BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DISTRIBUTED APPLICATION FACILITY IN RECURSIVE INTERNETWORK ARCHITECTURE SIMULATOR

ZAŘÍZENÍ PRO DISTRIBUOVANÉ APLIKACE V SIMULÁTORU REKURZIVNÍ SÍŤOVÉ ARCHITEK-

TURY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. KAMIL JEŘÁBEK
AUTOR PRÁCE

SUPERVISOR Ing. VLADIMÍR VESELÝ, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

This thesis provides a description of Distributed Application Facility (DAF), including its components, in Recursive InterNetwork Architecture (RINA). The DAF is equivalent to an application layer of today's network model. It also documents implementation of the DAF for the OMNeT++ simulation framework. The aim of this thesis is to extend the functionality of simulation library and to provide clear application programming interface.

Abstrakt

Tato práce popisuje Distributed Application Facility (DAF), včetně jejích komponent, v Recursive InterNetwork Architecture (RINA). DAF je ekvivalentem aplikační vrstvy dnešních sítí. Práce dále dokumentuje implementaci DAFu pro simulační framework OMNeT++. Cílem práce je rozšířit funkcionalitu simulační knihovny a poskytnout jednoduché aplikační rozhraní.

Keywords

Recursive InterNetwork Architecture, RINA, Distributed Application Facility, OMNeT++, RINASim

Klíčová slova

Rekurzivní mezisíťová architektura, RINA, Distributed Application Facility, OMNeT++, RINASim

Reference

JEŘÁBEK, Kamil. *Distributed Application Facility in Recursive InterNetwork Architecture Simulator*. Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Veselý Vladimír.

Distributed Application Facility in Recursive InterNetwork Architecture Simulator

Declaration

Hereby I declare that this master thesis was prepared as an original author's work under the supervision of Ing. Vladimír Veselý, Ph.D. The supplementary information was provided by Ing. Marcel Marek, Prof. John Day and Steve Bunch. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Kamil Jeřábek
May 24, 2017

Acknowledgements

I would like to thank to my supervisor Ing. Vladimír Veselý, Ph.D. for his support not only in formation of this thesis, and also for the opportunity to participate in PRISTINE project and RINASim developement. As a bonus I give him recipe to the only food I can prepare.

Slovakian Potato Dumplings with Bryndza Cheese
Ingredients
7 medium potatoes; 0.4 kg flour; 1 tsp salt; 250 grams smoked or regular bacon (block); 1/2 Tbsp oil; 300 g bryndza; 1 egg

Method
Grate the potatoes and place in a bowl. Mix flour, salt and egg with the potatoes until you get a thick, sticky dough. Chop bacon into small cubes and fry in 1/2 tablespoon of oil on low heat until crisp and brown. Cut the dough into boiling water. Cook it until halusky are ready. Once it is cooked, let it drain for 2-3 minutes and mix it with bacon and bryndza all together.

Many thanks also belong to Ing. Marcel Marek that helps create great environment within a team.
I also would like to thank to my family for their support during all my studies. Last but not least, many thanks belong to Pavlína Bártíková for psychical encouragement and an attempt to shift typography to a higher level. To František Šumšal for his support and countless nights spent in teahouse while working on this thesis. And also to Tomáš Hykel for a hints during the typing. To Martin Janoušek, my schoolmate, that carried the heavy burden of study with me.

Contents

List of Figures

Chapter 1

Introduction

These days, distributed applications, as well as cloud computing, is taking advantage over standard communications principles. The communication goes through the network and the *Software Defined Networking* (SDN) moves to the forefront, mainly in this environment.

Today's network technologies are built on a current underlying architecture based on TCP/IP model. This model is facing a lack of mobility mechanisms, multihoming, and more from the beginning. Even though, there is an effort to supplement such mechanisms to the model, it is hard to implement them.

Recursive InterNetwork Architecture (RINA) is a new network architecture; its author is John Day. It is based on principles described in his book *Patterns in Network Architecture*[5]. RINA represents a clean slate architecture that solves all of the mentioned problems from scratch.

RINA is based on the observation that all communication is just interprocess communication between two application processes and every application is a distributed application. Hence, it provides *Distributed Application Facility* that is an application layer compared to the TCP/IP model.

Current model needs explicit programming techniques (for example, using Open MPI) to create distributed applications. RINA should provide an easier approach in this way. It alludes to the fact that application programmers should not care about networking and they should just program the applications by manipulating application objects locally.

1.1 Goals

The theoretical part of this thesis aims to provide a detailed analysis of an application layer called *Distributed Application Facility* (DAF) of RINA architecture. It is completely different from current Internet architecture, hence the description is not limited to overall narration only, but it is focused on each component of DAF relevant to this thesis. In addition, there has to be included an application protocol, establishing of a connection, as well as a programming interface. It also brings a new programming model.

The technical report is focused on the description of the design and implementation of relevant DAF components, including all additional important parts of the OMNeT++ simulation framework. The goal is to use the potential of object-oriented programming to create easily extendable components with clear programming interfaces.

1.2 Structure

Chapter 2 provides detailed description of DAF including all its parts. The emphasis is primarily placed on the local programming interface and RIB Daemon component.

Chapter 3 gives a detailed view on how the applications in RINA RIB Based Programming Model should look like. The cooperation of all components in various cases is provided.

Chapter 4 describes design and implementation of each module created in RINASim during this work. The great attention is paid to implementation of API of each component and structure of Resource Information Base.

Chapter 5 covers testing use cases of the implemented model and overall discussion of impact of the solution on architecture conception.

Chapter 6 contains the final summarization of this thesis.

Chapter 2

Distributed Application Facility

RINA provides a bit different approach from today's networks that are based on TCP/IP model. The significant difference is that there exists only one layer that is recursively repeated. This layer is called *Distributed Application Facility* (DAF) and it is a designation of distributed application. We automatically assume that every application is distributed application and nothing else.

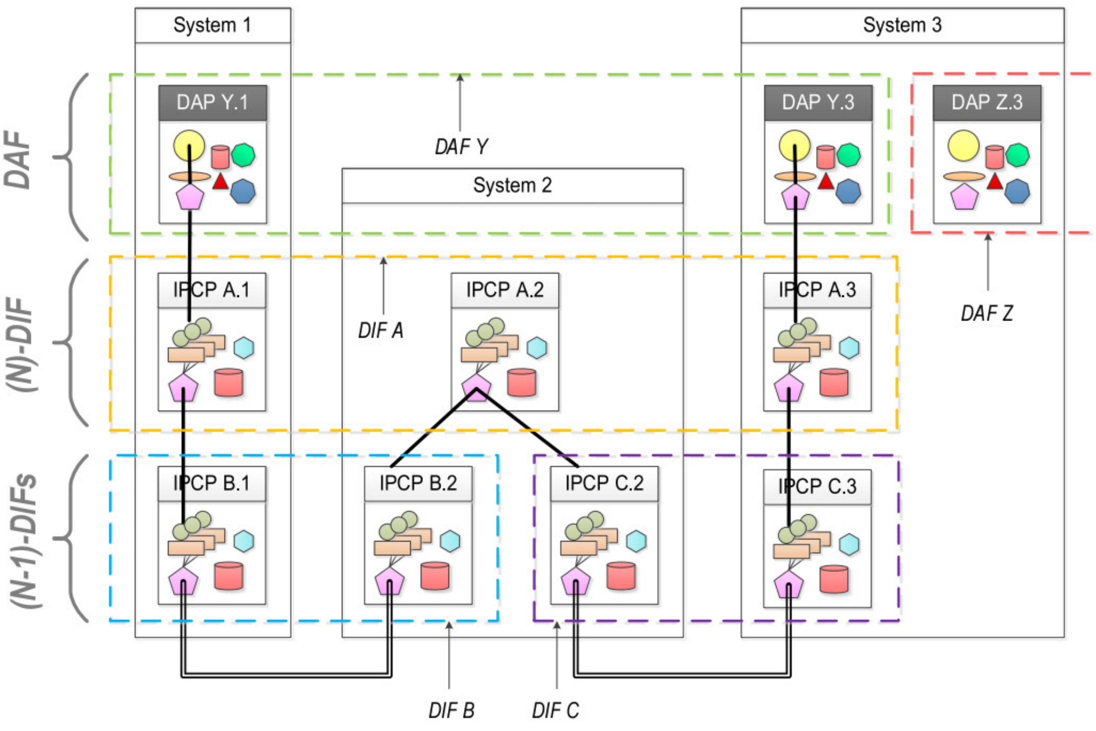


Figure 2.1: DIF, DAF, DAP and IPCP illustration [17]

The DAF is a collection of Application Processes that are cooperating to perform some function. We distinguish between two types of DAFs – *the heterogeneous* and *the homogeneous*. The heterogeneous DAFs are composed of Application Processes of different types.

On the opposite, the homogeneous DAFs are composed of the same types of Application Processes.

In RINA we distinguish one special case of DAF that is intended to provide InterProcess Communication (IPC) called *Distributed IPC Facility* (DIF). The DIF is one example of the homogeneous DAFs. All Application Processes of that DIF are of the same type and they contain the same components. More detailed description of DIFs is provided in RINA Specifications [11], [8].

An Application Processes on the same processing system may use different DIFs of different rank with a different scope to provide IPC. The DAF uses (operates over) (N)-DIF it has access to, as it is shown in more detail in the Figure 2.1. Application Processes of a single DAF have an access to the same DIF to be able to communicate.

A DAF will operate over a single DIF. There is no reason that DAF can operate over multiple DIFs. If Application Process has an access to multiple DIFs, it can leak information because of a different security level of the DIFs. [10]

The following sections provide the detailed description of the DAF components that are relevant to this thesis only. Summary of all RINA information of all components provided below came mostly from RINA Specifications and other sources: [3], [4], [7] [9], [10], [13], [11], [8], [18], [12], [16], [17] and [19].

2.1 Application Process

Application Process (AP) is a program intended to accomplish some purpose, which can be instantiated on a processing system. An Application Process contains one or more *Application Entities*, that are introduced in the following section, and functions for managing system resources, such as the processor, storage, and IPC.

Application Process must have at least one Application Entity. Otherwise, AP would have no input/outpu and would lack the state-sharing purpose.

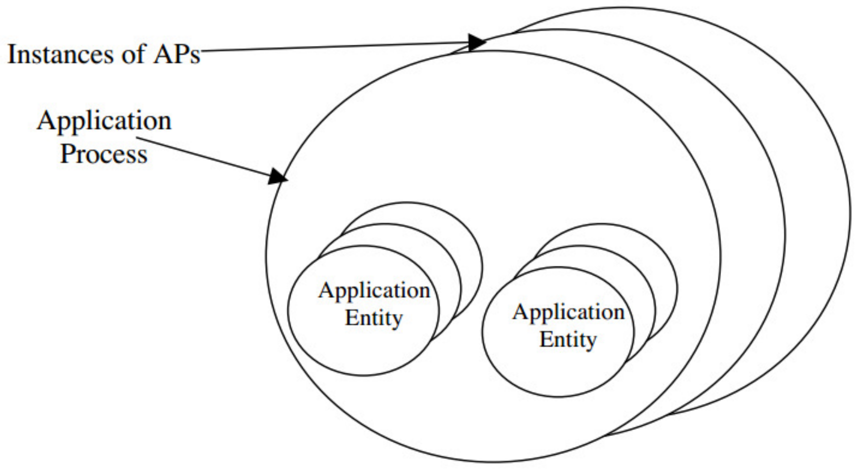


Figure 2.2: Application process with application entities [10]

2.2 Application Entity

Application Entity (AE) is a component (task) of an AP. An application needs to communicate with other applications for multiple purposes, potentially at the same time. These multiple communication connections (streams) need to be created a managed – this is the purpose of AEs. Application Entity implements an application protocol. Such protocol provides shared understanding of the purpose of communication, protocol, set of objects and their meaning that the two AEs exchange. There is used only one application protocol called *Common Distributed Application Protocol* for communication. It is described in more detail in section 2.5.

AEs should be implemented by subroutine libraries which should be hidden for application programmers – they would control it via API calls.

The example should provide a better understanding of the purpose of the Application Entities. Imagine an application that contains two different Application Entities. One AE should be responsible for serving requests for web pages, while another AE might be responsible for communicating with network database server. Each of these AEs has its defined purpose, set of objects they communicate, and communication protocol.

2.3 Instances of Application Processes and Application Entities

Application Process Instance (AP Instance) and *Application Entity Instance* (AE Instance) are instantiations of these tasks. On one processing system there can be many instances of the same AP. Also, it is possible to have many instances of the same AE in one AP.

It may be useful to have the opportunity to create an application connection specifically to one of the instances of AP and AE. Each instance can manipulate with unique data, it can have unique parameters, or it is known to have a defined purpose. Moreover, we want to connect back to it, if the connection fails.

A good example of how the AP and the AE instances might be used is „voice conference call“. Imagine, that one Application Instance is a video conference call with more than one participant. An AE implements a video-streaming protocol. And an AE Instance represents one camera of one of the participants.

2.4 Naming and Addressing

In sections above, DAF components that are participating in the communication were described. We need to distinguish endpoints of communication within DAF. RINA defines special identifiers for each component.

- *Distributed-Application-Name* (DAN) is whatever cast name[1] that identifies a distributed application and it is globally unambiguous. One DAF might have assigned more than one DAN with different access control properties.
- *Application Process Name* (APN) is a globally unambiguous synonym for an Application Process of DAF.

[1]Whatever cast name is a special synonym addressing/naming set of Application Processes.

- *Application Process Instance Identifier* (API-id) is an identifier bound to AP Instance to distinguish multiple AP Instances. It is unambiguous within the Application Process.
- *Application Entity Name* (AEN) is unambiguous within the scope of the Application Process.
- *Application Entity Instance Identifier* (AEI-id) is an identifier that is also unambiguous within single AP, and it helps us to identify different AE instances.

2.5 Common Distributed Application Protocol

Common Distributed Application Protocol (CDAP) is the only required application protocol in RINA. It provides a platform for building all distributed applications. CDAP allows distributed applications to deal with communication at the object level without the need to do an explicit serialization and other input/output operations. The CDAP unifies the approach of sharing data over the network, so we do not need to create any additional specialized protocols.

From the application perspective, the only operations, which can be performed on objects, are create/delete, read/write, and start/stop (execute/suspend). These operations are primarily supported by the CDAP. Also, all of these operations are subject to access rights on the objects.

The CDAP makes no restrictions on objects and their purpose or nature. There is also no error-check mechanism for provided objects, except for versioning mechanism[1] of the objects, which ensures that the objects are correctly interpreted by the application.

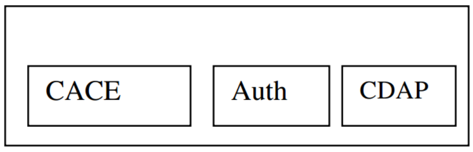


Figure 2.3: The Common Distributed Application Protocol [10]

CDAP is constructed of three different submodules (2.3):

- *Common Application Connection Establishment* (CACE) module is involved in establishing an application connection. It is based on a simplified version of the OSI ACSE protocol.
- *Authentication* (Auth) module provides authentication of the two communicating entities on an application connection. This is the only module that can be modified. It is assumed that there will be many available Authentication module variants (from null to passwords to something stronger). The Authentication module can generate its own protocol, if additional exchanges are needed.
- *Common Distributed Appplication Protocol* (CDAP) submodule.

[1]Discussed later in section 2.7.1 dealing with objects.

CDAP is a symmetric protocol. There is no distinguished party in the communication once the CDAP connection is established.

The CDAP uses only these eighteen messages (Table 2.1):

Message Opcode	**Purpose**
M_CONNECT	Initiate a connection from a source application to a destination application
M_CONNECT_R	Response to M_CONNECT, carries connection information or an error indication
M_RELEASE	Orderly close of a connection
M_RELEASE_R	Response to M_RELEASE, carries final resolution of the close operation
M_CREATE	Create an application object
M_CREATE_R	Response to M_CREATE, carries result of the create request, including identification of the created object
M_DELETE	Delete a specified application object
M_DELETE_R	Response to M_DELETE, carries result of the deletion attempt
M_READ	Read the value of a specified application object
M_READ_R	Response to M_READ, carries part or all of object values, or error indication
M_CANCELREAD	Cancel a prior read issued using M_READ for which a value has not been completely returned
M_CANCELREAD_R	Response to M_CANCELREAD, indicates outcome of the cancellation
M_WRITE	Write a specified value to a specified application object
M_WRITE_R	Response to M_WRITE, carries result of the write operation
M_START	Start the operation of a specified application object, used when the object has operational and non-operational states
M_START_R	Response to M_START, indicates the result of the operation
M_STOP	Stop the operation of a specified application object, used when the object has operational and non-operational states
M_STOP_R	Response to M_STOP, indicates the result of the operation

Table 2.1: CDAP message types [3]

2.5.1 Common Application Connection Eestablishment Phase

Common Application Connection Establishment Phase (CACEP) is the first phase that all communication must go through. It starts immediately after the initiating application process obtains a positive allocation response from the destination application.

The first CACEP within DAF occurs between Application Processes over 'management flow'. This is the first connection between two application processes. After the connection is established, other data transfer connections between AEs may proceed, as it is depicted in Figure 2.4. CACEP has to proceed on all connections. It is required to determine whether two AEs are communicating whom they think they do. Also, when the first CACEP is finished, another phase of communication – called *Enrollment* – is started. The description of this phase can be found in Section 2.6.

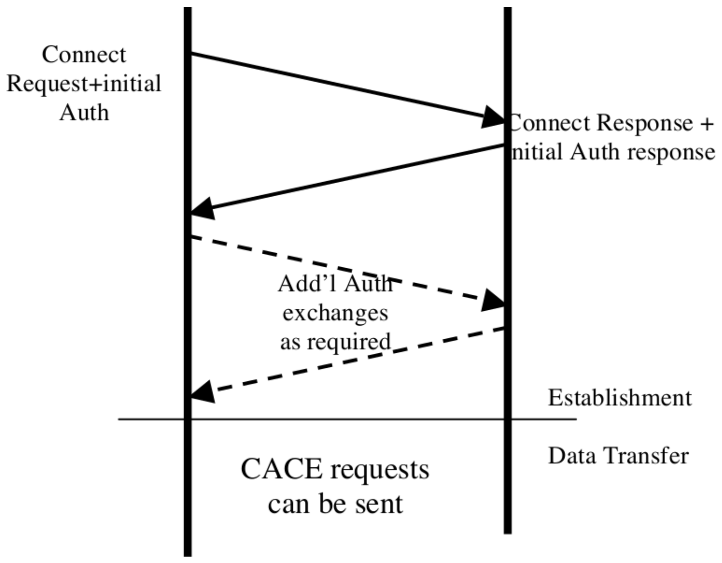

Connect Request+initial Auth
Connect Response + initial Auth response
Add'l Auth exchanges as required
Establishment
Data Transfer
CACE requests can be sent

Figure 2.4: Connection establishment over the newly allocated flow [10]

CDAP messages M_CONNECT and M_CONNECT_R are the only used messages while establishing an application connection. CDAP M_CONNECT messages carry initial authentication information in each direction. The authentication is application dependent, as it may differ from null to password to something more complex (in this case AEs can exchange additional messages).

The detailed description of this phase is depicted in the following Figures 2.5, 2.6, 2.7. These figures come from the detailed specification of CACE phase [4]. Figure 2.5 shows the procedure of the CACE phase from the initiating process perspective. The second diagram, depicted in Figure 2.6, shows the procedure from the responding process side.

The last diagram, Figure 2.7, depicts the release of the communication, both from the initiating and the responding process side. In this figure, both application processes are in „Established" state when initiator starts releasing. Releasing belongs to CACE phase as well.

In both diagrams, transitions are denoted with „input/action" labels. All diagrams show only positive progress. All failures, such as receiving unexpected CDAP messages, are indicated as „wrong input". Any process may initiate the deallocation in every state.

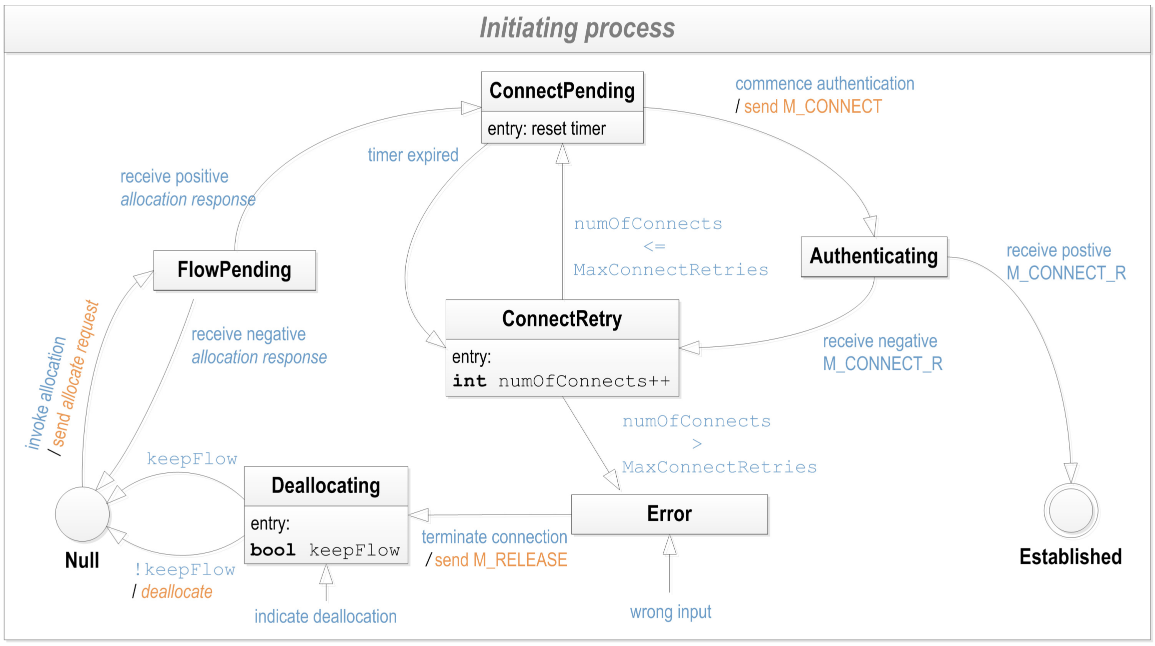


Figure 2.5: Initiating process CACE State Diagram [17]

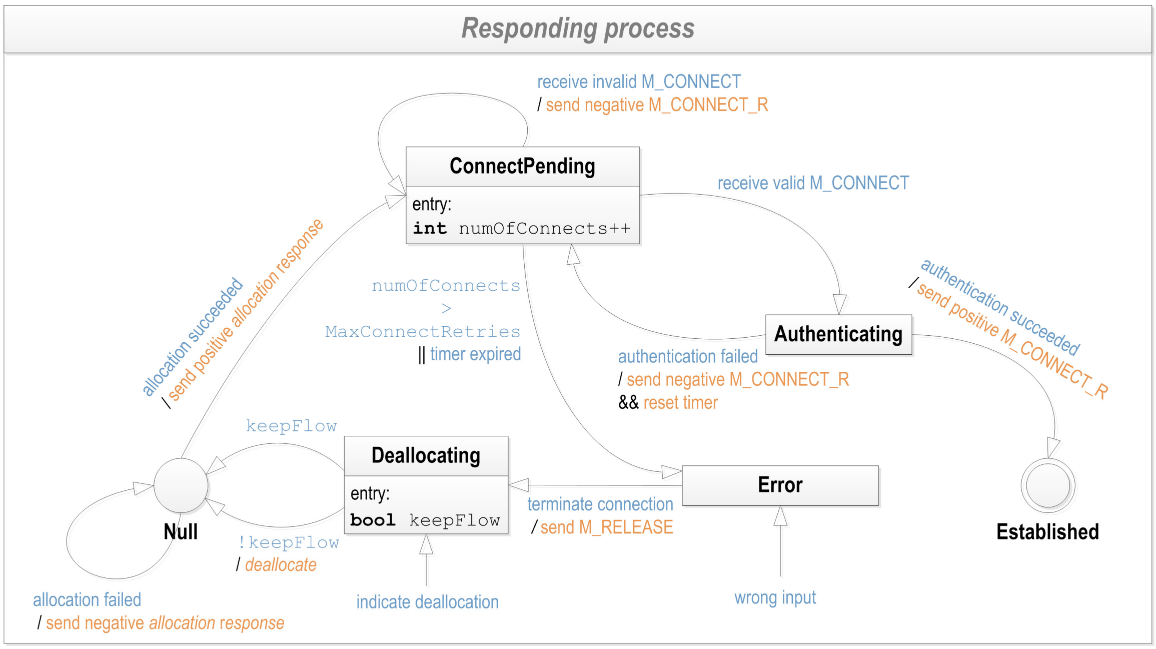


Figure 2.6: Responding process CACE State Diagram [17]

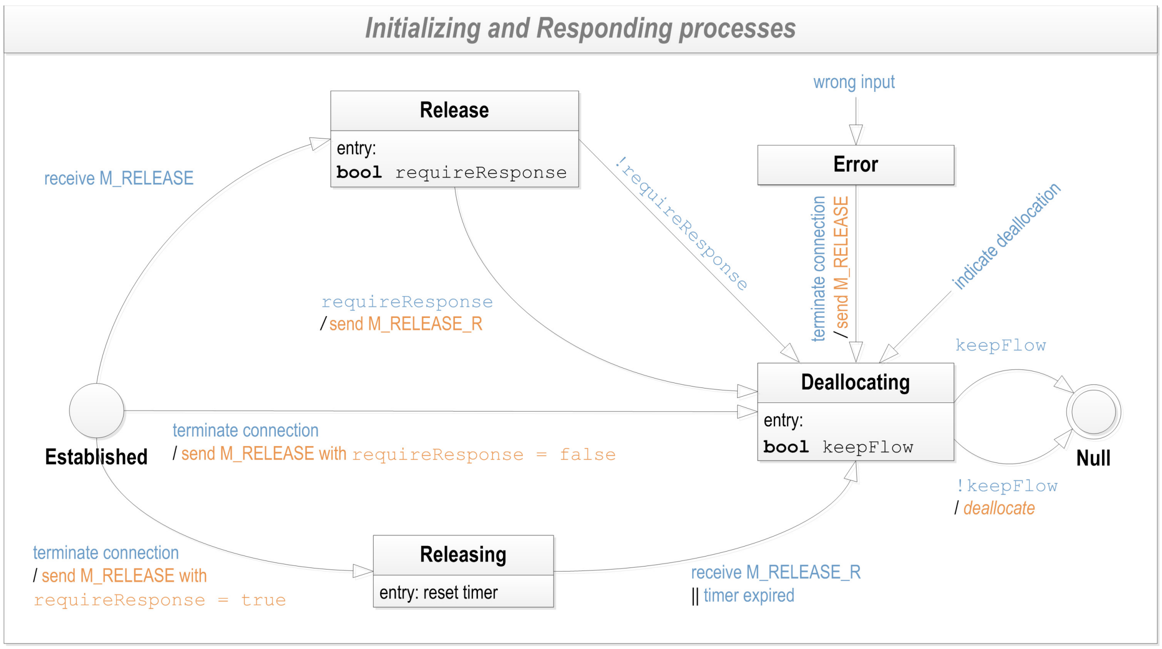


Figure 2.7: Both processes Release CACE State Diagram [17]

2.5.2 Local Programming Interface

Local Programming Interface (CDAP API) provides an easy way to control communication and manage shared objects. It is the main instrument for programming applications by application programmers in RINA. The following definitions are neither complete nor prescriptive.

The listing below provides a description for few shorthands used as arguments in following signatures. The listings are taken from CDAP Specification [3].

- *objectname*
  As noted later, there are two ways to identify an object. Either can be used.
- *objectvalue*
  The value of an object. Each value has a type and representation appropriate for the class of the object.
- *flow*
  A previously-established data transfer flow with appropriate characteristics.
- *CDAPconn*
  The handle that can be used to refer to a successfully-established CDAP connection.
- *result*
  The result field of a reply message (a missing field indicates Success).
- *success*
  The result (Boolean) of invoking an API call, indicating whether the arguments were acceptable and the operation was completed as requested.
- *APname, APinst*
  Application Process name and (if known) unique instance.
- *AEname, AEinst*
  Application Entity (AE) name and (if known) unique instance.
- *invokeID*
  A value is chosen by the application to uniquely identify an in-progress exchange of related messages. In this API description, we permit the invokeID to be omitted, by which we indicate that any potential reply message is to be suppressed.

The definitions are described as non-blocking function calls in a procedural programming language. Many of these calls are performed by RIB-encapsulating AE object. The API will only be used by application programmers, they will not explicitly program it.

The following API description came from CDAP Specification [3].

Lifecycle operations:

- **success = A_OPEN (invokeID, APname, APinst, AEname, AEinst, ...)**
  success = A_OPEN (invokeID, flow, ...)
  The A_OPEN operation requests a CDAP connection. The second signature could be used when reusing an existing flow for a subsequent CDAP connection.

- **CDAPconn, ..., result, success = A_GETOPEN_R (invokeID)**
  Retrieves a reply to a specific A_OPEN request. It returns the result of the earlier A_OPEN request, as well as any other values carried in the response message. If the result indicates success, the CDAPconn value is a valid connection handle and can be used in subsequent operations.

- **success = A_CLOSE (CDAPconn, optional invokeID)**
  Requests closing of the the CDAP connection. No future operations except A_GETCLOSE_R, if an invokeID was provided, are permitted on the CDAPconn. If no invokeID is present, the flow is immediately closed, without waiting for a response. If an invokeID is present, the flow remains open until the A_GETCLOSE_R invocation.

- **result, success = A_GETCLOSE_R (CDAPconn, invokeID, optional leaveopen = false)**
  Retrieves a reply to a previous A_CLOSE request. If the Boolean leaveopen argument is TRUE, the flow will not be closed.

Get/Set operations on valid objects:

- **success = A_READ (CDAPconn, optional invokeID, objectname)**
  Sends a request to retrieve the value of an object.

- **objectvalue, complete, result, success = A_GET_READ_R (CDAPconn, invokeID)**
  Retrieves the result of a previous A_READ request. The Boolean value complete is set if this response completes the A_READ request, indicating that no more replies will be received for this value of invokeID. If complete is false, then there will be at least one more reply message for the invokeID – such reply does not have to carry a value.

- **success = A_CANCELREAD (CDAPconn, optional invokeID)**
  Terminates an A_READ before the entire result has been returned.

- **success = A_CANCELREAD_R (CDAPconn, invokeID, result)**
  Responds to an earlier A_CANCELREAD operation. (Note that there is a race condition when an M_CANCELREAD is sent, but the opposite end has already sent the final response to the read, which causes the invokeID associated with it to be retired and the M_CANCELREAD to be discarded. The M_READ_R with the invokeID that completed the M_READ will have already been sent, so it performs the same function as the M_CANCELREAD_R would have, and the invokeID is retired upon receipt of either – only one will arrive.)

- **success = A_READ_R (CDAPconn, invokeID, result, objectname, objectvalue, optional complete = true)**
  Sends back a result, and if successful, the data requested by an A_READ operation with the specified invokeID. If the value is not complete, the complete Boolean must be set to *false*. The reader will keep the invoke_id valid, awaiting further data, until a subsequent A_SENDREAD_R operation with a complete value of true is received, at which point the A_READ is complete and the invokeID is no longer valid for replies.

- **success = A_WRITE (CDAPconn, optional invokeID, objectname, objectvalue)**
  Sends a request to set the value of object objname to the specified objvalue.
- **objectvalue, result, success = A_GETWRITE_R (CDAPconn, invokeID)**
  Retrieves the result of an earlier A_WRITE request. If provided as part of the reply, it also provides a value for the object (which can be either the prior value, the value after the update, or something else – this is an application behavior).
- **success = A_WRITE_R (CDAPconn, invokeID, result, objectname, optional objectvalue)**
  Sends back a result, and if supplied, an objectvalue, for the object written to by a prior A_WRITE with the same invokeID.

Creation/Destruction of an object:

- **success = A_CREATE (CDAPconn, optional invokeID, class, objectname, optional objectvalue)**
  Requests creation of an object with the specified class, objectname, and if supplied, initial value objectvalue.
- **success = A_CREATE_R (CDAPconn, invokeID, result)**
  Responds to a previous A_CREATE operation with the same invokeID.
- **success = A_DELETE (CDAPconn, invokeID, objectname)**
  Requests the deletion of the named object.
- **success = A_DELETE_R (CDAPconn, invokeID, result)**
  Responds to a previous A_DELETE operation.

Starting/Stopping of an object:

- **success = A_START (CDAPconn, optional invokeID, objectname, optional objectvalue)**
  Requests start of an object with the specified class, objectname, and if supplied, argument objectvalue.
- **success = A_START_R (CDAPconn, invokeID, result)**
  Responds to a previous A_START operation with the same invokeID.
- **result, success = A_GET_START_R (CDAPconn, invokeID)**
  Retrieves the result from a previous A_START operation with the same invokeID.
- **success = A_STOP (CDAPconn, invokeID, objectname, optional objectvalue)**
  Requests the stop of the named object, possibly providing an optional objectvalue.
- **success = A_STOP_R (CDAPconn, invokeID, result)**
  Responds to a previous A_STOP operation with the same invokeID.
- **result, success = A_GET_STOP_R (CDAPconn, invokeID)**
  Retrieves the result of a previous A_STOP operation.

2.6 Enrollment

Enrollment is the phase of communication that follows after the CDAP connection[1] is established with a member of a DAF. An Application Process must always be enrolled to become a member of a DAF. The phase is initialization of a new member of a DAF to become a full-featured member of a DAF. The Enrollment might perform, but is not limited to, the following operations (as it is described in [10]):

1. Determine the current state of the new member AP (if it is completely new, first time joining DAF, or it is a returning member);
2. Assigning capabilities of the new member;
3. Assigning synonyms to the new member for use within the DAF;
4. Initializing static aspects of the AP and initializing a DAF related policies;
5. Creating additional connections;
6. Initializing or synchronizing the RIBs; etc.

The list of operations is not complete. Additional operations may be performed. The operations might be diverse for different DAFs. Normal RIB update follows after successful Enrollment.

In RINA, there exist two types of Enrollment. The first one is within DIF and the second one is within DAF. They differ mainly from the perspective of the information that is exchanged during the phase. In this thesis, we deal with the Enrollment on DAF level only.

The following diagrams are created according to Enrollment specification [7]. The specification is primarily focused on Enrollment within a DIF, but it may be used also within a DAF. Figure 2.8 shows Enrollment of the initiating process. Figure 2.9 depicts Enrollment of the responding process of the communication. In both diagrams, transitions are denoted with „input/action“ labels.

The diagrams are self-explanatory except two parts. In Figure 2.9, we can see state called „CreatingObjects“, during which the member Application Process is sending M_CREATE messages to the initiating process. In this state, the initiating process creates important objects in its RIB to be able to operate within DAF.

After all necessary objects are created, the initiating process can request additional information about the DAF from the member Application Process.

[1]The CDAP connection is described earlier in Chapter 2.5.1 as CACE phase.

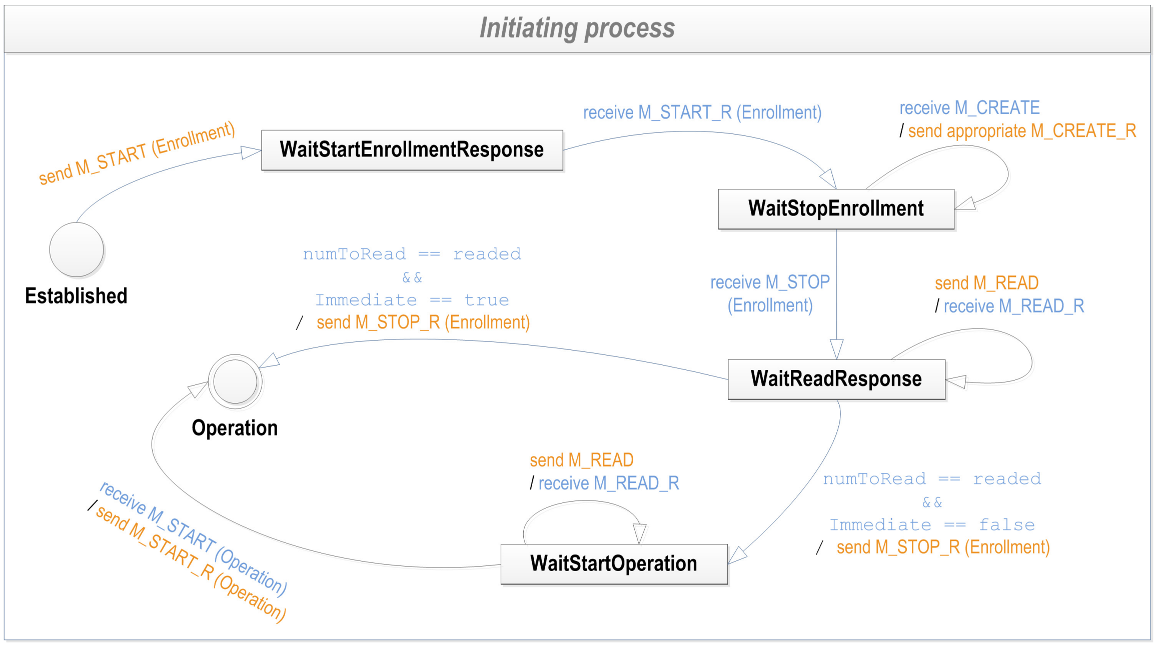


Figure 2.8: Initiating process Enrollment State Diagram [14]

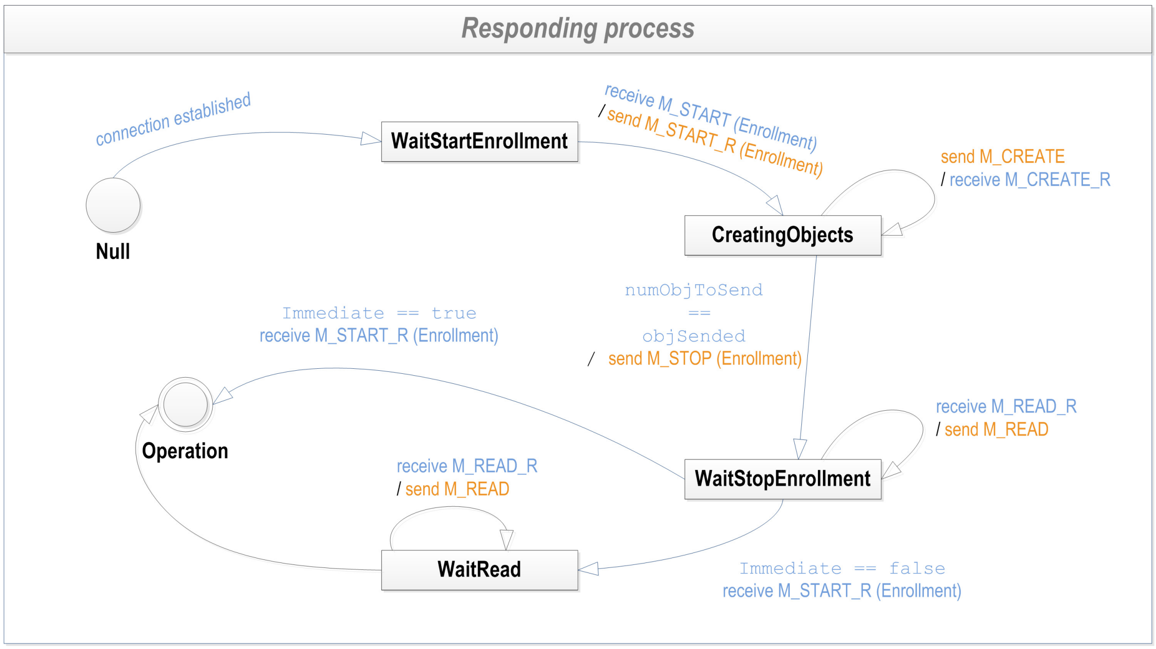


Figure 2.9: Responding process Enrollment State Diagram [14]

2.7 Resource Information Base

Resource Information Base (RIB) is the logical representation of the local repository of objects in DAF. Each member of a DAF has its own portion of the information stored in local RIB. All objects are accessible via RIB Daemon, that is responsible for managing and maintaining them. RIB Daemon is described in section 2.8.

The RIB is a storage, from the operating system point of view. There are no restrictions for implementation of the RIB. In DAF, it should most likely be implemented as some kind of database of an application related objects.

In current TCP/IP based networks, the RIB should be compared to SNMPs Management Information Base that is used for the same purpose (storing objects).

2.7.1 Objects

Object is the designation for a structured data that the CDAP is dealing with. All modern programming languages have the concept of objects; it is useful to deal with data on an object level from its perspective.

Objects are the main building block of the RIB. Two communicating AEs creates a shared object space, and they provide an access to the portion of the application's RIB.

All objects belong under some access rights. There should be at least two groups of objects. The first one is accessible only by AEs that created it and communicate it. The second one should contain the rest of objects of an application. These objects can be accessed by any component. But there should be more access rights models, than this easy one.

Objects stored in the RIB and relevant to CDAP have four core properties:

Class, or Type recursively captures all of the types of the outermost object and all contained objects.

Name is unique identifier among other objects of the same class.

Value contains actual value of the object.

Object Identifier is an integer identifier that is assigned to specific instance of an object by its owner, it is an alias to its class and name.

The objects that are stored in RIB are of two types. The *passive* objects that contain static information, and the *active* objects, which information has a side effect (e.g., a boolean value, which setting to true causes a reboot of a processing system).

2.7.2 RIB Based Programming Model

The RINA comes up with a different approach of programming distributed applications (or network applications) than it is in today's networks. As it is described earlier, the CDAP protocol, and one or more AEs, can be used to create a Distributed Application. All the data are stored in distributed RIB.

This model introduces an idea that application programmers will be more likely manipulating with local objects in RIB, rather than using an explicit communication operations. The AEs and CDAP are completely hidden from the application programmer. The application will not use AEs and CDAP APIs directly. They should use higher level application programming interface described in section 2.5.2 instead. It is primarily designed for manipulation with objects.

2.8 RIB Daemon

RIB Daemon is one of the key components of the Application Process. It is responsible for managing and maintaining information objects in RIB within DAF. The RIB Daemon should monitor all events that are occurring within DAF, because there could be a subscription that is triggered by the event. It can also keep a log of events if needed.

Application Process of a DAF may have several sub-tasks or threads. The RIB Daemon is a common component for those sub-tasks. If any sub-task has requirements for information from other participants in distributed application, it uses RIB Daemon to get it. The getting/setting of the information is done via subscriptions for objects.

All tasks of an AP can communicate with the RIB Daemon via special RIB Daemon API. The possible RIB Daemon subscription API is described in the following Section 2.8.1.

The RIB Daemon accepts subscriptions from tasks, and it performs the subscriptions as efficiently as possible. It contains mechanisms to avoid duplicate subscriptions. The subscription requests are time driven, event driven and direct. In DAF, it is expected that there will be a higher proportion of direct requests.

According to the RIB Based Programming model described in Section 2.7.2, the RIB Daemons role is also to make information available as efficiently as possible. It keeps the delay of the tasks that are using the data as little as possible. This needs an implementation of some pre-paging mechanism with the prediction of objects that will be needed next.

We can summarize the main functions of the RIB Daemon within DAF:

- upon events or periodically request or notify all sets of members of the current value of selected objects;
- monitor all events occurring within the DAF, in case it is bound to a subscription;
- immediate notification of an events arrival to tasks;
- to provide RIB Daemon API that can be used by APs sub-tasks;
- respond to requests for information from other members of DAF;
- maintain a mandatory log of received events;

2.8.1 RIB Daemon API

The RIB Daemon API is based on read/write subscriptions to objects. The receiving/sending of an object is based on an occurrence of an event or done periodically.

The API should look like (according to the RIB Daemon Specification [6]):

Create Subscription *<Read/Write> <relation-expression> <tolerance> <attribute_list> <notify/record> <member_list> <subscription-id>*

Delete Subscription *<subscription-id>*

Read Subscription *<subscription-id>*

Read Subscription is there for receiving the definition of a subscription. Note that there is no *Write Subscription*, the only possible procedure is to delete subscription and create a new one.

Description of parameters:

<*Read/Write*> define whether the subscription will cause receiving data from the members or it will send data to them.

<*relation-expression*> represents an expression that defines when this subscription will be invoked. This should be time periodicity, a boolean expression of events, or attribute values defining conditions upon which subscription will be invoked.

<*tolerance*> parameter provides an indication of the tolerance that is bound to relation-expression, under which the RIB Daemon can optimize requests that can occur „near“ each other.

<*attribute_list*> represents the information that will be read or written.

<*notify/record*> determines if the action is only recorded in the RIB or it should be delivered to the requester (requesting task), or both.

<*member_list*> is the list of names of the DAF members which attributes will be exchanged.

<*subscription-id*> is special identifier that uniquely identifies this subscription internally to this process.

Chapter 3

Applications Concept in RINA RIB Based Programming Model

RINA comes with a new view of how distributed computing should work. This chapter provides a deeper look into how current applications should be designed using RIB-Based Programming model. Each component has its defined purpose and functionality as it was described in the previous chapter. The following design description definitely not represent the only possible way. Even more, It describes our understanding and possible implementation of applications in RINASim.

3.1 Overview

Application Process is the highest level component encapsulating all other components. The programming interface described in section 2.5.2 is used here for programming applications. Most of operations behind the API calls will be handled by AEs, which is completely hidden from application programmers. The application programmers should not care about communication and distribution of objects behind the API calls and should instead manipulate objects using only through the high-level API – the emphasis is on AEs to cover communication. The main benefit of this approach is that RINASim users can define their way how the communication will be handled.

All DAFs should be secured and there has to be an authentication mechanism protecting not trusted APs to become an member of DAF. This is included in CACE and Enrollment phase when AP tries to connect to AP of DAF.

AEs has shared understanding of application objects that they are communicating with, i.e. an application protocol. That means for different purposes of communication there should be different AEs. AEs handle communication between nodes, as well as subscriptions with RIB Daemon. There are two possible ways of communication – using RIB Daemon or a direct exchange of messages between two AEs. It means that AEs stay in the border between an application, RIB Daemon, and other AEs in different APs. Connection to concrete AE Instance requires CACE including authentication. This protects accessing shared objects of communication between two concrete AE Instances.

Then there is RIB Daemon that is responsible for managing application objects in RIB. It is the only component in AP that should access and manage objects in RIB. There should be implemented an access control mechanism to determine, whether a component can perform a read/write/create/delete operation on an object on given level. RIB Daemon uses AEs for communication between nodes. All requests will be optimized according to subscriptions.

3.2 Streaming

One of the most common applications using communication through the network is `streaming`. This section provides a possible view on how the communication and interactions between the internal AP components look like.

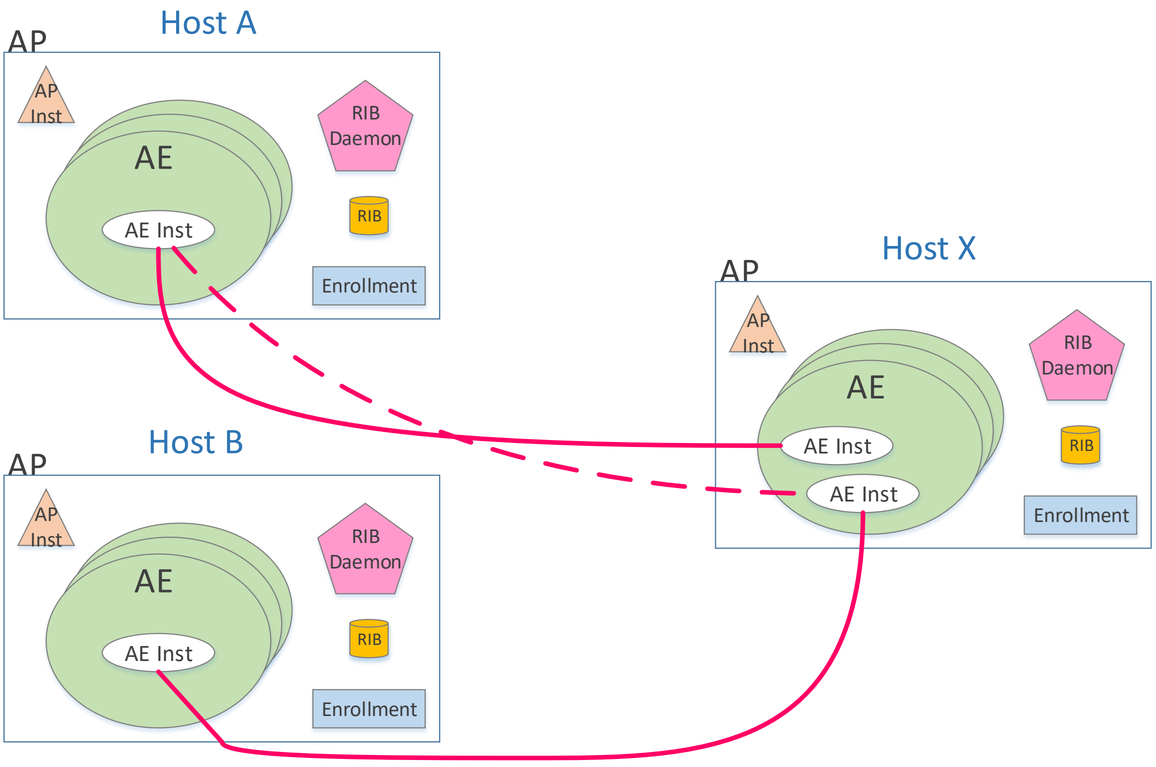


Figure 3.1: Streaming application between nodes

The Figure 3.1 shows how the connections look like in streaming application. The process is as follows:

1. At first, there should be an AP that continuously write data to an object in RIB. In our case, the AP on a host `Host X` spawns an AE Instance. It creates an object using AP API call. The API call forwards a request for object creation to AE Instance and it creates an object in RIB using RIB Daemons API. The AP then starts to write data to that object periodically. This means that the AP API call is periodically used. The AE Instance creates a subscription in RIB Daemon for writing an object value to the RIB on every API call. This is the type of subscription that is handled only once on this request, not periodically or on an event. This streams the data only locally.

2. The AP on host `Host A` wants to read the stream. In the first instance, the AP goes through Enrollment procedure to become a member of the same DAF as AP on `Host X`. It becomes a member of the DAF.

3. The AP on host `Host A` spawns an AE Instance and creates a connection to an AE Instance of AP on the `Host X`. The AP via AP API call requests the same object which the streaming AP is writing data to. The request is forwarded to the AE Instance. There are several options possible how the data should be requested and processed.

    - The AE Instance creates a subscription to RIB Daemon that will poll an object from the `Host X`'s RIB. This means, that it will periodically send CDAP `M_READ` messages and write the obtained object to the local RIB.
    - The second option is that AE Instance subscribes for an object to local RIB Daemon once. RIB Daemon generates CDAP `M_READ` message, that will be sent to the second AP. The receiving AE Instance creates a write subscription, based on the received request, to the local RIB Daemon, for an object that is in the local RIB. The RIB Daemon on the `Host X` then adds the requesting AE Instance to the write subscription list for the object. It can be a time driven or event-based subscription. Then the RIB Daemon sends CDAP `M_WRITE` messages to the requesting AE Instance. The AE Instance forwards it to RIB Daemon and the object is written to the local RIB.

4. The AP gets the data upon an AP API call `A_GET_READ_R` with the received object and response flag set, so that other replies will be received.

5. If the data are not needed, the AP API call `A_CANCELREAD` is forwarded to the AE Instance. The AE Instance generates CDAP `M_CANCELREAD` message and sends it to the second AE Instance. The receiver AE Instance deletes the previous write subscription for an object.

There are possible options how the two APs can be connected. The streaming AP may create a private stream, so that the data will be available only by connecting to one special AE Instance, only on AE level, or globally in DAF. This depends on where the data in RIB are stored.

If the object is stored under a concrete AE Instance, the connection to the concrete AE Instance on the remote host has to be created to receive this data by requesting AE Instance. This is indicated in Figure 3.1 by the dashed line from `Host A`'s AE Instance to the same AE Instance as `Host B`'s AE Instance is connected.

If the object is stored in the global level of DAFs RIB, the connection should be created to any AE Instance and the data can be received.

3.2.1 Object distribution

The RIB is a partially replicated database. The same data objects are spread among multiple RIBs in DAF. The data objects are distributed to nodes when an AP request it. But there is a possibility to spread the objects between multiple nodes, even, they are not requested. The reason for this is to make them replicated and available closer.

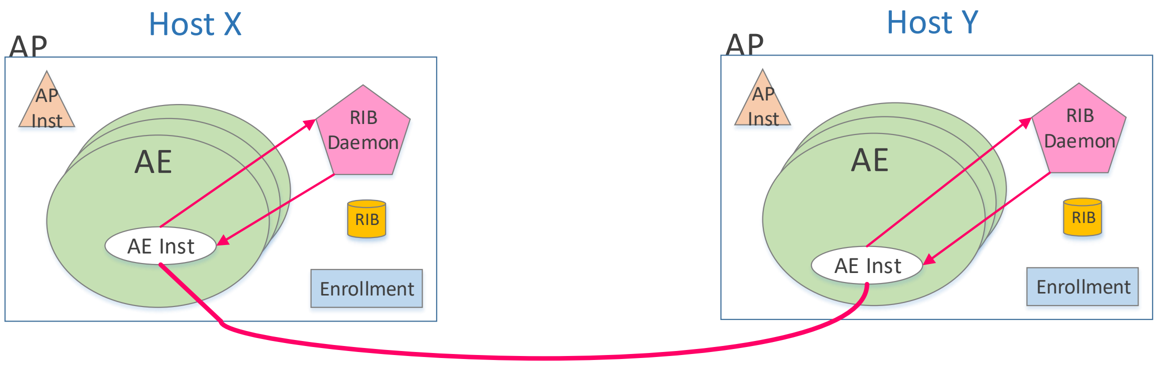


Figure 3.2: Forwarding data objects between nodes

In this case, there are spawned AE Instances between two nodes and objects are distributed by RIB Daemon from one RIB to another. This is applied especially to global objects that can be accessed by everyone. The only interactions are between AE Instance and RIB Daemon.

3.3 Conference Call

Another good example of a commonly used application using communication over the network is a conference call between multiple users. The process should look like:

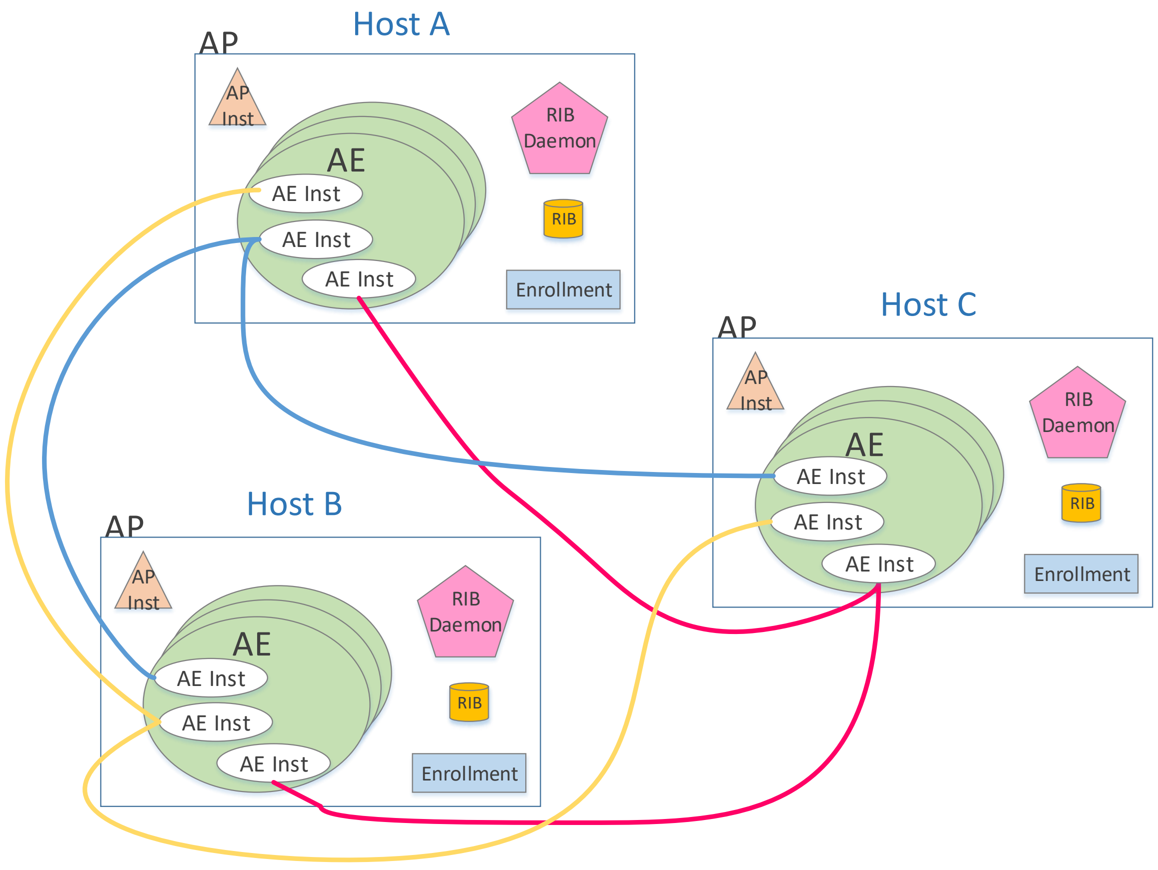


Figure 3.3: Conference call between multiple nodes case 1

1. The start should be similar as in the previous streaming case. The AP on `Host C` spawns an AE Instance which periodically writes data to the local RIB. This is done without any connection between other AE Instances of different AP.

2. If the AP on `Host A` wants to join the conference call, it creates a connection to the `Host C`'s AE Instance. The process is the same as in the streaming case. At first, the `Host A`'s AP has to be a member of the same DAF as the AP on the `Host C`. If not, the Enrollment has to proceed. The AE Instance reads data from remote RIB, or the write subscription is created from the remote AE Instance. The conference call works in both directions. There are more possibilities how the AE Instances can be connected.

   - There is another AE Instance on the `Host A` that writes data to local RIB. The AE Instance of `Host C` needs to create a connection to that AE Instance and get the data objects from it. This means that there are multiple streaming AE Instances and other AE Instances have to create connections to them to get the data. It is the same process as in the streaming case but repeated multiple times. The connections between AE Instances of different APs are depicted in Figure 3.3.

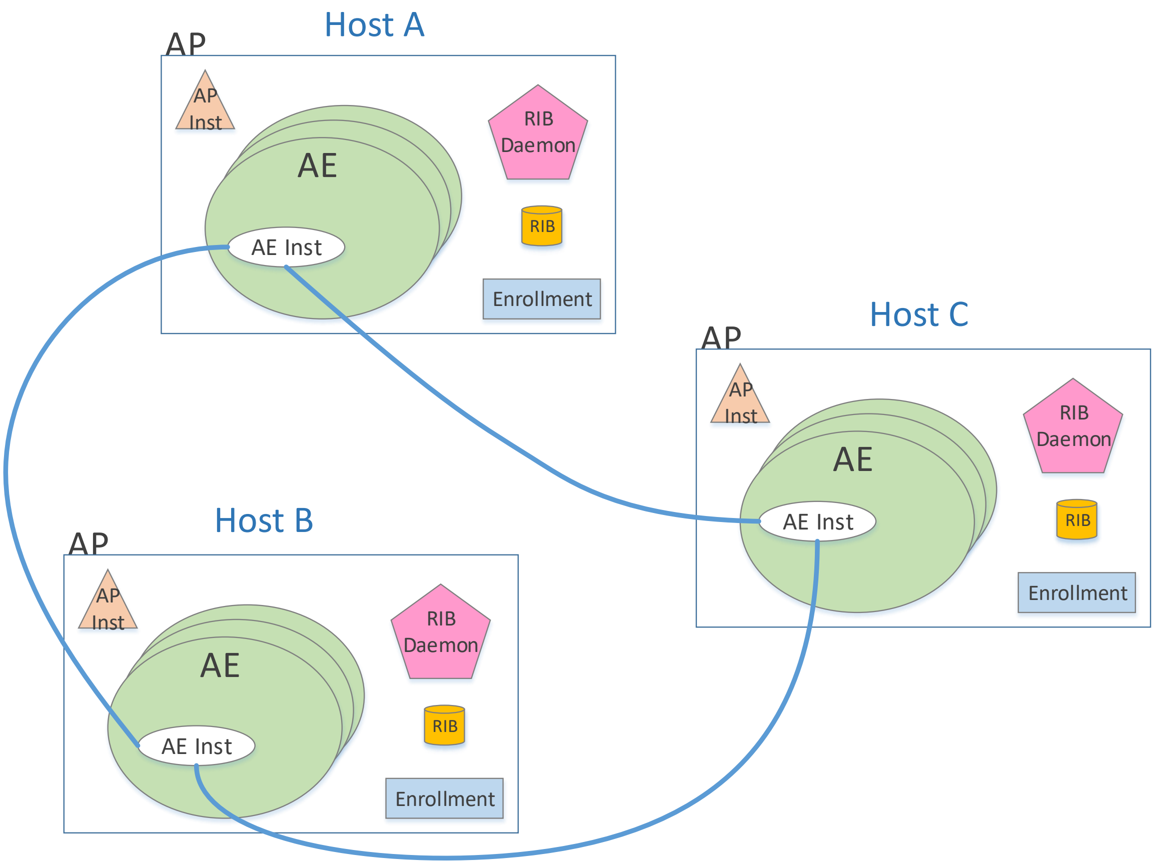


Figure 3.4: Conference call between multiple nodes case 2

   - Another possible approach is that there is only one AE Instance in each AP participating in the conference call. The AE Instance creates one connection to the remote AE Instance that serves for both directions of the communication – such connection requires QoS with higher bandwidth. Or there are created two connections between the same AE Instances for communication. The one connection is used for getting data from one AE Instance and the other for the second direction. The connections are indicated in Figure 3.4.

3. If the data are not needed, the AP cancels the read by issuing the `A_CANCELREAD` API call, and the data objects are not read anymore. The rest of the process is the same as in the streaming case.

3.4 Ping

Ping is one of the most basic applications that should be requested. In this case, each RIB contains a global, periodically updated object, containing the currently synchronized time. The application then creates the connection with the desired node and request the value of an object of that node. In the case of AE implementation, as it was mentioned earlier, two possible options of communication are possible.

- *using RIB Daemon* - The AE Instance subscribes for an object to RIB Daemon. The RIB Damon then requests an object from RIB from the desired node and returns it to AE that forwards this value to an AP.
- *using direct request* - The AE Instance sends a request for time from another AE Instance. This is a direct request between two AE Instances without using the RIB Daemon and RIB. The AE Instance forwards the obtained value to the AP.

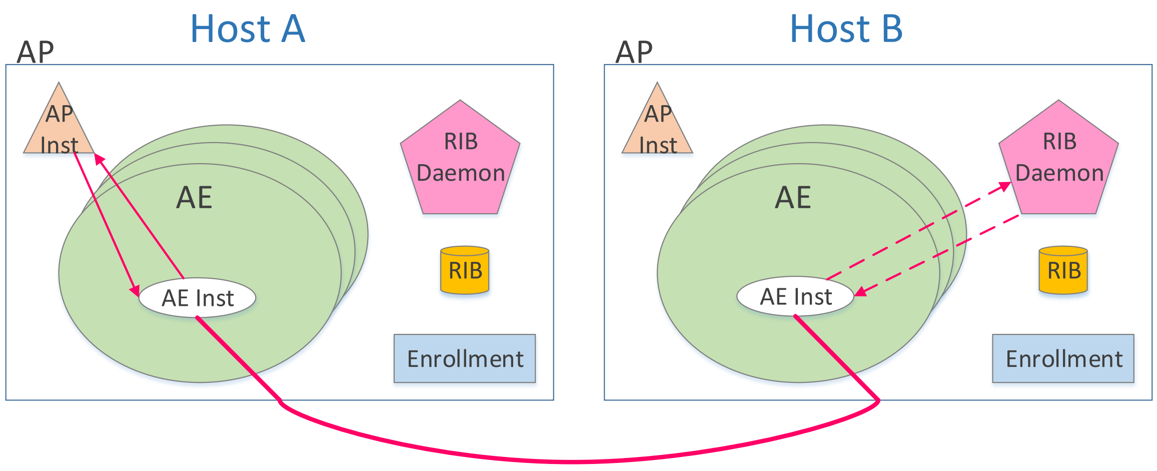


Figure 3.5: Design of Application Process

Figure 3.5 depicts both possible cases in one. The dashed lines represent the case where the RIB Daemon is incorporated into the process of obtaining the data.

Chapter 4

Design and Implementation of DAF

This chapter documents the design and implementation of Distributed Application Facility and its components in RINASim.

4.1 Development Environment

4.1.1 OMNeT++

OMNeT++ [1] is an open-source discrete event simulation framework used primarily for simulation of networks. Network simulation is not the only focus of this project; it is also used to simulate IT systems as well as hardware architectures.

OMNeT++ provides components (modules) that create the basic building blocks. These modules are hierarchically plugged and they communicate with each other. The modules should have defined input and output gates. Between the gates are created connections that all communication among modules goes through.

The component logic is programmed in C++ language. A special language called NED is used for the structural definition of composite components. OMNeT++ runs on every Unix-like platform as well as on Windows.

4.1.2 RINASim

RINASim (Recursive InterNetwork Architecture Simulator) is an open-source simulation library for the OMNeT++ framework. The purpose of RINASim is to provide a way to model and simulate RINA networks. It is developed by the NES@FIT research group at the Faculty of Information Technology of Brno University of Technology.

The development of this library has started within the PRISTINE [15] EU-7FP-ICT project. It is hosted on GitHub [2] under MIT license. It was successfully presented at OMNeT++ Summit 2016 and got officially added to supported OMNeT++ frameworks.

4.2 Modules

This section describes the design and implementation of modules participating in an AP. It is necessary to pay extra attention to each module as each module has its special purpose in the AP. Modules are designed in such way so that they can be extended or replaced by a new module.

The OMNeT++ allows us to use three different techniques of passing objects between modules – using signals, messages, or direct API calls. The signals are emitted by one module transmitting an object and caught by any module that has a subscription to this signal. This thesis makes use of all mentioned techniques. Figure 4.1 depicts the interaction between modules within an AP.

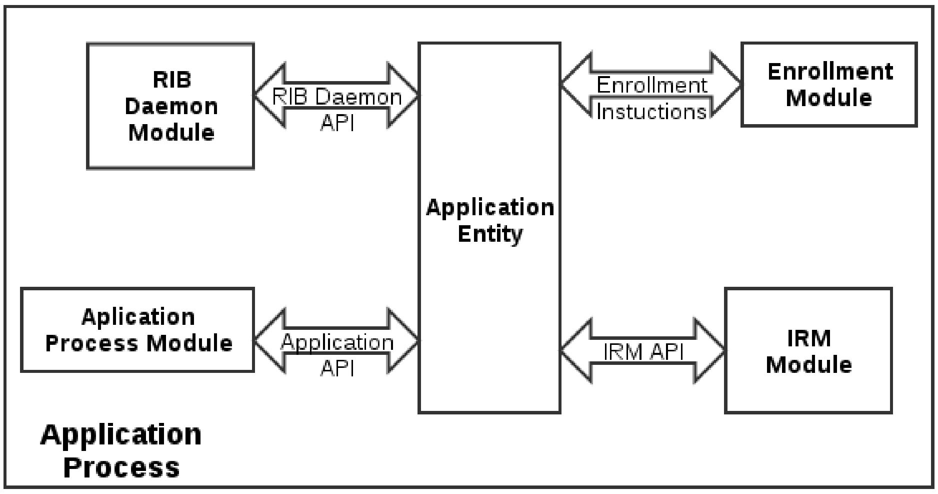


Figure 4.1: Design of Application Process

4.3 Application Process

The AP is a module wrapper that contains all internal modules as it is depicted in Figure 4.2. The current level of abstraction enables us to create one static AP Instance. The apInst represents the implementation of the AP Instance functionality, and it is the core module of AP. The purpose of this module is to provide the API (described in the 4.3.1) to users so they can create its applications.

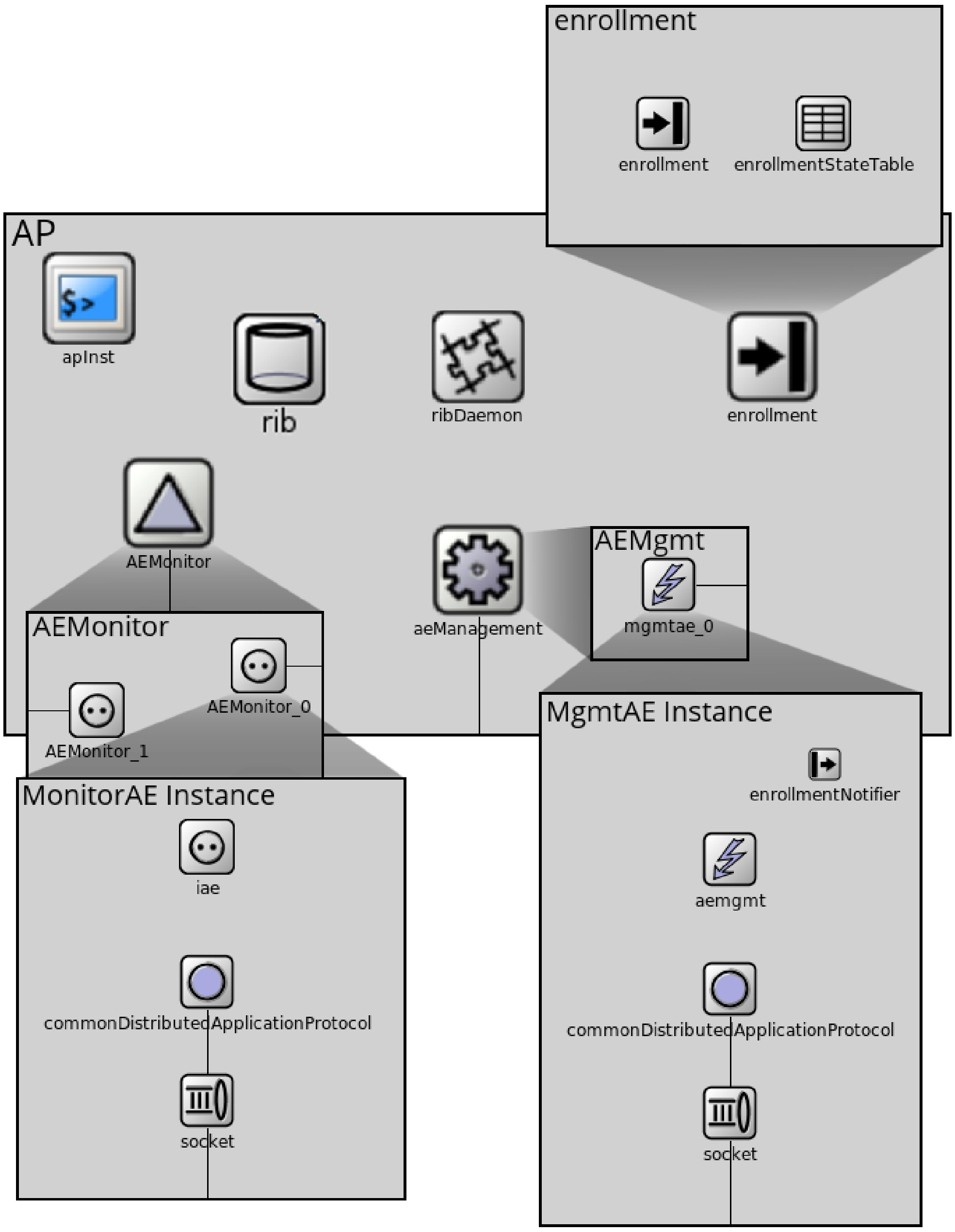


Figure 4.2: Application Process module structure

4.3.1 AP API

The OMNeT++ is a discrete event simulator, so each API call has to be bound to a scheduled event. The blocking calls are not possible to implement here. The AP API is designed so that every scheduled API call, described in 4.1, has its callback method for receiving a result of the operation. These callback methods are under the explicit control of application programmers. The following API calls are implemented, and it can be used when programming AP Instance.

Method	Description

`a_open(int invokeID,`
`string APname,`
`string APinst,`
`string AEname,`
`string AEinst)`
	- checks if the AP is already enrolled
	- the input parameters represents a destination naming information, source naming parameters are generated from them
	- creates an AE Instance dynamically based on the input parameters
	- issues start of AE Instance functionality that leads to the creation of CDAP connection

`a_close(int CDAPConn,`
`int invokeID = 0)`
	closes the connection

`a_read(int CDAPConn,`
`string objName,`
`int invokeID = 0)`
	reads object matched by `objName` using the CDAP Connection

`a_write(int CDAPConn,`
`string objName,`
`object_t *obj,`
`int invokeID = 0)`
	writes obj value to an object matched by `objName`

`a_cancelread(int CDAPConn,`
`int invokeID = 0)`
	cancel the reading of an object on CDAP Connection matched by `invokeID`

`a_create(int CDAPConn,`
`string clas,`
`string objectname,`
`object_t *obj = NULL,`
`int invokeID = 0)`
	creates an object with desired `objectname`, if `obj` is not `NULL` then value is also assigned

`a_delete(int CDAPConn,`
`string objectname,`
`int invokeID = 0)`
	deletes an object matched by `objectname`

`a_start(int CDAPConn,`
`string objectname,`
`object_t* objectvalue = NULL,`
`int invokeID = 0)`
	starts an object with desired `objectname`, if `objectvalue` is not `NULL` then value is also assigned

`a_stop(int CDAPConn,`
`string objectname,`
`object_t* objectvalue = NULL,`
`int invokeID = 0)`
	stops an object with desired `objectname`

Table 4.1: AP API calls

This second list covers all callback functions that can be redefined while programming a user specific AP Instance. All callbacks are called when the result is obtained for a particular previous API call.

The APIResult object is passed as a parameter to all of the named methods in the following list. The APIResult class contains the result of an operation, identification of the connection, and application object if needed. It is created in AE Instance and passed via signal back to AP Instance.

Method
`onA_getOpen(APIResult* result)`
`onA_getRead(APIResult* result)`
`onA_getWrite(APIResult* result)`
`onA_getCreate(APIResult* result)`
`onA_getDelete(APIResult* result)`
`onA_getStart(APIResult* result)`
`onA_getStop(APIResult* result)`
`onClose(APIResult* result)`

Table 4.2: AP API callback methods

All of these callbacks are called immediately after the appropriate operation is done, if not stated otherwise by an AE Instance implementation.

4.4 AE

Application Entity implements an application protocol – shared understanding of objects it is dealing with. This means that there should be several different AE instances with different purposes, which will deal with objects in different ways.

The aei is a core module, in which all main functionality is implemented. This should be implemented by RINASim users by themselves. The interface classes AEBase and AE are provided, which extend the interface class by basic functionality. The AE class provides methods that handle reactions on CDAP messages obtained from commonDistributedApplicationProtocol, and reactions on calls from apInst API and RIB Daemons API. The list of methods and their meanings is named in 4.4.2.

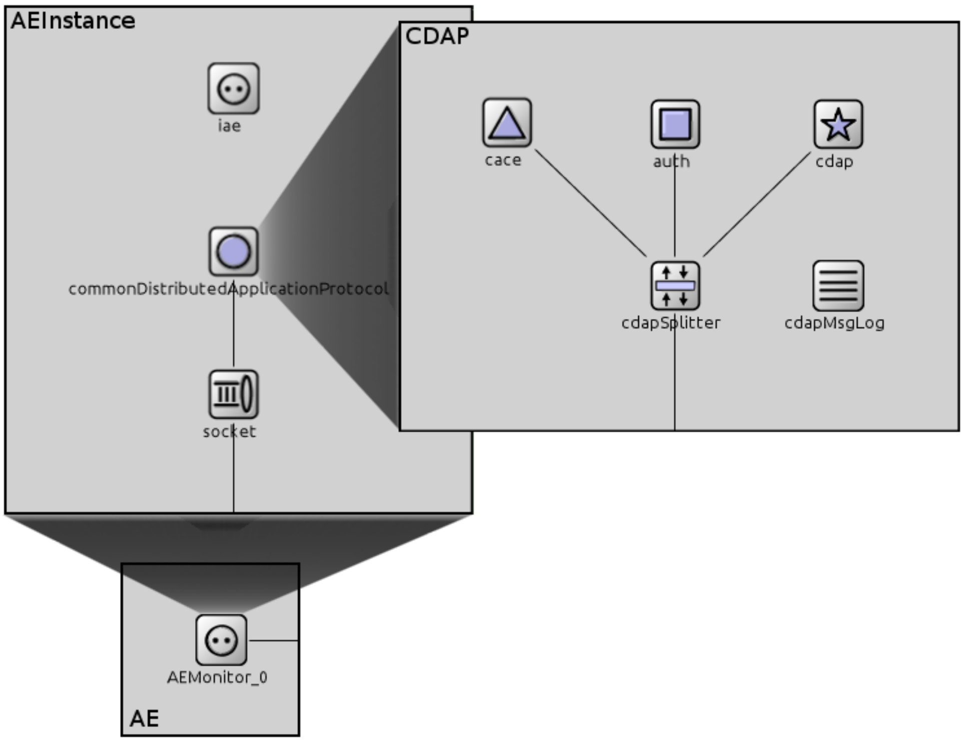


Figure 4.3: Application Entity Instance module structure

The `commonDistributedApplicationProtocol` is a submodule provided in each AE Instance. Its purpose is to forward and process CDAP protocol messages. It is modeled as a compound module consisting of five submodules:

- `cace` - Common Application Connection Establishment protocol forwarding CDAP `M_CONNECT` and `M_RELEASE` messages;
- `auth` - providing advanced authentication exchanges;
- `cdap` - providing forwarding of other CDAP messages;
- `cdapSplitter` - splitter forwarding messages to appropriate upper modules;
- `cdapMsgLog` - logger for all processed messages.

The last module is the `socket`. It should work as a buffer for incoming data. If the flow is overwhelmed with incoming data it should return an error.

4.4.1 Parameters

Parameters of the AE Instance are divided into three parts. The first part contains naming information with the source and destination of the connection it is managing. The values are set during dynamic creation of the module initiated by an API call from `apInst`.

Data type	**Name**	**Description**
`string`	`aeName`	AE name of this AE
`string`	`aeInstance`	AE Instance name of this AE Instance
`string`	`dstApName`	destination AP name
`string`	`dstApInstance`	destination AP Instance name
`string`	`dstAeName`	destination AE name
`string`	`dstAeInstance`	destination AE Instance name

Table 4.3: AE naming parameters

The next two parts are fully configurable. The first one contains all QoS settings for a given connection, as can be seen in the following listing.

Data type	**Name**	**Description**
`int`	`averageBandwidth`	bits/s
`int`	`averageSDUBandwidth`	SDU/s
`int`	`peakBandwidthDuration`	peak bandwidth-duration
`int`	`peakSDUBandwidthDuration`	peak SDU bandwidth-duration
`int`	`burstPeriod`	burst period
`int`	`burstDuration`	burst duration
`double`	`undetectedBitErr`	undetected bit error rate
`double`	`pduDroppingProbability`	PDU dropping probability
`int`	`maxSDUsize`	maximum SDU size
`bool`	`partialDelivery`	partial delivery
`bool`	`incompleteDelivery`	incomplete delivery
`bool`	`forceOrder`	SDUs must be delivered in-order
`int`	`maxAllowGap`	maximum allowable gap in SDUs
`int`	`delay`	delay in usecs
`int`	`jitter`	jitter in usecs
`int`	`costTime`	$/ms
`int`	`costBits`	$/Mb

Table 4.4: AE QoS parameters

The last part contains the configuration of the authentication used during CACE phase. This is also a fully configurable part. By default, no authentication is used, so the authentication always passes.

Data type	**Name**	**Description**
`int`	`authType`	authentication type
`string`	`authName`	authentication type name
`string`	`authPassword`	password
`string`	`authOther`	another type of authentication

Table 4.5: AE authentication parameters

4.4.2 AE API

The API of the AE is divided into three separate parts depending on from which module the methods are issued. All of them are composed of callback methods, which are redefined by AE programmer.

The first group represents reactions on API calls from an AP Instance. The `APIReqObj` object contains all parameters needed for each API call and the type of the issued call.

Method
`onA_read(APIReqObj* obj)`
`onA_write(APIReqObj* obj)`
`onA_create(APIReqObj* obj)`
`onA_delete(APIReqObj* obj)`
`onA_start(APIReqObj* obj)`
`onA_stop(APIReqObj* obj)`
`onA_cancelread(APIReqObj* obj)`

Table 4.6: AE callbacks handling reactions from AP API

The second group is a collection of callback functions that are issued in case that message from `commonApplicationDistributionProtocol` is passed to this AE Instance. The input parameter here is `CDAPMessage` object.

Method
`processMRead(CDAPMessage* msg)`
`processMReadR(CDAPMessage* msg)`
`processMWrite(CDAPMessage* msg)`
`processMCreate(CDAPMessage* msg)`
`processMCreateR(CDAPMessage* msg)`
`processMDelete(CDAPMessage* msg)`
`processMDeleteR(CDAPMessage* msg)`
`processMCancelread(CDAPMessage* msg)`
`processMCancelreadR(CDAPMessage* msg)`
`processMStart(CDAPMessage* msg)`
`processMStartR(CDAPMessage* msg)`
`processMStop(CDAPMessage* msg)`
`processMStopR(CDAPMessage* msg)`

Table 4.7: AE CDAP message processing methods

The last group represents only one method `RIBSendData` that passes an object from RIB Daemon to CDAP module directly.

4.5 RIB

The RIB is simple module provided by every AP. It provides an interface implemented in `RIBBase` class, and the core functionality is extended in `DAFRIB` class. The `RIB` class contains RIB tree-structured database, and it implements operations for manipulation with such database.

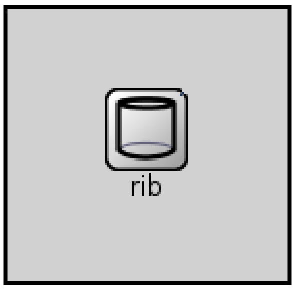


Figure 4.4: RIB module structure

The only module that can do operations in RIB is RIB Daemon. This restriction leads us to use a friend class to RIB Daemon class and declares all methods as private.

4.5.1 RIB Structure

Resource Information Base serves as a database of application objects in the AP. RIB in each AP represents only one portion of the entire replicated database of all objects in DAF. It implements all necessary data operations.

The objects are addressed by their names. Each name has special a format, by which it should be addressed on a different level in the hierarchy of objects. The format is following:

```
/DAF/[LABEL|AE_NAME]/[LABEL|IAE_NAME]/ObjName
```

Each object is subject to access rights related to a level in the hierarchy it is stored on. For example, the object is stored under a concrete AE Instance, then only two AE Instances that has the connection between them can access this object.

Tree Structure

Sorted binary search tree (BST) with reverse pointers was chosen to design the RIB database of objects and hierarchy tree. The hierarchy tree is a structure of multiple trees that model different levels in the database, as it is depicted in Figure 4.5.

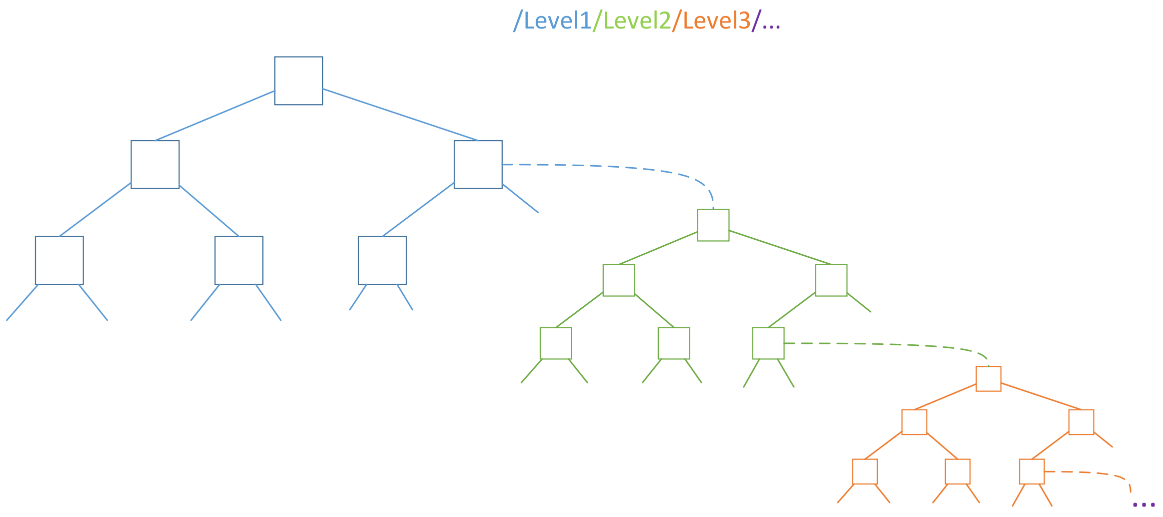


Figure 4.5: The hierarchy tree structure

The whole database is composed of two different trees. Both trees are BST but with different nodes and purpose. The purpose of a tree hierarchy is to create a structure with different levels of plumbing, depending on which level the concrete object belongs to.

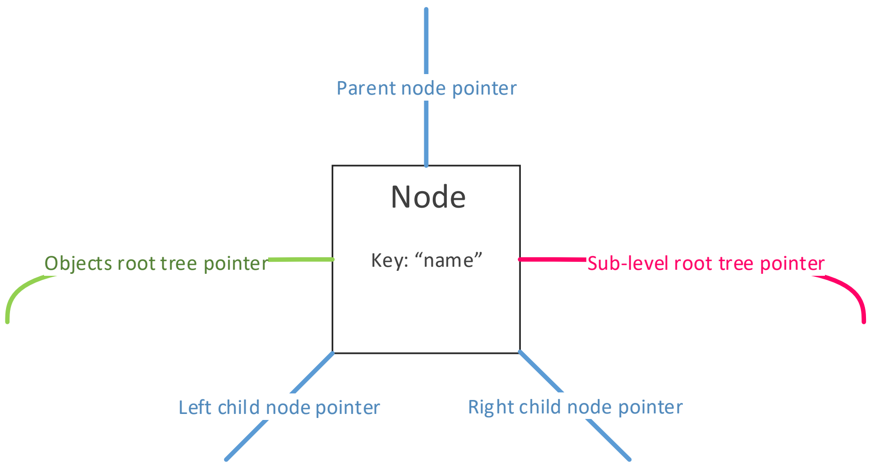


Figure 4.6: The hierarchy tree node

Each node of this tree has five pointers, as it is captured in Figure 4.6. First three pointers are standard pointers of BST (parent node and two child nodes). Then there is a pointer to a sub-level tree, which allows us to create more levels, like one AE with AE Instances – all AE Instances then have their nodes in this sub-level tree. The last item is a pointer to an object tree. The name of the label in current level is used as a key for this node.

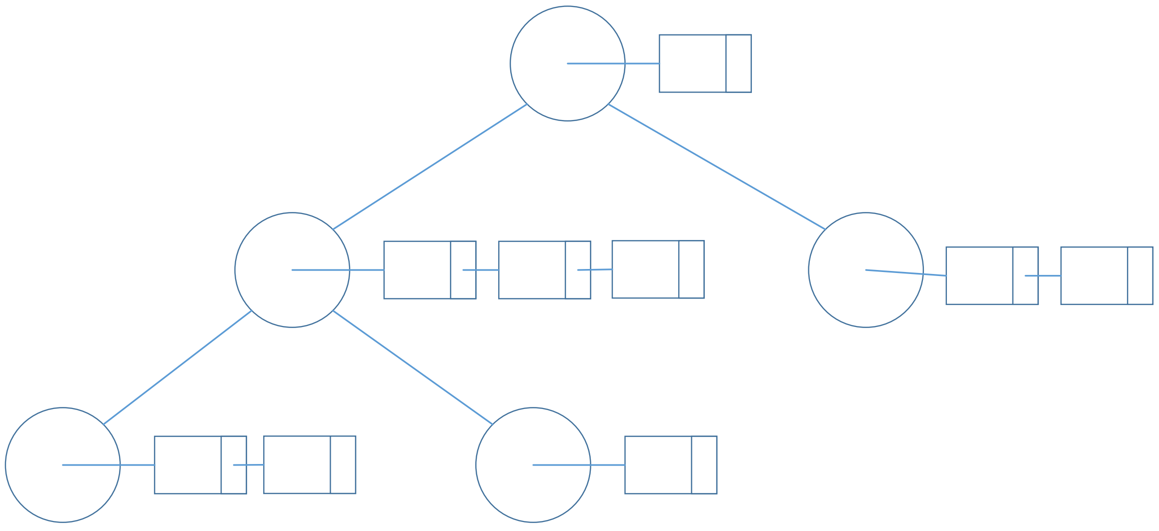


Figure 4.7: The Object tree

The object tree is the second different BST, shown in Figure 4.7. Its purpose is to store all objects that belong under some label in the hierarchy tree. The tree is sorted by the object name. Each node contains a list of object values and the name as a key. The list of objects contains different versions of instances of an object. The key for each node is the name of an object.

4.5.2 Objects

Application objects are stored in the separate object tree, it is implemented by `RIBTreeNodeObj`. This class contains a list of objects, the object is represented by structure `object_t`.

```
struct object_t {
    string objectClass
    string objectName
    int objectInstance
    ObjectPtr objectVal
}
```

4.5.3 RIB API

RIB API supports two types of operations. The first group manipulates with the tree structure and the tree levels. This group supports only operations for creating AE and AE Instance levels in the tree. All these operations are described in Table 4.8.

Method	**Description**
`createAE(string AEName)`	creates tree node at AE level
`createIAE(string IAEName, AEBase* ae)`	creates tree node at AE Instance level and assigns a pointer of the AE Instance to it
`deleteAE(string AEName)`	deletes tree node with AE, also destroys all subtrees
`deleteIAE(string IAEName)`	deletes tree node with AE Instance, also destroys all subtrees

Table 4.8: RIB API for manipulation with tree structure

The second group of methods covers operations with objects, namely read/write/create/delete. These methods are described in Table 4.9.

Method	Description
`createObj(string objName, object_t *obj)`	creates object according to name an assigns the value of it
`deleteObj(string objName)`	deletes object in the object tree
`writeObj(string objName, object_t *obj)`	writes object value to object matched by name
`readObj(string objName)`	reads object value of object matched by name

Table 4.9: RIB API for manipulation with data objects

4.6 RIB Daemon

The RIB Daemon is a simple module that stands on the border between AEs and RIB. It manages all objects stored in the RIB. The core functionality is implemented in `DAFRIBd` class, which extends the interface `DAFRIBdBase` class. Application entities can use the RIB Daemon API to create, read, or delete subscriptions for objects in the RIB. Three types of subscriptions are supported – on an object change, periodically, or once on request. Each of the subscriptions is stored in one of the two lists, and it is checked based on its type.

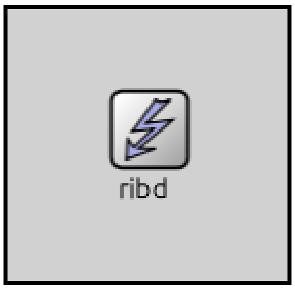


Figure 4.8: Rib Daemon module structure

The RIB Daemon generates messages and uses appropriate AE Instances for sending them to destinations. On the other side, RIB Daemon receives messages from AE Instances or reacts on AE Instance requests.

4.6.1 RIB Daemon API

RIB Daemon provides the following API. The primary user of this API are AE Instances. The API is divided into two parts. The first one covers operations with subscriptions, which are described in Table 4.10.

The second part contains only supporting operations – creation and deletion of AE Instance record in RIB structure. The description can be seen in Table 4.11.

Method	Description
`createSubscription( SubscriptionOption option, SubscriptionWhen when, SubscriptionOperation operation, string obj, string member, int subscId)`	creates a subscription for an object in RIB based on input parameters - `option` - `READ|WRITE` whether it is read or write subscription - `when` - `ON_REQUEST|ON_CHANGE` whether the subscription is on request, or on object change - `operation` - `NOTIFY|STORE` whether the action is only recorded in RIB, or sent back to requester - `obj` - object name - `member` - member to which the action belongs - `subscId` - subscription id
`deleteSubscription(int subscId)`	deletes subscription based on subscriber id as an input parameter
`readSubscription(int subscId)`	reads subscription based on subscriber id

Table 4.10: RIB Daemons subscription API

Method	Description
`createIAE(AEBase* iae)`	creates IAE entry in RIB tree structure on the appropriate level
`deleteIAE(AEBase* iae)`	deletes IAE entry in RIB

Table 4.11: RIB Daemons addition methods

4.7 Enrollment

The enrollment module encapsulates `enrollment` and `enrollmentStateTable`. The purpose of `enrollment` is to manage two phases of connections, i.e., generates and reply on received messages according to finite state machine states.

Both state machines are implemented in separate classes. The `CACEMgmt` class implements CACE phase, and the Enrollment phase is implemented in `DAFEnrollment` class, which contains the core functionality of the `enrollment` module. It implements both initiator and responder roles in one module.

This module spawns Management AEs and uses them for communication. The module catches all relevant CDAP messages to both phases as an input and reacts to them according to recorded states.

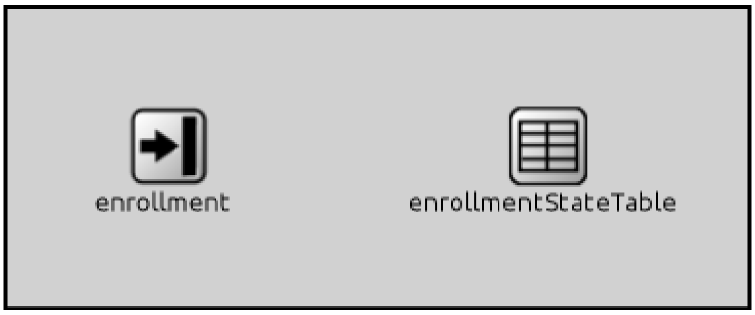
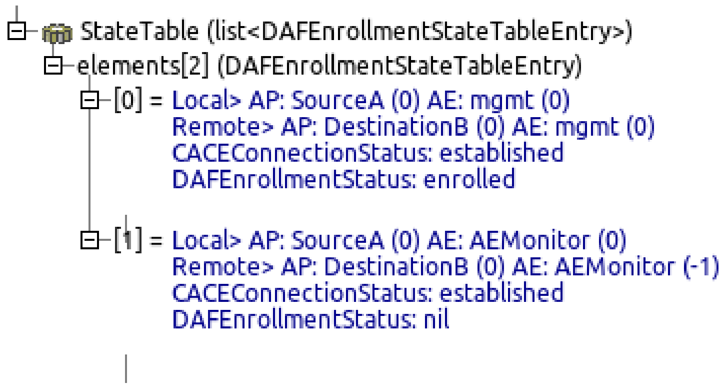


Figure 4.9: Enrollment module structure

The Enrollment has to maintain a state of all application connections. This is ensured by class `EnrollmentStateTable` that implements a storage of entries. This state information is stored separately from RIB because it should be managed only by the enrollment module. The visualization of this data in the simulator is also more transparent to users, as it is depicted in Figure 4.10.



```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[2] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled

    [1] = Local> AP: SourceA (0) AE: AEMonitor (0)
          Remote> AP: DestinationB (0) AE: AEMonitor (-1)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: nil
```

Figure 4.10: Enrollment state table entries

4.7.1 Initial Phases of Communication

The initial phases of the connection are CACE and Enrollment. At first, when the user tries to create an application connection to another AP, the *apInst* module sends the signal to the enrollment module, to check whether the AP is already enrolled to this DAF. If not, the first AE Management is spawned, which tries to create an application connection to the destination AP.

The CACE takes place over this new connection. This phase is managed by `CACEMgmt` class that implements a finite state machine described in section 2.5.1. This class is also part of the enrollment module.

The Enrollment phase starts right after the connection is successfully established. The entire phase is managed by `DAFEnrollment` class, which implements the Enrollment finite state machines mentioned in section 2.6.

After successful enrollment, the Enrollment module sends a notification signal back to *apInst*, that AP is enrolled to the DAF and other connections can be created. The `apInst` then spawns an appropriate AE Instance. According to the parameters, the AE Instance allocates connection and also goes through the CACE phase. This CACE is handled by

CACEGeneric class, which is similar to CACEMgmt class and also implements the same finite state machines.

Both connections are stored in EnrollmentStateTable as it is depicted in Figure 4.10. The difference is that DAFEnrollmentStatus is not set in normal connections.

4.8 Management AE

Management AE is a special AE designed to manage and communicate through management connections, hence the special attention is devoted to this module. Only the Enrollment and RIB Daemon module can use this AE, and it is completely independent. It has almost the same structure as normal AE as it is shown in Figure 4.11.

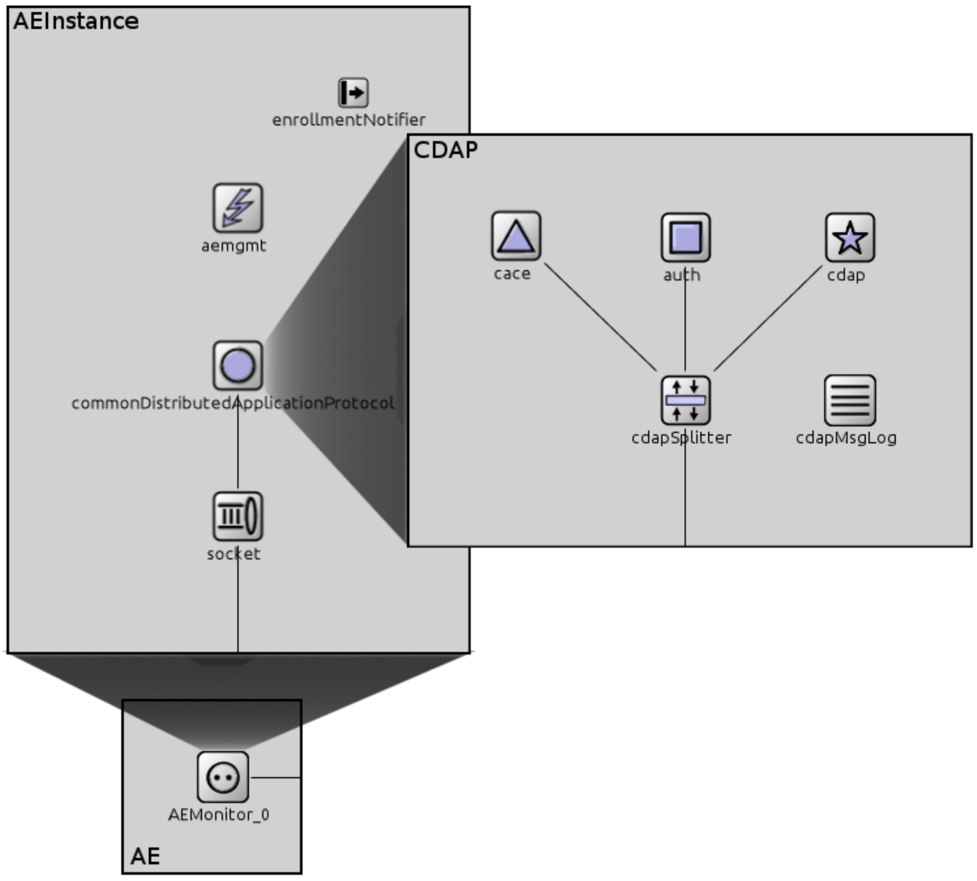


Figure 4.11: Management Application Entity module structure

The parameters are the same as in the case of normal AE Instance. The difference is only in values that are assigned statically, such as QoS of connection.

In addition, there is one special submodule called enrollmentNotifier. This component is implemented to forward signals between Enrollment module and aemgmt module. This enables more clear implementation of aemgmt component because it works with only two signals from enrollmentNotifier.

Chapter 5

Testing and Evaluation

The testing scenario is focused on testing the whole implemented model. The first part is devoted to testing finite state machines and transitioning to appropriate states both for CACE and Enrollment. The second part tests simple ping scenario with reading data from remote AP using RIB Daemon. The topology is very simple and contains only two hosts and two APs communicating with each other. The whole scenario is depicted in Figure 5.1.

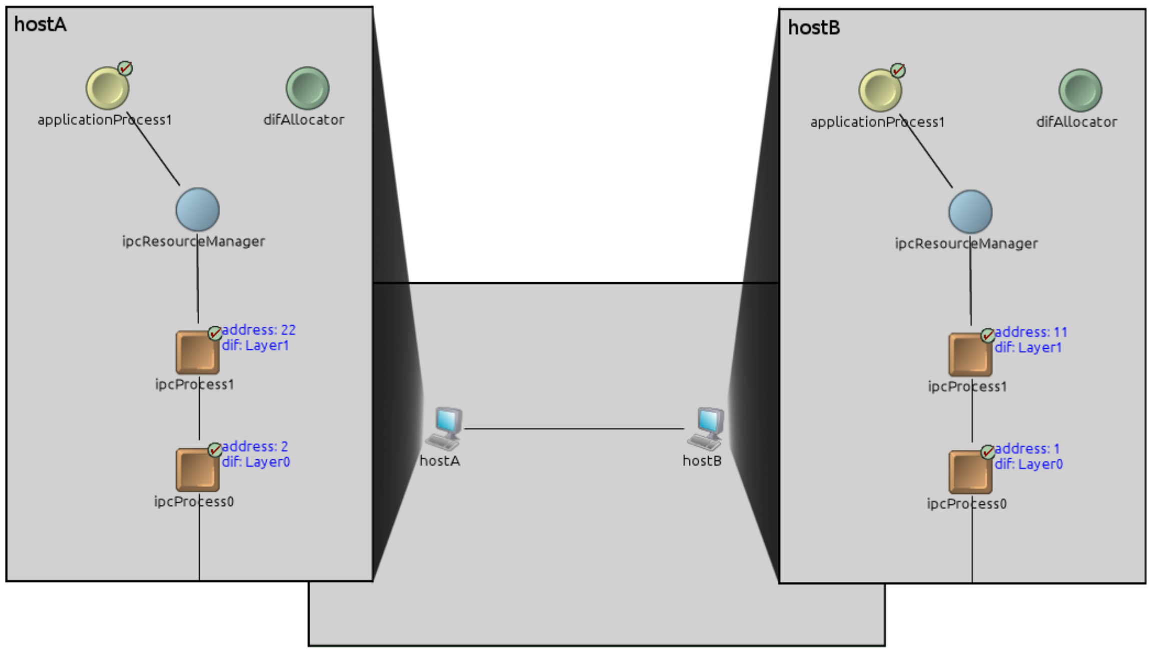


Figure 5.1: Two hosts and two AP topology

The AP `SourceA` is supposed to create one application connection with destination AP `SourceB`. This is enough to test both Enrollment and CACE, one over management connection and normal application connection. This process also comprises the dynamic creation of AE modules and appropriate cooperation of the modules in AP. Appropriate sent messages and final states of successful CACE and Enrollment are depicted in accompanying Figure 5.2.

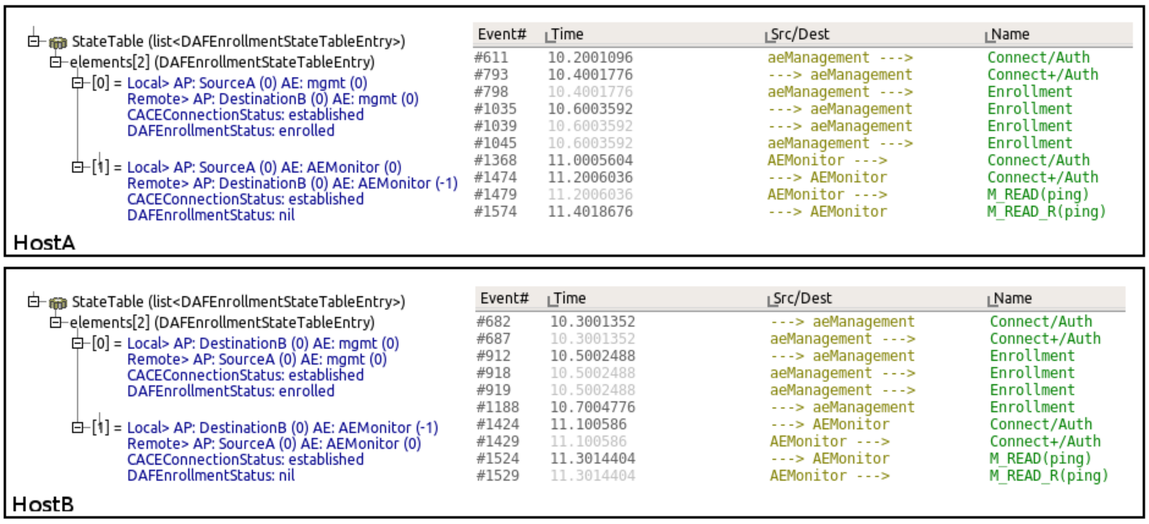
The process follows:

1. The simulation starts with an API request for a new connection. This results in a check if the AP is enrolled to DAF. If not, the `Enrollment` module spawns an `aeManagement` instance, creates an entry in an `enrollmentStateTable` and `aeManagement` instance allocates a new management connection.

2. After a successfully allocated connection, the CACE state machine takes the initiative and sends the first CDAP `M_CONNECT` message. This message carries an authentication that is set by the configuration. The message is passed through the connection to the second host. If the authentication was successful, a positive `M_CONNECT_R` message (indicated by `Connect+/Auth` in Figure 5.2) is sent back as a response. With that, the CACE phase is complete, and Enrollment phase may proceed.

3. When CACE phase is finished, the initiative is taken by Enrollment state machine. This includes an exchange of four CDAP messages `M_START(Enrollment)`, `M_STOP(Enrollment)`, `M_STOP_R(Enrollment)` and `M_START(Operation)`. Neither the initiator nor the responder needs to read or create any objects. Hence there are no more exchanges, and the Enrollment procedure is short.

4. After the last message received, the Enrollment notifies that the AP is successfully enrolled and the other connections within DAF may be created. The `AEMonitor` is spawned and a new application connection with desired parameters is allocated. Then an entry in the `enrollmentStateTable` is created, and CACE state machine manages the CACE phase. The exchange of messages is the same as in the case of previous management connection. After a successfully established connection, the `M_READ` messages is sent.



```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[2] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled

    [1] = Local> AP: SourceA (0) AE: AEMonitor (0)
          Remote> AP: DestinationB (0) AE: AEMonitor (-1)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: nil
HostA
```

Event#	Time	Src/Dest	Name
#611	10.2001096	aeManagement --->	Connect/Auth
#793	10.4001776	---> aeManagement	Connect+/Auth
#798	10.4001776	aeManagement --->	Enrollment
#1035	10.6003592	---> aeManagement	Enrollment
#1039	10.6003592	---> aeManagement	Enrollment
#1045	10.6003592	aeManagement --->	Enrollment
#1368	11.0005604	AEMonitor --->	Connect/Auth
#1474	11.2006036	---> AEMonitor	Connect+/Auth
#1479	11.2006036	AEMonitor --->	M_READ(ping)
#1574	11.4018676	---> AEMonitor	M_READ_R(ping)

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[2] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: DestinationB (0) AE: mgmt (0)
          Remote> AP: SourceA (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled

    [1] = Local> AP: DestinationB (0) AE: AEMonitor (-1)
          Remote> AP: SourceA (0) AE: AEMonitor (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: nil
HostB
```

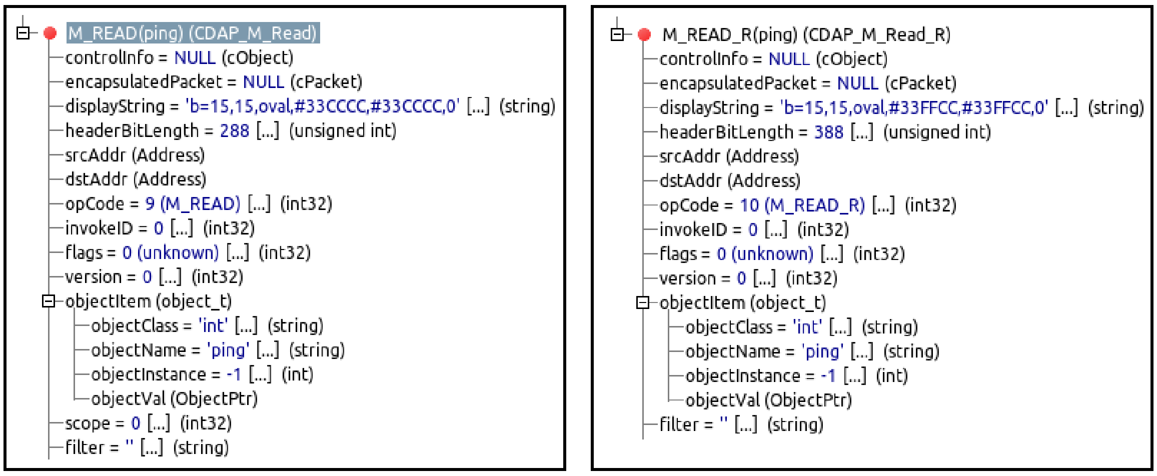
Event#	Time	Src/Dest	Name
#682	10.3001352	---> aeManagement	Connect/Auth
#687	10.3001352	aeManagement --->	Connect+/Auth
#912	10.5002488	---> aeManagement	Enrollment
#918	10.5002488	aeManagement --->	Enrollment
#919	10.5002488	aeManagement --->	Enrollment
#1188	10.7004776	---> aeManagement	Enrollment
#1424	11.100586	---> AEMonitor	Connect/Auth
#1429	11.100586	AEMonitor --->	Connect+/Auth
#1524	11.3014404	---> AEMonitor	M_READ(ping)
#1529	11.3014404	AEMonitor --->	M_READ_R(ping)

Figure 5.2: Resulting EnrollmentStateTable and message transitions log

The extract of `enrollmentStateTable` entries shows the resulting states on the left side in Figure 5.2. State transitioning is depicted in Figure A.1.

Right after Enrollment and CACE over allocated application flow between two `AEMonitor` instances passed, the reading of data may proceed.

1. The AP issues `A_READ` method call. The AP forwards the request to `AEMonitor` instance, which creates subscription request to RIB Daemon with option `ON_REQUEST` and `NOTIFY` set. The RIB Daemon, based on subscription, sends `M_READ` message using `AEMonitor` instance to the remote AP.
2. The `AEMonitor` instance in the remote AP forwards the message to RIB Daemon, it gets required object from RIB and send it in `M_READ_R` message using local `AEMonitor` instance back to the requestor.
3. The requesting `AEMonitor` instance receives this CDAP `M_READ_R` message, forward it to RIB Daemon and forward the resulting object back to AP.



```
M_READ(ping) (CDAP_M_Read)
 ├controlInfo = NULL (cObject)
 ├encapsulatedPacket = NULL (cPacket)
 ├displayString = 'b=15,15,oval,#33CCCC,#33CCCC,0' [...] (string)
 ├headerBitLength = 288 [...] (unsigned int)
 ├srcAddr (Address)
 ├dstAddr (Address)
 ├opCode = 9 (M_READ) [...] (int32)
 ├invokeID = 0 [...] (int32)
 ├flags = 0 (unknown) [...] (int32)
 ├version = 0 [...] (int32)
 ├objectItem (object_t)
 │ ├objectClass = 'int' [...] (string)
 │ ├objectName = 'ping' [...] (string)
 │ ├objectInstance = -1 [...] (int)
 │ └objectVal (ObjectPtr)
 ├scope = 0 [...] (int32)
 └filter = '' [...] (string)
```

```
M_READ_R(ping) (CDAP_M_Read_R)
 ├controlInfo = NULL (cObject)
 ├encapsulatedPacket = NULL (cPacket)
 ├displayString = 'b=15,15,oval,#33FFCC,#33FFCC,0' [...] (string)
 ├headerBitLength = 388 [...] (unsigned int)
 ├srcAddr (Address)
 ├dstAddr (Address)
 ├opCode = 10 (M_READ_R) [...] (int32)
 ├invokeID = 0 [...] (int32)
 ├flags = 0 (unknown) [...] (int32)
 ├version = 0 [...] (int32)
 ├objectItem (object_t)
 │ ├objectClass = 'int' [...] (string)
 │ ├objectName = 'ping' [...] (string)
 │ ├objectInstance = -1 [...] (int)
 │ └objectVal (ObjectPtr)
 └filter = '' [...] (string)
```

Figure 5.3: Resulting EnrollmentStateTable and message transitions log

Figure 5.3 depicts the CDAP `M_READ` and `M_READ_R` containing application object. In the first message, it is specified that what object to receive. The second message contains also object value.

All parts of the model was validated and experimentally tested multiple times. The modules reacts as it was described in design.

5.1 Discussion

The development of RINASim was primarily focused on IPC Process and its function. The AP contained only one AE and nothing more. The AE served as a traffic generator to test underlying DIFs. There was no API to use. The CDAP messages were passed directly to the underlying IPC process. There were no AP Instances and the AE was static. The old state is depicted in Figure 5.4 on the left.

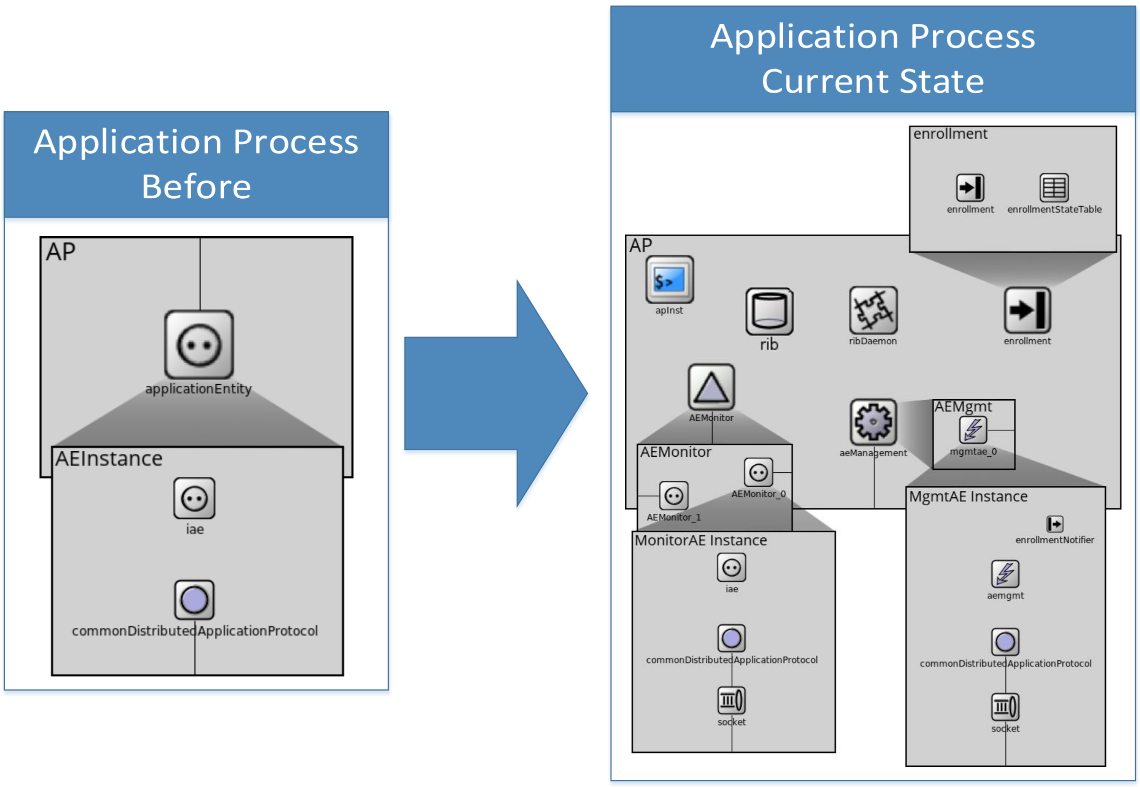


Figure 5.4: Progress between old and current AP

The work of this thesis aimed at extending AP of all its components so that the user of RINASim could expect the AP behaviour as it is described in the specifications. The current implementation can be seen in Figure 5.4 on the right.

It allows the user to experiment with the full-fledged model of AP that can finally participate in DAF and create distributed applications. The AP was extended with phases of communication, such as CACE and Enrollment. The programming of RINASim applications has moved to a higher level, the AP Instance became the highest unit that can be programmed. The AP Instance has its API that the users can use. The AEs are created dynamically based on AP Instance API calls.

The AE is created as an interface module, providing several APIs for communication with different modules. This brings new possibilities for experimenting with the model. Each user can create its AE in case it does not fit our understanding of the AP behaviour.

The model was also extended with RIB that serves as a partially replicated database. There was added RIB Daemon as component taking care of management and distribution of application objects, and it also provides an API available for communication with other modules.

Chapter 6

Conclusion

The thesis aimed to understand each component participating in DAF. The main goal of the work was to design the components according to specifications, so they provide an easily usable API and fulfill the desired purpose. All the outlined goals were accomplished.

The RINASim contained only a single Application Entity that served as a traffic generator between two static nodes. This work brings a full-featured Application Process capable of creating multiple connections, dynamic creation of modules, and primarily dealing with communication on a data object level. The emphasis is on the design of a new approach called RIB based programming model and distribution of data between nodes.

The significant part of the work is design and implementation of an API that provides easier and uniform creation of applications in RINASim. It also helps with the better understanding of how to create distributed applications in RINA. The RINASim was extended with the implementation of models of CACE, Enrollment, Application Process, Application Entity, Management Application Entity, RIB Daemon and Resource Information Base. The implementation was tested and validated according to the specifications.

The implementation is continuously added to new releases of the official version of RINASim. RINASim is used worldwide by many researchers to help understand how RINA works and to experiment with its model. The simulator is marked as one of the official OMNeT++ frameworks.

This work was supported by the Brno University of Technology organization and by the research grant PRISTINE EU-7FP-ICT.

Part of this thesis was accepted to the student's conference Excel@FIT in years 2016 and 2017. The work has been awarded in two categories, Expert panel award, and the Professional community award, at the conference in 2016.

The work brought the opportunity to participate in continual research of a completely new network architecture. The continual research required discussion with other researchers, even with the author of the RINA architecture.

The discussions about parts of the architecture are still evolving. The future work will be focused on extending the model in RINASim during my Ph.D. studies. Also, several different AEs have to be created to represent different approaches in the design of communication. The measurement of the efficiency of distribution of objects between APs should also create a significant role in future work.

Bibliography

[1] OMNeT++ Community Site. http://www.omnetpp.org. accessed: 2016-1-9.

[2] Brno University of Technology: kvetak/RINA - GitHub. https://github.com/kvetak/RINA. 2016.

[3] Bunch, S.: CDAP - Common Distributed Application Protocol Reference. December 2010.

[4] Bunch, S.; Day, J.; Trouva, E.: Patterns in Network Architecture - Recursive IPC Network Architecture - Common Application Connection Establishment Phase (CACEP). 2012.

[5] Day, J.: *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall. 2008. iSBN-13: 978-0-132-25242-3.

[6] Day, J.: OIB/RIB Daemon. 2010.

[7] Day, J.: Patterns in Network Architecture - Recursive IPC Network Architecture - Basic Enrollment Specification. 2012.

[8] Day, J.: Recursive IPC Network Architecture - The Interina Reference Model - Part 3: Distributed InterProcess Communication - Chapter 2: DIF Operations. 2012.

[9] Day, J.: Patterns in Network Architecture - Recursive IPC Network Architecture - The interina Reference Model - Part 1: Basic Concepts of Distributed Systems. 2016.

[10] Day, J.: Recursive IPC Network Architecture - The Interina Reference Model - Part 2: Distributed Applications - Chapter 1: Basic Concepts of Distributed Applications. 2016.

[11] Day, J.: Recursive IPC Network Architecture - The Interina Reference Model - Part 3: Distributed InterProcess Communication - Chapter 1: Fundamental Structure. 2016.

[12] Day, J.; Matta, I.; Mattar, K.: ReArch'08 - Re-Architecting the Internet. December 2008.

[13] Day, J.; Trouva, E.: Recursive IPC Network Architecture - The Interina Reference Model - Part 2: Distributed Applications - Chapter 2: Introduction to Distributed Management Systems. 2016.

[14] Jeřábek, K.: *RECURSIVE IPC NETWORK ARCHITECTURE: ANALYSIS AND MODELLING OF ENROLLMENT*. Bachelor thesis. Brno University of Technology, Faculty of Information Technology. 2014.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=17386

[15] PRISTINE consortium: PRISTINE will take a major step forward in the integration of networking and distributed computing. http://ict-pristine.eu. 2016.

[16] Trouva, E.; Grasa, E.; Day, J.; et al.: Is the Internet an unfinished demo? Meet RINA! 2011.

[17] Veselý, V.: *A NEW DAWN OF NAMING, ADDRESSING AND ROUTING ON THE INTERNET*. PhD. Thesis. Brno University of Technology, Faculty of Information Technology. 2016.
Retrieved from: http://www.fit.vutbr.cz/study/DP/PD.php?id=515

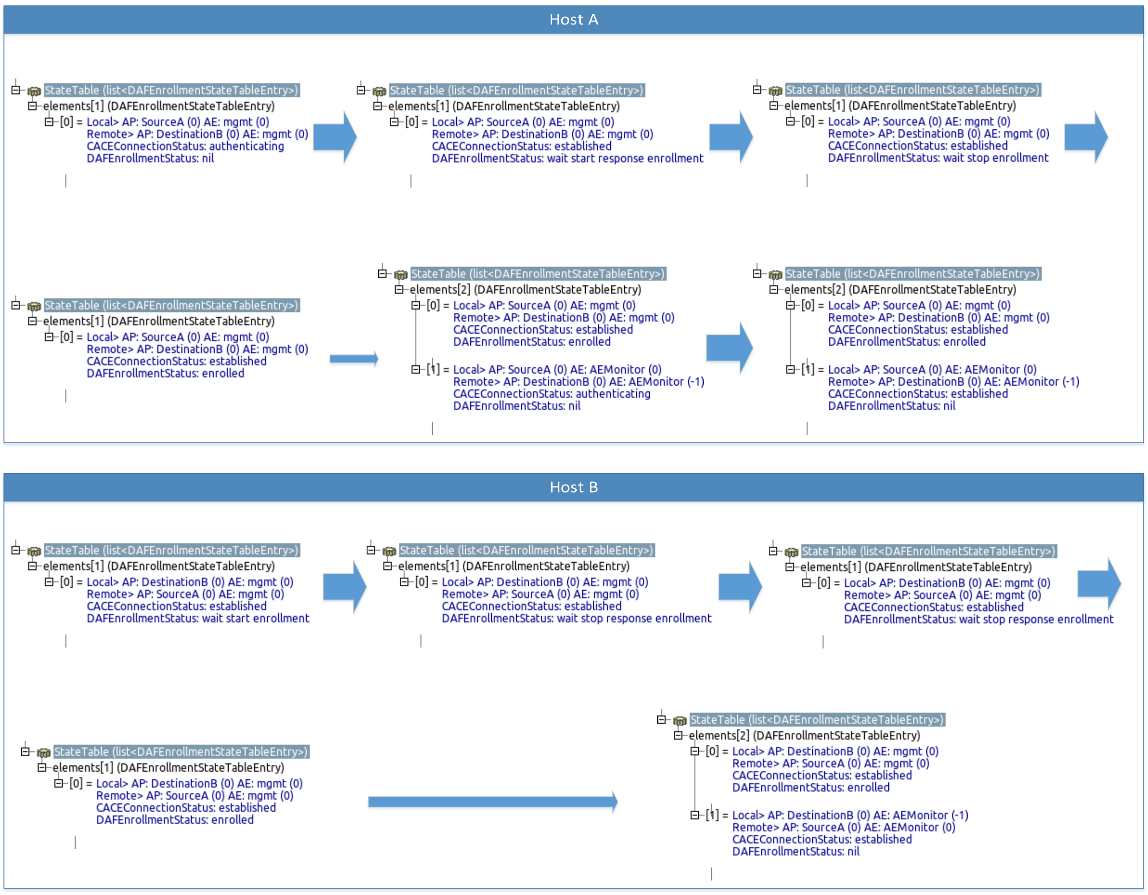
[18] Veselý, V.; Marek, M.; Hykel, T.; et al.: RINASim: Your Recursive InterNetwork Architecture Simulator. September 2015.

[19] Veselý, V.; Marek, M.; Jeřábek, K.; et al.: Deliverable 2.6: RINASim - advanced functionality. http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d26-rina_sim-draft.pdf. 2015.

Appendices

Appendix A

Enrollment Progress



```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: authenticating
          DAFEnrollmentStatus: nil
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: wait start response enrollment
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: wait stop enrollment
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[2] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled
    [1] = Local> AP: SourceA (0) AE: AEMonitor (0)
          Remote> AP: DestinationB (0) AE: AEMonitor (-1)
          CACEConnectionStatus: authenticating
          DAFEnrollmentStatus: nil
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[2] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: SourceA (0) AE: mgmt (0)
          Remote> AP: DestinationB (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled
    [1] = Local> AP: SourceA (0) AE: AEMonitor (0)
          Remote> AP: DestinationB (0) AE: AEMonitor (-1)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: nil
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: DestinationB (0) AE: mgmt (0)
          Remote> AP: SourceA (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: wait start enrollment
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: DestinationB (0) AE: mgmt (0)
          Remote> AP: SourceA (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: wait stop response enrollment
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: DestinationB (0) AE: mgmt (0)
          Remote> AP: SourceA (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: wait stop response enrollment
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[1] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: DestinationB (0) AE: mgmt (0)
          Remote> AP: SourceA (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled
```

```
StateTable (list<DAFEnrollmentStateTableEntry>)
  elements[2] (DAFEnrollmentStateTableEntry)
    [0] = Local> AP: DestinationB (0) AE: mgmt (0)
          Remote> AP: SourceA (0) AE: mgmt (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: enrolled
    [1] = Local> AP: DestinationB (0) AE: AEMonitor (-1)
          Remote> AP: SourceA (0) AE: AEMonitor (0)
          CACEConnectionStatus: established
          DAFEnrollmentStatus: nil
```

Figure A.1: EnrollmentStateTable Contents Progress