaces, no encoding/decoding classes have been implemented, since protobuf-format messages are exchanged [12] and protobuf encoders and decoders (`ProtobufDecoder`, `ProtobufEncoder`) are already implemented and provided by Netty [8]. However, the respective Handler (`RAEHandler` and `CmeHandler`) and PipelineFactory (`RAEPipelineFactory` and `CmePipelineFactory`) classes have been implemented for both interfaces.
- Timer classes under `timer` package, used for fulfilling required tasks during specific periods of time. `CachedPathsTask` and `ExpiredStreamsTask` are developed, extending the `TimerTask` class of `java.util` package, used for removing paths with expired TTL and expired streams from local database.

5.1.2.2 Interfaces

5.1.2.2.1 CME-CC interface

CME and CC exchange messages following a DNS-like format, described in Section 11.2.1. The controller classes and methods which parse and handle received messages are:

- **`DNSPipelineFactory implements ChannelPipelineFactory`**
    - o **`public ChannelPipelinegetPipeline()`**: It creates the channel pipeline to handle incoming UDP, DNS-like packets, using custom DNS-based decoder and encoder, and attaching the business logic handler at the end.
- **`DNSDecoder extends OneToOneDecoder`**
    - o **`protected Object decode(ChannelHandlerContextctx, Channel channel, Object msg)`**: Decodes incoming bytes of DNS-like packets and returns an object with the required parameters to be handled by `DNSHandler`.
    - o **`private intgetBit(byte[] data, int pos)`**: Returns the integer at position pos of byte array data.
    - o **`private void printStringBin(byte[] in)`**: Prints the integer of a byte array.

© Copyright 2012, the Members of the COMET Consortium

- **private int intFrom2Bytes(byte[] ba):** Returns the respective integer from a byte array of size 2.
- **DNSEncoder extends OneToOneEncoder**
  - **protected Object encode(ChannelHandlerContextctx, Channel channel, Object msg):** Encodes the object received from `DNSHandler` to a `ChannelBuffer` object.
  - **private void setBit(byte[] data, intpos, intval):** Sets the bit at a specified position in byte array data into a required value.
  - **private byte[] intTo2Bytes(intnum):** Returns the respective byte array of an integer.
  - **private byte[] appToHex(String app):** Returns the byte array of the received application protocol string, based on COMET specification.
  - **private byte[] transToHex(String transp):** Returns the byte array of the received transport protocol string, based on COMET specification.
- **DNSHandler extends SimpleChannelUpstreamHandler**
  - **public void messageReceived(ChannelHandlerContextctx, MessageEvent e):** This method handles the DNS-like messages, performs Content Name resolution, decision making and prepares the object that will be returned to client.

5.1.2.2.2 CME-RAE Interface

Protobuf [12] messages are exchanged in CME-RAE interface, following the .proto file defined in Section 11.2.2. The controller classes and methods which parse and handle received messages are:

- **RaePipelineFactory implements ChannelPipelineFactory**
  - **public ChannelPipelinegetPipeline():** This method creates the channel pipeline using frame decoder for the TCP packets, protobuf encoder and decoder for the protobuf messages and the business logic handler.
- **RaeHandler extends SimpleChannelUpstreamHandler**
  - **public void messageReceived(ChannelHandlerContextctx, MessageEvent e):** This method handles the `GenericRequest` protobuf messages received in the interface.

5.1.2.2.3 inter-CME interface

Protobuf [12] messages are exchanged in inter-CME interface, following the .proto file defined in Section 11.2.2. The controller classes and methods which parse and handle received messages are:

- **CmePipelineFactory implements ChannelPipelineFactory**
  - **public ChannelPipelinegetPipeline():** This method creates the channel pipeline using frame decoder for the TCP packets, protobuf encoder and decoder for the protobuf messages and the business logic handler.
- **CmeHandler extends SimpleChannelUpstreamHandler**
  - **public void messageReceived(ChannelHandlerContextctx, MessageEvent e):** This method handles the `GenericRequest`protobuf messages received in the interface.

© Copyright 2012, the Members of the COMET

5.1.2.3 Design

5.1.2.3.1 Class Diagrams

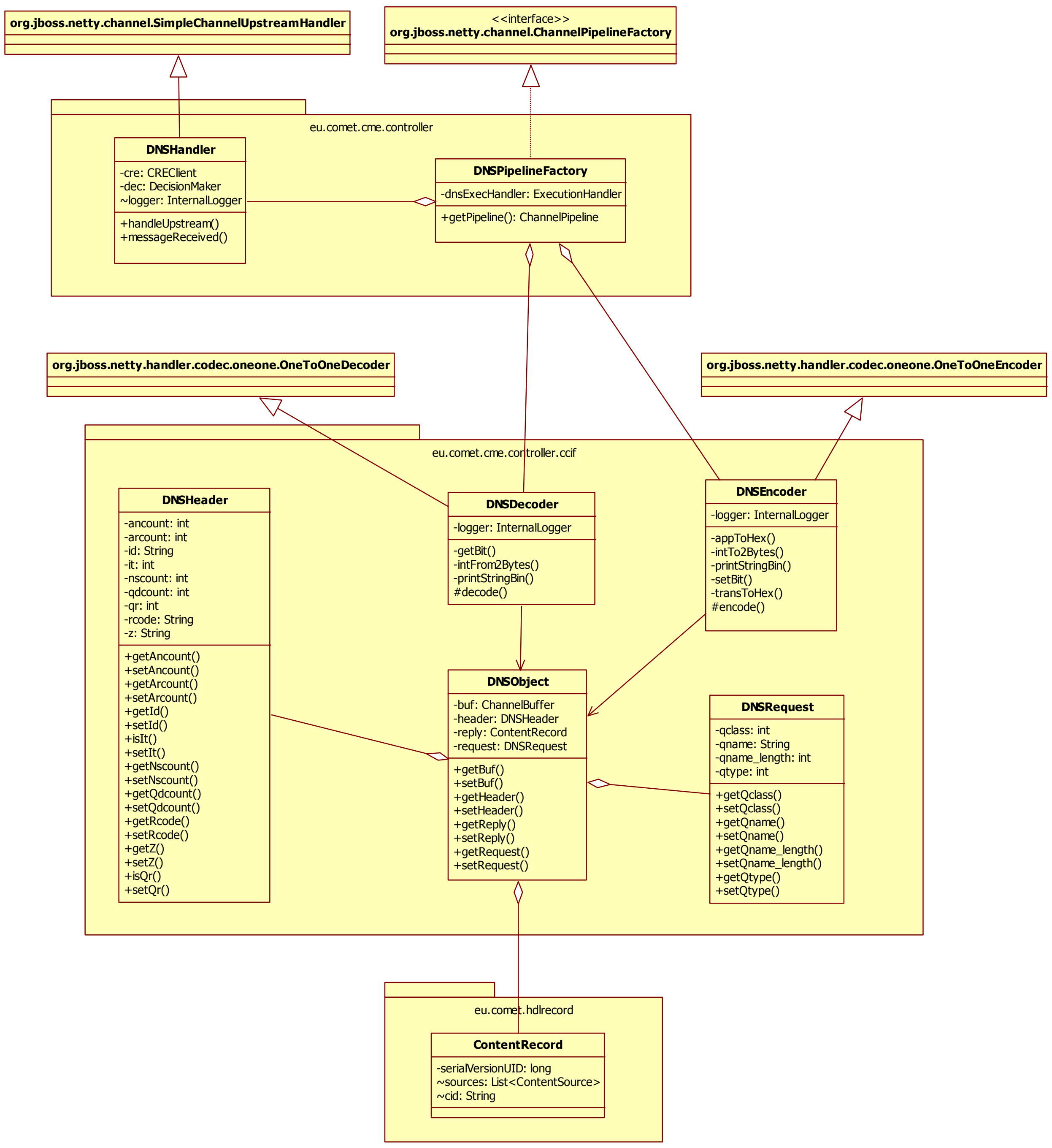


Figure 7: CC-CME interface class diagram

Figure 7 presents the class diagram for the CC-CME interface, in which `DNSPipelineFactory` implements Netty's `ChannelPipelineFactory` and has a `DNSHandler` (extending `SimpleChannelUpstreamHandler`), a `DNSDecoder` (extending `OneToOneDecoder`) and a `DNSEncoder` (extending `OneToOneEncoder`). In addition to this, `DNSDecoder` uses `DNSObject`, which contains `DNSHeader`, `DNSRequest` and `ContentRecord`.

© Copyright 2012, the Members of the COMET Consortium

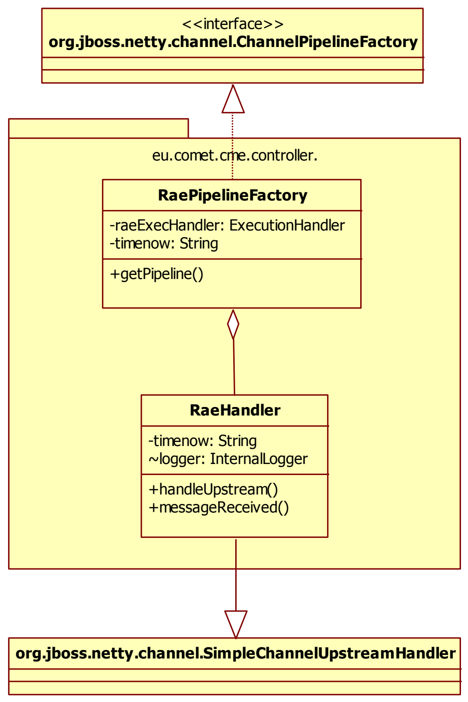


Figure 8: CME-RAE interface class diagram

Figure 8 presents the class diagram of CME-RAE interface, in which `RAEPipelineFactory` implements Netty's `ChannelPipelineFactory` and has a `RaeHandler`, which extends Netty's `SimpleChannelUpstreamHandler`. This is also the case for the inter-CME interface, which is presented in Figure 9.

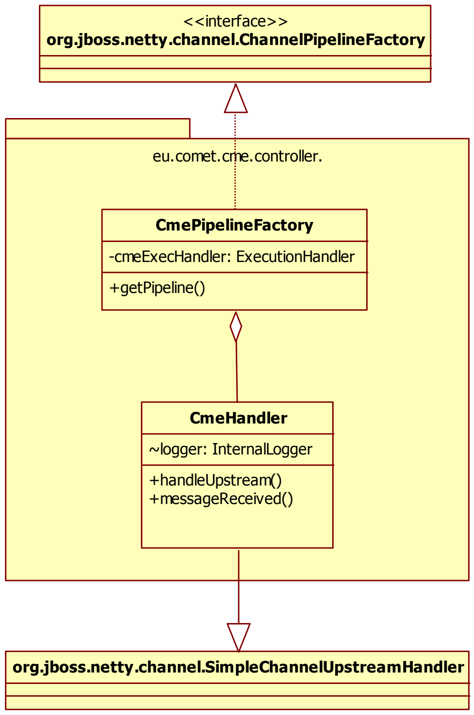


Figure 9: inter-CME interface class diagram

© Copyright 2012, the Members of the COMET

In addition, Figure 10 presents the class diagram for the class involved when Controller is invoked. More specifically, methods from the `DecisionMaker`, `CREClient` and `PathConfiguration` interfaces are called.

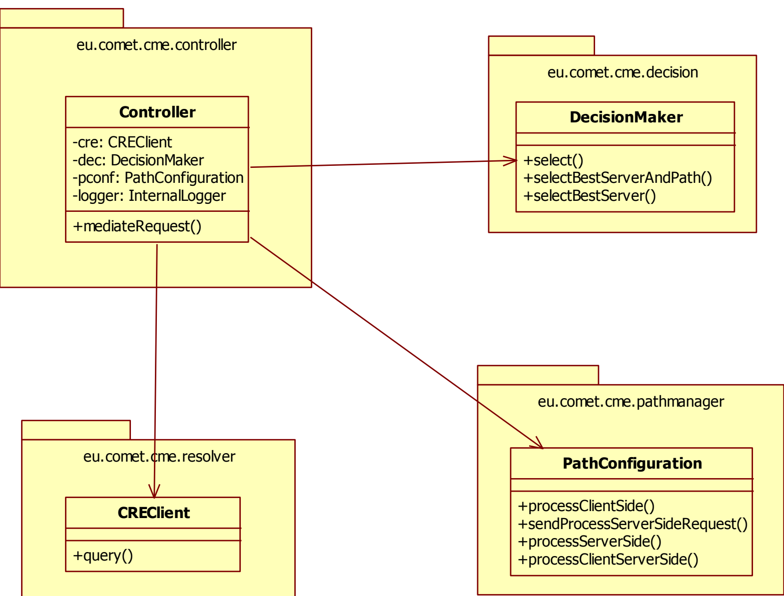


Figure 10: Mediator class diagram

5.1.2.3.2 Sequence Diagrams

In Figure 11 the sequence diagram is presented when a message is received at CME-CC interface. The `DNSEncoder` comes first in the `DNSPipeline` and parses the received bytes, returning a `DNSObject`, containing all required information (e.g. content name), which is provided to `DNSHandler`. The request is handled by the `Controller`, which will coordinate all other CME components. First, it queries the content name from `HandleClient` and receives the content record stored in CRE (this is described in resolver component at Section 5.1.3), which is then sent to `DecisionMaker`, responding with the selected content source and path (`srcpath`). If both the Content Client and selected Content Server belong to the same domain, then `PathConfigurator` is invoked twice, to configure path in client- and server-side, otherwise only in client-side. Finally, `DNSHandler` receives the selected content record, handled then by the `DNSDecoder` which writes the bytes of the response to the channel.

© Copyright 2012, the Members of the COMET Consortium

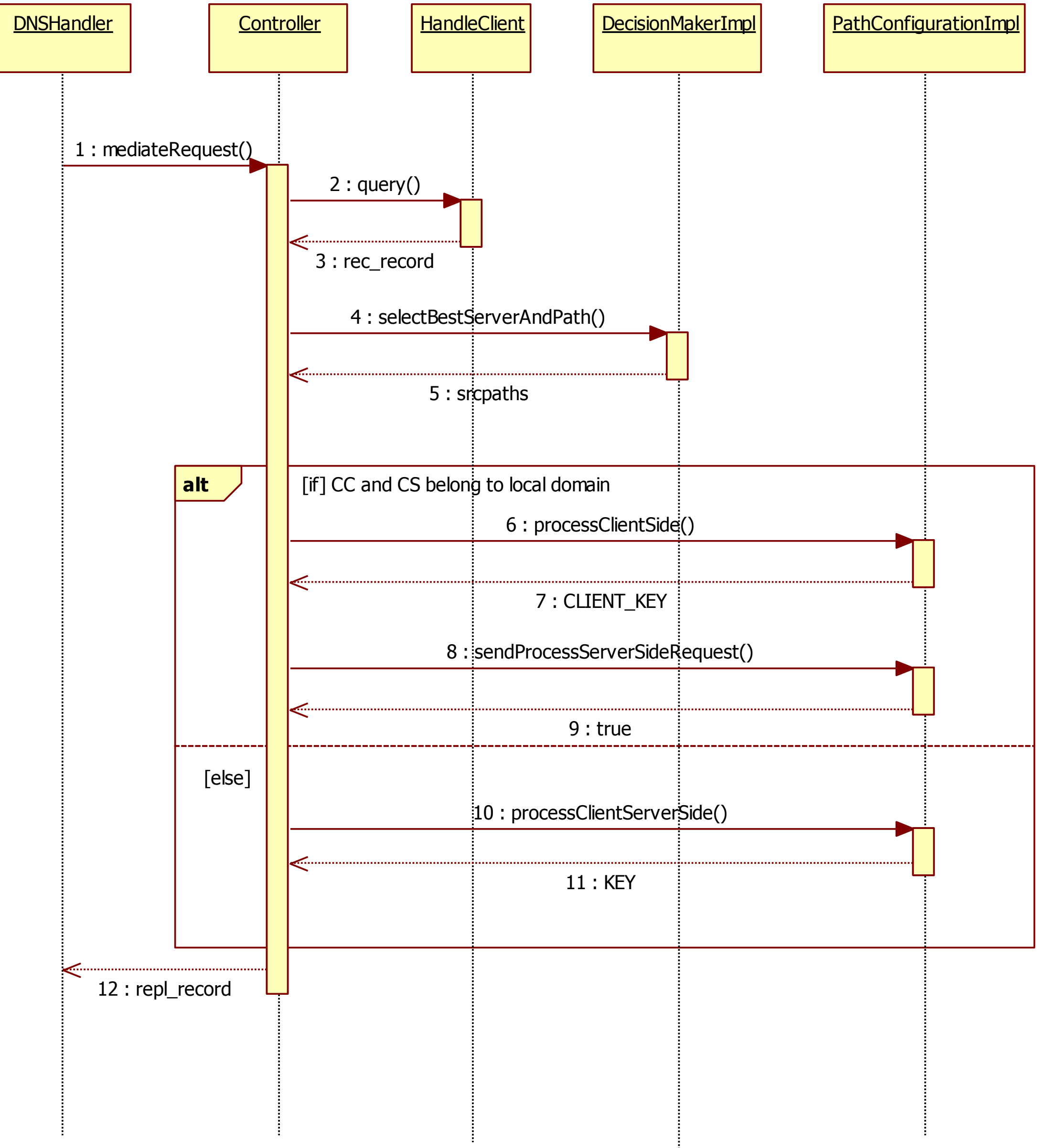


Figure 11: Mediator sequence diagram

In CME-RAE and inter-CME interfaces, when a new message arrives at the `RAE/CmePipelineFactory`, `ProtobufDecoder` handles to decode the received bytes into a `GenericRequest` object, which is then received by `RAE/CmeHandler`, which just responds with another `GenericRequest` message back to `ProtobufEncoder`, returning the bytes sent back to RAE.

5.1.2.4 Testing and Test scenarios

Testing of controller includes:

- `DNSEncoder/Decoder` testing, by providing test data and checking their output (`DNSPipelineTest`)
- CME-RAE interface testing, by running an instance of CME server and a test Client, sending test data (`RAEPipelineTest`)

© Copyright 2012, the Members of the COMET

5.1.2.4.1 DNSPipelineTest

`DNSPipelineTest` implements the following method:

- **`public void testDNSPipeline():`** This method tests `DNSDecoder` and `DNSEncoder` classes. More specifically, test data (raw bytes) are inserted in an embedding decoder and checks if output is the correct `DNSObject`. In addition to this, a `DNSObject` is provided to an embedding encoder, checking if bytes returned are the ones expected.

5.1.2.4.2 RAEPipelineTest

`RAEPipelineTest` includes the following methods:

- **`public static void starting():`** Sets up the test CME Server.
- **`public void testRaePipeline():`** Checks the CME's RAE interface by constructing a VERSION message to be sent by the client and waits for the CME's reply.
- **`public static void finish():`** Finalizes the test CME server.

5.1.3 Resolver

The resolver component is responsible for resolving received content name to content record, through interacting with the CRE. The content record includes parameters described in section 3.3.

In the context of COMET, the resolver is actually a Handle System client interacting with the CRE, which in this case is a standalone Handle System server [11], containing all available content records.

5.1.3.1 Description of functionality

As previously stated, the resolver component interacts with CRE in order to resolve the content name received from the controller component to its associated content record, stored in the CRE.

The resolver component is actually the `HandleClient` class, which implements the `CREClient` interface and uses the essential methods from the Handle System Java API [11] to perform name resolution. It also uses available CRE configuration information (IP address and port) stored in the database.

5.1.3.2 Interfaces

The resolver component implements the CME-CRE interface and consists of the following methods:

- **`public ContentRecord query(String cname):`** The method for querying a content name from the CRE. It returns a `ContentRecord` object, containing all defined content record parameters. In case of unsuccessful name resolution or unavailability of CRE parameters in the database, returns a null object.
- **`public Object toObject (byte[] bytes):`** The method `toObject` is used to transform a byte array to a Java object. This method is essential, because content record parameters are stored as byte arrays in the CRE.

5.1.3.3 Design

5.1.3.3.1 Class Diagrams

Figure 12 presents the class diagram for the resolver component, in which the `HandleClient` class implements the `CREClient` interface.

© Copyright 2012, the Members of the COMET Consortium

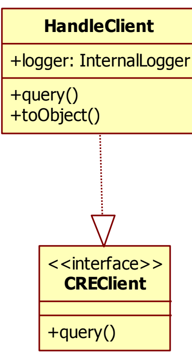


Figure 12: Resolver class diagram

5.1.3.3.2 Sequence Diagrams

During content name resolution (Figure 13), the `HandleClient` performs 2 queries in Config table in CME database, requesting the IP address and port of root CRE, using methods of `ConfigDAO` interface. If the queries are successful, then `HandleClient` sends a TCP handle request to root CRE to receive the IP address of local CRE responsible for the received naming authority and then sends a TCP handle request to local CRE to receive the associated content record.

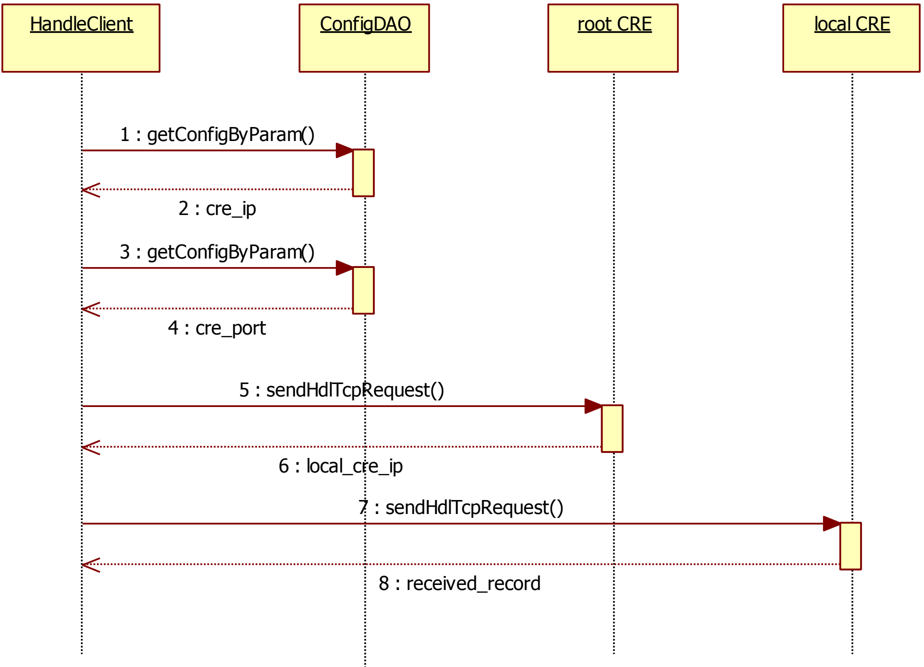


Figure 13: Content Name Resolution sequence diagram

5.1.3.4 Testing and Test scenarios

Taking into account that the resolver component implements the CME-CRE interface, it is difficult to test its direct functionality, because a CRE instance running would be essential. However, failure tests have been implemented using JUnit [13], to test component's behavior in case that either the CRE configuration parameters are not stored in the database, or the CRE is not running. The tests are performed by the `resolverTest` class, which includes the following methods:

© Copyright 2012, the Members of the COMET

- **`public void testResolutionFailures()`**: The `testResolutionFailures()` is the method testing the failure cases in resolver component. It tries to perform name resolution, while there are no CRE configuration information stored in the database, returning a null `ContentRecord` object, it performs name resolution when only the CRE IP address is stored in the database, successfully returning a null `ContentRecord` and finally, performs name resolution, while all CRE configuration information are stored and CRE is not running, returning a null `ContentRecord`.

5.1.4 Decision Maker

Decision Maker is responsible for deciding the best server and path, based on the received content record, information stored in path storage and decision parameters configured in CME database.

5.1.4.1 Description of functionality

Decision Maker contains the multi-criteria decision algorithm to rank available candidates and select the best of them. It consists of the `DecisionMakerImpl` class, which implements the `DecisionMaker` interface, as well as some helper classes (`DecisionMakerUtil`, `SourceAndPath`, `ServerPathCandidate`), containing methods and classes used during decision process.

5.1.4.2 Interfaces

The `DecisionMakerImpl` class implements the following 2 methods, also defined in the `DecisionMaker` interface:

- **`public ContentRecord select(ContentRecord cr)`**: This method is used to select the first content source and the first content server from the received content record and returns a new `ContentRecord` object with the selected parameters.
- **`public SourceAndPath selectBestServerAndPath(ContentRecord cr, String ccip)`**: The method used to select the best Content Source and path, based on content record parameters, path storage information, servers' load and decision parameters configured in CME database, when user's CoS is higher or equal than BTBE.
- **`public SourceAndPath selectBestServer(ContentRecord cr, String ccip)`**: The method used to select the best Content Server, based on servers' load, when user's CoS is equal to BE.

5.1.4.3 Design

5.1.4.3.1 Class Diagrams

Figure 14 presents the class diagram for the Decision maker component, in which the `DecisionMakerImpl` class implements the `DecisionMaker` interface. In addition, `DecisionMakerImpl` calls methods from `PathDiscoveryImpl` and `ServerAwarenessImpl`, as well as from the helper classes, included in `eu.comet.cme.decision.util` package, `DecisionMakerUtil`, `ServerPathCandidate` and `SourceAndPath`.

© Copyright 2012, the Members of the COMET Consortium

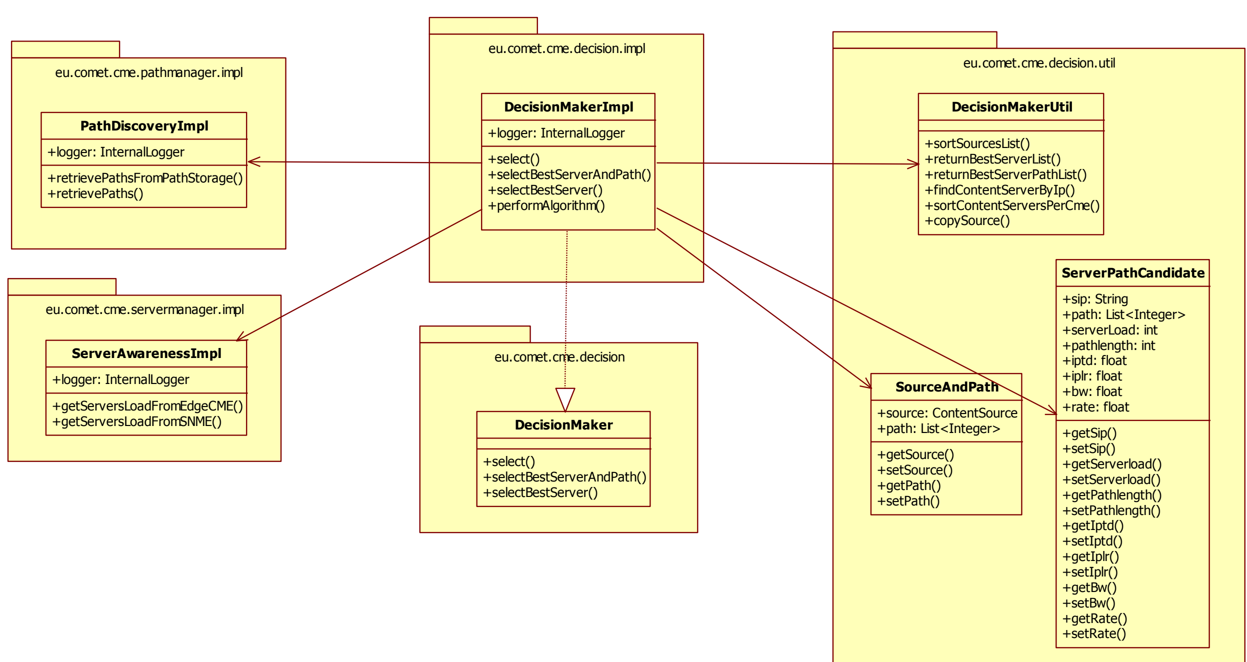


Figure 14: Decision Maker class diagram

5.1.4.3.2 Sequence Diagrams

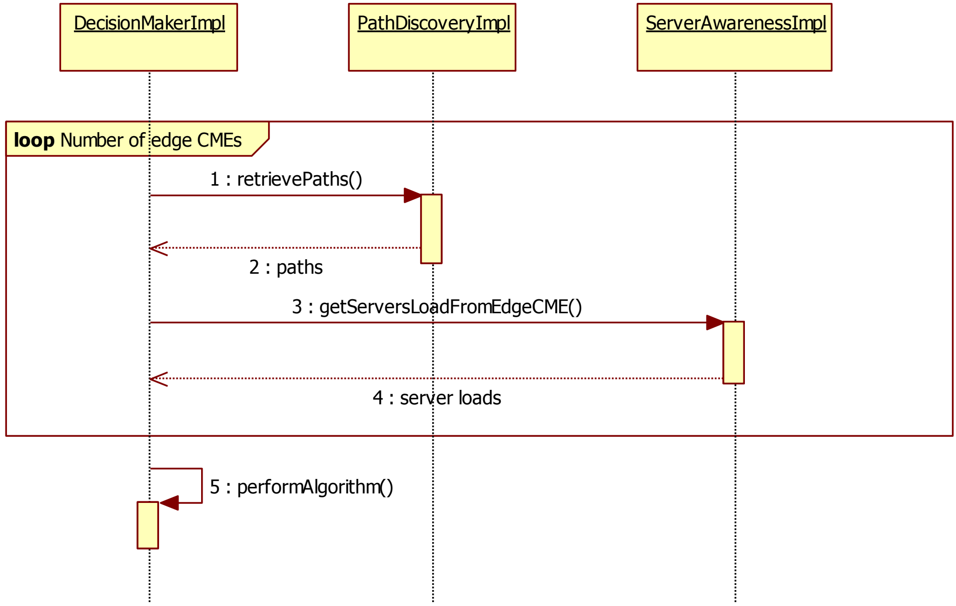


Figure 15: Decision process sequence diagram

Figure 15 presents the sequence diagram for the `selectBestServerAndPath` method of the decision process, in which `DecisionMakerImpl` calls the `retrievePaths` method from the `PathDiscoveryImpl`, in order to receive discovered paths and `getServersLoadFromEdgeCME` method of `ServerAwarenessImpl`, to receive loads for all content servers, per their adjacent CME. Finally, it uses `performAlgorithm` method to select the best server and path, which is returned to Controller. In the case of `selectBestServer` method, it only requires servers' loads.

© Copyright 2012, the Members of the COMET

5.1.4.4 Testing and Test scenarios

Decision Maker is currently tested using JUnit [13] for multiple cases and purposes:

- **DecisionMakerUtilTest**
  - `public void testListSorting()`: Method used for testing if content sources list is performed correctly.
- **DummyDecisionMakerTest**
  - **`public void OneSourceMultipleServers()`**: This method tests the dummy server selection algorithm for a content record containing one content source with multiple content servers.
  - **`public void MultipleSourcesMultipleServers()`**: This method tests the dummy server selection algorithm for a content record containing multiple content sources with multiple content servers.
- **AlgorithmTest**
  - **`public void testRankingAlgorithm()`**: Method for testing that decision algorithm is performed correctly.
  - **`public void testSelectServer()`**: Method for testing that selection of best content server is performed successfully.

5.1.5 DB

The DB is the database-responsible component, containing all methods for setting up, configuring and querying parameters from the database. Currently, the DB component handles only 16 tables in the CME database:

- CACHED_PATHS: Stores information about paths from a prefix to another
- CACHEDPATH_ASES: Stores the actual cached paths (lists of AS numbers)
- CAFES_KEY: Stores assigned key sequence, for a given list of CAFEs
- CAFESKEY_CAFES: Stores the actual list of cafes (list of IP addresses)
- CONFIG: Stores multiple configuration parameters (local CRE IP address and port, local SNME address and port, administrator's username and password, CME's AS number, BW_AGGREGATE, REFRESH_TIMEOUT, etc.)
- DECISION_PARAMS: Stores aspiration and reservation level values and types for decision process variables
- PATHINF: Path information received from CME-RAE interface
- PATHS: Paths' parameters
- PATH_ASES: The actual paths stored (list of AS numbers)
- PROVINF: Provisioning information received from CME-RAE interface
- PREFIXES_CAFES: Associates prefixes to IP addresses of CAFES
- PATHS_CAFES: Maps paths (list of AS numbers) to edge CAFES
- PATHCAFES_ASES: Stores the actual paths
- PATH_KEY: Stores the key for a specific path
- PATHKEY_ASES: Stores the actual path (list of AS numbers)
- USER_COS: Maps users' IP addresses to their assigned CoS
- STREAM_INFO: Contains configured streams' information

5.1.5.1 Description of functionality

The DB component is implemented using the Hibernate Framework [14], which is a Java persistence framework and contains 4 types of classes, the Data Access Objects (DAOs), which contain all required queries for the respective tables in CME database, the Data Transfer Objects

© Copyright 2012, the Members of the COMET Consortium

(DTOs) which map to the tables created in the database, the `PrefixesUtil` class, containing helper-methods and the `HibernateUtil` class, which initiates and provides the Hibernate session factory.

5.1.5.2 Interfaces

All DAOs implemented are described below:

- **`ConfigDAO`**
  - `public void insertAdmin():` Inserts administrator's username and password in Config table
  - `public void insertOneConfig(Config c):` Insert one row in Config table
  - `public void insertMultipleConfig(List<Config> configs):` Inserts multiple rows
  - `public void deleteAllConfigData():` Deletes all rows
  - `public String getConfigByParam(String param):` Queries a row for a config parameter name
- **`DecisionParamsDAO`**
  - `public void deleteAllDecisionParams():` Deletes all rows from Decision_Params table
  - `public List<DecisionParams> getAllDecisionParams():` Returns all rows
  - `public void insertDecisionParams(List<DecisionParams> params):` Insert multiple decision params
  - `public void insertOneParam (DecisionParams pr):` Inserts one row to table
  - `public String getTypeByVariable(String var):` Returns type for a decision variable
  - `public String getResLevelByVariable(String var):` Returns reservation level for a decision variable
  - `public String getAspLevelByVariable(String var):` Returns aspiration level for a decision variable
- **`CachedPathDAO`**
  - `public void deleteAllCachedPaths ():` Deletes all rows
  - `public void insertOnePathToCache (CachedPath path):` Inserts one row
  - `public void insertMultiplePathsToCache(List<CachedPath> paths):` Inserts multiple rows
  - `public List<CachedPath> getAllCachedPaths():` Returns all rows
  - `public List<CachedPath> getCachedPathsByPrefixesCos (Prefix src, Prefix dest, String cos):` Returns the cached paths for given source and destination prefixes and CoS
  - `public void deleteCachedPathsOnTTL(long current_time, long ttl_millis):` Deletes cached paths that their TTL has expired
- **`CafesKeyDAO`**
  - `public void deleteAllCafesKeys():` Deletes all rows of table
  - `public boolean insertCafeKey(CafesKey ck):` Insert one row in the table

© Copyright 2012, the Members of the COMET

- public List<CafesKey> getAllCafesKeys(): Returns all rows
- public CafesKey getCafeKeyById(String id): Returns a row for a given row ID
- public boolean updateReservedCapacity(String id, long bw_aggregate): Updates reserved capacity for a specific CAFEs pair.

- **PathKeyDAO**
  - public void deleteAllPathKeys (): Deletes all rows
  - public PathKey getPathKeyById (String id): Returns a row for a given ID
  - public boolean insertPathKey (PathKey pk): Inserts one row
  - public boolean updateUsedBW(String id, int br): Updates used bw with content's bit rate
  - public boolean removeUsedBW(String id, int br): Removes content's bit rate from used bw
- **PathsCafesDAO**
  - public void deleteAllPathsCafes(): Deletes all rows
  - public boolean insertPathCafe(PathsCafes pa): Inserts one row
  - public PathsCafes getPathCafeById(String id): Returns one row for a given ID
  - public List<PathsCafes> getAllPathsCafes(): Returns all rows
  - public List<String> getAllPaths(): Returns all stored paths
  - public boolean updateReservedCapacity(String id, long bw_aggregate): Updates reserved capacity for a specific AS path
- **PrefixesCafesDAO**
  - public void deleteAllPrefixesCafes(): Deletes all rows
  - public boolean insertPrefixCafe(PrefixesCafes pre): Inserts one row
  - public List<PrefixesCafes> getAllPrefixesCafes(): Returns all rows
  - public Prefix checkIPtoLocalPrefixes (String ip): Returns stored prefix for a given IP address
  - public String getCafeByPrefix (Prefix pre): Returns the IP address of the CAFE assigned to given prefix
- **ProvInfDAO**
  - public void deleteAllProvInfData(): Deletes all rows
  - public boolean deleteProvInf (DomainEdge src, DomainEdge dst, String cos): Deletes row for given source and destination edges and CoS
  - public ProvInf getProvInfByEdgesCos (DomainEdge src, DomainEdge dst, String cos): Returns row for given source and destination edges and CoS
  - public ProvInf getProvInfByPrefixesCos (Prefix src, Prefix dst, String cos): Returns row for given source and destination prefixes and CoS
  - public ProvInf getProvInfByPrefixEdge (Prefix src, int dstEdge, String cos): Returns row for given source prefix and destination AS number and CoS

© Copyright 2012, the Members of the COMET Consortium

- public void insertProvInf(DomainEdge src, DomainEdge dst, String cos, QoSParams params) : Insert one row to table

- **UserCosDAO**
  - public void deleteAllUserCos() : Deletes all rows
  - public boolean insertUserCos(String ip, String cos) : Inserts one row
  - public List<UserCos> getAllUserCos() : Returns all rows
  - public String getCosByIp(String ip) : Returns associated CoS of a user's IP address
- **StreamInfoDAO**
  - public void deleteAllStreamInfo() : Deletes all rows
  - public void insertStreamInfo (StreamInfo str) : Inserts one row
  - public boolean deleteStreamInfo (long streamid, String pathkeyid) : Deletes one row for specific id
  - public long getAvailableStreamID() : Returns the next available id

© Copyright 2012, the Members of the COMET

5.1.5.3 Design

5.1.5.3.1 Database schema

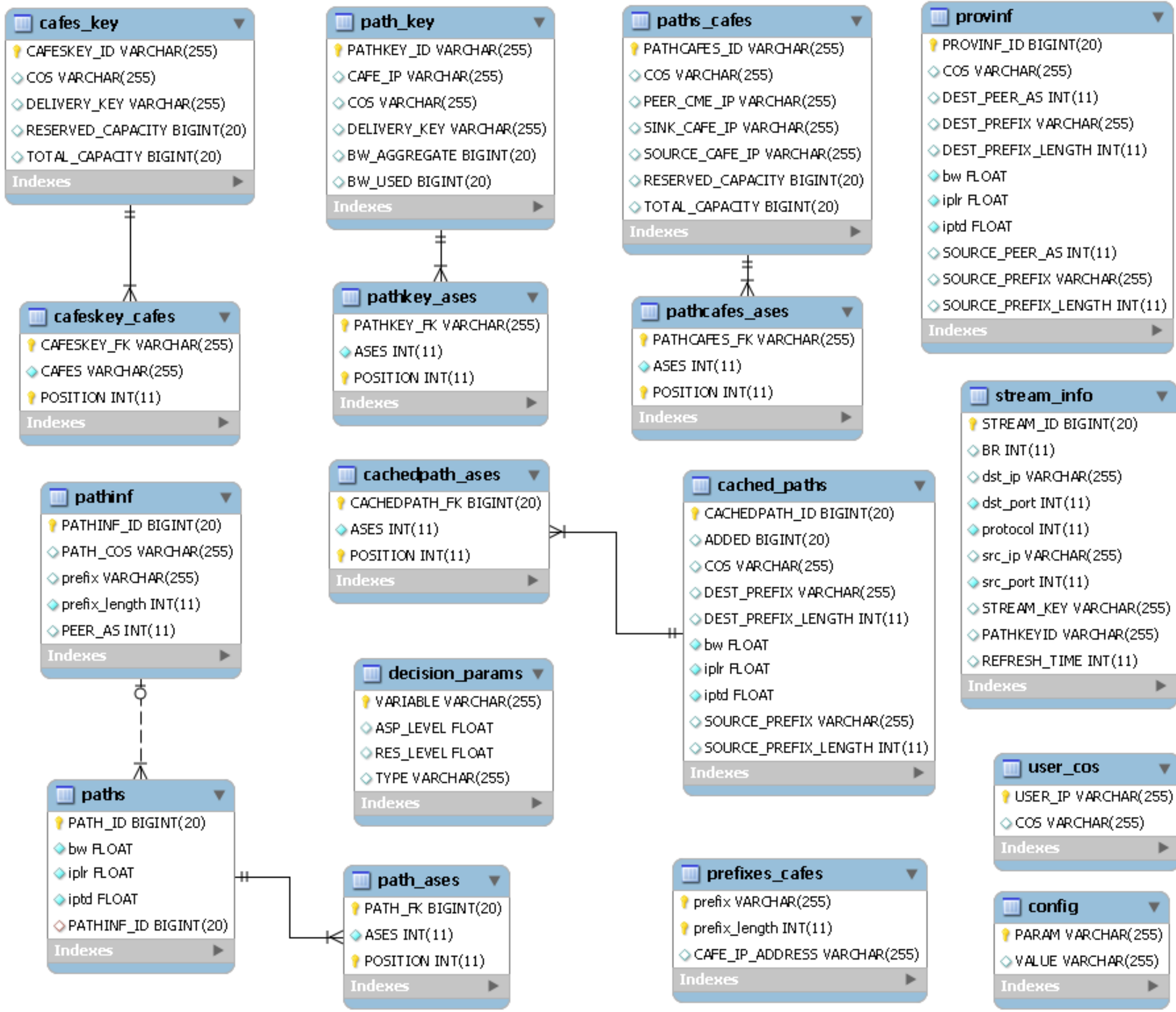


Figure 16: CME database schema

5.1.5.4 Testing and Test scenarios

JUnit [13] tests were provided for all DAOs of DB component, aiming to check the validity of all queries used. Therefore, each DAO has its respective test class with simple test data and scenarios. In addition, JUnit tests for `PrefixesUtil` class were also provided, to check if helper methods provide successful results:

- **`CachedPathTest`**
  - `public void testCachedPathQueries()`
- **`CafesKeyTest`**
  - `public void testCafesKeyQueries()`
- **`ConfigTest`**
  - `public void testConfigQueries()`
- **`DecisionParamsTest`**
  - `public void testDecisionParamsQueries()`
- **`PathInfTest`**

© Copyright 2012, the Members of the COMET Consortium

  - `public void testPathInfQueries()`
- **PathKeyTest**
  - `public void testPathKeyQueries()`
- **PathsCafesTest**
  - `public void testPathsCafesQueries()`
- **PrefixesCafesTest**
  - `public void testPrefixesCafesQueries()`
- **ProvInfTest**
  - `public void testProvInfQueries()`
- **UserCosTest**
  - `public void testUserCosQueries()`
- **PrefixesUtilTest**
  - `public void testPrefixesUtilMethods()`
- **StreamInfoTest**
  - `public void testAllStreamInfoQueries()`

5.1.6 Path Manager

Path Manager is the component responsible for handling all path-related operations of CME, and more specifically, path storage, discovery, configuration and provisioning and CAFE configuration processes.

5.1.6.1 Description of functionality

All classes of Path Manager exist under `pathmanager` package and are divided in 3 main categories:

- Path-related processes' interfaces and classes implementing them:
  - `PathStorage` and `PathStorageImpl`, containing methods for handling and storing messages received in CME-RAE interface
  - `PathDiscovery` and `PathDiscoveryImpl`, containing methods for retrieving paths, either from local database or from server-side CME
  - `PathConfiguration` and `PathConfigurationImpl`, responsible for configuring selected path, both in client- and server-side CMEs
  - `PathProvisioning` and `PathProvisioningImpl`, containing methods for provisioning selected path
  - `CafeConfiguration` and `CafeConfigurationImpl`, containing methods for configuring server-side CAFE
- Inter-CME client and helper classes:
  - `InterCmeClient` and its method `send(GenericRequest req, String cmeip)`, are used for opening a connection with another CME and sending a request.
  - `InterCmeHandler` extending JBoss Netty's `SimpleChannelUpstreamHandler`, used for handling responses received from other CMEs
  - `InterCmePipelineFactory`, implementing `ChannelPipelineFactory`, configuring the pipeline for the `InterCmeClient`, and including all required encoders and decoders for protobuf messages.
- CME-CAFE client and helper classes:
  - `CafeClient` and its method `send(GenericRequest req, String cafeip)`, are used for opening a connection with CAFE and sending a request.

© Copyright 2012, the Members of the COMET

    - `CafeHandler` extending JBoss Netty's `SimpleChannelUpstreamHandler`, used for handling responses received from CAFEs.
    - `CafePipelineFactory`, implementing `ChannelPipelineFactory`, configuring the pipeline for the `CafeClient`, and including all required encoders and decoders for protobuf messages.
- `PathManagerUtil` and `CafeUtil` class, containing helper methods for all processes.
- `ResourceManager` class, containing methods for resource management in CME.

5.1.6.2 Interfaces

- **`PathStorage`**
    - `public void processRAEdata(GenericRequest req)`: The method processing GenericRequest messages received by RAE. There are 6 types of messages that can be processed, RESET, VERSION, INSERT_PROVISIONING, REMOVE_PROVISIONING, INSERT_PATHS, REMOVE_PATHS.
- **`PathDiscovery`**
    - `public List<CachedPath> retrievePathsFromPathStorage(String clip, String srvip, String cos)`: The method called in server-side domain when "RETRIEVE_PATHS" message is received in `CmeHandler`.
    - `public Map<String, List<CachedPath>> retrievePaths(String ccip, List<String> srvips, String cos, String cmeip)`: The method called in client-side domain after successful name resolution. It checks local db, and if they don't exist, sends a "RETRIEVE_PATHS" request to server-side CME.
- **`PathConfiguration`**
    - `public List<String> processClientSide(String ccip, List<Integer> path, String cos)`: The method used to configure client-side CME after successful decision process
    - `public boolean sendProcessServerSideRequest(List<Integer> path, String ccip, String srvip, String trans_proto, String trans_port, String cmeip, String br, String cos, List<String> key)`: The method used to send and receive process server-side requests and responses after successful client-side configuration.
    - `public ProcessServerSideResponse processServerSide (ProcessServerSideRequest r)`: The method used to process server-side CME after successful client-side processing
    - `public boolean processClientServerSide(List<Integer> path, String ccip, String srvip, String trans_proto, String trans_port, String cmeip, String br, String cos)`: The method used to process client-side CME when both content server and client belong to local domain
- **`PathProvisioning`**
    - `public PathKey startPathProvisioning(List<Integer> path, String cos, List<String> key_client)`: The method initiating and finishing Path provisioning in server-side CME, if key for received path does not exist. It initiates path provisioning process towards client-side CME.

© Copyright 2012, the Members of the COMET Consortium

- `public ProvisionPathResponse handleProvisionRequests (ProvisionPathRequest r)`: The method handling path provisioning requests in any CME. It checks if CME is final or transit and responds accordingly.

- **CafeConfiguration**
  - `public boolean configureStream(String cafeip, List<Integer> path, String ccip, String srvip, String trans_proto, int trans_port, int br, String cos, List<String> key)`: The method used to connect to all adjacent CAFEs and collect their expired streams.
  - `public List<StreamInformation> collectExpiredStreams()`: The method used to configure a stream to the respective CAFE.
- **InterCmeClient**
  - `send(GenericRequest req, String cmeip)`: The method used to send inter-CME requests to other CMEs and return inter-CME responses.
- **CafeClient**
  - `send(GenericRequest req, String cafeip)`: The method used to send requests to CAFE and return ACK or NACK response.

5.1.6.3 Design

5.1.6.3.1 Class Diagrams

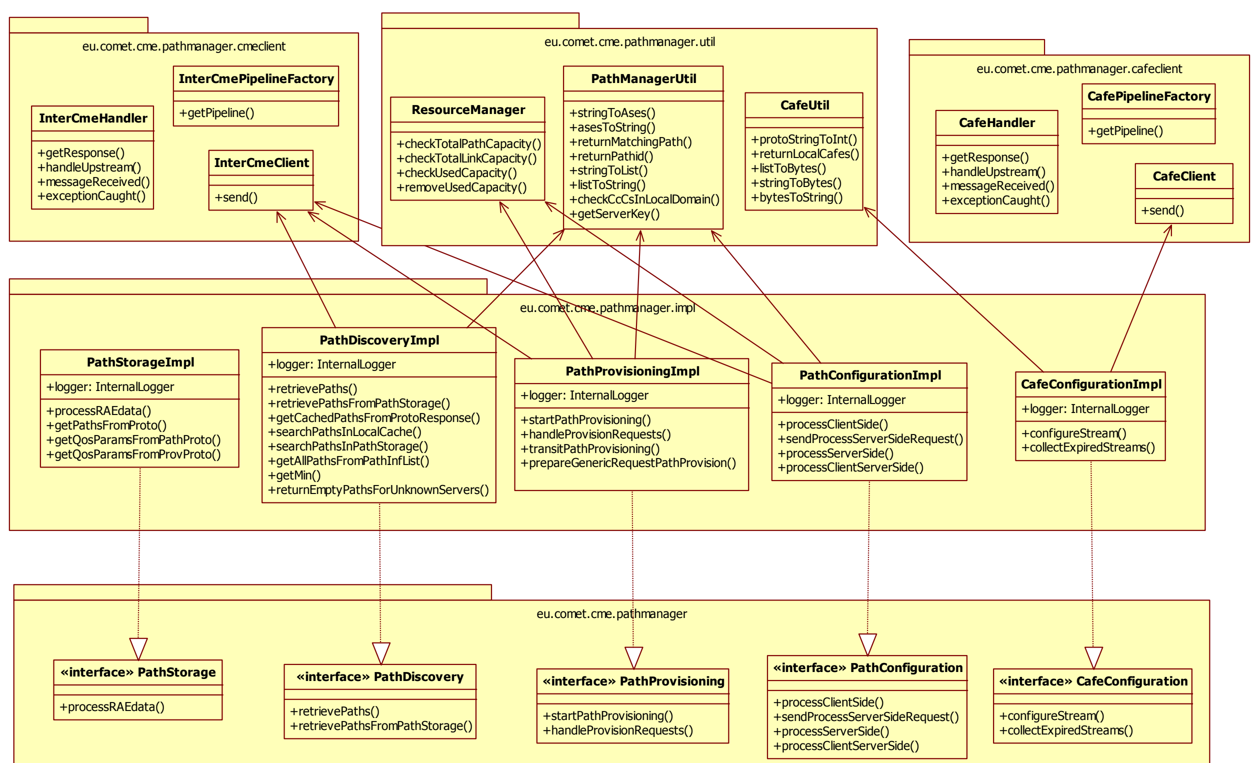


Figure 17: Path Manager class diagram

Figure 17 presents the class diagram for the Path Manager component. `PathStorageImpl`, `PathDiscoveryImpl`, `PathConfigurationImpl`, `PathProvisioningImpl`, and `CafeConfigurationImpl` implement their respective interfaces `PathStorage`, `PathDiscovery`, `PathConfiguration`, `PathProvisioning` and `CafeConfiguration`. In addition, `PathDiscoveryImpl` uses the `InterCmeClient`, and `PathManagerUtil` classes,

© Copyright 2012, the Members of the COMET

`PathProvisioningImpl` invokes methods from `InterCmeClient`, `ResourceManager` and `PathManagerUtil` classes, `PathConfigurationImpl` uses `InterCmeClient`, `ResourceManager` and `PathManagerUtil` classes, while `CafeConfigurationImpl` calls methods from `CafeClient` and `CafeUtil` classes.

5.1.6.3.2 Sequence Diagrams

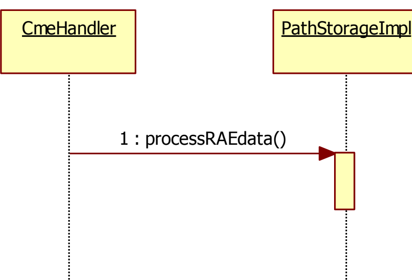


Figure 18: Path storage sequence diagram

Figure 18 presents the sequence diagram of the path storage process. The operation is invoked from `CmeHandler`, whenever a message is received in rae-cme interface. `PathStorage` interface then checks the type of message and performs the requested-by-RAE operation (storage or removal of path or provisioning information).

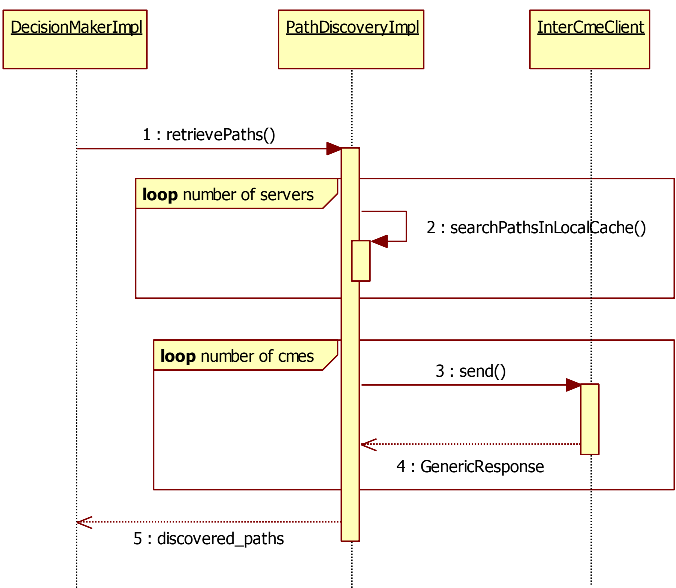


Figure 19: Sequence diagram of path discovery in client domain

Figure 19 presents the path discovery process in the domain attached to content client, when there are no paths stored in local cache. The process is invoked by the Decision Maker component, through the `retrievePaths` method of the `PathDiscovery` interface, and then for all the content servers within the request, local database is checked for cached paths. When there are no cached paths, then content servers are grouped per their adjacent CMEs and for each CME, a request is sent to the inter-CME interface, waiting for a response with discovered paths.

© Copyright 2012, the Members of the COMET Consortium

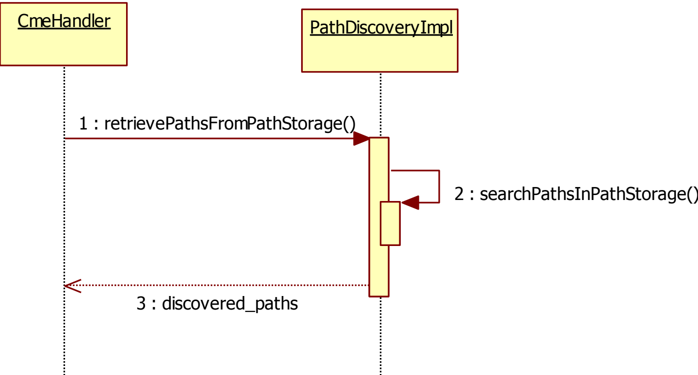


Figure 20: Sequence diagram of path discovery in server domain

On the other hand, Figure 20 presents the path discovery process performed in the domain attached to a content server. Path Discovery process is invoked by `CmeHandler` whenever a RETRIEVE_PATHS message is received in inter-CME interface, and the `retrievePathsFromPathStorage` method of the `PathDiscovery` interface is called, aiming to return all discovered paths stored in local database.

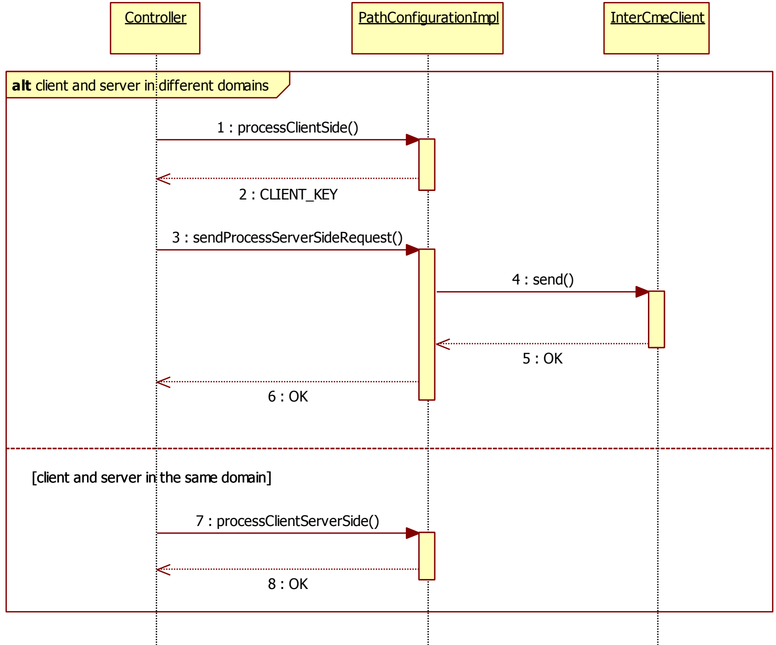


Figure 21: Sequence diagram of path configuration in client domain

© Copyright 2012, the Members of the COMET

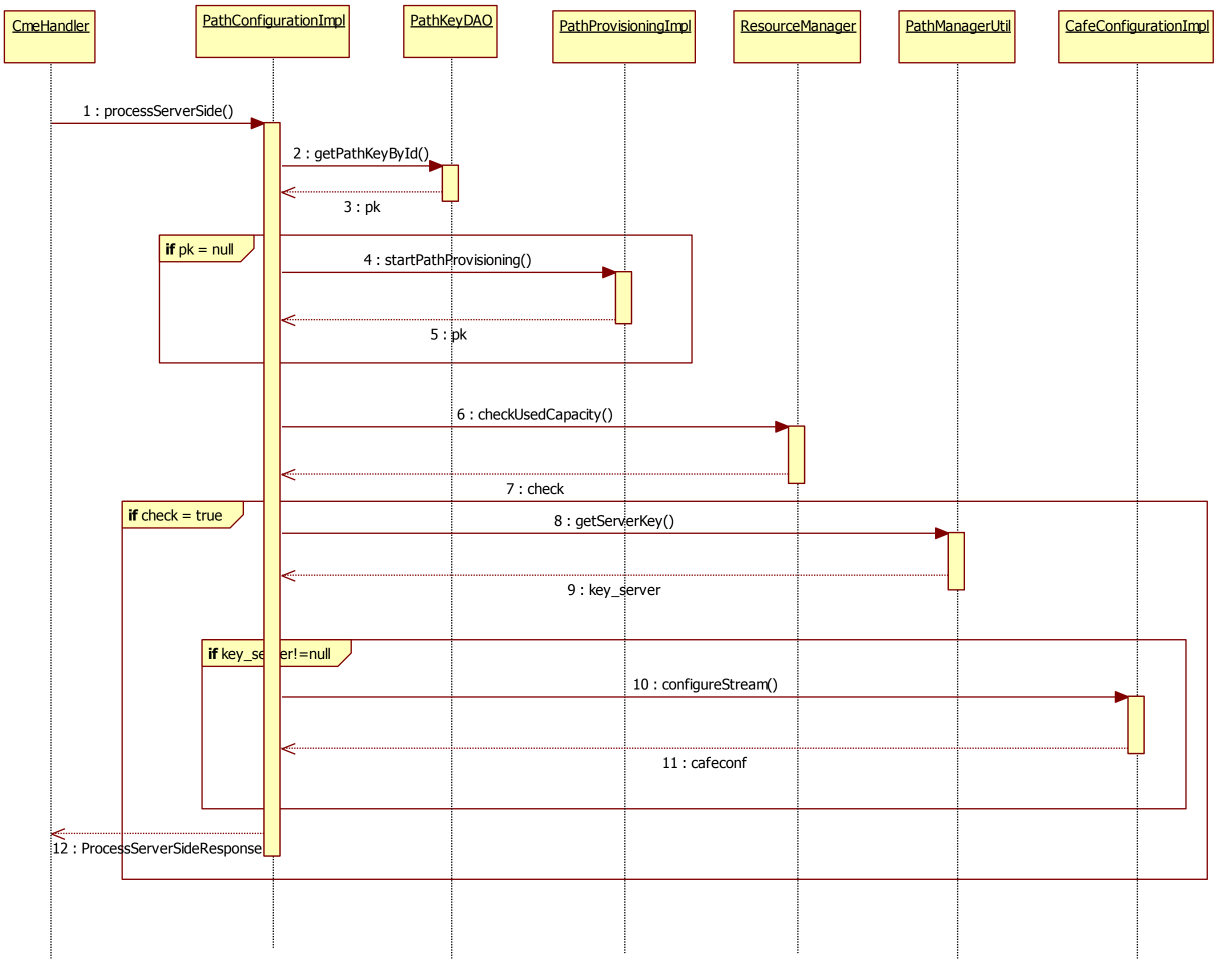


Figure 22: Sequence diagram of path configuration in server domain

Figures Figure 21 and Figure 22 present the path configuration processes in client and server domains accordingly. More specifically, path configuration is initiated in client domain by the `Controller`, which calls the `processClientServerSide` method of `PathConfiguration` interface (if content client and server belong to the same domain) or the `processClientSide` method and then sends a PROCESS_SERVER request to the server-side CME by the send method of `InterCmeClient` (if content client and server belong to different domains).

In the server-side CME, path configuration is invoked by the `CmeHandler` whenever a PROCESS_SERVER request is received in inter-CME interface and then the `processServerSide` method is called, aiming to find or produce the key of the selected path and finally configure the CAFE, adjacent to content server. Initially, local database is checked for stored keys, and if there aren't any, path provisioning process is initiated through the `startPathProvisioning` method of `PathProvisioning` interface. Then, `ResourceManager` is invoked in order to check if there are available resources for the selected path, and if successful, the key_`server` is produced via `PathManagerUtil` class. If all operations are successful, then CAFE configuration is invoked, and a response is sent back to `CmeHandler`.

© Copyright 2012, the Members of the COMET Consortium

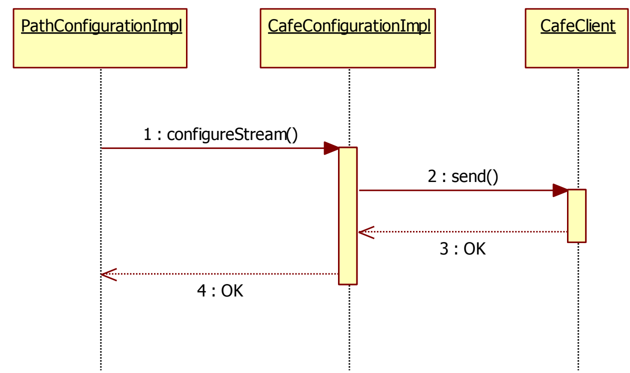


Figure 23: CAFE configuration sequence diagram

In Figure 23, CAFE configuration process is presented, in which `configureStream` method of `CafeConfiguration` interface is invoked, and a CONFIGURE_STREAM request is sent to CAFE, using the `send` method of `CafeClient` class.

5.1.6.4 Testing and Test scenarios

Since path-related operations involve inter-CME or CME-CAFE communication in most cases, JUnit [13] tests were provided for cases when required data exist in local CME. 7 classes were created for these tests:

- **`PathConfigurationTest`**
  - `public void testProcessClientSide()`: used for testing successful client-side configuration.
  - `public void testSendProcessServerSideRequest()`: method for testing path configuration interface when no server-side CME is running.
  - `public void testProcessClientServerSide()`: used for testing client-side configuration when both client and server belong to the same domain.
  - `public void testProcessServerSide()`: method used for testing server-side configuration.
- **`CafeUtilTest`**
  - `public void testCafeUtilMethods()`: method for testing all methods of CafeUtil.
- **`PathManagerUtilTest`**
  - `public void testPathManagerUtilMethods()`: method for testing all methods of PathManagerUtil.
- **`ResourceManagerTest`**
  - `public void testResourceManagerMethods()`: method for testing all methods of ResourceManager.
- **`CafeConfigurationTest`**
  - `public void testCafeConfigurationMethods()`: method for testing all methods of CafeConfiguration interface.
- **`PathDiscoveryTest`**
  - `public void testPathDiscoveryAtServerSide()`: method for testing if path discovery is performed correctly in server-side CME.

© Copyright 2012, the Members of the COMET

  - `public void performLocalCacheQueries()`: method for testing path discovery in client-side CME, when cached paths exist in local db.
- **PathProvisioningTest**
  - `public void testStartPathProvisioning()`: method for testing initialization of path provisioning.
  - `public void testHhandleProvisionRequests()`: method used to test handling of path provisioning requests.

5.1.7 Server Manager

Server Manager is the CME component responsible for gathering servers' loads, either from CMEs adjacent to servers or directly from attached SNME.

5.1.7.1 Description of functionality

`ServerAwarenessImpl` is the key class of the component, implementing the `ServerAwareness` interface, while `snmeclient` package contains the `SnmeClient`, `SnmeHandler` and `SnmePipelineFactory` classes, which are used to contact SNME in the CME-SNME interface.

5.1.7.2 Interfaces

- **ServerAwareness**
  - `public Map<String, Integer> getServersLoadFromEdgeCME (List<String> srvips, String cmeip)`: The method that handles client-side requests and sends to inter-CME interface for server loads. In case that CME is attached to the server, then loads are requested directly from local SNME.
  - `public List<ServerLoadResponse> getServersLoadFromSNME (List<ServerLoadRequest> slreqs)`: The method that sends and receives server load queries to CME-snme interface. It may return null if no response received or no snme info exist in db.
- **SnmeClient**
  - `ResponseServerStatus send(QueryServerStatus req, String snmeip, String snmeport) throws Exception`: The method used to send requests to SNME and return responses back to Server Manager component.

© Copyright 2012, the Members of the COMET Consortium

5.1.7.3 Design

5.1.7.3.1 Class Diagrams

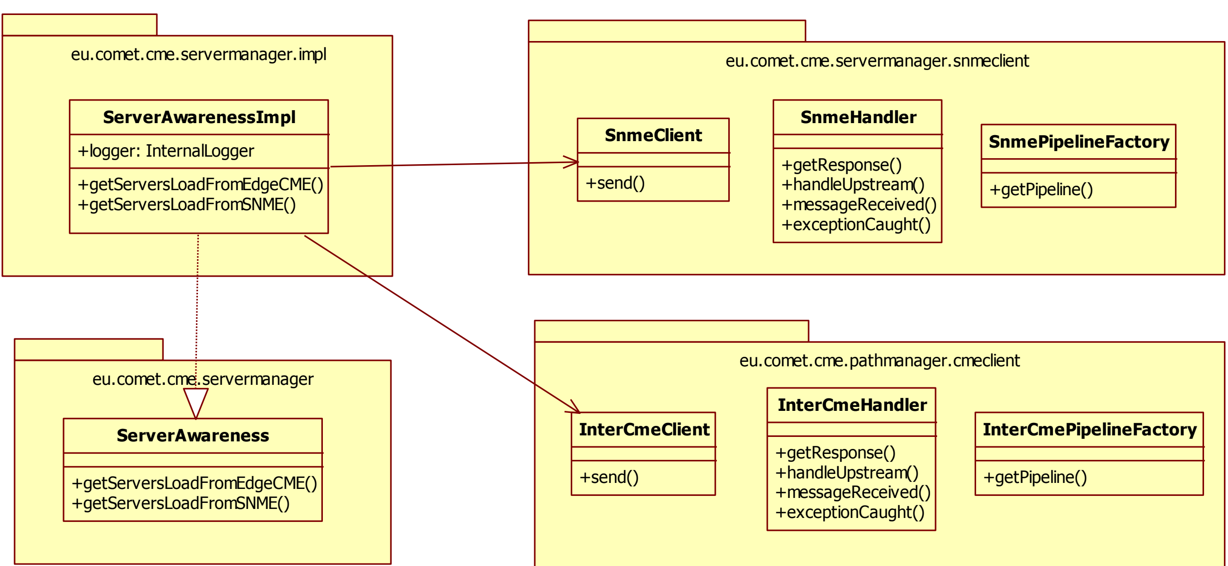


Figure 24: Class diagram of Server Manager

Figure 24 presents the class diagram of the Server Manager component. `ServerAwarenessImpl` class implements the methods of the `ServerAwareness` interface and also uses the `SnmeClient` and `InterCmeClient` classes for connecting to SNME and remote CME accordingly, as well as their respective helper classes.

5.1.7.3.2 Sequence Diagrams

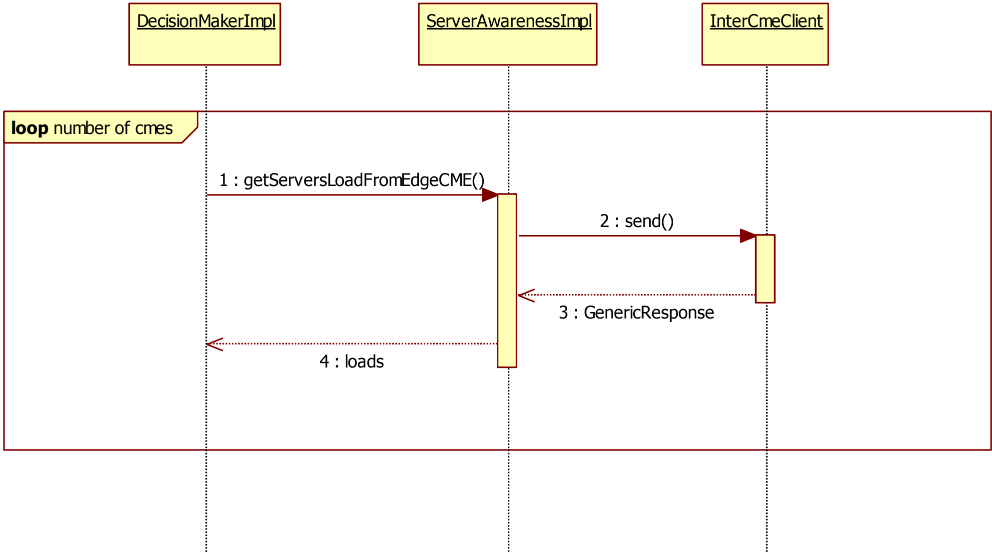


Figure 25: Sequence diagram of server awareness in client domain

© Copyright 2012, the Members of the COMET

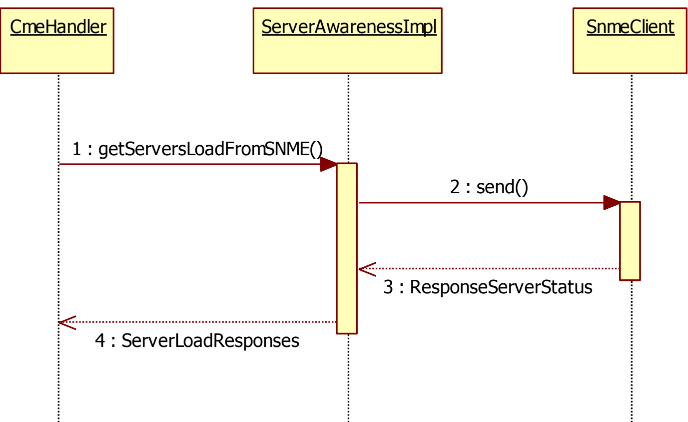


Figure 26: Sequence diagram of server awareness in server domain

Figure 25 and Figure 26 present the sequence diagrams of server awareness process in client and server domains accordingly. In the client domain, server awareness is initiated by `DecisionMakerImpl` through `getServersLoadFromEdgeCME` method of the `ServerAwareness` interface (for all edge CMEs within the content record), which then sends a SERVER_LOAD request to remote CME, using the send method of `InterCmeClient`.

In the server-side domain, `CmeHandler` receives the SERVER_LOAD request and calls the `getServersLoadFromSNME` method of the `ServerAwareness` interface, aiming to send `ServerStatus` requests to adjacent SNME, using the `SnmeClient` class. Finally, received loads are returned to `CmeHandler`.

5.1.7.4 Testing and Test scenarios

JUnit [13] tests were provided for failure case of server awareness component (missing CME-SNME communication, etc.) from `ServerAwarenessTest` class:

- `public void testNoResponseFromCME()`: Tests the case when there is no communication between CME and SNME.

5.1.8 Admin

Admin is the web administration component of the CME, responsible for configuring required parameters for almost all operations performed by CME:

- Password update (Profile Update)
- Root CRE IP address and port (Name Resolution)
- Cached Paths TTL (Path Storage)
- Maximum number of server/path candidates and decision process variables (type, aspiration and reservation level) (Decision Process)
- SNME IP address and port (Server Awareness)
- Tables for IP Prefixes-CAFEs mapping, AS peering-CAFEs mapping and CAFEs peering-key mapping, as well as aggregate value for BW, refresh time and CME's AS number (Path Configuration)
- Users' IP addresses and CoS mapping (Content Resolution)

5.1.8.1 Description of functionality

Admin component contains 2 types of classes under package `admin`, the `WebSrv` class which starts the embedded Jetty web server [9] and deploys the admin web application and the `Controller`

© Copyright 2012, the Members of the COMET Consortium

class under `admin.web` package containing all methods for handling parameters inserted by the administrator in the web interface.

5.1.8.2 Interfaces

Figure 27, Figure 28, Figure 29, Figure 30, Figure 31, Figure 32, Figure 33, Figure 34 and Figure 35 present some screenshots from CME Administration web page for logging in, updating administrator's password, configuring root CRE's IP address and port, cached paths TTL, decision variables, SNME configuration, path configuration properties and users' CoS respectively. CME administrator can access the webpage at http://{cme.ip.address}:8090/CME/index.jsf.

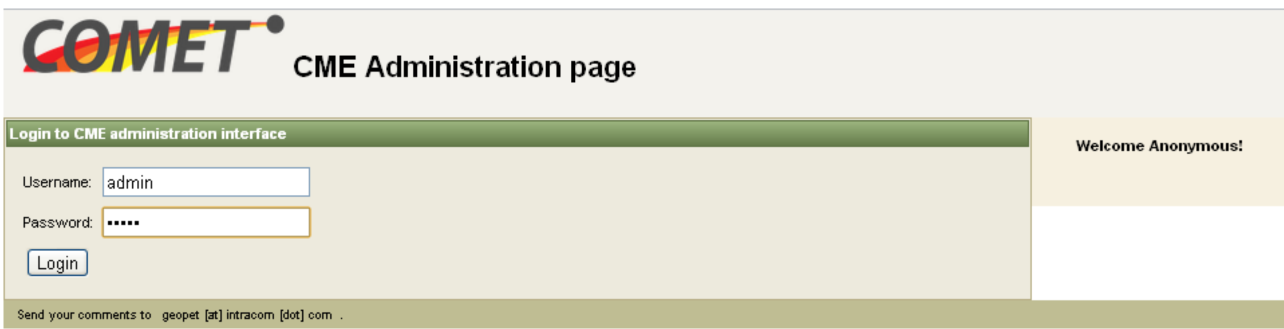


Figure 27: Login page

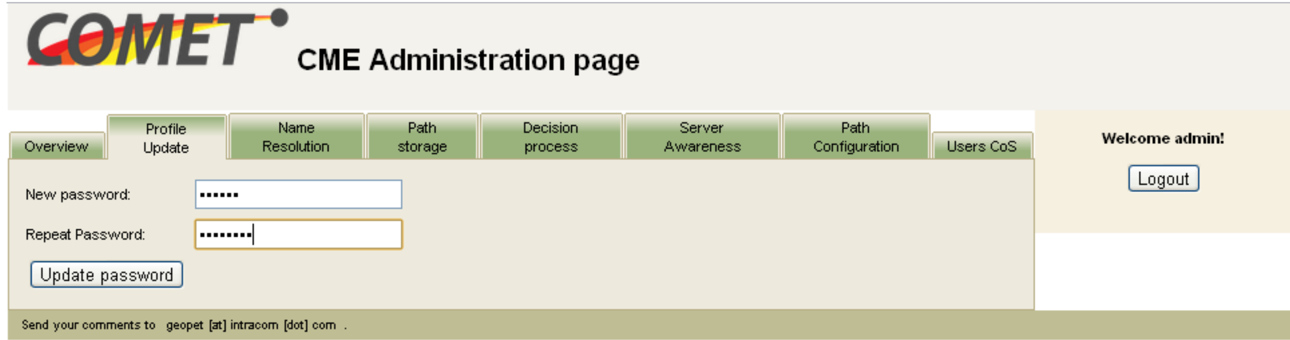


Figure 28: Profile Update

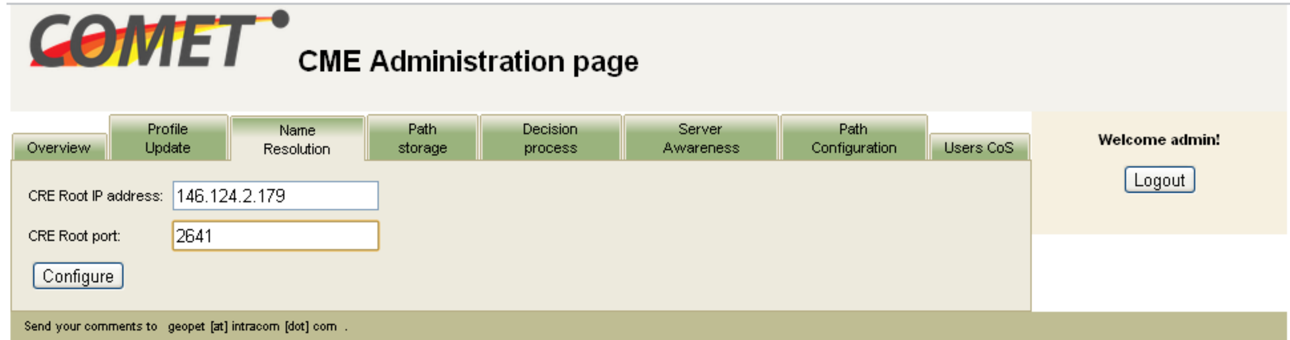


Figure 29: root CRE configuration

© Copyright 2012, the Members of the COMET

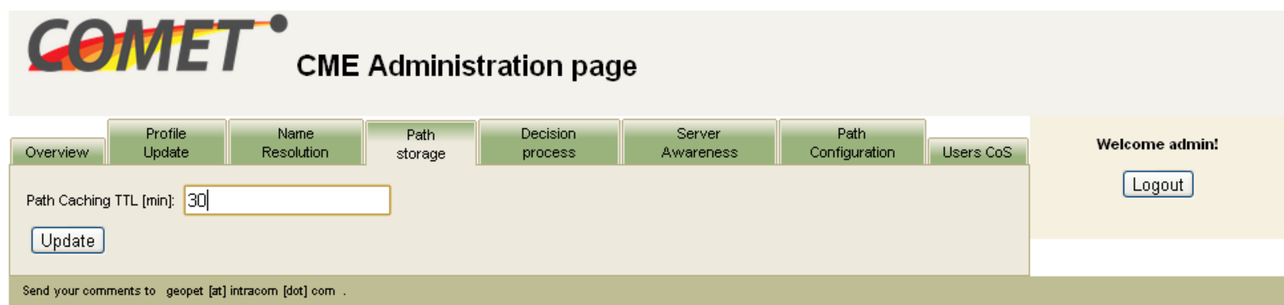


Figure 30: Cached paths TTL configuration

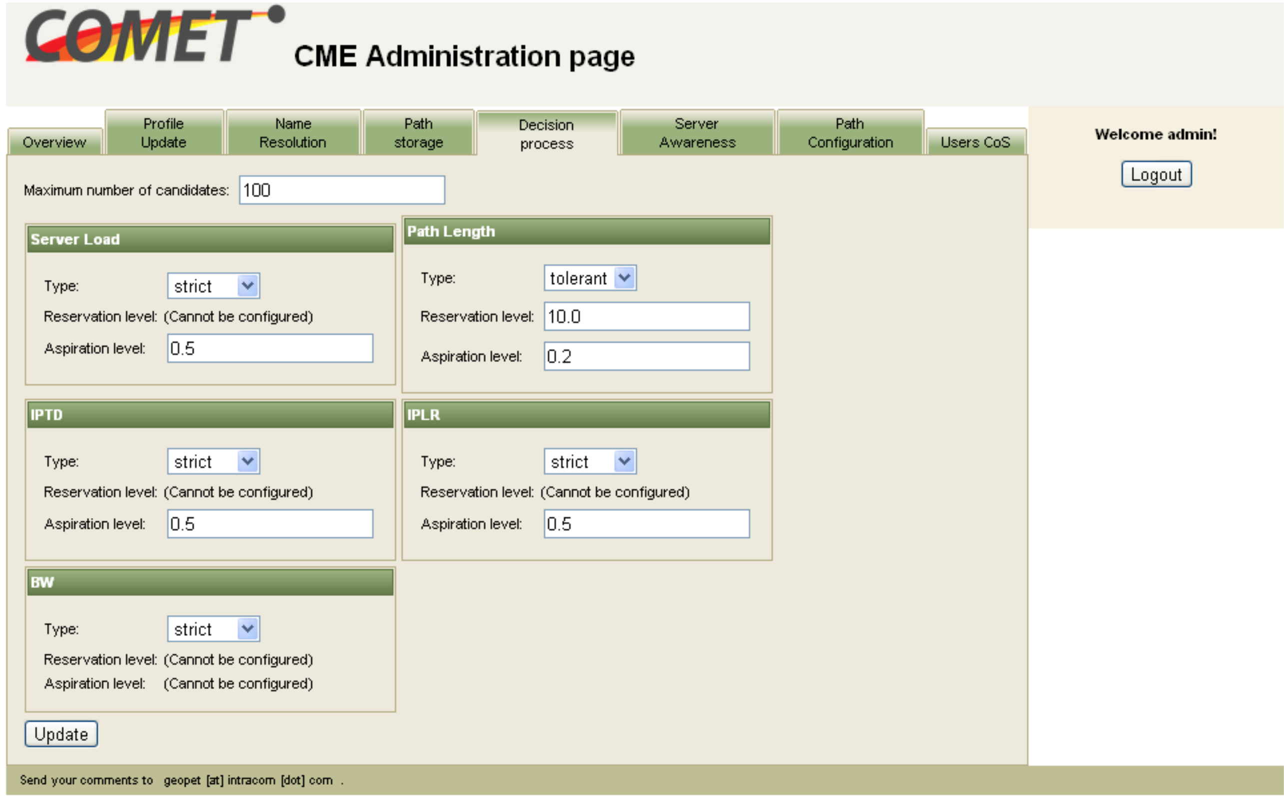


Figure 31: Decision variables configuration

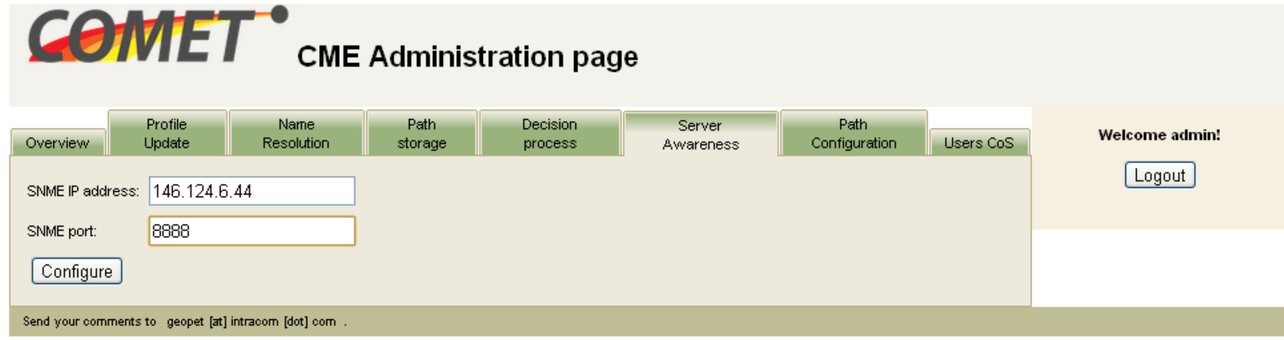


Figure 32: SNME configuration

© Copyright 2012, the Members of the COMET Consortium

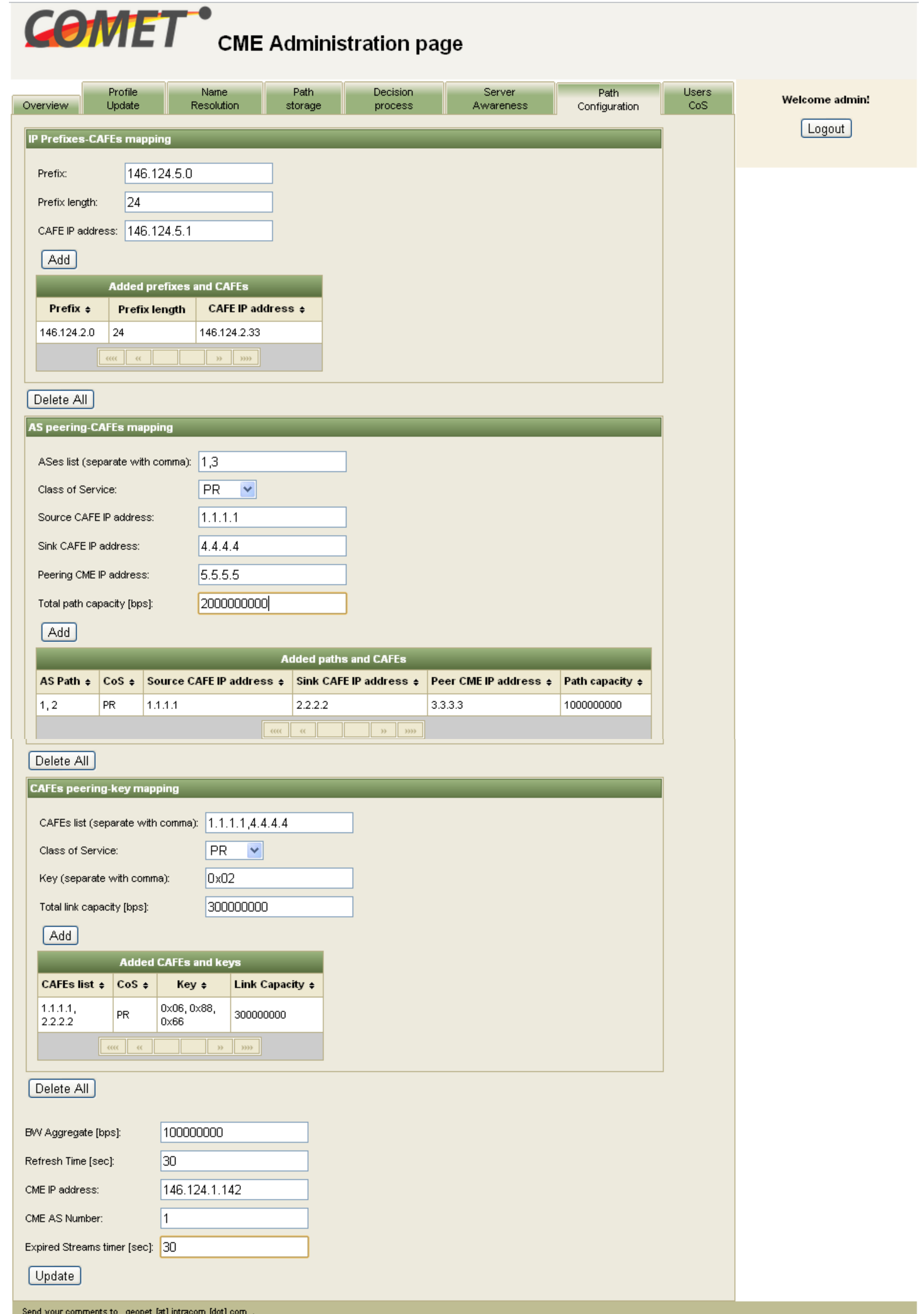


Figure 33: Path Configuration parameters configuration

© Copyright 2012, the Members of the COMET

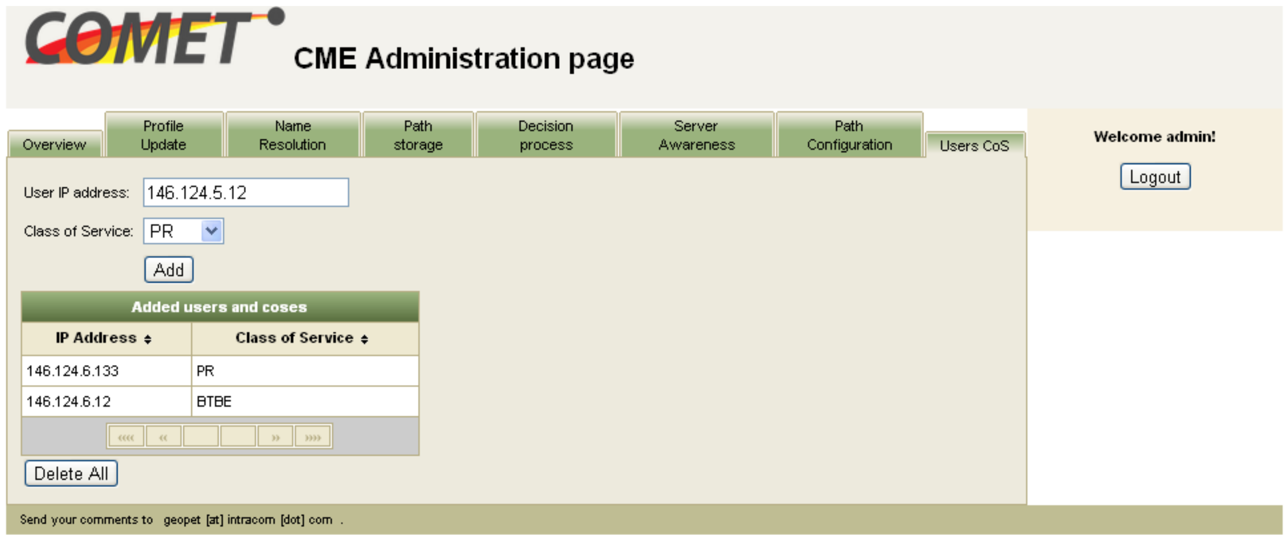


Figure 34: User-CoS mapping

COMET

CME Administration page

Overview | Profile Update | Name Resolution | Path storage | Decision process | Server Awareness | Path Configuration | Users CoS

Welcome admin!

Logout

Name Resolution

Path Storage

Decision Process

Server Awareness

Path Configuration

Added prefixes and CAFEs

Prefix	Prefix length	CAFE IP address
146.124.2.0	24	146.124.2.33

Added paths and CAFEs

AS Path	CoS	Source CAFE IP address	Sink CAFE IP address	Peer CME IP address	Path capacity
1, 2	PR	1.1.1.1	2.2.2.2	3.3.3.3	1000000000

Added CAFEs and keys

CAFEs list	CoS	Key	Link capacity
1.1.1.1, 2.2.2.2	PR	0x06, 0x88, 0x66	300000000

BW Aggregate [bps]: 100000000
Refresh Time [sec]: 30
CME IP address: 146.124.6.14
CME AS Number: 1
Expired Streams timer [sec]: 30

Users CoS

Send your comments to geopet [at] intracom [dot] com .

Figure 35: Configuration overview

© Copyright 2012, the Members of the COMET Consortium

5.1.8.3 Design

The Controller class consists of the following methods, which handle inserted parameters from user:

- `public String login ()`: used for handling username and password from login page and authenticating administrator
- `public String configurecre()`: method for configuring CRE IP address and port
- `public String updatepassword()`: method for updating administrator's password
- `public String addusercos()`: method for handling users' IP addresses and CoS mapping
- `public String deleteallusercos()`: method for deleting all existing users-CoS mappings
- `public String addpathstorage()`: used for configuring cached paths TTL
- `public String adddecisionparams()`: used for configuring decision variables
- `public String configuresnme()`: used for configuring SNME properties
- `public String addprefixcafe()`: method for handling prefixes-CAFEs mapping
- `public String deletet1()`: method for deleting all prefixes-CAFEs mappings
- `public String addt2()`: used for handling AS paths-CAFEs mapping
- `public String deletet2()`: used for deleting all AS paths-CAFEs mappings
- `public String addt3()`: used for handling CAFEs-key mapping
- `public String deletet3()`: used for deleting all CAFEs-key mappings
- `public String addpathconfparams()`: method for configuring path configuration properties
- `public String logout()`: used for logging out from web interface

5.1.8.4 Testing and Test scenarios

No particular self-contained tests have been implemented for testing CME web application. Manual tests were performed during software's validation, in order to prove that it works as expected.

5.2 Content Resolution Entity

5.2.1 Description of overall functionality

CRE is the COMET entity responsible for storing content records and responding to content name resolution requests during content consumption. It has interfaces to:

- CME
- CP

CRE is a Handle System server, using the Handle System API [11] to store and update content records and resolve content names, hence its internal structure and procedures are transparent to COMET. 2 types of CREs are introduced and implemented, *local CRE*, containing all content records for one or more naming authorities and *root CRE*, containing the IP address of the local CRE per naming authotiry.

5.2.2 Interfaces

As previously stated, CRE interacts with:

- Resolver component of CME during name resolution
- CP during content publication

© Copyright 2012, the Members of the COMET

For both operations, CRE exposes an interface at specific TCP or UDP port (default is 2641). Both Resolver component of CME and CP, use the defined Handle System protocol [6][7] in their interaction with CRE.

5.2.3 Design

CRE source code consists of one package, `eu.comet.cre.handleserver`, in which 4 classes are included: `SimpleSetup`, `DBTool`, `ConfigCommon` and `StartServer`, which extend particular classes from the Handle System API (as depicted in Figure 36). Attributes and methods are not shown in the class diagram, since they were implemented from Handle System.

`SimpleSetup` class is responsible for setting up the CRE and creating all required resources and files to run and uses certain methods from the supporting classes `DBTool` and `ConfigCommon`, while `StartServer` class is used to run and shutdown the CRE.

Currently, CRE stores content records and responds to content name resolution requests for naming authorities specified in its configuration files.

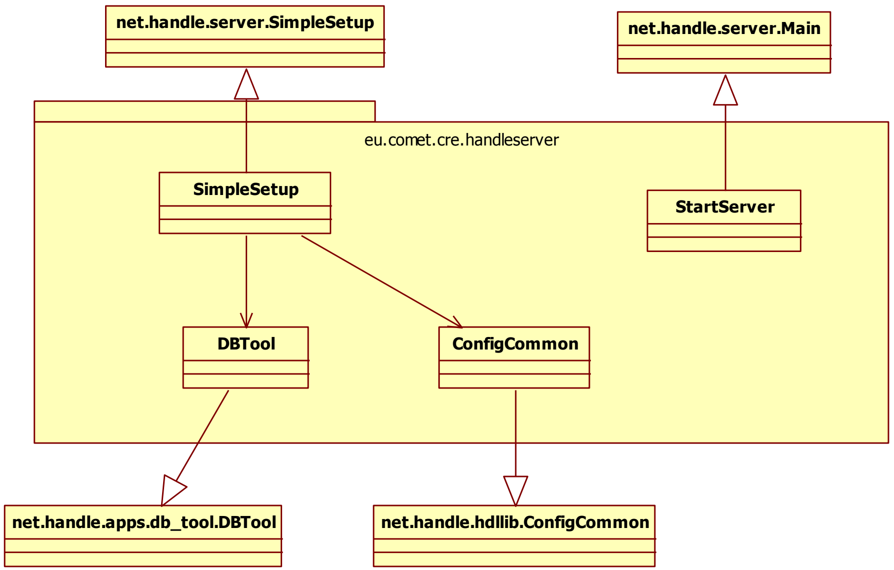


Figure 36: CRE class diagram

5.2.4 Testing and Test scenarios

Currently, there are no self-contained tests for the CRE.

5.3 Content Publisher

5.3.1 Description of overall functionality

CP is the COMET entity responsible for content publication. More specifically, it is used to create, update and delete content records from the CRE database. It is a web application which interacts with a local CRE, using required classes and methods from the Handle System API [11].

Content Publisher application can be accessed through any web browser, by any user with administrative credentials. It is assumed that a Content Provider has already registered to COMET its naming authority (e.g. “com.gmail@user”) and its admin handle (e.g. com.gmail@user/ADMIN) is stored in its respective local CRE database, including administrator’s username and password.

© Copyright 2012, the Members of the COMET Consortium

Content Record parameters consist of:

- Content Name (CN)
- Content ID (CID)
- List of Content Sources:
    - MIME type
    - Class of Service (CoS)
    - Bit Rate (BR)
    - IP Transfer Delay (IPTD)
    - IP Loss Ratio (IPLR)
    - Priority
    - Application protocol
    - Transport protocol
    - Transport port
    - List of Content Servers:
        - IP address
        - Server path
        - CME IP address

5.3.2 Interfaces

CP currently has 2 interfaces:

- CP-CRE interface, for handle system authentication, content record registration, deletion or update
- Web interface, in order to be accessed through every web browser at http://{cp.ip.address}:8090/Publisher/index.jsf.

Figure 37, Figure 38, Figure 39, Figure 40 and Figure 41 present features of the Content Publisher web interface, for logging in, creating, deleting, updating content records, and processing batch files.

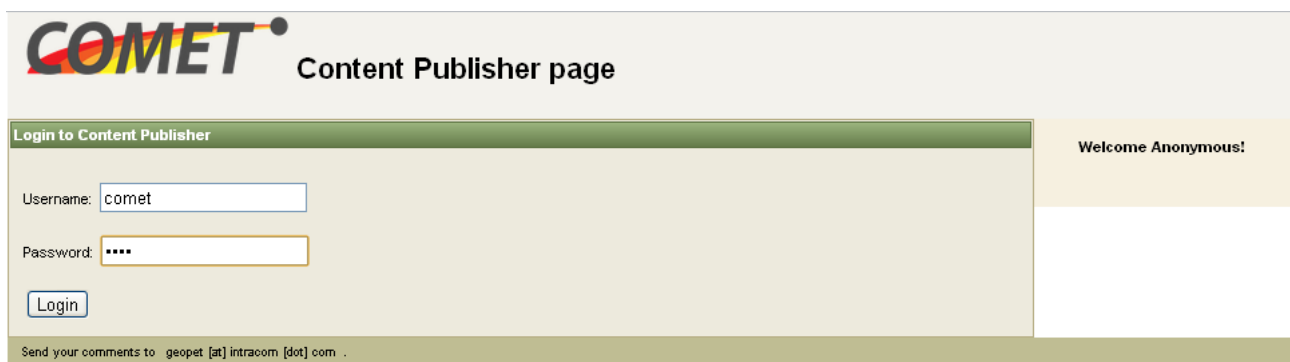


Figure 37: Login page

© Copyright 2012, the Members of the COMET

Figure 38: Content Record Creation

Figure 39: Content Record Deletion

© Copyright 2012, the Members of the COMET Consortium

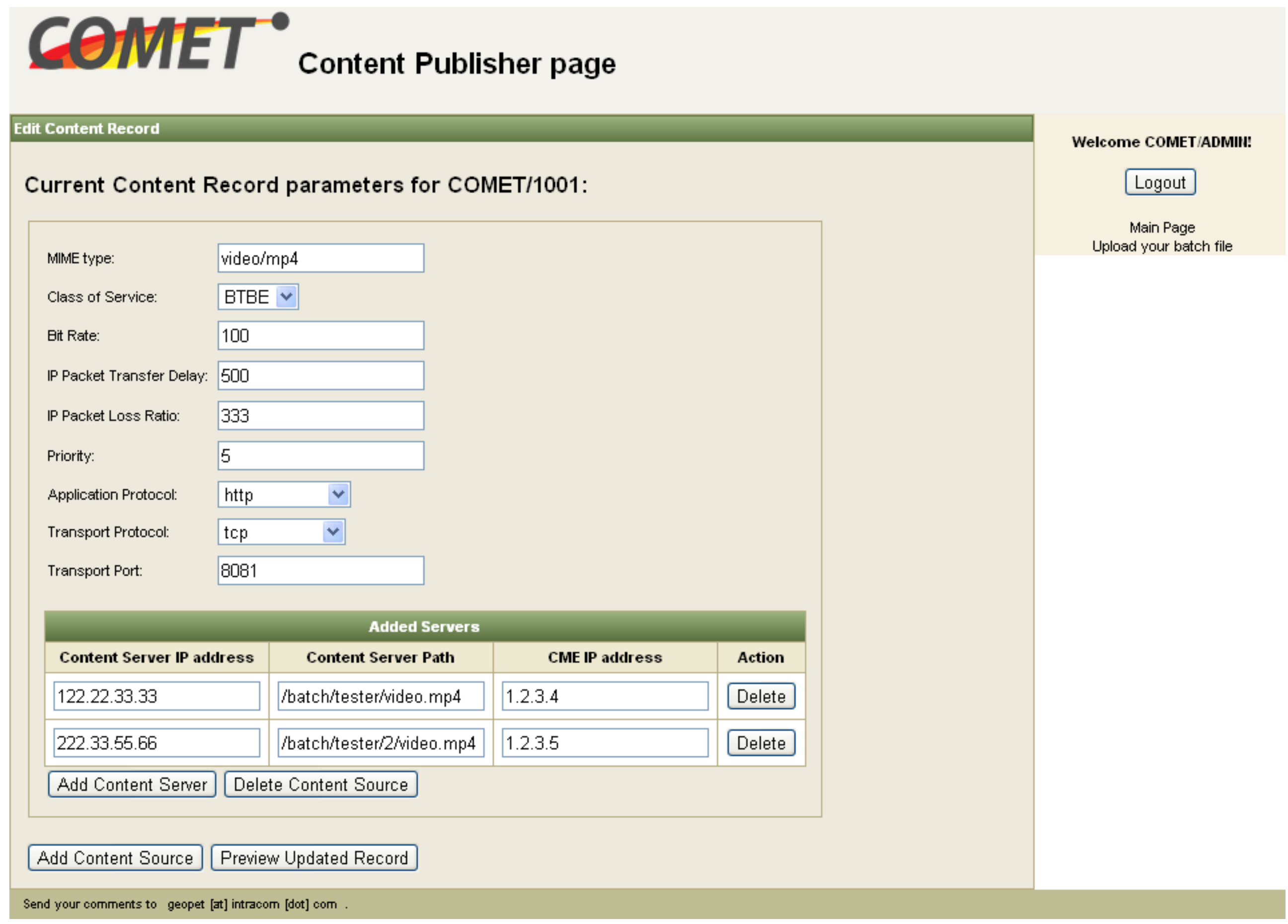


Figure 40: Content Record Update

Figure 41: Batch file processing

5.3.3 Design

Content Publisher source code consists of 3 packages:

- `eu.comet.cp.handle`, containing:
    - `HandleClient` class, which includes all methods for making Handle system requests [11]:
        - `public AbstractResponse sendRequest(AbstractRequest request) throws HandleException, IOException`: The method for sending a request to CRE using the Handle System protocol

© Copyright 2012, the Members of the COMET

    - public int authenticate(String adminHandle, String password) throws HandleException, IOException: The method for authenticating a user
    - public List<String> listhandles(String adminHandle, String password) throws HandleException, IOException: The method for requesting all the content records under a specific naming authority
    - public int createHandle(String handle, ContentRecord cr, String adminHandle, String password) throws HandleException, IOException: The method for creating a content record
    - public int deleteHandle(String del_handle, String adminHandle, String password) throws HandleException, IOException: The method for deleting a content record
    - public int editValue(String handle, String handle_type, int idx, Object obj, String adminHandle, String password) throws HandleException, IOException: The method for editing a content record
    - private static AdminRecord getAdminRecord(String adminHandle): The method for creating a Handle System Admin record
    - public static byte[] toByteArray (Object obj): The method for transforming an object to byte array
    - public static byte[] getBytesFromFile(File file) throws IOException: The method for transforming a file to byte array
    - public String[] getDataFromFileInWar() throws IOException: The method for getting configuration parameters from the text file inside the war file
    - public String convertStreamToString(InputStream is) throws IOException: The method for transforming an inputstream to a string
    - public Object toObject (byte[] bytes): The method for transforming a byte array to an object
    - public ContentRecord getRecord(String qhandle) throws HandleException, IOException: The method for getting the stored content record for a content name
- CidGenerator class, which includes the methods for generating the CID:
    - public String generate (String cn): The method for generating the Content ID from content name
    - public String returnHashedString(String S): The method for providing the hash of a given string
- BatchProcessor class, with all methods used in batch file processing:
    - public Map<String, List<String>> readfile(String filename): The method for reading the batch file and returning the results of the batch processing

© Copyright 2012, the Members of the COMET Consortium

    - `public static void readLines(List<String> lines) throws HandleException, IOException`: The method for reading the lines of the batch file
    - `private static void processPreviousRequest(String action, String content_name, List<ContentSource> sources) throws HandleException, IOException`: The method for processing a request
- `eu.comet.cp.web`, containing the controlling class of web application:
  - `Controller` class, containing all required methods for handling inserted parameters from web interface and perform the required handle system operations:
    - `public String login () throws HandleException, IOException`: The method for authenticating administrator in CP
    - `public String gotoCreate()`:The method used for redirecting to insert_record page
    - `public String addserver()`:The method used for adding a server in already created content source
    - `public String previewrecord()`:The method for adding all content source and servers' parameters in content record and redirecting to preview_record page
    - `public String addsources()`:The method for redirecting to a new insert_record page
    - `public String createrecord() throws HandleException, IOException`: The method for creating a content record
    - `public String gotoEdit() throws HandleException, IOException`: The method for redirecting to edit_record page for the selected content name
    - `public String logout()`:The method for logging out from CP
    - `public String delete() throws HandleException, IOException`: The method for deleting the content record for the selected content name
    - `public String deletesource()`:The method for deleting the target content source from the content record
    - `public String deleteserver()`:The method for deleting the target content server from the target content source of the content record
    - `public String addnewserver()`:The method for adding a content server in the target content source of the content record
    - `public void listener(UploadEvent event) throws Exception`: The method for uploading the batch file and processing it using Batch Processor
    - `public String gotoMain() throws HandleException, IOException`: The method for redirecting to main page
    - `public static void copy(File src, File dst) throws IOException`: The method for copying a file to another file
- `eu.comet.hdlrecord`, containing:
  - `ContentRecord`: The content record object
  - `ContentSource`: The content source object

© Copyright 2012, the Members of the COMET

o `ContentServer:` The content server object

Figure 42 presents the class diagram of Content Publisher.

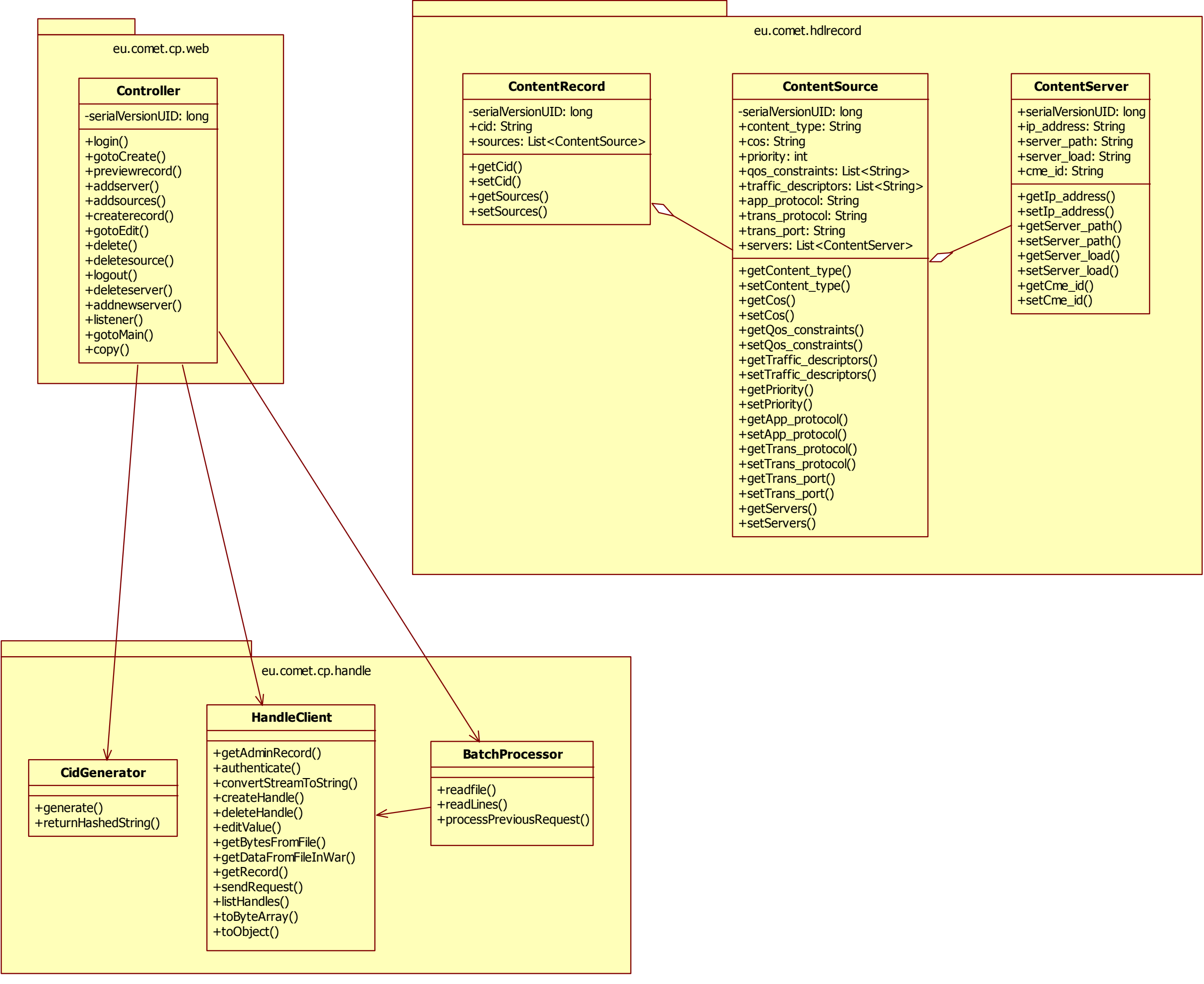


Figure 42: Content Publisher class diagram

5.3.4 Testing and Test scenarios

Current version of Content Publisher does not include any self-contained tests, since they would require an instance of CRE up and running.

5.4 Content Client

5.4.1 Description of overall functionality

Content Client is responsible for:

- Requesting a particular content from CME,
- Handling Content Record parameters received from CME,
- Building the direct request to the Content Server.

© Copyright 2012, the Members of the COMET Consortium

5.4.2 Interfaces

5.4.2.1 CC-CME Interface

CC and CME exchange messages following a DNS-like format, described in Section 11.2.1. The Content Client classes and methods which parse and handle sent/received messages are:

- **`class UDPSocket : public CommunicatingSocket : public Socket`**
  - `intUDPSocket::sendTo(const void *buffer, intbufferLen, const string &foreignAddress, unsigned short foreignPort):`

  It creates an UDP socket to send DNS-like packets to the server.
  - `intUDPSocket::recvFrom(void *buffer, intbufferLen, string &sourceAddress, unsigned short &sourcePort) :`

  It creates an UDP socket and waits for incoming DNS-like packets from the server.
- **`class ContentHandler`**
  - `string createRequestMessage(char *qname):`

  It creates a data buffer following the specification of DNS-like format in order to send the Content Request to CME.
  - `void parseResponseMessage(u_char *recvBuffer):`

  It parses the incoming datagram packet from CME in order to get correctly the parameters of the requested Content Record.

5.4.3 Design

5.4.3.1 Class Diagrams

Content Client is a C/C++ based application. Due to the fact that part of the code is developed with C which is a non object-oriented language, the class diagram cannot show the entire relations between the different entities of the application. Figure 43 presents content client's class diagram including `Socket`, `CommunicatingSocket` and `UDPSocket` classes, which contain the methods to create and handle sockets in CC and CME communication.

`ContentHandler` class contains the methods for parsing sent/received messages and building the direct URL to the content.

The diagram also contains the different data structures `st_header`, `st_query`, `st_query_str`, `st_answer` and `st_answer_str` of the message, based on the CC-CME interface specifications.

© Copyright 2012, the Members of the COMET

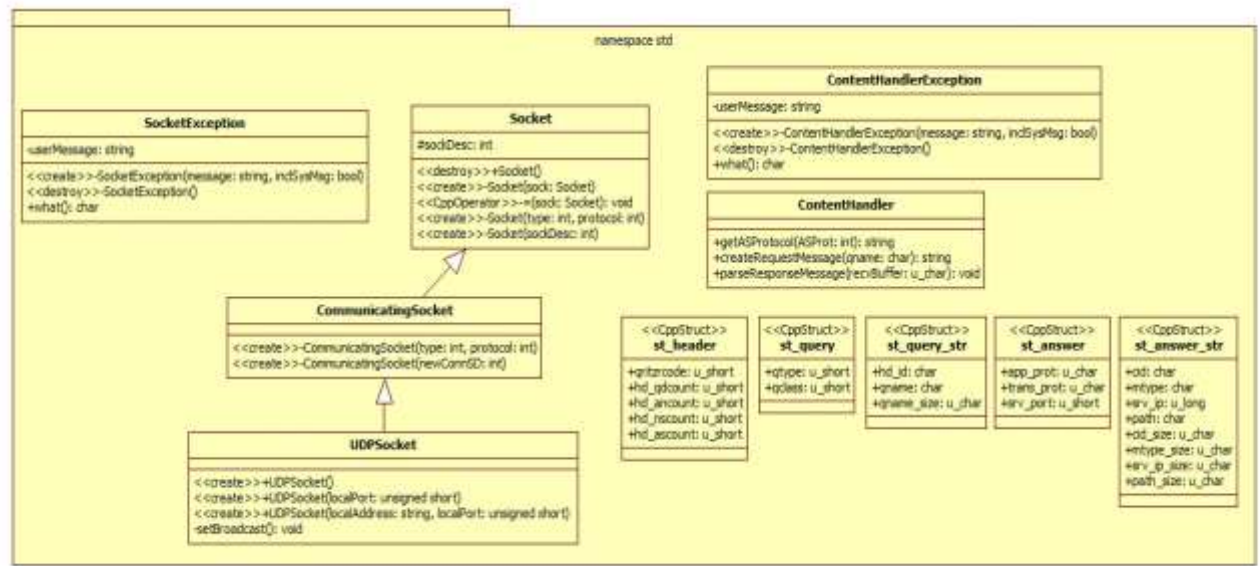


Figure 43: Content Client class diagram

5.4.3.2 Sequence Diagrams

During content request, Content Client requests a particular content from the CME, including its content name or content ID. At this point, we assume that an end-user has found the content name or the content ID of the content he wants to consume.

After this message, the CME will start the Content Resolution process and respond to Content Client with the selected content record parameters.

Upon receiving the answer from CME, the Content Client extracts Content Server's IP address, the application/session protocol, the transport protocol, the Content Server's port and path to the content, from the message, and with these parameters, builds the direct request to the Content Server. More specifically, the Content Client extracts these parameters from the answer received from the CME and forms the URL with these parameters, which is passed to the Operating System in order to launch the appropriate application (e.g. web browser) to send a request to the chosen server.

The process of the Content Request is depicted in the sequence diagram of Figure 44:

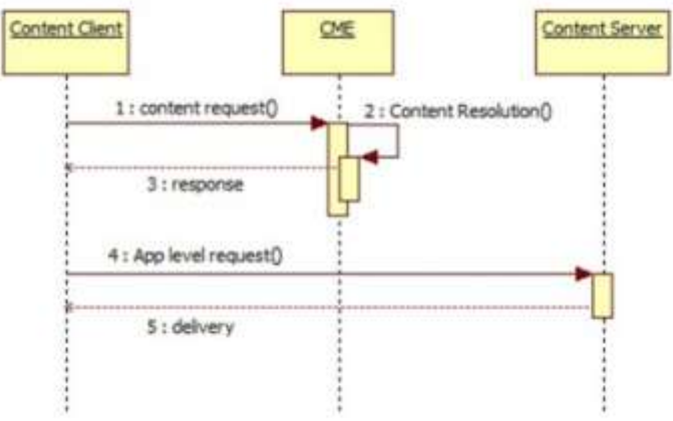


Figure 44: Content Client sequence diagram

© Copyright 2012, the Members of the COMET Consortium

5.4.4 Testing and Test scenarios

COMET repository hosts the self-contained tests for the Content Client.

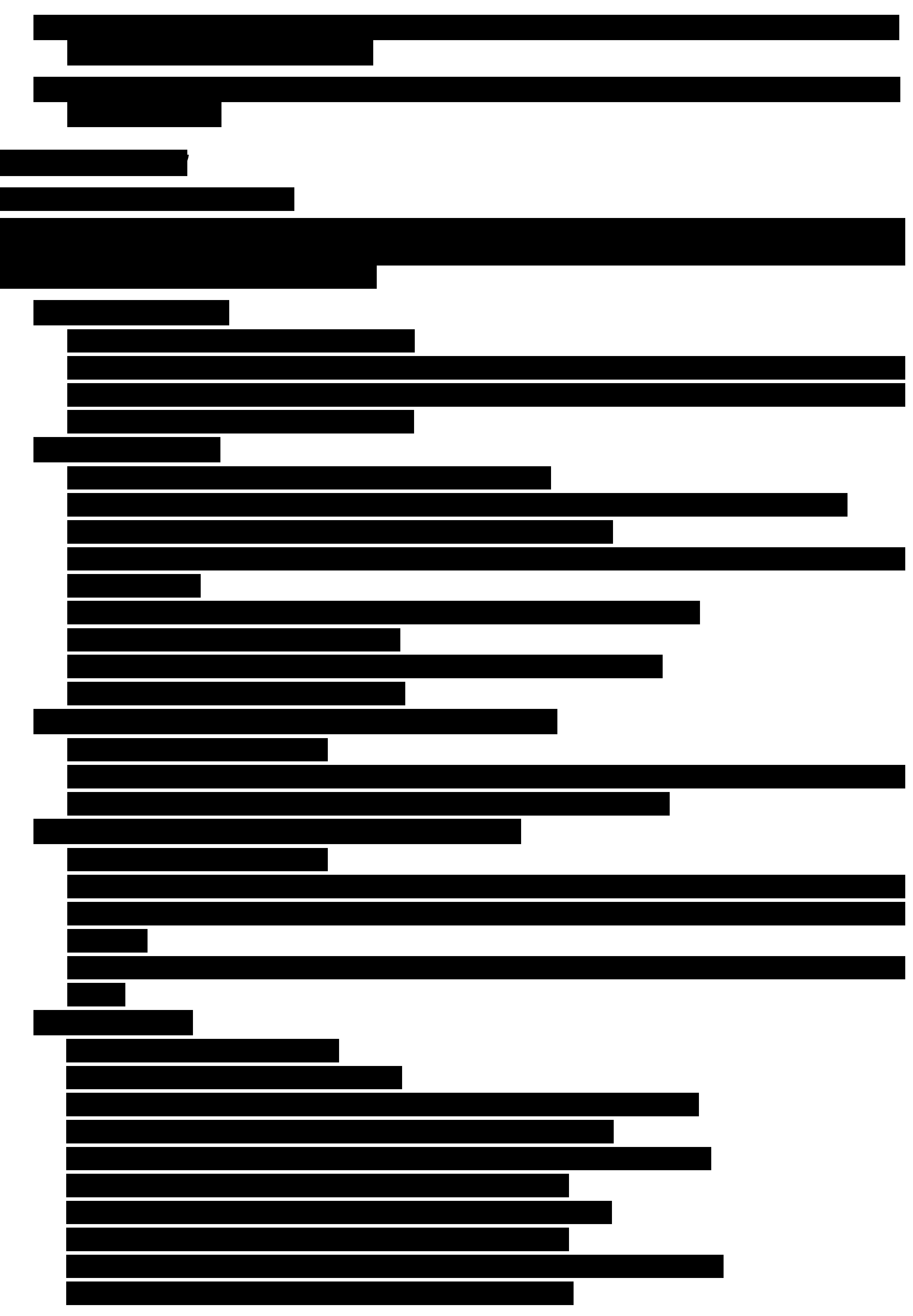
The components of the tests are the following:

- Test Client application.
- Dummy Server application.

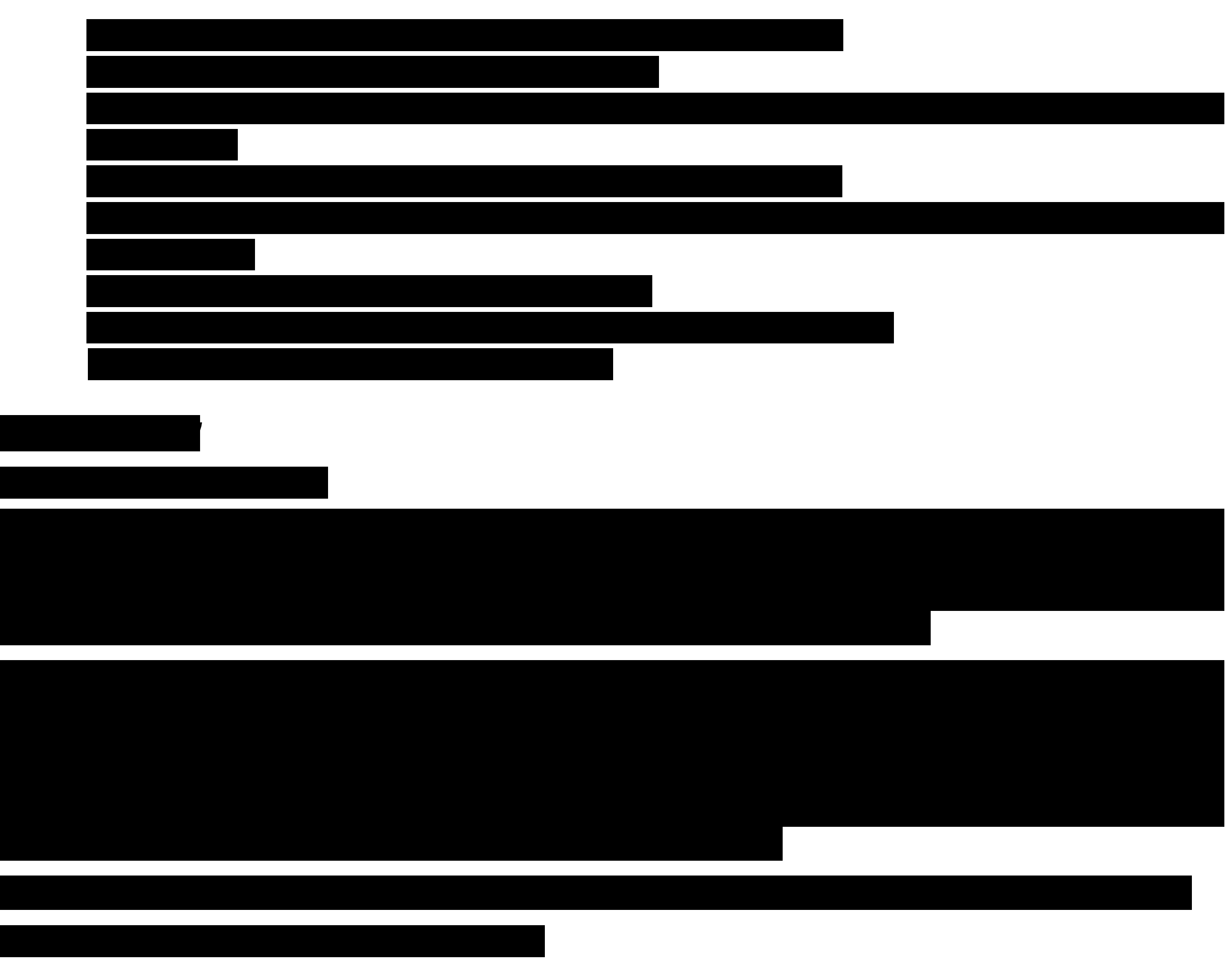
Test Client application works at the same way that Content Client application but sends a request to the Dummy Server. The Dummy server application checks the correct format of the request message, based on CC-CME interface specification, and sends a pre-defined response message. Test Client receives and parses the incoming message and builds the full URL for content consumption and compares it with a pre-defined URL.

5.5 Server Network Monitoring Entity

© Copyright 2012, the Members of the COMET

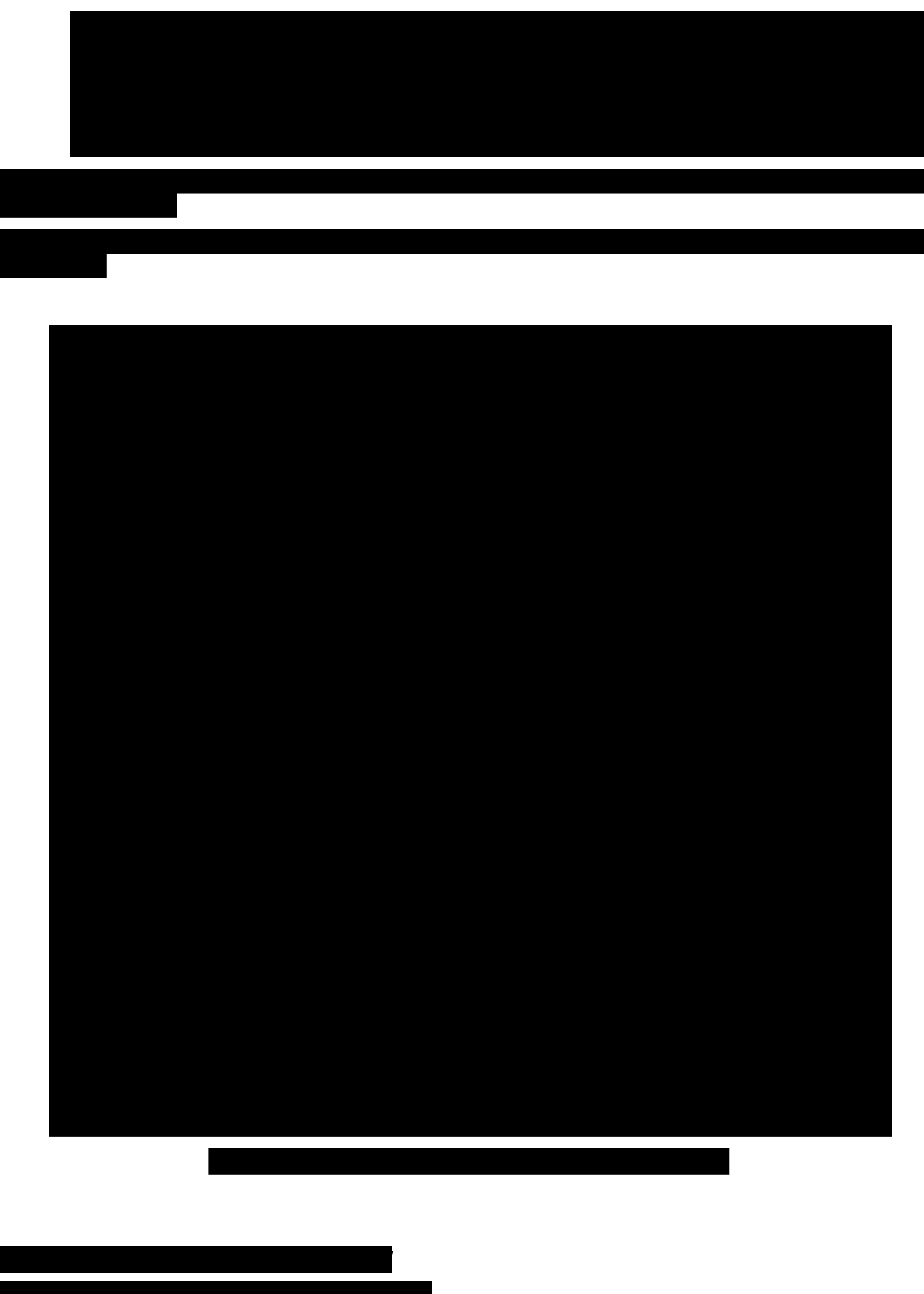


© Copyright 2012, the Members of the COMET Consortium

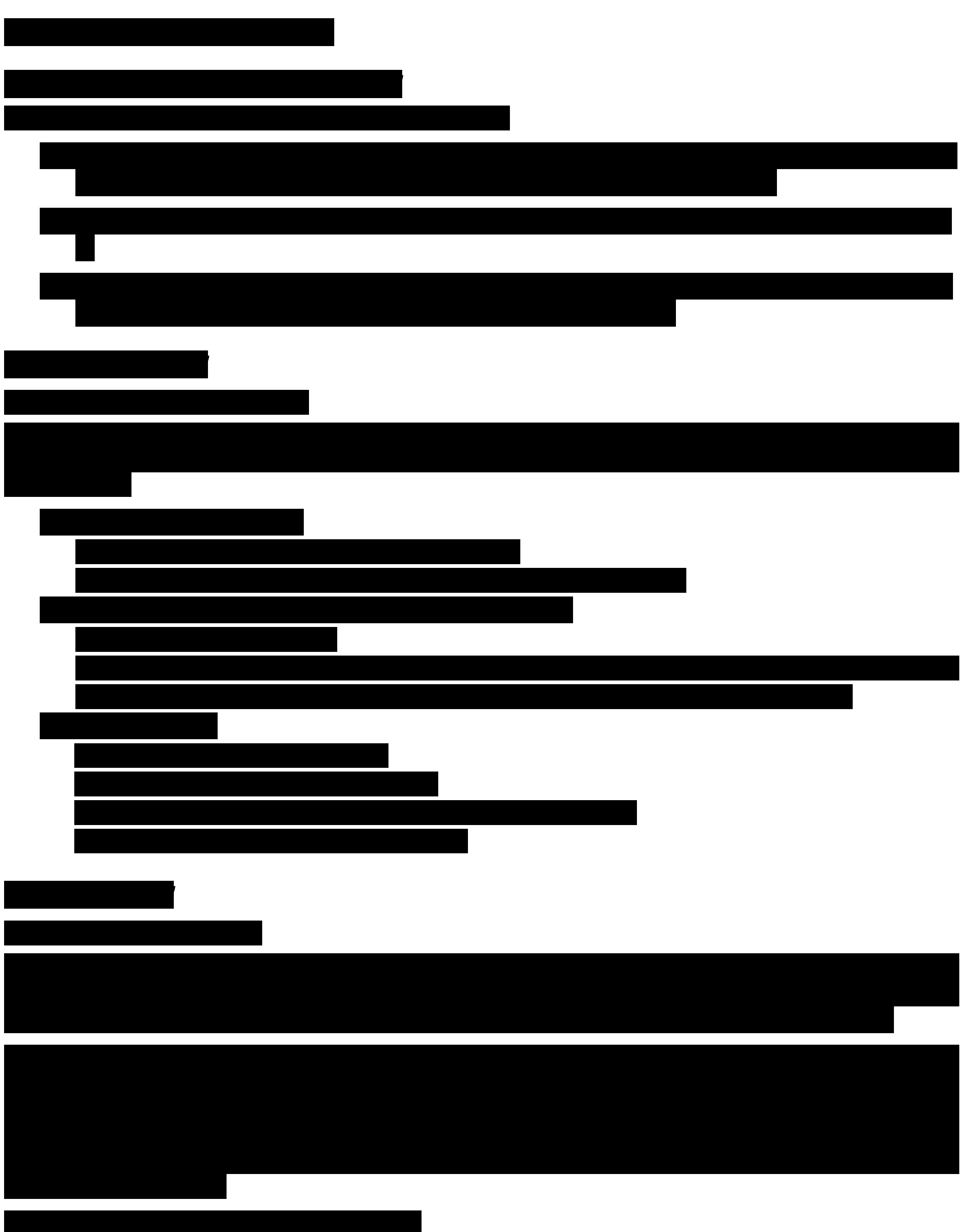


© Copyright 2012, the Members of the COMET

© Copyright 2012, the Members of the COMET Consortium

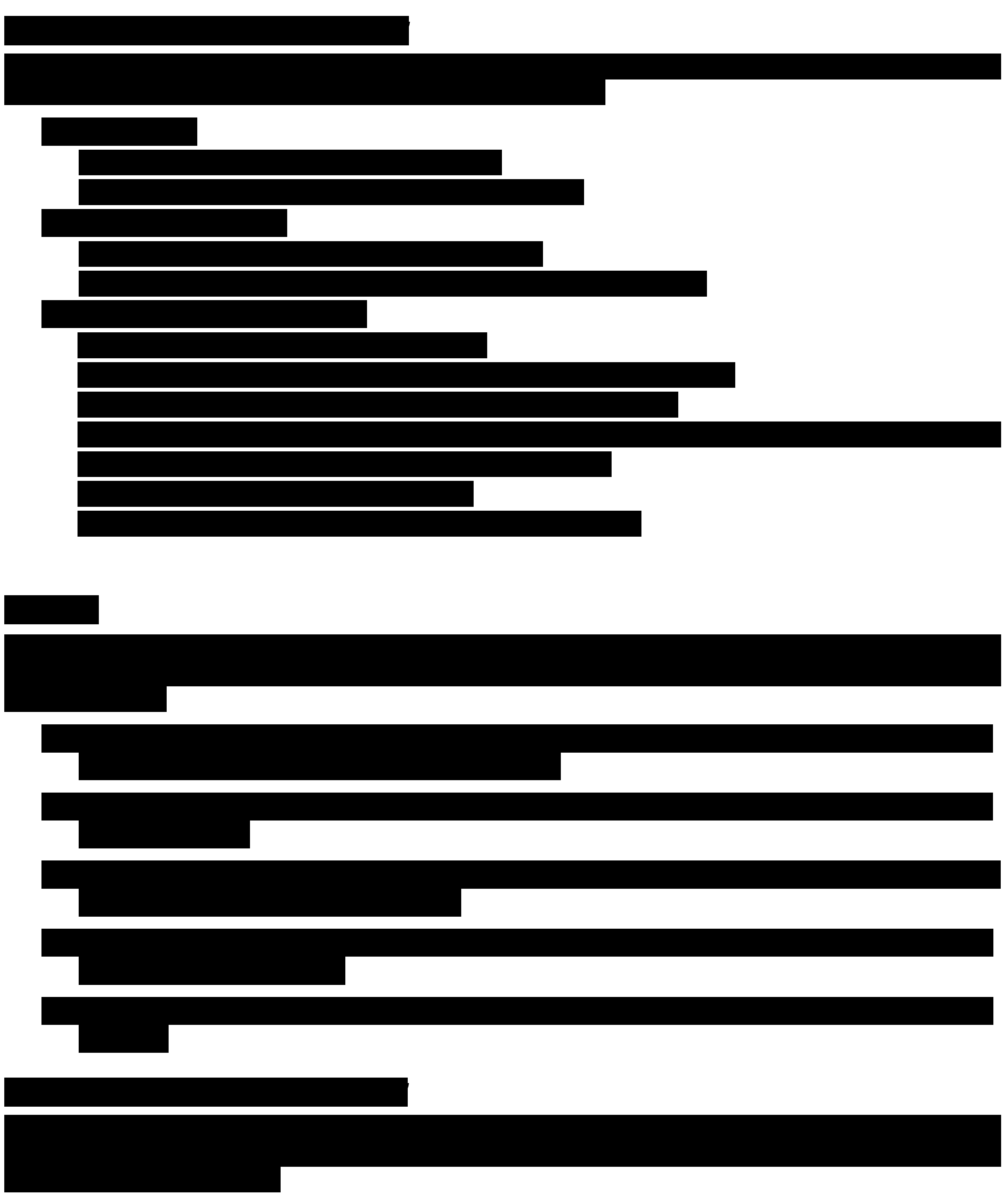


© Copyright 2012, the Members of the COMET



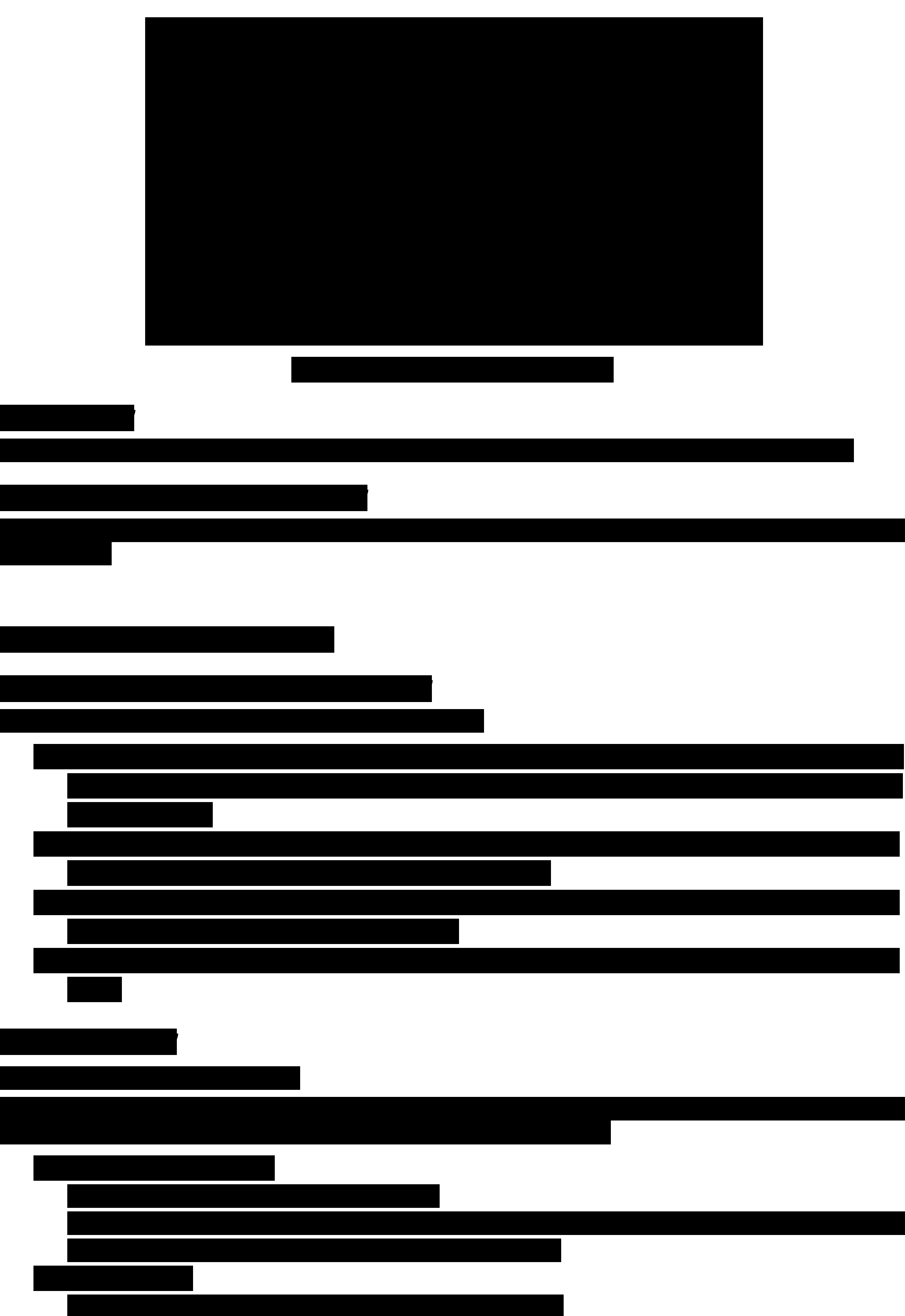
© Copyright 2012, the Members of the COMET Consortium

© Copyright 2012, the Members of the COMET



© Copyright 2012, the Members of the COMET Consortium

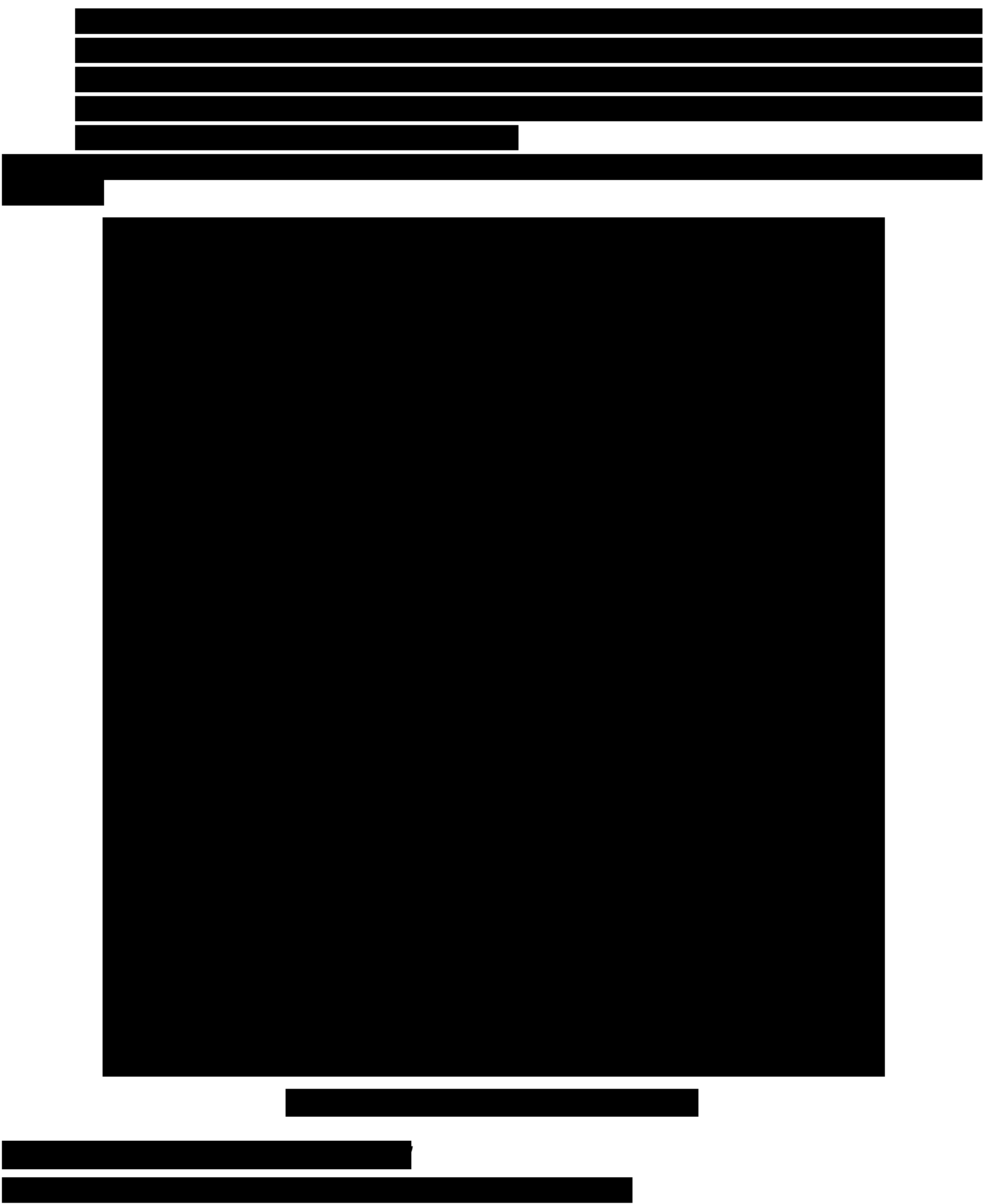
© Copyright 2012, the Members of the COMET



© Copyright 2012, the Members of the COMET Consortium

© Copyright 2012, the Members of the COMET

© Copyright 2012, the Members of the COMET Consortium



© Copyright 2012, the Members of the COMET

6 Coupled Approach Entities

6.1 Content Resolution and Mediation Entity

6.1.1 Description of overall functionality

In the coupled approach, the content resolution and mediation entity (CRME) is the main entity in coordinating the publication, resolution and delivery of content. It interfaces with the following entities:

- Local content clients (CCs)
- Routing Awareness Entity (RAE)
- Its neighbor CRMEs
- Local content-aware forwarding entities (CAFEs)
- Local content publishers (indicating the corresponding content servers (CSs))

CRME's main functionality includes:

- Handling content publication messages from local content publishers (i.e., the `Register` message)
- Handling content registration messages from neighbouring CRMEs (i.e., the `Publish` message)
- Handling content requests from local CCs and neighbouring CRMEs (i.e., the `Consume` message)
- Configuring the related CAFEs within the domain for the delivery of requested content
- Managing and maintaining the content management repository
- Receiving NLRI from RAE

6.1.2 Interfaces

Communication between CRME and other entities is carried out over four main types of interfaces: an inter-CRME interface, a CRME-CC interface, a CRME-CS interface and a CRME-CAFE interface. In the following subsections, we provide a detailed description of these interfaces and the communication methods used across them.

6.1.2.1 Inter-CRME Interface

The inter-CRME interface is that over which control-plane messages are exchanged between the neighbouring CRMEs, namely, the `Publish`, `Consume`, and `Error` messages, the descriptions of which are given in Table 6-1. Figure 53 illustrates the communication between two neighbouring CRMEs.

Table 6-1: Inter-CRME interface messages

Message	Information Passed	Description
`Publish`	- `contentID`	Sent by a CRME to its provider CRME following the *provider router forwarding route* when a content is being published.
`Consume`	- `contentID`	Sent by a CRME to its provider CRME(s) based on the provider route forwarding route to locate the requested content.

© Copyright 2012, the Members of the COMET Consortium

Error	- contentID	Initiated by a tier-1 CRME when the requested content cannot be found.

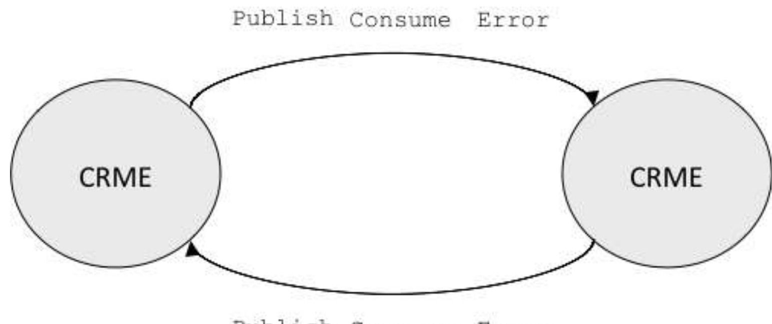


Figure 53: Inter-CRME communication.

6.1.2.2 CRME-CC Interface

This interface enables the communication between content consumers with their local (or immediate) CRME. There are two types of message being exchanged between these two entities; namely `Consume` and `Error` messages, the descriptions of which are given in Table 6-2. Figure 54 illustrates the communication between a CRME and a CC.

Table 6-2: CRME-CC interface messages

Message	Information Passed	Description
Consume	- contentID	Sent by a content consumer to its local CRME for requesting a content.
Error	- contentID	Forwarded by the local CRME to the original content consumer indicating that the requested content cannot be found / unavailable.

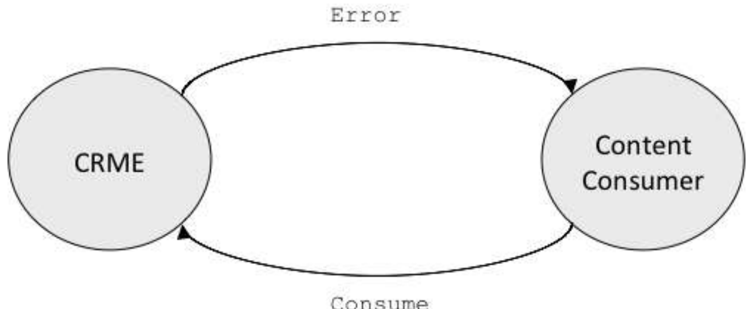


Figure 54: CRME-CC communication.

6.1.2.3 CRME-CS Interface

This interface enables the communication between content publishers (along with its content servers) with their local (or immediate) CRME. There are two types of message being exchanged between these two entities; namely `Register` and `Consume` messages, the descriptions of which are given inTable 6-3. Figure 55 illustrates the communication between a CRME and a CP/CS.

© Copyright 2012, the Members of the COMET

Table 6-3: CRME-CS interface messages

Message	Information Passed	Description
Register	- contentID	Sent by a content publisher to its local CRME to publish a content.
Consume	- contentID	Forwarded by the local CRME to the content server hosting the requested content (i.e., the requested content has been found).

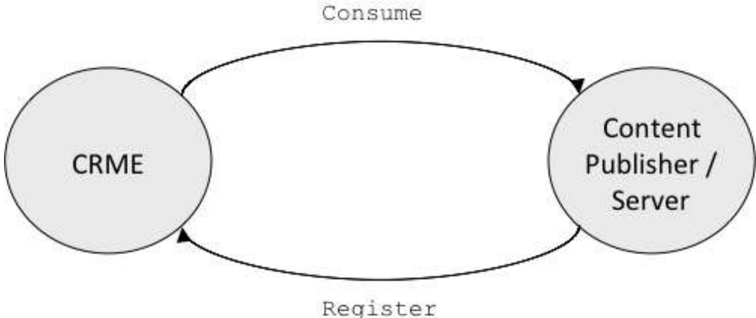


Figure 55: CRME-CS communication.

6.1.2.4 CRME-CAFE Interface

The CRME-CAFE 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 D4.3 [5] where the relevant content delivery operations handled by CAFE are described.

6.1.3 Design

Being the main entity in the coupled approach, the CRME is involved in every aspect of the content lifecycle beginning from content publication to resolution and finally the delivery of the content. In this section, the design of the content publication and resolution processes handled by CRME is described.

Figure 56 shows the class diagram of the proof-of-concept implementation of the coupled approach, which consists of four main classes corresponding directly to the four main COMET entities used for the coupled approach. These classes are the `Crme`, `ContentServer`, `ContentConsumer`, and `Cafe`.

© Copyright 2012, the Members of the COMET Consortium

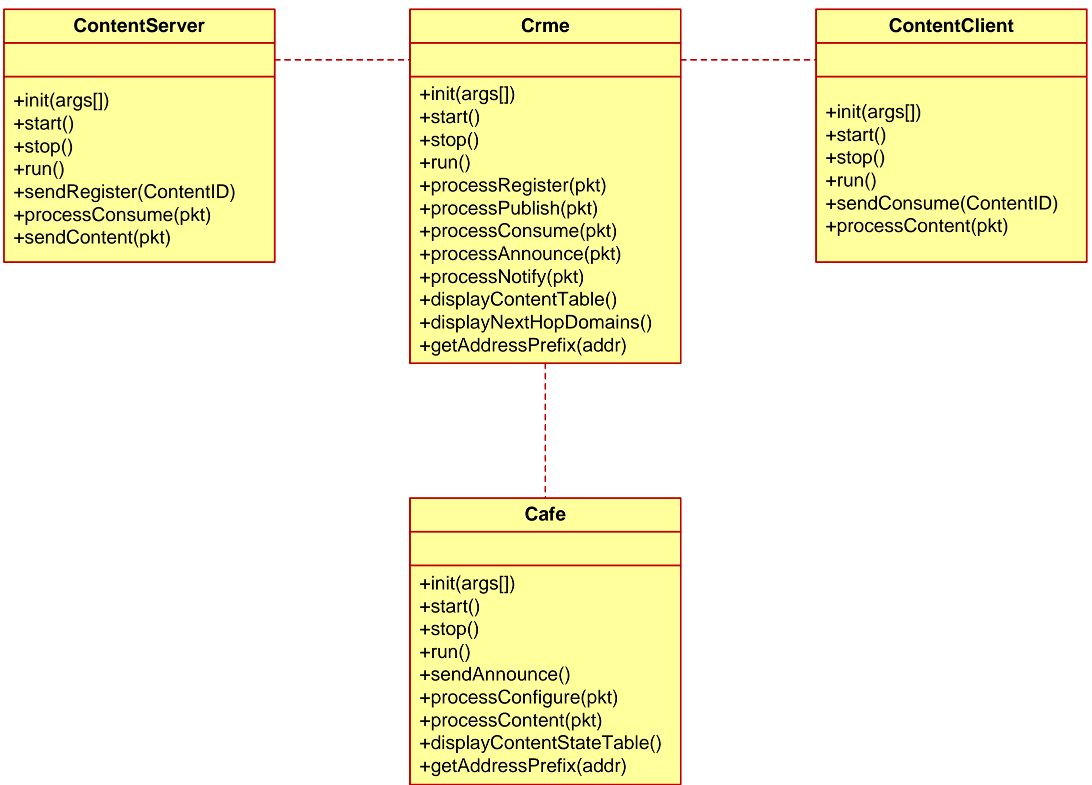


Figure 56: UML class diagram of the coupled approach proof-of-concept emulator implementation

The `Crme` class contains three main functions related to content mediation:

- **`processRegister(pkt)`** – processes `Register` messages by creating the relevant entries in the `ContentTable` and forwarding a `Publish` message to the next-hop CRME.
- **`processPublish(pkt)`** – processes `Publish` messages by creating the relevant entries in the `ContentTable` and forwarding the message to the next-hop CRME.
- **`processConsume(pkt)`** – processes `Consume` messages it receives by looking up in its `ContentTable` the next-hop CRME to which to forward the `Consume` message. If an entry for the content requested does not exist in the `ContentTable`, the `Consume` message is forwarded to a parent CRME according to the resolution rules specified in D3.2 [3].

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

- **`displayContentTable()`** – prints to screen all entries in its `ContentTable`.
- **`displayNextHopDomains()`** – prints to screen information about its neighbouring CRMEs.

Figure 57 shows the UML state diagram illustrating the design of the CRME class, in particular the parts of that class relating to content publication and resolution. The parts related to content delivery and route optimisation are described in D4.3 [5].

Once initialised, the CRME is put to idle state and wait for incoming messages.

For the content publication process, two messages are involved; namely the `Register` and `Publish` message. If the CRME receives a `Register` message, it will first create a new entry for this content (indicated by the message) in its `contentTable`. A `Publish` message will then be created with the same content ID and forwarded to its provider CRME. The CRME will follow

© Copyright 2012, the Members of the COMET

similar operations if a `Publish` message is received (only that instead of creating a new `Publish` message, the received message is forwarded without modifications).

A content resolution process is initiated once a content consumer sends a `Consume` message. When a CRME receives a `Consume` message, it first checks its `contentTable` if there is a matching entry for the requested content. If a matching entry is found, then after sending the appropriate configuring information through a `Configure` message to the involved CAFEs, it will forward the Consume message to the next CRME towards the content server as indicated in its `contentTable` (i.e., the corresponding `nextHopAddr`). The handling of the `Configure` messages by CAFE is detailed in D4.3 [5]. If, however, the content is not found in this CRME (i.e., no matching entry is found), then the `Consume` message will be forwarded based on the *provider route forwarding rule*. Essentially, the `Consume` message will be forwarded to the provider CRME. The relevant states for this content request will be installed at the relevant CAFEs via `Configure` messages. In the case where the content is still not found at the tier-1 level, then the tier-1 CRME will create an `Error` message to be forwarded back to the content consumer.

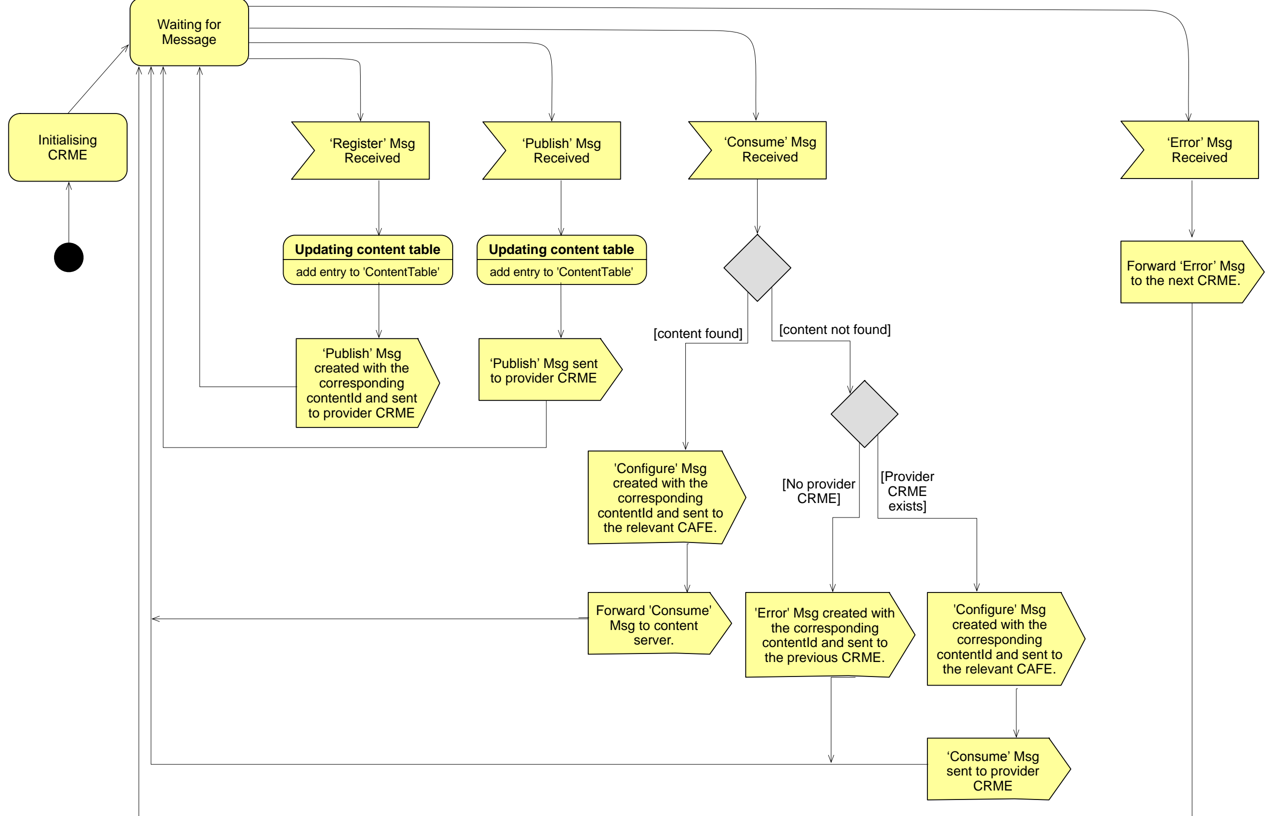


Figure 57: UML state diagram for content publication and resolution aspects of the CRME.

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

```
+time ON_ROUTER router_addressusr.curling.Crme myport nneighbourdomain
<neighbourCRMEaddr, rel, neighbourCRMEport> nCAFE <CAFEaddr, CAFEport>
nPrefixes <Prefix  NextHopCrmeNextHopCafe>
```

where:

`router_address` = the address or name of the current virtual router

`myport`= the port used by this CRME

© Copyright 2012, the Members of the COMET Consortium

nneighbourdomain = number of domains directly linked to this domain

<neighbourCRMEaddr, rel, neighhourCRMEport> = nneighbourdomain x tuple of the address of the neighbour CRME, the inter-domain relationship (either provider, customer or peer) and the neighbour CRME port.

nCAFE= number of CAFEs within the domain

<CAFEaddr, CAFEport>= the duple of the local CAFE address and port

nPrefixes = number of entries in the CRME's Loc-RIB (BGP) table

<Prefix,NextHopCrme,NextHopCafe> = nPrefixes x tuple of the address prefix of a domain, the address of the next hop CRME and next-hop CAFE towards the destination domain, respectively.

In the implementation of the CRME entity, there exists two tables:

(1) Content management repository (ContentTable)

ContentID	<NextHopCRMEaddr, NextHopCRMEport>

(2) BGP (Loc-RIB) routing table (RoutingTable)

DomainPrefix	<NextHopCRMEaddr, NextHopCAFEaddr>

(3) CRME-level forwarding table (NextHopDomains)

NextHopDomain	<LocalCAFEaddr, NextHopCAFEaddr, NextHopCAFEport>

where

ContentID = the string identifier of the content

NextHopCRMEaddr = the address of the next-hop CRME

NextHopCRMEport = the port number of the next-hop CRME

DomainPrefix = the address prefix of the destination domain

LocalCAFEaddr= the address of the local egress CAFE involved in the delivery of the content

NextHopCAFEaddr= the address of the CAFE directly connected to the local CAFE via the inter-domain link for this specific content delivery

NextHopCAFEport= the corresponding port used by the next CAFE

6.1.4 Testing and test scenarios

6.1.4.1 Basic Content Publication Operation

To test the content publication operation, an inter-domain topology shown in Figure 58 and Figure 59 are setup. The test simply involves content server attached to 5.3 registering different content (i.e., issuing Register messages for different content).

This enables the validation of the following:

© Copyright 2012, the Members of the COMET

- The correct forwarding of `Register` messages from CP to its local CRME (via intermediate CAFE(s))
    - The `Register` message should traverse from CP to node 5.1 via node 5.3 for test topology 1 and node 7.1 via node 7.3 for test topology 2.
- The correct handling of `Register` messages at the local CRME (i.e., node 5.3 in test topology 1 and node 7.3 in test topology 2)
    - The local CRME should be able to create a new entry in its `contentTable` for each new content being registered.
- The correct creation of the `Publish` messages from local CRME
    - The content ID must be the same as the corresponding `Register` message.
- The correct forwarding of the `Publish` messages
    - Test topology 1: The `Publish` messages should be forwarded following the *provider route forwarding rule* (i.e., node 5.1 → node 5.2 → node 3.3 → node 3.1 → node 3.2 →node1.3 → node 1.1)
    - Test topology 2: The `Publish` messages should be forwarded following the *provider route forwarding rule* (i.e., node 7.1→ node 7.2 → node 4.3 → node 4.1 → node 4.2 →node 1.4→ node 1.1)
- The correct handling of `Publish` messages at each CRME
    - Each CRME receiving the `Publish` messages should be able to create a new entry in its `contentTable` for each new content being published.

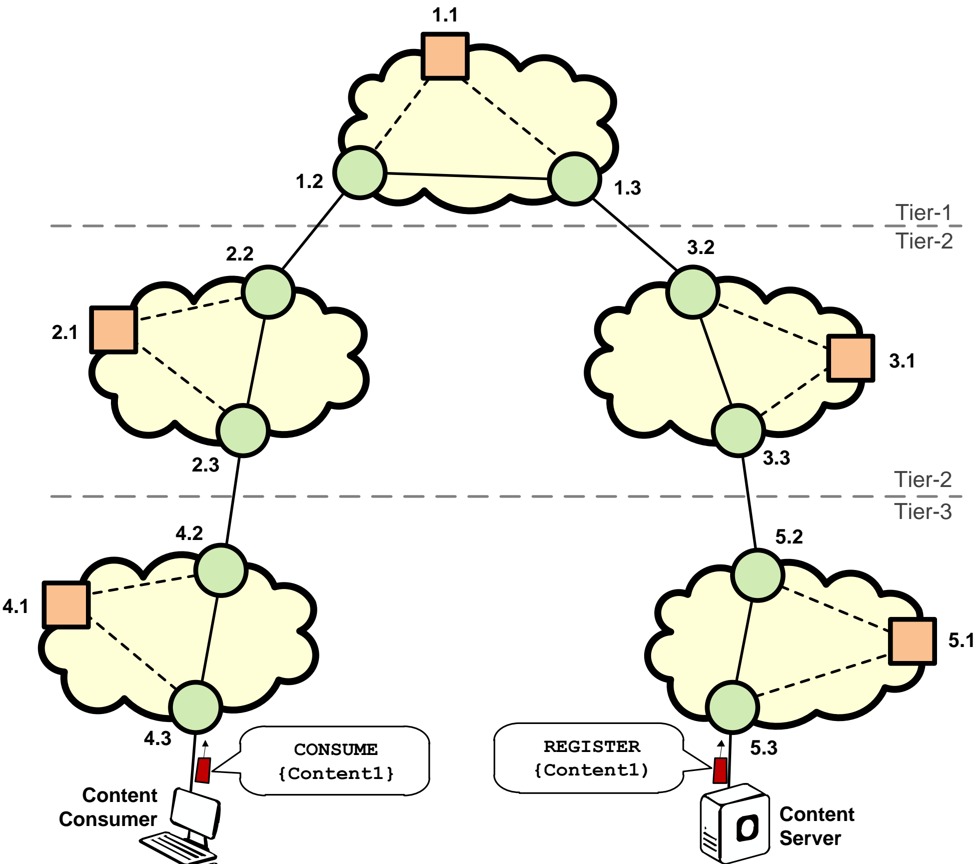


Figure 58: Test topology 1.

© Copyright 2012, the Members of the COMET Consortium

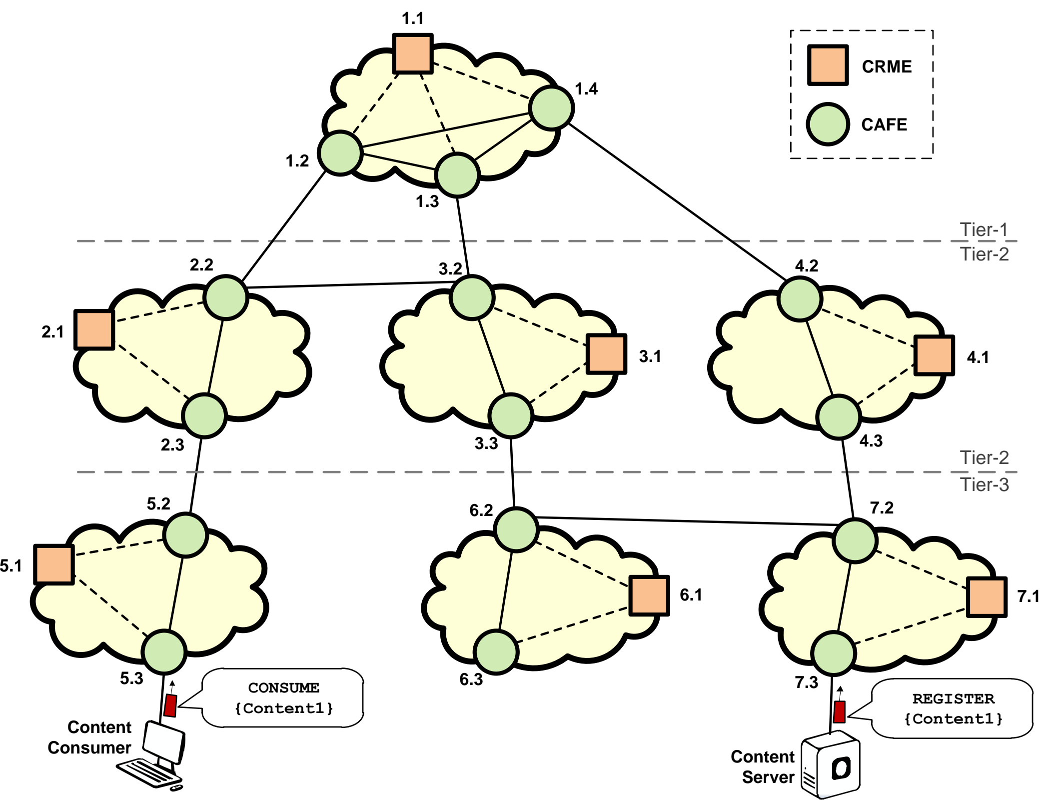


Figure 59: Test topology 2.

6.1.4.2 Basic Content Resolution Operation

To test the content resolution operation, an inter-domain topology shown in Figure 59 is setup. It is initialized with a single content (i.e., `Content1`) being published as shown in the figure. The test involves attaching CC at different points of the network and requesting for the content.

This enables the validation of the following:

- The correct forwarding of `Consume` messages from CC to the CP via intermediate CRME(s)
  - The `Consume` message should follow the *provider route forwarding rule* while the requested content is still not found.
  - The `Consume` message should follow the entry in the CRMEs' `contentTable` once the requested content is found.

Note that the relevant validation of the configuration of forwarding states at CAFEs is detailed in D4.3 [5].

6.2 Content Publisher

6.2.1 Description of overall functionality

The content publisher is implemented to enable content publication in the PoC. In our PoC, content publisher and content server are assumed to be co-located. As such, this implementation includes both the facility to publish a content (describe in this document) and to serve content requests (describe in D4.3 on content delivery under the coupled approach).

CP's main functionality includes:

© Copyright 2012, the Members of the COMET

- Creating and transmitting content publication messages (i.e., the `Register` message)
- Handling content requests from local CRME (i.e., the `Consume` message)
- Transmit requested content after receiving a valid `Consume` message.

6.2.2 Interface

In the coupled approach, a content publisher only has a specific interface with its local CRME. It may also have link to one or more CAFE(s) in its domain. However, communication with CAFEs is limited to content transference. The description of the CC-CRME interface is given in section 6.1.2.3.

6.2.3 Design

The `ContentServer` class contains two main functions related to content publication and consumption:

- **`sendRegister(ContentID)`** – sends a `Register` message to its local CRME to register content with the passed `ContentID`.
- **`processConsume(pkt)`** – processes `Consume` messages it receives and reads the `ContentID` being requested.

Figure 60 shows the UML state diagram for CP. Immediately after initialisation, a CP will publish its content by creating the necessary `Register` message(s). The local CRME is then responsible for forwarding and publishing the content.

Then, it comes to an idle state, waiting to serve any content requests (i.e., `Consume` message requesting its content). When it receives a `Consume` message, it will then start transmitting the requested content to its immediate CAFE. Note that it does not know the route to or the identity of the CC requesting its content. The entire delivery of the content relies on the correct coordination of CRMEs and CAFEs via the content publication and resolution processes.

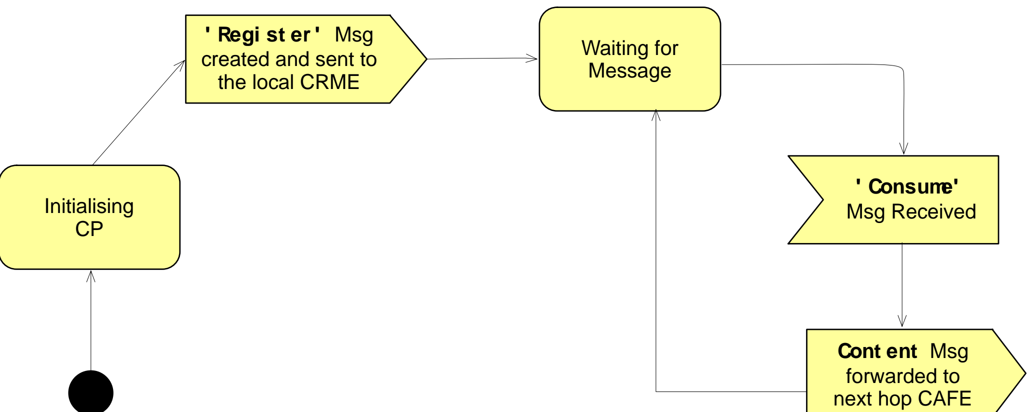


Figure 60: UML state diagram for a CP.

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

`+time ON_ROUTER router_addressusr.curling.ContentServer targetCRMEaddr myport targetCRMEport contentname` where:

`router_address`= the address or name of the current virtual router

`targetCRMEaddr`= the router name of the local CRME

© Copyright 2012, the Members of the COMET Consortium

`myport` = the port used by the content publisher

`targetCRMEport`= the port used by the local CRME

`contentname` = the name of the content being registered

6.2.4 Testing and test scenarios

6.2.4.1 Registering content

To test the content registration capability of CP, we setup multiple CPs connected to a CRME via an intermediate CAFE. These CPs then send `Register` messages to the CRME.

This enables the validation of the following:

- The correct creation of `Register` messages.

6.3 Content Client

6.3.1 Description of overall functionality

The content client is implemented to enable content resolution in the PoC.

CC's main functionality includes:

- Creating and transmitting content consumption messages (i.e., the `Consume` message)
- Receive and storing corresponding content after issuing a valid `Consume` message.

6.3.2 Interface

In the coupled approach, a content client only has a specific interface with its local CRME. It may also have link to one or more CAFE(s) in its domain. However, communication with CAFEs is limited to content transference. The description of the CS-CRME interface is given in section 6.1.2.2.

6.3.3 Design

The `ContentClient` class contains one main functionrelated to content mediation:

- **`sendConsume(ContentID)`** – sends a `Consume` message to the its local CRME requesting content with the passed `ContentID`.

Figure 61 shows the UML state diagram for CC. Immediately after initialisation, a CC will request a content by creating the necessary `Consume` message. The local CRME is then responsible for forwarding this `Consume` message to find the content.

Then, it comes to an idle state, waiting for the requested content. Once the requested content is received, it will save it into the disk.

© Copyright 2012, the Members of the COMET

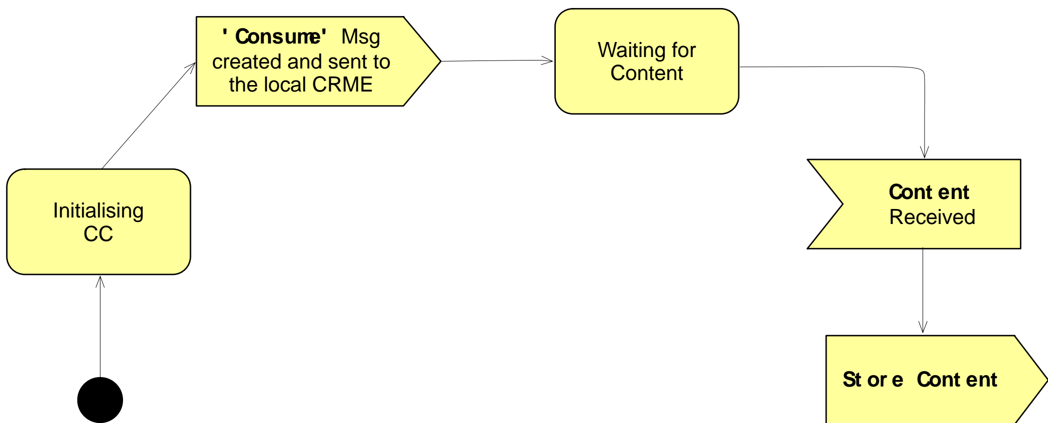


Figure 61: UML state diagram for a CC.

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

`+time ON_ROUTER router_nameusr.curling.ContentClient targetCRMEaddr myport targetCRMEport contentname` where:

`router_address` = the address or name of the current virtual router

`targetCRMEaddr`= the router name of the local CRME

`myport` = the port used by the content client

`targetCRMEport`= the port used by the local CRME

`contentname` = the name of the content being requested

6.3.4 Testing and test scenarios

6.3.4.1 Requesting content

To test the content request capability of CC, we setup multiple CCs connected to multiple CPs via an intermediate CRME (with CAFE co-located at the same node). The test is initialised with the CPs registering some content at the CRME. Then, the different CCs will start requesting for the registered content.

This enables the validation of the following:

- The correct creation and transmission of `Consume`messages.

© Copyright 2012, the Members of the COMET Consortium

7 Conclusions

This deliverable presented the final set of implementation details for the content mediation plane (CMP) in the overall COMET architecture and the respective implementation of both specified approaches. It fully utilizes the COMET architecture presented in D2.2 [1], and specified CMP mechanisms included in D3.2 [3].

Hence, this deliverable presented the set of detailed interfaces offered and used by all COMET architecture entities and components of CMP for the final releases of both approaches, as well as provided the final documentation of the implemented components and functionalities, offering the possibility to extend any COMET component.

Most high-level CMP entities were implemented in Java because it is a high-level and platform-independent OO language, while for the edge (CC) and network (lower-level) elements (RAE and CAFE) of COMET system, C++ was selected, due to its higher performance. A justification for all implementation choices was provided. These choices were made considering the higher aim of rapid implementation cycles, rapid integration cycles and development of an efficient prototype for validating and testing all specified mechanisms of the COMET system. Although certain improvements and alterations would be required in order to meet production requirements, all CMP entities could be easily deployed in existing systems, without disrupting their operation.

With regard to the interfaces, proprietary protocols on top of UDP were implemented for the edges of the COMET system. The established Handle protocol was preferred for the CRE, due to its content-oriented nature. Finally, Google's protobuf was used for all other interfaces between COMET entities, aiming to build a robust, homogeneous and efficient content mediation system. In addition, all COMET entities were designed and developed to be deployed in both IPv4 and IPv6 environments, but not in mixed ones, leading to an easily-deployable system to existing and future network environments.

The final documentation of CFP components and interfaces is described in D4.3 [5]. Details about the integration technologies and procedures used to develop, integrate and test the COMET software will be included in forthcoming deliverable D5.1. The aim of D5.1 will be to document the integration procedure applied and the conducted system tests, resulting in a commercially integrated and tested software, following industry standards.

© Copyright 2012, the Members of the COMET

8 References

[1] Comet Deliverable D2.2 High level Architecture of the COMET System

[2] Comet Deliverable D3.1 Interim Specification of Mechanisms, Protocols and Algorithms for the Content Mediation System

[3] Comet Deliverable D3.2 Final Specification of Mechanisms, Protocols and Algorithms for the Content Mediation System

[4] Comet Deliverable D4.2 Final Specification of Mechanisms, Protocols and Algorithms for Enhanced Network Platforms

[5] Comet Deliverable D4.3 Prototype Implementation and System Integration Interfaces for Enhanced Network Platforms

[6] http://www.ietf.org/rfc/rfc3650.txt

[7] http://www.ietf.org/rfc/rfc3651.txt

[8] Jboss Netty NIO client server framework: http://www.jboss.org/netty

[9] Jetty web server: http://jetty.codehaus.org/jetty/

[10] MySQL Database: http://www.mysql.com/

[11] Handle System software: http://www.handle.net/

[12] Protobuf buffers: http://code.google.com/p/protobuf/

[13] JUnit testing framework: http://www.junit.org/

[14] Open source Java persistence framework project: http://www.hibernate.org/

[15] Apache Tomcat web server: http://tomcat.apache.org/

[16] VLC open source cross-platform multimedia player: http://www.videolan.org/vlc/

[17] µTorrent lightweight BitTorrent client: http://www.utorrent.com

[18] Apache MINA network application framework: http://mina.apache.org/

[19] JBoss Application Server: http://www.jboss.org/jbossas

[20] Glassfish Application Server: http://glassfish.java.net/

[21] Hibernate ORM framework: http://www.hibernate.org/

[22] JBoss Richfaces: http://www.jboss.org/richfaces

[23] "Connecting IPv6 Routing Domains Over the IPv4 Internet", Carpenter et al., The Internet Protocol Journal, Volume 3, Number 1, March 2000.

© Copyright 2012, the Members of the COMET Consortium

9 Abbreviations

API	Application Programming Interface
BR	Bit Rate
BGP	Border Gateway Protocol
CAFE	Content-Aware Forwarding Entity
CC	Content Client
CFP	Content Forwarding Plane
CME	Content Mediation Entity
CMP	Content Mediation Plane
CoS	Class of Service
CP	Content Publisher
CRE	Content Resolution Entity
CRME	Content Resolution and Mediation Entity
CS	Content Server
DAO	Data Access Object
DB	Database
DNS	Domain Name System
DTO	Data Transfer Object
HS	Handle System
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IPLR	IP Loss Ratio
IPTD	IP Transfer Delay
JVM	Java Virtual Machine
NLRI	Network Level Routing Information
OS	Operating System
PoC	Proof of Concept
RAE	Routing Awareness Entity
SA	Server Awareness
SMA	Server Monitoring Agent
SNME	Server and Network Monitoring Entity
STREP	Specific Targeted Research Project
TCP	Transmission Control Protocol
TTL	Time to live
UDP	User Datagram Protocol
UML	Unified Modelling Language

© Copyright 2012, the Members of the COMET

10 Acknowledgements

This deliverable was made possible due to the large and open help of the WP3 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.

© Copyright 2012, the Members of the COMET Consortium

11 Appendix

11.1 Application and Transport Protocol codification

11.1.1 Application Protocol

Hex code	Protocol	Hex code	Protocol	Hex code	Protocol
0x01	http://	0x0C	telnet://	0x17	tcpobex://
0x02	https://	0x0D	imap:	0x18	irdaobex://
0x03	tel:	0x0E	rtsp://	0x19	file://
0x04	mailto:	0x0F	urn:	0x1A	urn:epc:id:
0x05	ftp://	0x10	pop:	0x1B	urn:epc:tag:
0x06	ftps://	0x11	sip:	0x1C	urn:epc:pat:
0x07	sftp://	0x12	sips:	0x1D	urn:epc:raw:
0x08	smb://	0x13	tftp:	0x1E	urn:epc:
0x09	nfs://	0x14	btspp://	0x1F	urn:nfc:
0x0A	dav://	0x15	btl2cap://		
0x0B	news:	0x16	btgoep://		

11.1.2 Transport Protocol

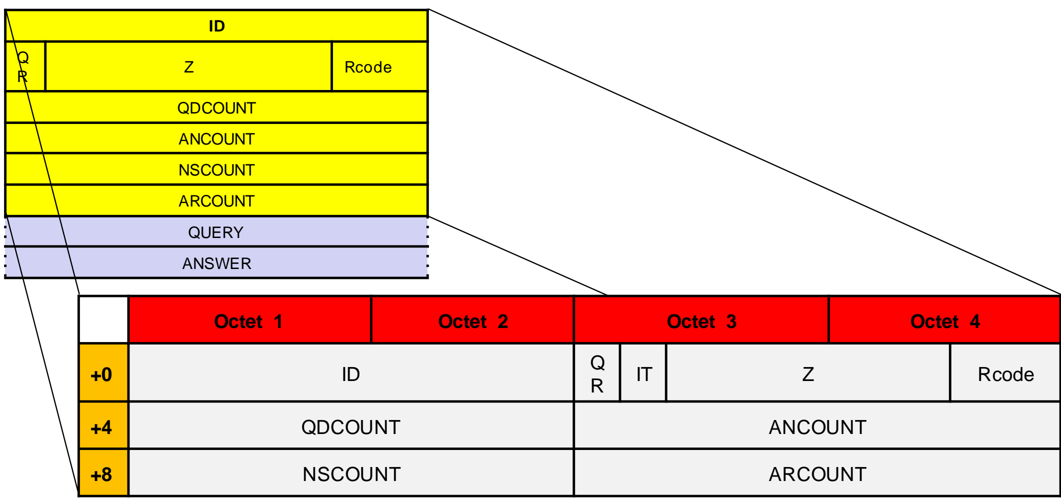
Hex code	Protocol	Hex code	Protocol
0x00	Hopopt	0x08	EGP
0x01	ICMP	0x09	IGRP
0x02	IGMP	0x11	UDP
0x03	GGP	0x29	IPv6 over IPv4
0x04	IP in IP encapsulation	0x2E	RSVP
0x05	ST	0x2F	GRE
0x06	TCP	0x59	OSPF
0x07	UCL, CBT	…	…

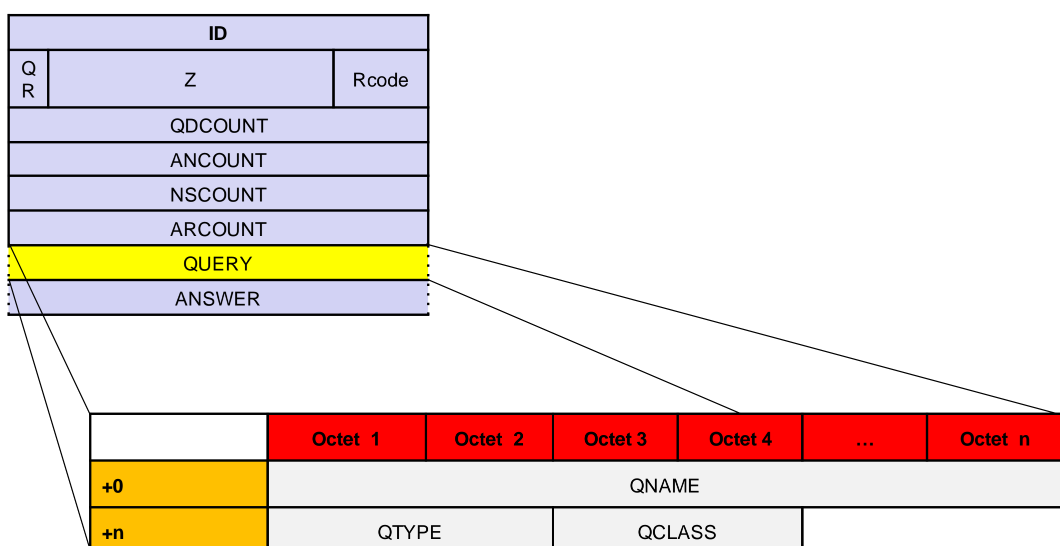
© Copyright 2012, the Members of the COMET

11.2 Interfaces specification

11.2.1 CME-CC

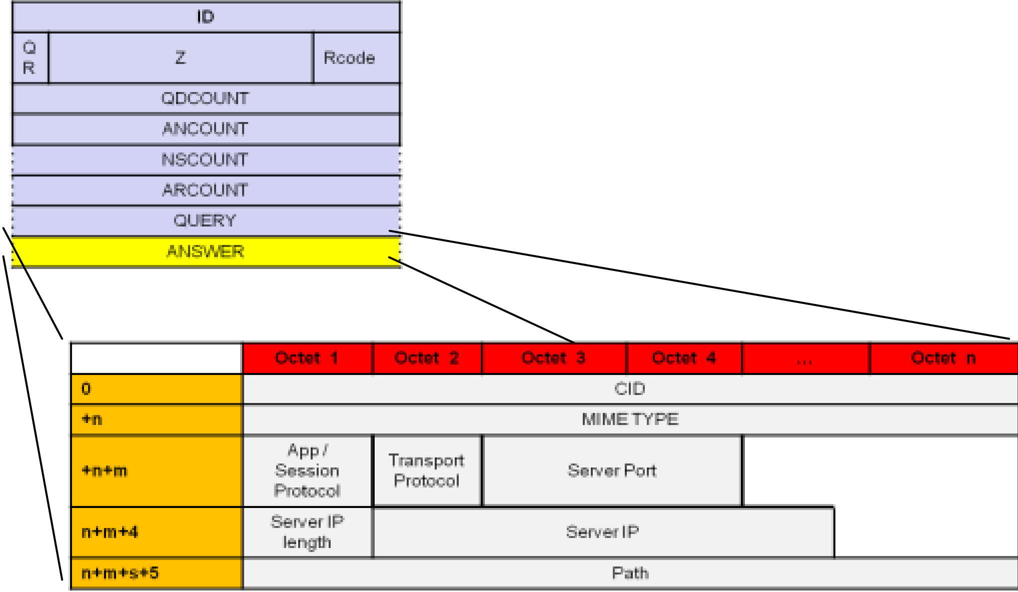
11.2.1.1 Query message





© Copyright 2012, the Members of the COMET Consortium

11.2.1.2 *Response*



11.2.2 CME-RAE

```
package comet.cmerae;
option java_package = "eu.comet.raeprotoc";
option java_outer_classname = "RaeProtoc";

message GenericRequest {
	enum Type {
		RESET = 1;
		VERSION = 2;
		INSERT_PROVISIONING = 3;
		REMOVE_PROVISIONING = 4;
		INSERT_PATHS = 5;
		REMOVE_PATHS = 6;
	}
	required Type type = 1;
	optional ProvInfprov = 2;
	optional PathInf path = 3;
}

message QoSParameters {
	optional float packet_delay = 1;
	optional float packet_loss = 2;
	optional float supported_bandwidth = 3;
}

message DomainEdge { // modes A and B are exclusive
	// mode A: peering number is set
	optional int32 peering_as_number = 1;
	// mode B: access network prefix is set
	optional int32 prefix_length = 2;
	optional bytes prefix = 3;
}

message ProvInf {
	required DomainEdge source = 1;
	required DomainEdge sink = 2;
```


© Copyright 2012, the Members of the COMET

```
      required string class_name = 3;
      optional QoSParametersqosparams = 4;
}

message Prefix {
      required int32 prefix_length = 1;
      required bytes prefix = 2;
}

message Path {
      repeated int32 as = 1 [packed=true];
      optional QoSParametersqosparams = 2;
      required int32 edge_as = 3;
}

message PathInf {
      required Prefix prefix = 1;
      required string class_name = 2; // e.g., BE, BTBE, PR, PR-L(ive), PR-
R(ecorded), ...
      repeated Path paths = 3;
}

message Reply {
      required string version = 1;
      optional bool error = 2;
}
```

© Copyright 2012, the Members of the COMET Consortium

11.2.3 CME-CAFE

```
package comet.cmecafe;
option java_package = "eu.comet.cme.controller.cafeif";
option java_outer_classname = "CafeProtoc";

message GenericRequest {
enum Type {
        CONFIGURE_STREAM = 1;
        COLLECT_EXPIRED_STREAMS = 2;
    };
    required Type type = 1;
    optional ConfigureStreamMessage configure = 2;
    optional CollectExpiredStreamsMessage collect = 3;
}

message ConfigureStreamMessage{
    required int32 id = 1;
    required Filter filter = 2;
    required int32 bandwidth = 3;
    required string cos = 4;
    required bytes key = 5;
    required int32 refresh_time = 6;
    repeated int32 as_path = 7;
}

message Filter {
    optional string ip_source = 1;
    optional string ip_destination = 2;
    optional int32 protocol = 3;
    optional int32 port_source = 4;
    optional int32 port_destination = 5;
}

message CollectExpiredStreamsMessage {
}

message GenericResponse {
enum Type {
        CONFIGURE_STREAM_RESULT = 1;
        COLLECT_EXPIRED_STREAMS_RESULT = 2;
    };
required Type type = 1;
optional ConfigureStreamResult configure = 2;
optional CollectExpiredStreamsResult collect = 3;
}

message ConfigureStreamResult {
enum Type {
        CAFE_SUCCESS = 1;
        CAFE_FAILURE = 2;
    };
    required int32 id = 1;
    required Type result = 2;
    optional string description = 3;
}

message CollectExpiredStreamsResult {
    repeated StreamInformation id = 1;
}
```


© Copyright 2012, the Members of the COMET

```
message StreamInformation {
    required int32 id = 1;
    optional Filter filter = 2;
    optional int32 bandwidth = 3;
    required string cos = 4;
    repeated int32 as_path = 5;
    optional int64 transferred_bytes = 6;
    optional int32 duration = 7;
}
```

11.2.4 inter-CME

```
package comet.intercme;
option java_package = "eu.comet.cme.controller.cmeif";
option java_outer_classname = "InterCmeProtoc";

message GenericRequest {
      enum Type {
            RETRIEVE_PATHS= 1;
            PROCESS_SERVER = 2;
            PROVISION_PATH = 3;
            SERVER_LOAD = 4;
      }
      required Type type = 1;
      repeated RetrievePathsRequest retrieve_request = 2;
      optional ProcessServerSideRequest process_request = 3;
      optional ProvisionPathRequest provision_request = 4;
      repeated ServerLoadRequest load_request = 5;
}

message GenericResponse {
      enum ResponseType {
            RETRIEVE_PATHS= 1;
            PROCESS_SERVER = 2;
            PROVISION_PATH = 3;
            SERVER_LOAD = 4;
      }
      required ResponseType responsetype = 1;
      repeated RetrievePathsResponse retrieve_response = 2;
      optional ProcessServerSideResponse process_response = 3;
      optional ProvisionPathResponse provision_response = 4;
      repeated ServerLoadResponse load_response = 5;
}

message QoSParameters {
      optional float packet_delay = 1;
      optional float packet_loss = 2;
      optional float bw = 3;
}

message RetrievePathsRequest {
      optional string client_ip = 1;
      optional string server_ip = 2;
      optional string cos = 3;
}

message RetrievePathsResponse {
      optional string status = 1;
      repeated Path paths = 2;
      optional string server_ip = 3;
```


© Copyright 2012, the Members of the COMET Consortium

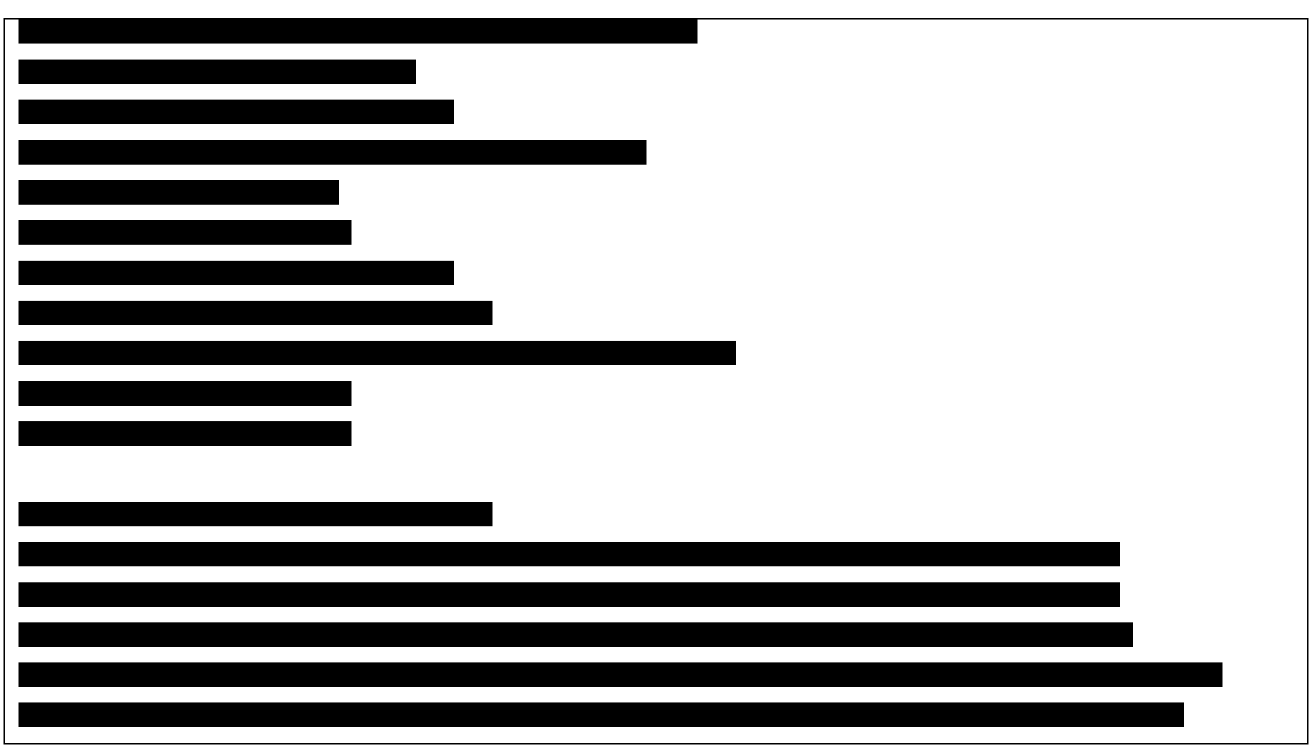
```
}

message Prefix {
      optional int32 prefix_length = 1;
      optional string prefix = 2;
}

message Path {
      repeated int32 as = 1 [packed=true];
      optional Prefix source = 2;
      optional Prefix destination = 3;
      optional QoSParameters qosparams = 4;
}

message ProcessServerSideRequest {
      repeated int32 as = 1 [packed=true];
      optional string client_ip = 2;
      optional string server_ip = 3;
      optional string trans_protocol = 4;
      optional int32 trans_port = 5;
      optional int32 bw = 6;
      optional string cos = 7;
      repeated string key = 8;
}

message ProcessServerSideResponse {
      optional string status = 1;
}

message ProvisionPathRequest {
      repeated int32 as = 1 [packed=true];
      optional string cafe_ip = 2;
      optional int64 bw_aggregate = 3;
      optional string cos = 4;
      repeated string key_new = 5;
}

message ProvisionPathResponse {
      optional string status = 1;
      optional string cos = 2;
      repeated string key_new = 3;
}

message ServerLoadRequest {
      optional string server_ip = 1;
}

message ServerLoadResponse {
      optional string server_ip = 1;
      optional int32 server_load = 2;
}
```


© Copyright 2012, the Members of the COMET

11.3 Databases Specification

© Copyright 2012, the Members of the COMET Consortium

© Copyright 2012, the Members of the COMET



© Copyright 2012, the Members of the COMET Consortium