Pristine

Deliverable-5.1

Draft specification of common elements of the management framework

Deliverable Editor: Sven van der Meer,LMI

Publication date:	28-June-2014
Deliverable Nature:	Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	Programmability In RINA for European supremacy of virTualised NEtworks
Website:	www.ict-pristine.eu
Keywords:	DIF, management, system, RIB, elements, common
Synopsis:	This deliverable describes the Common Elements for the DIF management system (DMS). It outlines the distributed design principles used to create the DMS, and structure the corresponding Resource Information Base (RIB). It also specifies the resulting initial architecture for the common elements of the DMS.

The research leading to these results has received funding from the European Community's Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 619305.

Executive Summary

Recursive Inter-Network Architecture (RINA) is an emerging clean-slate programmable networking approach, centring on the Inter-Process Communication (IPC) paradigm, which will support high scalability, multi-homing, built-in security, seamless access to real-time information and operation in dynamic environments. The heart of this networking structure is naturally formed and organised by blocks of containers called “Distributed Information Facilities (DIFs)” where each block has programmable functions to be attributed to as required. The term "Distributed Application Facility" (DAF) refers to application processes that use a Distributed Information Facility (DIF) to exchange information.

Network Management (NM) refers to the activities, methods, procedures, and tools that pertain to the operation, administration, maintenance, and provisioning of networked systems [clem2006]. In RINA, network management is preformed by a DIF Management System (DMS). The common elements of such a system are the focus of this document. The rest of the document is structured as follows.

A summary of RINA’s approach to network management is provided. A RINA DIF Management System (DMS) follows a manager-agent model in its network management. This allows some flexibility in the management interactions between managers, agents, and inter-agent (agent to sub-agent).

In RINA, two protocols are used to assist the DIF Management System (DMS) in its operation:

- **Common Distributed Application Protocol (CDAP)** enables distributed applications to deal with communications at an object level, rather than forcing applications to explicitly deal with serialization and input/output operations.
- **Common Application Connection Establishment Phase (CACEP)** allows Application Processes to establish an application connection.

A review of the current state-of-the-art (SoTA) is presented, representing a summary of the work done by leading standards organisations in the area of network management systems. The SoTA covers the languages used to

capture management information, and includes work by the International Standards Organisation (ISO) and International Telecommunications Union (ITU) in its X.7xx series, Tele-Management Forum's (TM-Forum) - Information Framework (called SID) and Internet Engineering Task Force's (IETF) data modelling language - YANG. The IETF's earlier work on Simple Network Management Protocol (SNMP), and corresponding data model Structure Management Information (SMI) are excluded as they are somewhat superseded by the NETCONF protocol and YANG data model work. The Distributed Management Task Force's (DMTF) Common Information Model (CIM) model is also excluded as this work was further advanced within the TM-Forum.

This is followed in Part II by a high level description of the design decisions made in the design of the RINA Managed Object (MO) model. A **single inheritance** model is used to define the Managed Object (MO) model. **Notifications** are modelled as explicit attribute reads and writes, to defined attributes of type Event. A MO **containment** model is defined to capture runtime relationships

A set of utility classes and corresponding inheritance tree are presented for the RINA managed object model. This captures the static and typing information available in the DIF Management System (DMS). Dynamic relationships (runtime relationships) are captured in the containment model. A containment model is presented for both the DIF and DAF.

The next two sections, discuss and describe the impact of the design decisions on both the DMS Manager, and the Management Agent. In both cases, a high-level architecture of the common components of the DMS are presented.

The final section covers future directions for the DMS. In particular, outlining future work on defining the tooling and supported concrete syntaxes for the Managed Objects when there are transmitted over the wire. It is also expected that some work is needed in providing transaction support (for management operations) to RINA's protocol set.

Table of Contents

1. Introduction

The Internet as the global communications infrastructure has been successful in shaping the modern world by the way we access and exchange information. The Internet architecture, designed in the 1960's, has been supporting a variety of applications and offering a number of services till now, but emerging applications demand better quality, programmability, resilience and protection. Any alterations to the Internet architecture have become restricted to simple incremental updates and plug-ins instead of radical changes by introducing new solutions. The Internet as the global communications infrastructure has been successful in shaping the modern world by the way we access and exchange information.

The Recursive Inter-Network Architecture (RINA) is an emerging clean-slate programmable networking approach, centring on the Inter-Process Communication (IPC) paradigm, which will support high scalability, multi-homing, built-in security, seamless access to real-time information and operation in dynamic environments. The heart of this networking structure is naturally formed and organised by blocks of containers called *"Distributed Information Facilities - DIFs"* where each block has programmable functions to be attributed to as required. A DIF is seen as an organising structure, grouping together application processes that provide IPC services and are configured under the same policies [DIF].

1.1. Scope

Based on the above concepts, this deliverable focuses on the common architecture and design principles of multi-layer management for handling configuration, performance and security aspects within RINA.

This deliverable can be considered in two parts:

Part I (RINA theory and specifications)
The first part covers the various parts of RINA theory or suggested practice, as proposed by the RINA approach. Influences from other standards bodies are discussed in the state-of-the-art review.

Part II (PRISTINE implementation)
These sections cover the PRISTINE implementation, managed object model and general design principles.

The final section outlines the future direction of this work and the potential impact on clarifying some implementation aspects.

2. Network Management in RINA

This section describes the view of network management as proposed by the Recursive Inter-Network Architecture (RINA). RINA takes some inspiration from previous OSI and IETF network management work, and follows a similar manager/agent paradigm in defining the roles of the various management components. This section begins with some definitions of terms used in defining RINA management concepts.

2.1. Definitions

In order to aid comprehension of the subsequent sections, the following definitions are reproduced here for convenience.

- **Application Process, AP**. The instantiation of a programme executing in a processing system intended to accomplish some purpose. An Application Process contains one or more tasks or Application-Entities, as well as functions for managing the resources (processor, storage, and IPC) allocated to this AP. Tasks are also Application Processes.
- **Common Application Connection Establishment Phase, CACEP**. CACEP allows two Application Processes to establish an application connection. During the application connection establishment phase, the APs exchange naming information, optionally authenticate each other, and agree in the abstract and concrete syntaxes of CDAP to be used in the connection, as well as in the version of the RIB. It is also possible to use CACEP connection establishment with another protocol in the data transfer phase (for example, HTTP).
- **Common Distributed Application Protocol, CDAP**. CDAP enables distributed applications to deal with communications at an object level, rather than forcing applications to explicitly deal with serialisation and input/output operations. CDAP provides the application protocol component of a Distributed Application Facility (DAF) that can be used to construct arbitrary distributed applications, of which the DIF is an example. CDAP provides a straightforward and unifying approach to sharing data over a network without having to create specialised protocols.
- **Distributed Application Facility (DAF)**. A collection of two or more cooperating APs in one or more processing systems, which exchange information using IPC and maintain shared state. In some Distributed

Applications, all members will be the same, i.e. a homogeneous DAF, or may be different, a heterogeneous DAF.

- **Distributed IPC Facility (DIF), Layer**. A collection of two or more Application Processes cooperating to provide Inter-Process Communication (IPC). A DIF is a DAF that does IPC. The DIF provides IPC services to Application Processes or IPC Processes of other DIFs via a set of API primitives that are used to exchange information with the application's peer.
- **IPC Process**. An application process whose primary purpose is managing IPC.
- **Processing system**. The hardware and software capable of executing programs instantiated as Application Processes that can coordinate with the equivalent of a "test and set" instruction, i.e. the tasks can all atomically reference the same memory.
- **Resource Information Base (RIB)**. The logical representation of information held by the IPC Process for the operation of the DIF.

2.2. Network Management - Distributed Management Systems (NM-DMS)s

A high-level overview of the RINA architecture is provided by Figure 1, "Graphical model of the RINA architecture". Each computing system (rectangle boxes) can run one or more IPC processes, implementing one or more DIFs in that system. IPC Processes in a system are managed by the Management Agent, who has read and write permissions to the IPC Processes' Resource Information Base (RIB)s. There is (at least) one Management Agent on each processing system (node).

DIF Managers on the other hand, perform network management related tasks within an administrative domain, so there is at least one DMS in an administrative domain. In some configurations one or more DIF Managers communicate with the agents in each system of its management domain to provide central configuration, fault, security and performance management.

The management agents and the DIF Managers together form a distributed application that manages elements of one or more DIFs, and is called Network Management - Distributed Management System (or NM-DMS in short).

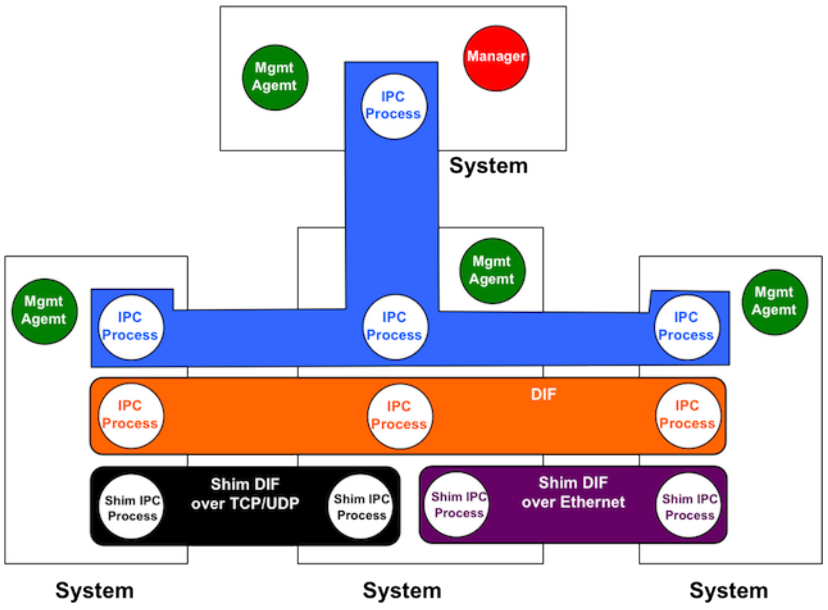


Figure 1. Graphical model of the RINA architecture

While the IPC-Processes that comprise the DIF are exchanging information on their operation and the conditions they observe, it is generally necessary to also have an outside window into the operation of DIFs comprising the network. While the members of a DIF may reach a local optimisation, it is often more complex to discover global optimisations without an “outside” perspective.

Therefore a distinction is made between control and management behaviour. In these systems, control must be automatic, as events are happening far too fast and state is changing too rapidly for a centralised system to be effective. Thus management restricts itself to deciding on which of the available policies should be applied and ensuring the appropriate policies are in place, in advance, as well as monitoring the effectiveness of those control policies and providing higher level insights into network operations.

Furthermore, the nature of distributed systems always opens the possibility for partitioning. Hence, it must be possible for distributed systems to fail-safe without central control. A NM-DMS will perform the traditional functions of network management [clem2006]. The DAF model can be applied to network management to represent the whole range from distributed (autonomic) to centralised (traditional).

In the traditional centralised network management architecture, an NM-DMS would be a heterogeneous DAF consisting of one or more DAPs providing management functions, with other DAPs providing telemetry. The management DAPs might be subdividing roles or tasks within network management or providing management for sub-domains and redundancy for each other. A typical DMS will have the usual tasks of event management, configuration management, fault management, resource management, performance management. This also has the advantage of shifting the focus away from boxes to a distributed system model.

The NM-DMS DAPs in the traditional agent role (see Figure 2, “Possible interactions between entities of the RINA management framework”) function like the sensory nervous system collecting and pre-processing data. The Agent will have access to the DAF Management task of all IPC Processes (and associated DAPs) in the processing systems that are in its domain. While there is no constraint, it is likely that an NM-DAF would have one “Agent DAP” for monitoring in each processing system with a DIF or DAF in its domain. The DAF Management task of each DAF or DIF in the NM-DMS domain is a kind of “sub-agent.” A Management Agent may be designed to seek out its DMS or alternate DMSs in the event of failures.

In order to interact with each system, the manager process needs to have a DIF in common with it. There are several ways of achieving this, ranging from using a single DIF dedicated to interconnect the manager with each Management Agent (as it is shown in Figure 2, “Possible interactions between entities of the RINA management framework”), to using different DIFs for different systems. Once the manager has allocated a flow to a Management Agent, it establishes an application connection to it via CACEP [CACEP], which includes optional authentication. Once the application connection is in place, the DMS and the management agent can communicate by performing remote operations on the RIBs of IPC Processes via CDAP - the Common Distributed Application Protocol [CDAP].

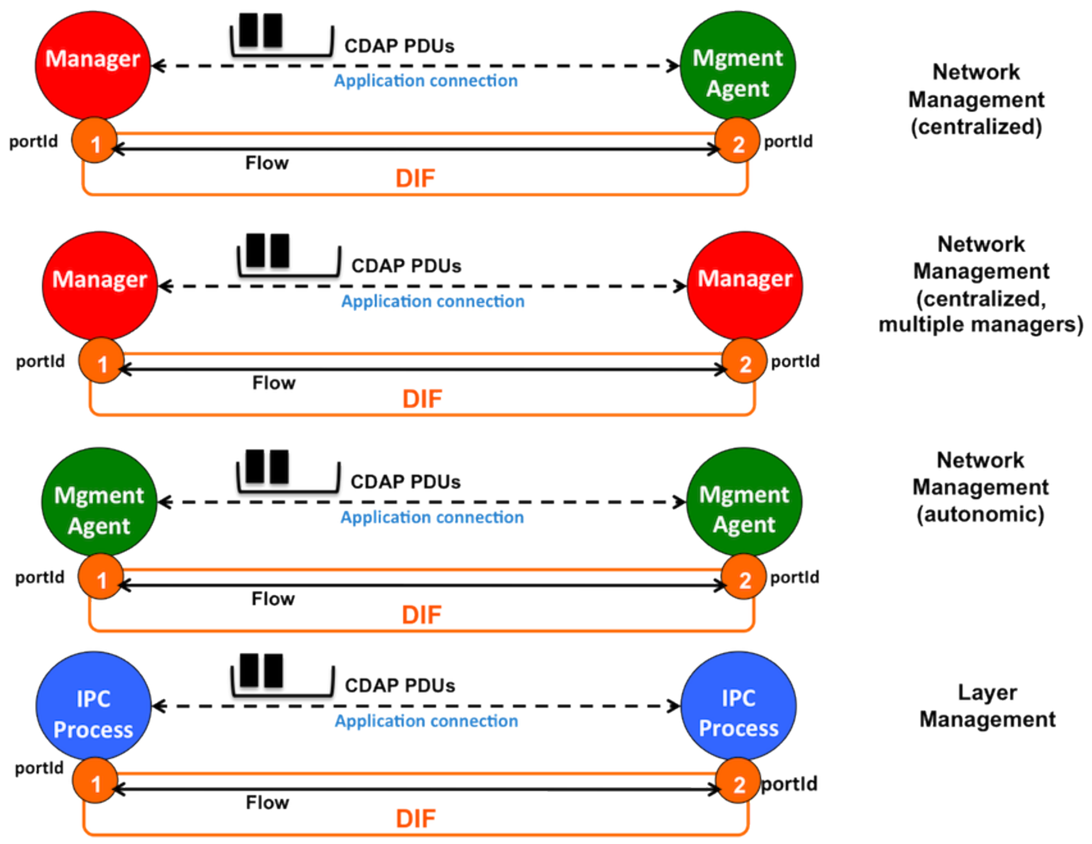


Figure 2. Possible interactions between entities of the RINA management framework

The Figure above illustrates the different types of management interactions that are foreseen in the RINA architecture:

- **Manager-Management Agent** interaction. The most common interaction in the traditional configuration of Network Management Systems, in which a Manager process uses agents in each Computing System in order to monitor those systems and update its configuration when needed.
- **Manager-Manager** interaction. In some cases, like when a DIF is owned by multiple independent entities, it is necessary to partition the management of one or more DIFs into multiple management domains. At least one Manager process is responsible for managing one of those individual domains. Therefore, Manager to Manager interactions are also required.
- **Management Agent - Management Agent** interaction. In this configuration the Management Agents have more autonomy to take certain decisions based on the information learned from other neighbour Management Agents. The degree of autonomy can vary depending on the configuration in use: from using Management Agents

to aggregate management informations in "sub-domains" to Network Management Systems with no central Managers at all.

- **IPC Process - IPC Process** interactions. While this type of interaction doesn’t purely fall in the category of "Network Management", IPC processes in a DIF (layer) use the same tools to exchange information: CACEP for application connection establishment and CDAP to operate on the objects of the neighbouring IPC Processes RIBs. Examples of usage of this interaction are enrolment, routing, flow allocation or resource allocation.

Since many of the tools required by Network Management in RINA are still in its infancy, PRISTINE will initially focus in traditional centralised Network Management configurations, assuming a single Management Domain. As the project evolves and Network Management tools mature (specially the definition of the RIB), PRISTINE may also consider configurations in which Management Agents have more autonomy. Finally, interactions between IPC Processes for layer management are in the scope of PRISTINE, but addressed in other work packages (mainly WP3 and WP4).

3. CDAP: the management protocol

The RINA specifications describe two complementary protocols used to i) establish application connections and ii) exchange messages between communicating Application Entities (AEs, communicating entities on different application processes). The first protocol is used to establish an application connection between AEs, and is called the Common Application Connection Establishment Phase (CACEP). Application establishment is required in order for the two AEs to have enough information to understand each other and to optionally authenticate each other. Amongst this information there is:

- **Abstract syntax** id. The specific version of the CDAP protocol message declarations that the message conforms to.
- **Concrete syntax** id. This identifier selects the concrete syntax used to encode the CDAP protocol messages on a wire format.
- **RIB version**. Version of RIB and object set to use in the conversation between AEs, including its encoding. Note that the abstract syntax refers to the CDAP message syntax, while version refers to the version of the AE RIB objects, their values, vocabulary of object ids, and related behaviours that are subject to change over time.
- **Source naming information**. This information (process name, process instance, entity name and entity instance) identifies the AE that is requesting the establishment of an application connection.
- **Destination naming information**. This information (process name, process instance, entity name and entity instance) identifies the AE that is being targeted for the establishment of an application connection.
- **Authentication information**. Identifies the procedure to be used for authentication (if any), and contains the information required to carry out this procedure (such as the credentials of the source AE).

Once an application connection is established, a second protocol called Common Distributed Application Protocol (CDAP) is used to exchange data on operations over the objects in the RIB exposed by the application entities. CDAP is an object-oriented protocol modelled after CMIP (the Common Management Information Protocol) [cmip] that allows two AEs to perform six operations on the objects exposed by their Resource

Information Bases (RIBs). These fundamental remote operations on objects are: create, delete, read, write, start and stop. Since in RINA there is only one application protocol (CDAP), the different AEs in the same application process do not identify different application protocols, but the subsets of the RIB available through a particular application connection. That is, different AEs provide different levels of privileged access into an Application Process RIB - as illustrated in the Figure below.

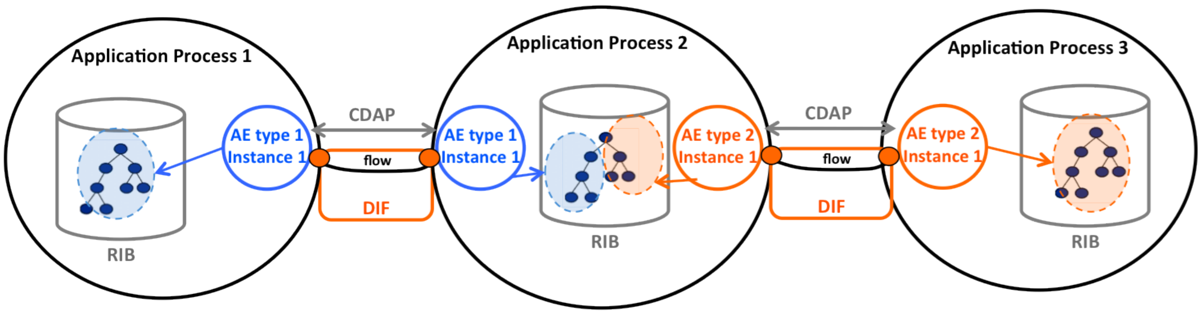


Figure 3. Graphical example illustrating CDAP operation

A more in-depth overview of both protocols is given in the next section.

3.1. CACEP, Common Application Connection Establishment Phase

CACEP allows two Application Entities in different Application Processes (APs) to establish an application connection. During the application connection establishment phase, the AEs exchange naming information, optionally authenticate each other, and agree on the abstract and concrete syntaxes of CDAP/RIB to be used in the connection, as well as on the version of the RIB. This version information is important as RIB model upgrades may not be uniformly applied to the entire network at once. Therefore, it must be possible to allow multiple versions of the RIB to be used, to allow for incremental network management upgrades.

As illustrated in Figure 4, “Operation of CACEP”, CACEP operates in the following way. The Initiating Process first allocates an (N-1)-flow with a destination application. When this is complete, it sends an *M_Connect Request* with the appropriate parameters (mainly source and destination application naming information, identification of the authentication mechanism to be used (if any), ids of the abstract and concrete syntaxes, and version of the RIB) and initiates the authentication policy. Depending on the complexity of the authentication policy, zero or more CDAP messages will be exchanged between the communicating Application Processes. When the authentication policy completes, a positive or negative *M_ Connect Response* is returned by the destination application process and the connection is established.

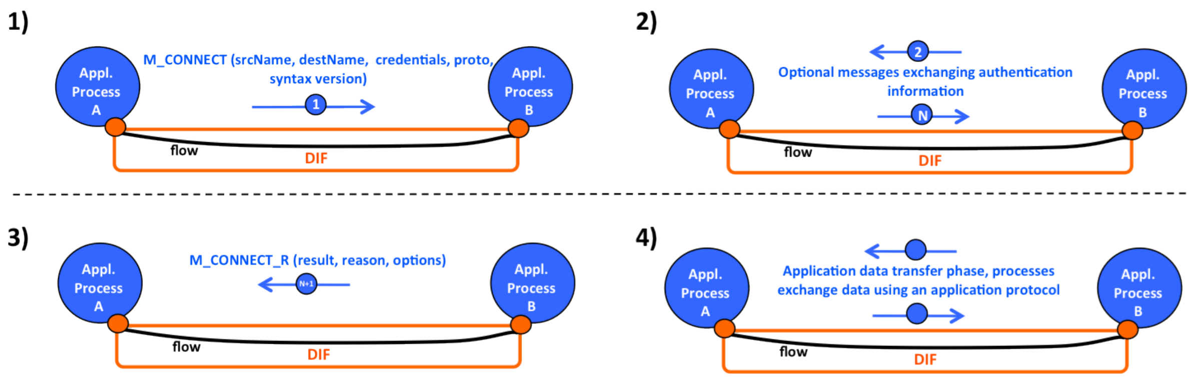

1)
Appl. Process A
M_CONNECT (srcName, destName, credentials, proto, syntax version)
1
Appl. Process B
flow
DIF
2)
Appl. Process A
2
Optional messages exchanging authentication information
N
Appl. Process B
flow
DIF
3)
Appl. Process A
M_CONNECT_R (result, reason, options)
N+1
Appl. Process B
flow
DIF
4)
Appl. Process A
Application data transfer phase, processes exchange data using an application protocol
Appl. Process B
flow
DIF

Figure 4. Operation of CACEP

When any of the two APs wishes to terminate the application connection, it sends an *M_release* Request message to its neighbour, which may or may not require an associated *M_ release Response* message. After that the application connection is over. The two APs may choose to deallocate the flow that was supporting the application connection, or to re-use the same flow for a new application connection. It is also possible to use CACEP connection establishment with another protocol in the data transfer phase (for example, HTTP).

3.2. CDAP, Common Distributed Application Protocol

The Common Distributed Application Protocol (CDAP) is used by communicating RINA applications to exchange structured application-specific data. The RINA architecture uses CDAP to construct specialised distributed applications that cooperate to create a Distributed IPC Facility (DIF), which provides network transport to other applications. However, it can be used by any application that needs to share information or initiate state changes with another application over a network. The protocol itself is not application-specific.

CDAP enables distributed applications to deal with communications at an object level, rather than forcing applications to explicitly deal with serialisation and input/output operations. CDAP provides the application protocol component of a Distributed Application Facility (DAF) that can be used to construct arbitrary distributed applications, of which the DIF is an example. CDAP provides a straightforward and unifying approach to sharing data over a network without having to create specialised protocols.

CDAP is modelled after Common Management Information Protocol (CMIP), a straightforward standards-defined protocol [x711]. While CMIP concepts are employed so that we can benefit from the experience of prior implementations, many details are different and the protocol is no longer CMIP per se. The reasons for modelling CDAP after CMIP are that CMIP is a distributed object-oriented intermediate language; that performs create/delete, read/write (get/put), action (start/stop) on objects with a certain schema. CMIP provides almost all that is required to create a universal application protocol and nothing more (almost, justifying the changes introduced by CDAP). It may be seen as too basic, but adding more features to the protocol would end up with the creation of a new full-blown programming language, since between the basic form and a programming language there is no natural place to stop. CDAP provides the following main changes with respect to CMIP:

- CDAP does not provide for generalised actions (M_ACTION). The only provided actions are M_START and M_STOP.
- There is no M_EVENT message. Applications may generate M_WRITE messages (with or without confirmation) to create the same effect. Or, applications can selectively register for asynchronous notifications by issuing an M_READ to a particular notification object at the start of a connection, and the opposite application can then send non-final M_READ_Reply messages to indicate any number of event occurrences. Both methods provide the capability of returning application-defined data objects along with the notification.
- There is no M_ABORT message. M_RELEASE with no confirmation requested has indistinguishable semantics for CDAP.
- Scope/Filter: Filter is left out of CDAP for the time being (see scope/filter discussion below).
- Authentication: Multiple plug-in policies will be defined, with means for both standards-defined and user extension.
- Concrete encoding: CDAP can be encoded in multiple ways. The original concrete encoding specified has been Google Protocol Buffers (GPB) [gpb] [gpb], but many others are possible and will be defined in the future (such as the concrete encodings of ASN.1 [asn1], JSON [json], XML [xml], etc.).

The design of CDAP explicitly takes into account that implementations will be done in multiple languages and on various platforms of widely different scales, so some lowest-common-denominator choices have been adopted rather than using approaches that might narrow the acceptance or unduly complicate implementations.

3.2.1. Objects

CDAP allows applications to send and receive data using structured information that we refer to as Objects. We refer to the set of objects stored within an application that is available via CDAP as its local Resource Information Base (RIB). All modern programming languages have an object concept, though the component types and aggregation capabilities are richer in some languages than others. In the CDAP model, the Application Entities (AE) that are communicating with one another create a shared object space, providing access to the portion of the application's RIB that is relevant to the AE's purpose, and allowing a distributed application to selectively create and share distributed objects.

Provided that an application can encode an object into a sequence of bytes, and that the application it is communicating with can decode them properly, CDAP places no restrictions on application object values, however, for its own operation, CDAP uses a simple and easily-represented definition of the general object concept, consisting of entities that are:

- A limited set of scalar types (a single scalar is a degenerate object),
- aggregations of objects of the same type (arrays), or
- aggregations of potentially-dissimilar objects (structures).

The objects in an application have four properties of concern to CDAP:

- A Class, or type, that recursively captures all of the types of the outermost object and those of any contained objects,
- a name that is unique among other objects of the same class,
- a value,
- an object identifier, an integer that may be assigned to a specific instance of an object by its owner as an alias to its class and name.

The object identifier is a shorthand name for an object. In CDAP and other RINA protocols, it is never mandatory to use it, its purpose is solely to shorten messages and avoid the necessity of repeatedly mapping class/ names to objects. Either the class and name, or the object identifier, or both may be supplied in a message (if both are supplied, they are validated for consistency). The object identifier value may have a previously-agreed-upon value known to both applications, or it may be dynamically assigned for use on an application connection. Pre-defined manifestly-known object identifier values that are known a priori to map to specific names/classes may be provided outside the context of an application connection, for example in a standard, in which case the applications must both be aware of and conform to the pre-agreed mapping (the specific pre-agreed object ids may be associated with the version number describing the RIB).

3.2.2. Scope and Filter

Scope and filter allow the user of CDAP to operate on multiple objects with a single CDAP message. Scoping selects objects to be operated upon within the managed object containment tree. As shown in the Figure below, the scope of an operation is defined relative to a base managed object:

- Operation applies to the base object only
- Operation applies to the Nth level subordinate objects only
- Operation applies to the base object plus all of its subordinates (entire sub-tree)

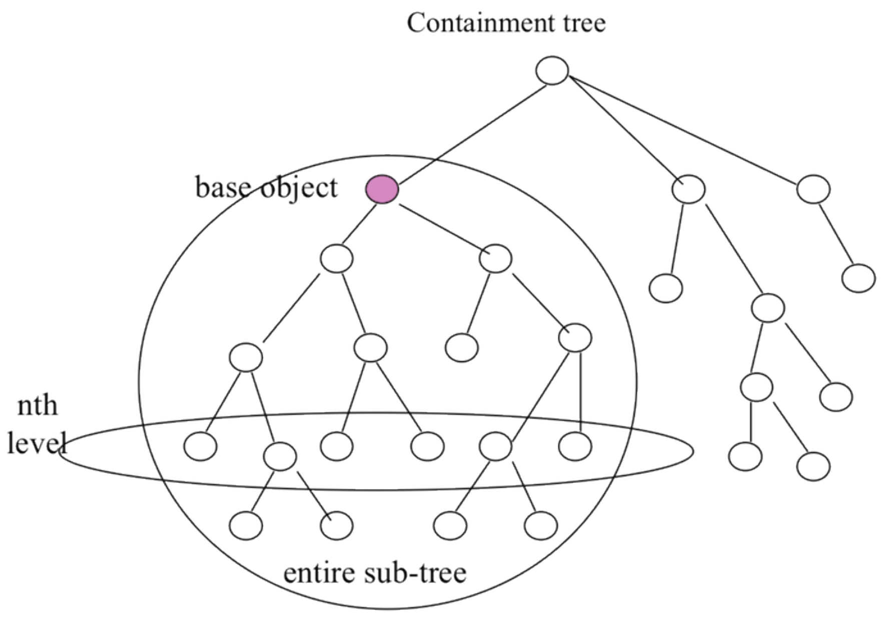


Figure 5. Graphical representation of scope in a CDAP operation

Filtering permits objects within scope to be selected according to test criteria, allowing the CDAP user to select a subset of all the objects in the scope of an operation. The operation is then applied to all selected objects. Although it is clear that the filter mechanism is useful, it is far from clear that the filter definition provided by CMIP is the best one. [1] Therefore, the definition of the rules for constructing filters (and even the decision of including filter in the protocol) is left to future research.

3.2.3. Concrete syntaxes

The encoding of the message type (*opCode*) and other values in the message into a form used for communication on a *wire* is referred to as the concrete syntax of the message. Agreement between communicating applications on the concrete syntax to use is a fundamental requirement for communication. Once this is established, it becomes possible to discuss other aspects of the communication, such as the version of the message declarations (the abstract syntax) and object definitions and values at the application level (usually summarised as a *version*) in use.

For an application connection, after the data transfer flow is established the first message sent from one application (the requester) to another (the responder) is a request for connection. The first messages exchanged, the M_CONNECT and M_CONNECT_R messages, must include the concrete syntax value that corresponds to the CDAP message encoding method being used. This is done by encoding the concrete syntax in the first byte(s) of the message so that it can be examined before making the decision to use a certain syntax to interpret the remainder of the message. In the M_CONNECT message, the value in the first byte(s) of the message designates the concrete syntax used to encode the entire message (After application connection establishment, the applications are free to use a different concrete syntax, if their agreed-to protocol version so indicates). In the M_CONNECT_R response, the first byte of the message has the same value, representing acceptance, or a different one, representing a counter-proposal. If either party fails to recognise the syntax value it receives (either the concrete or abstract syntax), or understands it, but refuses to use that syntax, it discards the message and the connection establishment fails.

[1]CMIP filter expressions take more resources when processing them. So it is a balance between usefulness and resource usage.

Google Protocol Buffers (GPB)

Google Protocol Buffers was the first concrete syntax defined to encode and decode the CDAP protocol messages. The reasons for choosing GPB are that it provides an efficient encoding (both in terms of parsing/ generation time and bit efficiency), it is being used in production environments such as massive scale distributed systems by Google and there are free, open tools for developers in many programming languages. Other encoding schemes such as JSON, XML or ASN.1 concrete encodings may be used in some environments and will be studied in the future.

In the defined GPB encoding of CDAP messages, encoding the value of the abstract syntax identifier as the very first field of the message will produce the constant value of 0x08 (a 32-bit Variant with field number = 1) in the first byte of the message. Encoding the GPB message fields in canonical order, the usual behaviour, will achieve this automatically. The CDAP protocol engine can check for this value to recognise that the syntax is GPB, and can then safely parse the remainder of the message. If other concrete syntaxes are defined for CDAP in the future, they will use a different value for the first byte of the message, making it possible for multiple concrete syntaxes to exist contemporaneously.

4. RIB: State of the Art

4.1. Overview of the different options

Any survey of the network management landscape and standards bodies, will yield extended work from a variety of established and trusted standardisation organisations. Among the most influential are:

- **OSI - CMIP and X700** In particular OSI's work on the X.700 series of specifications which are used as the basis for Telecommunications Management Networks, as specified in the M.3000 series of specifications.
- **IETF - SNMP and SMI** This was a response to the OSI work, where a simpler protocol was specified an interim solution, hence the "Simple" in the title. However, this was seen as a capable and flexible management solution in its own right and was widely adopted by equipment vendors (SNMP has the largest deployed implementation) It had some limitations, and has had two major version revisions.
- **IETF - NetConf and YANG** This is a more recent standardisation activity, with a goal to updating the network management techniques to a more recent technology stack. If provides mechanisms to install, manipulate, and delete the configuration of network devices. It uses an Extensible Mark-up Language (XML)-based data encoding for the configuration data as well as the protocol messages, although alternate encodings have been proposed.
- **DMTF - WBEM and CIM** This is an effort to address some of the perceived shortcomings of SNMP for enterprise systems management. [2] Some new concepts were introduced, and UML was used as the specification language. This was implemented on enterprise operating systems but not widely deployed on network equipment.
- **TM-Forum - SID** The TM-Forum's SID (Shared Information and Data model) is one of the frameworks of the TMForum NGOSS (New Generation Operations Systems and Software) Framework suite. It is

[2] Interoperability - Most modules published by the IETF proved useful for monitoring purposes. However, for configuration tasks, vendors chose expedience over interoperability by implementing their own private enterprise modules. Security - SNMP v1 had no security built in. Ease of use - Use familiar technologies and web based configuration tools. DTMF used Web Based Enterprise Management (WBEM) which is basically HTTP over SSL.

also called the Information Framework. SID can be considered as the language of the NGOSS. The aim of the model is to identify the business entities that play a role in the business processes of a telecommunications service provider.

From the above, we can see there are three main management lineages. The first is the OSI work contained in X700 series. The second is the IETF's work, here we focus on IETF's newer work on NETCONF protocol and its data model YANG. The final lineage is the UML based one, from DMTF - CIM model to the TM-Forums SID model. Here we focus again on the more recent work done by the TM-Forum, which built upon concepts developed in the DMTF - CIM model. An outline of the OSI ITU-T x.700, SID and IETF YANG are given below.

4.1.1. Review of current E.U. projects

The current E.U. projects in the Future Networks Management area, have been reviewed. The aim of each project was discerned, and an short review into the aspects of Network Management covered by the project carried out.

- **COSIGN** - The Combining Optics and Software Defined Networks(SDN) in next Generation data centre Networks (COSIGN) project [COSIGN] aims to define and implement a flat, scalable Data-Centre-Network architecture facilitated by optical technologies and SDN based network control. – (Optical path routing) - fully optical interconnection path from server to server within racks and between racks.
- **NetIDE** – The NetIDE project [NETIDE] aims to build a single Integrated Development Environment (IDE) to support the complete development life-cycle of network controller programs in a vendor-independent fashion. – (Software Defined Network configuration) - protocol-independent cross controller IDE.
- **T-NOVA** The T-NOVA project [TNOVA] aims to implement a management/orchestration platform for the automated provision, configuration, monitoring, and optimisation of Network Functions-as-a-Service (NFaaS) specifically tailored towards cloud environments.
- **UNIFY** – The UNIFY project [UNIFY] aims to create a service abstraction model and an associated domain-specific creation language

and programming interfaces to automate and optimise the deployment of service chains. – (Service Provider DevOps) – new management technologies for service delivery in cloud environments.

- **GreenICN** – The Green ICN project [GreenICN] aims to address how Information Centric Networking (ICN) networks and devices can operate in a scalable and energy-efficient manner in the aftermath of disasters constrained by fragmented networks with intermittent connectivity and for efficient pub/sub video delivery. – (Network Architecture) – for content delivery networks in constrained resource scenarios.
- **STRAUSS** – The STRAUSS project [STRAUSS] aims to define a global optical infrastructure for Ethernet transport that includes heterogeneous transport and network control plane technologies. – (Optical path routing) – Ethernet transport architecture using Software Defined Networks (SDN) and Network Function Virtualisation (NFV) approach over optical networks.
- **LEONE** – The LEONE project [LEONE] aims to build a network management framework that integrates network management and performance characteristics from a large number of diverse resources in a bid to increase the Quality of Experience (QoE) experienced by the end users. – (Network data analysis) – measuring QoE as perceived by the end-users.
- **Trilogy 2** – The Trilogy 2 project [TRILOGY2] aims to provide a converged architectural framework capable of orchestrating, provisioning, and controlling the usage of heterogeneous resources such as bandwidth, storage and processing as demanded by emerging highly distributed applications. – (Architectural Framework) – for sharing extraneous heterogeneous resources regardless of the contributors.

On completing a short review, most of these projects can be characterised as development projects for management systems for specific environments. PRISTINE takes a more general approach to network management, by leveraging the commonality already in the RINA model to reveal more about the nature of network management. Thus currently these projects, do not occupy the same network management research space as PRISTINE’s DMS.

It should be noted that this is a snapshot of these projects, based on available published papers and deliverables, and some of these projects may produce work of interest in future deliverables.

4.1.2. OSI, ITU-T x.700

X.700 series of specifications are a set of standards used by OSI and ITU to specify the capability for managers to gather information and to exercise control, and the capability to maintain an awareness of, and report on, the status of resources within an OSI management environment (OSIE).

In essence they define the objects and functionalities needed for a network management system, along with specifying tools and services needed to monitor, control and coordinate management activities.

Managed objects

A managed object is the OSI management view of a resource that is subject to management, such as an item of physical communications equipment. Thus, [m3010] defines a managed object as *the abstracted view of a network resource (physical or functional) that represents its properties as seen by (and for the purposes of) management*.

A managed object is defined in terms of attributes it possesses, operations that may be performed upon it, notifications that it may issue and its relationships with other managed objects.

Management is achieved through cooperation between one or more components of the management activity taking a *managing* role and others taking a *managed* role. The role played by a particular system may be static or may change over time. This is referred to as manager-agent roles, elsewhere in this document.

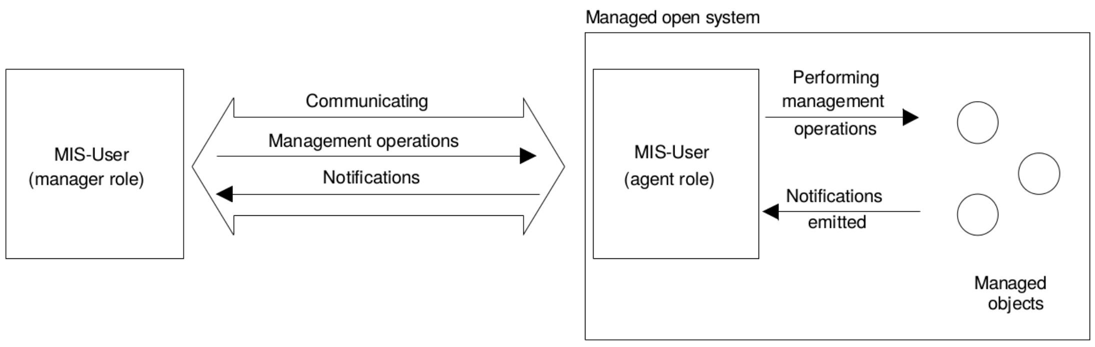


Figure 6. OSI management information flow

As defined in [x710], the set of managed objects within a system, together with their attributes, constitutes that system's management information base (MIB).

OSI Management functional areas

OSI management is required for a number of purposes. These requirements are categorized into a number of functional areas as defined in [m3400]:

Fault management encompasses fault detection, isolation and the correction of abnormal operation of the OSI environment. Faults may be persistent or transient. Faults manifest themselves as particular events (e.g. errors) in the operation of a system.

Configuration management identifies, exercises control over, collects data from and provides data to systems for the purpose of preparing for, initializing, starting, providing for the continuous operation of, and terminating interconnection services.

Accounting management allows charges to be established for the use of resources in the OSIE, and for costs to be identified for the use of those resources. In can be argued that Accounting management operates on a subset of the data made available through performance management, so PRISTINE focuses on the performance management aspects.

Performance management enables the behaviour of resources in the OSIE and the effectiveness of communication activities to be evaluated. It allows, statistical information to the gathered, logs of system state to be examined, and provides a means to determine system performance under natural or artificial conditions.

Security management. The purpose of security management is to support the application of security policies. This includes controlling security services and mechanisms, and the reporting of security-relevant events.

These areas are widely used when describing the functional scope of management systems in general.

4.1.3. SID

The Information Framework (SID) model is standardised by the TeleManagement Forum (TM Forum) in [SID]. The SID is part of the

Framework (formerly known as NGOSS) knowledge base and thus closely linked to the TM Forum's architecture and business modelling (eTOM). The SID provides business and system view definitions for designing and managing a telecommunications network. The SID defines domains and entities on several levels of detail. It enables the design of services and network resources in conjunction with products and customers, thus providing the necessary associations to link all resources to business activities. For example, services are categorised as customer or resource facing, where the former are services such as a Virtual Private Network (VPN) that a customer is directly aware of, whilst the latter are internal network services, such as Multi Protocol Label Switching (MPLS) or Border Gateway Protocol (BGP), that the customer is unaware of.

The SID model represents business concepts, their characteristics and relationships, described in an implementation independent manner. The model provides a detailed understanding of all areas of Business. It has many uses including internal modelling work, defining a common business terminology (e.g., for integration activities), and understanding business concepts and their relationships. The SID can also be used as a tool-kit that allows modellers to select particular aspects of the model they need to use to model specific applications.

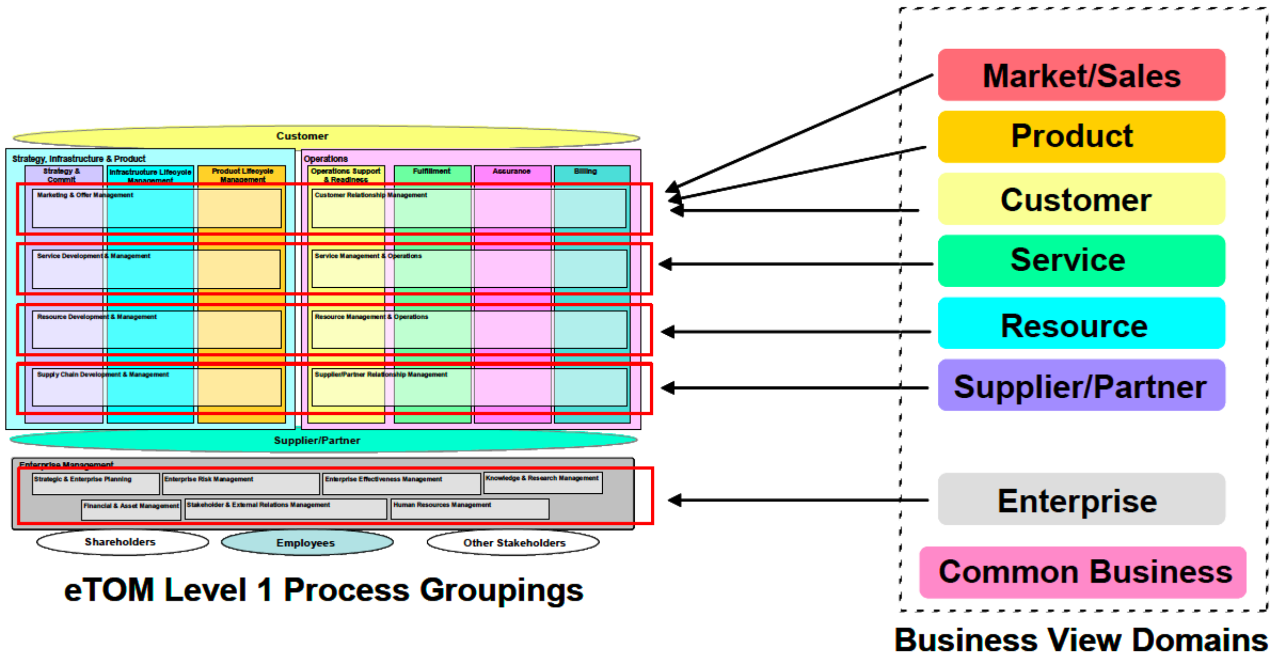

Customer
Strategy, Infrastructure & Product
Strategy & Commit
Infrastructure Lifecycle Management
Product Lifecycle Management
Operations
Operations Support & Readiness
Fulfillment
Assurance
Billing
Marketing & Offer Management
Customer Relationship Management
Service Development & Management
Service Management & Operations
Resource Development & Management
Resource Management & Operations
Supply Chain Development & Management
Supplier/Partner Relationship Management
Supplier/Partner
Enterprise Management
Strategic & Enterprise Planning
Enterprise Risk Management
Enterprise Effectiveness Management
Knowledge & Research Management
Financial & Asset Management
Stakeholder & External Relations Management
Human Resources Management
Shareholders
Employees
Other Stakeholders
eTOM Level 1 Process Groupings
Market/Sales
Product
Customer
Service
Resource
Supplier/Partner
Enterprise
Common Business
Business View Domains

Figure 7. eTom Business View Domains

The Information Framework business view model can be viewed as complementary to the Business Process Framework by providing an information/data reference model and a common information/data vocabulary from a business entity perspective. The business view model uses the concepts of domains and aggregate business entities (or sub-domains) to categorise business entities in order to reduce duplication and overlap.

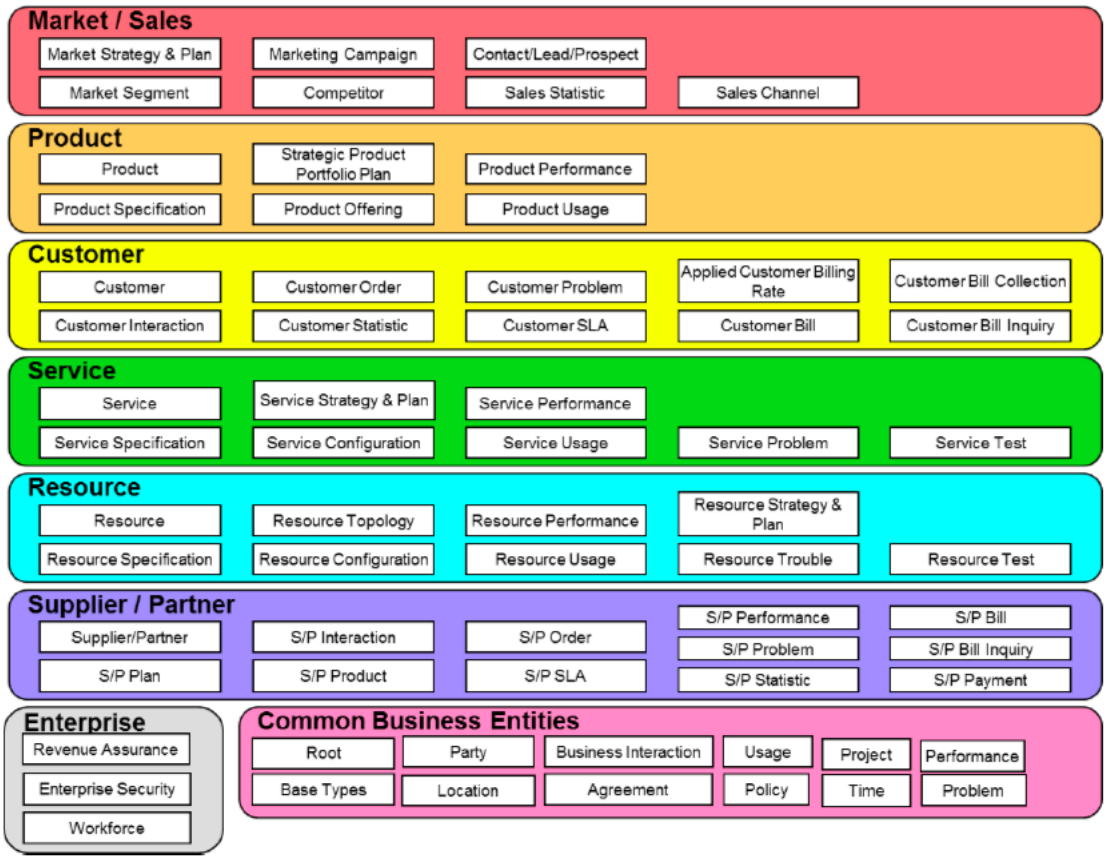


Figure 8. SID model

Together the Business Process Framework and Information Framework provide enterprises with a process and entity view of their business. Essentially, the Information Framework provides the definition of the things that are to be affected by the business processes defined in the Business Process Framework.

4.1.4. IETF - YANG

YANG [rfc6020] is a language used to model data for the NETCONF protocol. NETCONF [rfc4741] is a protocol devised by an IETF WG in order to support network configuration. Its design was heavily influenced by Juniper Networks JUNOscript application programming interface. The NETCONF protocol supports several features required for configuration management that were lacking in other network management protocols, such as SNMP [rfc3410]. It operates on data-stores and represents the configuration of a device as an XML document. It provides primitives to help coordinate and facilitate configuration changes over multiple devices (including locking and transactions). This is seen as one of its key strengths.

NETCONF has a simple layered model, where protocol operations are executed as RPC calls over a secure communications channel. This layering is shown in the following diagram.

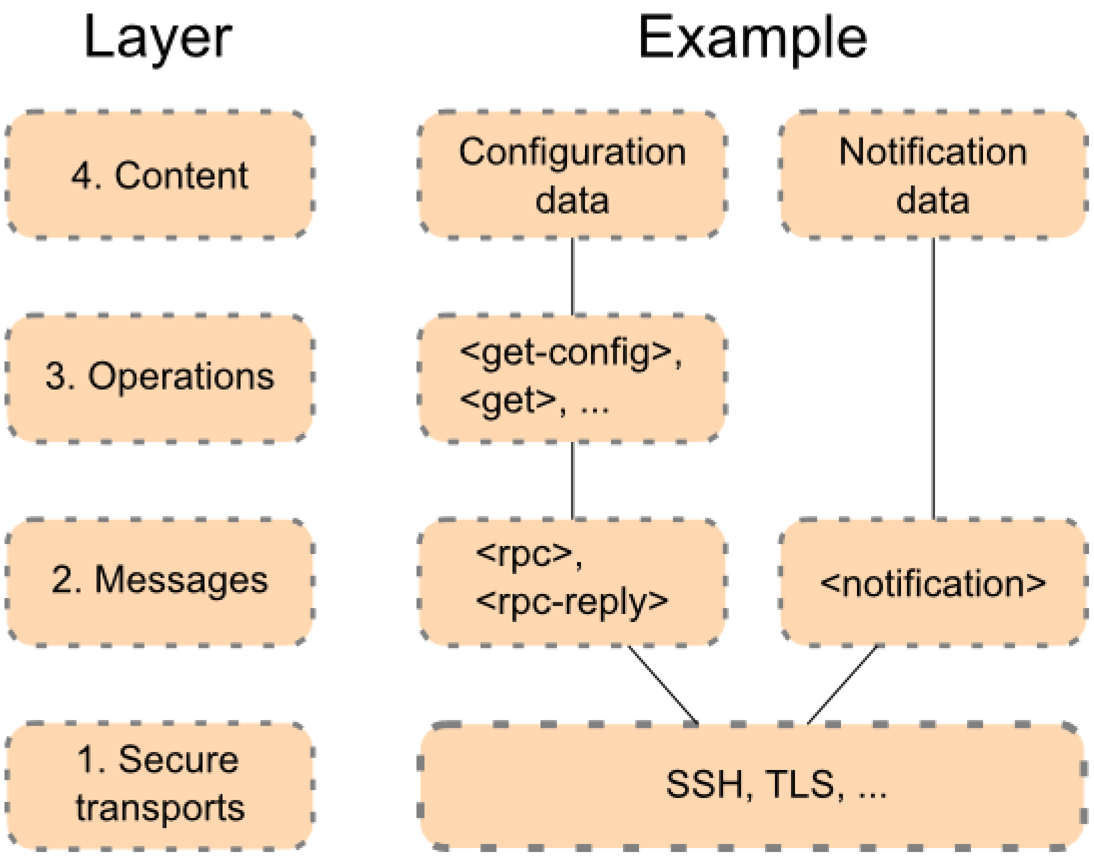


Figure 9. NETCONF protocol layers

YANG defines the data model used in NETCONF. A YANG module defines a hierarchy of data that can be used for NETCONF based operations, including configuration, state data, Remote Procedure Calls (RPCs), and notifications. This allows a complete specification of all data sent between a NETCONF client and server.

YANG [rfc6020] structures data models into modules and sub-modules. A module can import data from other external modules, and include data from sub-modules. The hierarchy can be augmented, allowing one module to add data nodes to the hierarchy defined in another module. This augmentation can be conditional, with new nodes appearing only if certain conditions are met.

YANG unsurprisingly defines a set of built-in types, and has a type mechanism through which additional types may be defined. Derived types can restrict their base type's set of valid values using mechanisms like range or pattern restrictions that can be enforced by clients or servers. They can also define usage conventions for use of the derived type, such as a string-based type that contains a host name.

YANG also introduces explicit support for some best practices, for example, allowing data-type distinctions between static/configuration data and dynamic/operational data. Operational or dynamic state is data that is derived by other means than configuration activities, for example, by signalling or routing protocols between peer devices. This clear separation allows *true* configuration data to be manipulated separately from other operational state, for example, backed up, or used to seed another device. To take advantage of this capability, NETCONF defines some explicit protocol operations.

get-config allows only the configuration data to be retrieved from the specified target object, omitting any operational state.

copy-config allows the configuration to transferred from a source to a target object. For example, making the current running configuration the default startup configuration.

delete-config allows configuration data to be removed from a device or datastore.

edit-config allows configuration changes to be made, by merging changes with the existing configuration. This can be thought of as applying a patch to a data-store to derive a new configuration. It can take a filter or test operation to determine whether the configuration changes are applied to the object in question.

A summary of differences between NETCONF and SNMP [rfc3410] are outlined in [rfc3535] and summarised below:

- Distinction between configuration and state data
- Selective data retrieval with filtering

- Multiple configuration data stores (candidate, running, startup)
- Configuration change transactions
- Extensible procedure call mechanism
- Streaming and playback of event notifications

In summary, NETCONF adds the features above, avoids some limitations in using UDP directly, and has standard security and commit mechanisms.

4.2. Managed Object models

The following section describes the various managed object models, and discusses them under various headings, for example, how objects are defined, what they are composed of, what types of relationships are allowed, etc.

4.2.1. OSI, ITU-T x.700

The x.700 series specifications define a mechanism and framework for capturing, formally describing, and managing Managed Objects.

Components of a managed object class

Managed objects that share the same definition are instances of the same managed object class [x722]. Different instances of a given class will share the attributes, operations, notifications and behaviour defined in mandatory packages of the class, and will share those defined in conditional packages to the extent that the instances satisfy the conditions associated with those packages.

Packages. A package is a collection of characteristics, i.e. attributes, notifications, operations and/or behaviour, which is an integral module of a managed object class definition. Packages are specified as either mandatory or conditional when referenced in a managed object definition. A mandatory package must be present in all instances of a given managed object class. A conditional package is a package that shall be present in a managed object for which the explicit condition associated with that package in the managed object class definition is TRUE.

Packages can only exist as part of a managed object. Packages become an integral part of the managed object and cannot be accessed outside of a managed object.

Attributes. Managed objects have attributes. An attribute has an associated value, which can exhibit structure, i.e. it can consist of a set or sequence of elements. Attributes are defined to be in packages.

Behaviour. Part of the definition of a managed object class is behaviour. The behaviour can define the semantics of the attributes, operations and notifications. They also can be used to define, preconditions, postconditions or invariants that apply to the entire managed object.

Relation between objects

There are two relationship types between objects in x.700 specifications as defined in [x720]:

Inheritance. One managed object class is specialised from another managed object class by defining it as an extension of the other managed object class. Such an extension is made by defining further packages that include one or more of the following:

- new management operations;
- new attributes;
- new notifications;
- new behaviour;
- extensions to the characteristics of the original managed object class.

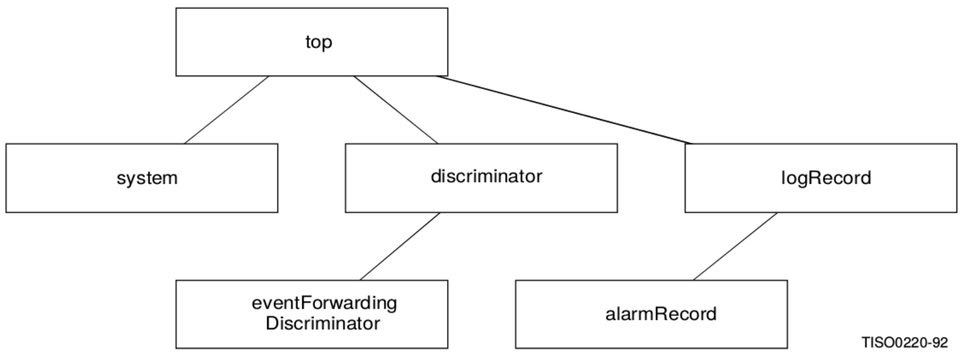


Figure 10. Sample inheritance tree from [x720]

Containment and Naming. A managed object of one class can contain other managed objects of the same or different classes. This containment relationship is a relationship between managed object instances, not classes. The containment relationship is used for naming managed objects. Objects that are named in terms of another object are termed subordinate objects of that object. The object that establishes the naming context for other objects is called the superior object of these subordinate objects [x720]. A subordinate object is named by the combination of:

- the name of its superior object;
- information uniquely identifying this object within the scope of its superior object.

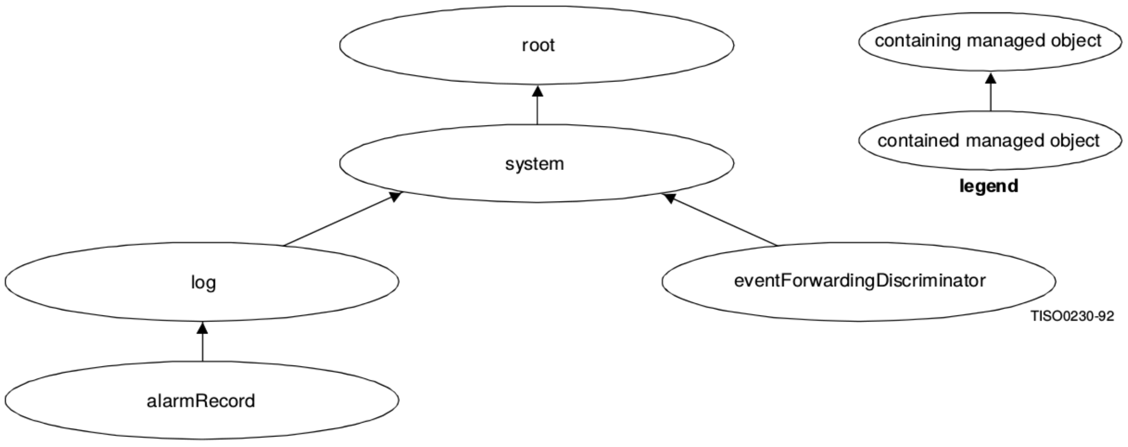


Figure 11. Sample containment tree from [x720]

Managed object identification

Each managed object is identified within the scope of its superior object by means of an attribute value assertion (AVA) that a specified attribute has a specified value. When used for naming in this way, an AVA is called a relative distinguished name (RDN), and must have the property of unambiguously identifying a single managed object within the scope of its superior object [x720].

A system managed object represents the managed system. Each system managed object has systemTitle and systemId attributes. Either of these attributes may be used in naming the system managed object. The systemId attribute is single-valued and its ASN.1 type is a choice of the following:

- a GraphicString;

- an INTEGER;
- a NULL.

Systems management operations

Two kinds of management operations are defined: those which can be sent to a managed object to be applied to its attributes, and those which apply to the managed object as a whole.

Attribute oriented operations. The following management operations can be sent to a managed object to be applied to its attributes:

- get attribute value;
- replace attribute value;
- replace-with-default value;
- add member;
- remove member.

Operations that apply to managed objects as a whole. The following management operations apply to managed objects as a whole and their impact is generally not confined to modifications of attribute values:

- Create;
- Delete;
- Action.

4.2.2. The Information Framework (SID)

Entities – An Entity represents a collection of instances of the same type (i.e. a chair entity represents the concept of chairs, not just a single chair). An entity name is usually a noun in the singular, such as Animal, Cable Section, Customer Request, etc. Entities represent both physical (e.g. mobile phone) and conceptual (e.g. ownership) things. Entities in the Framework are shown using either a box with 3 segments or a simplified representation of a single box. Methods are not shown in the information model. An attribute represents a characteristic of an Entity (e.g. a chair may have attributes of weight & manufacture date). An attribute may optionally have its type specified in the model. Complex attributes may be split into a separate Entity using the Composite Pattern (e.g. Person Entity and PersonName Entity).

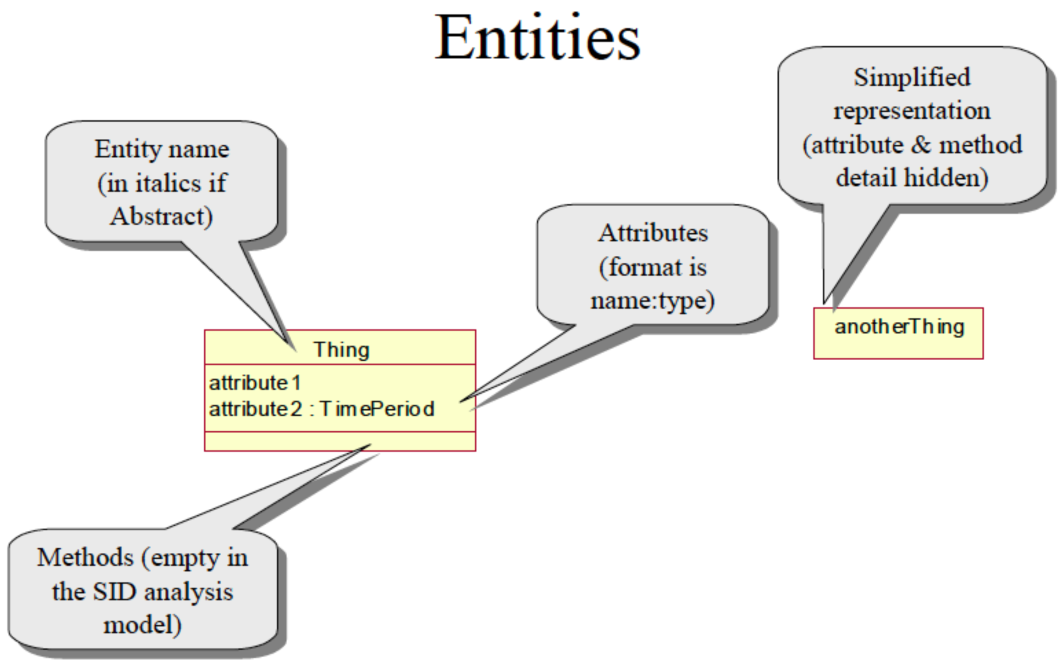

Entities
Entity name (in italics if Abstract)
Simplified representation (attribute & method detail hidden)
Attributes (format is name:type)
Thing
attribute1
attribute2 : TimePeriod
anotherThing
Methods (empty in the SID analysis model)

Figure 12. SID entities

Associations – An association is shown as a solid line joining two entities used to show relationships between entities. This relates to business information such as “Customers purchase Products”. When an association has its own separate properties, it is shown as an Association Entity, with a dashed line joining it to the association that it is representing. Association names help to comprehend what the association represents. The name will usually be a verbal phrase, like “Person has Name” or “Customer orders a Product”. Associations are also assigned a cardinality at each end, indicating the number of instances participating in the relationship. In some cases, an association may have an association role defined. An association role is used when an entity at one end plays a role in the association and is often used in self joins. E.g. an association between two individuals in a family to model the business rule “a parent has one or more children; each child has two parents”.

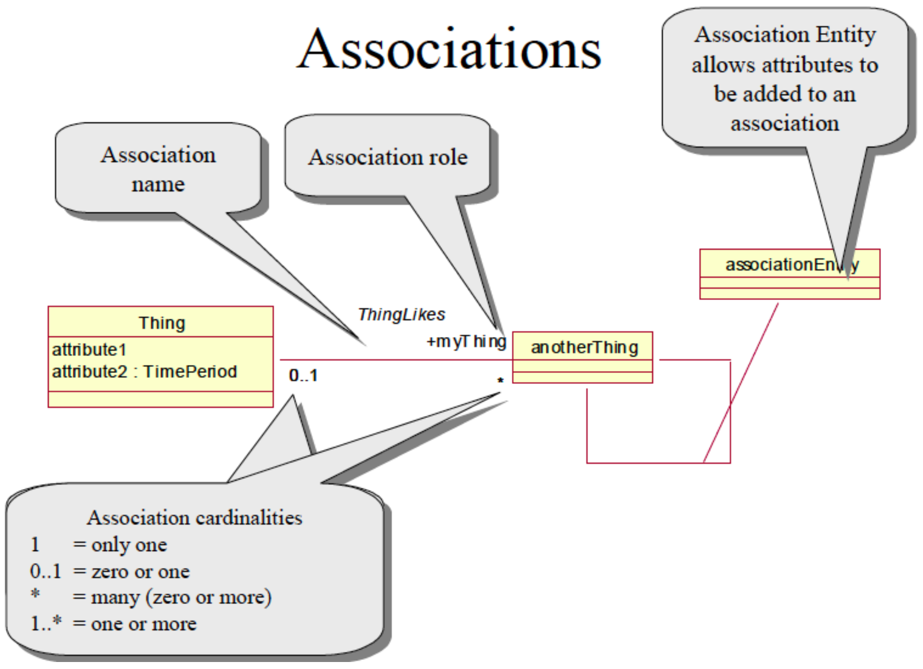


Figure 13. SID associations

Aggregation Association – The aggregation association is an association that indicates that there is a closer relationship than with a normal association, such as a whole / part relationship. The aggregation association is shown with a 'hollow diamond' at the Entity that groups the parts. Aggregation often relates to a business rule like "A has one or more Y, each Y is a part of an X".

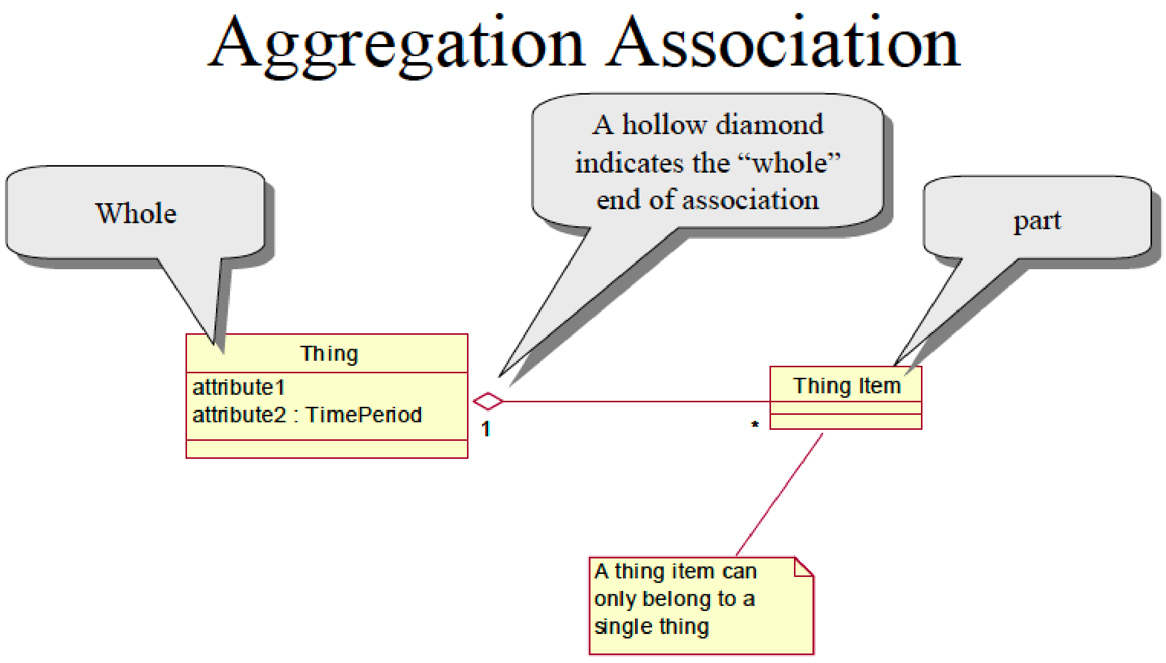

Aggregation Association
Whole
A hollow diamond indicates the "whole" end of association
part
Thing
attribute1
attribute2 : TimePeriod
1
*
Thing Item
A thing item can only belong to a single thing

Figure 14. SID aggregation association

Inheritance / Specialisation – Inheritance relates generic entities to more specific variants. An typical example of this is "A mobile phone is a type of phone". Overuse of inheritance can produce poor models and so simple "X is a type of Y" concepts are not always modelled in the SID model using inheritance. In the SID model, parent entities are often shown as being abstract (the name is shown in italics). Attributes in the parent entity that are also present in the child entity are not repeated on the UML diagrams.

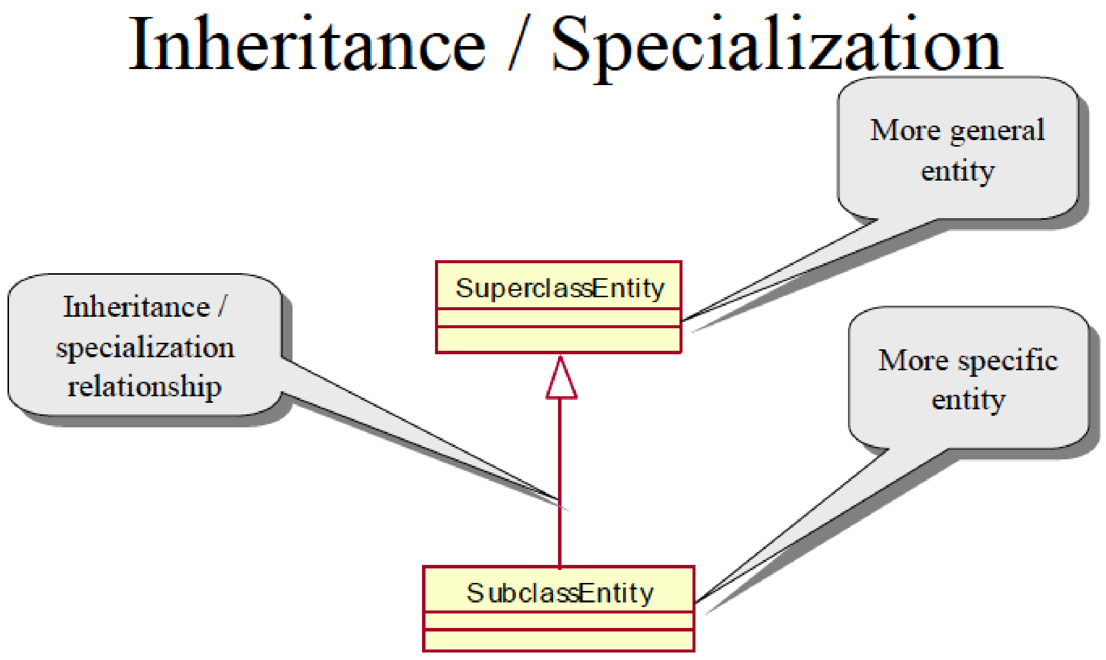

Inheritance / Specialization
More general entity
SuperclassEntity
Inheritance / specialization relationship
More specific entity
SubclassEntity

Figure 15. SID inheritance / specialisation

Composite – The Composite pattern is used when there is a business concept where a single thing or a collection of those things can be used interchangeably. For example, in a warehouse the concept of a "stock item" may include parts, sub-assemblies and complete items. When composites are formed from physical things, these often form tree structure. Composites of logical things and composites of specifications often form directed acyclic graphs.

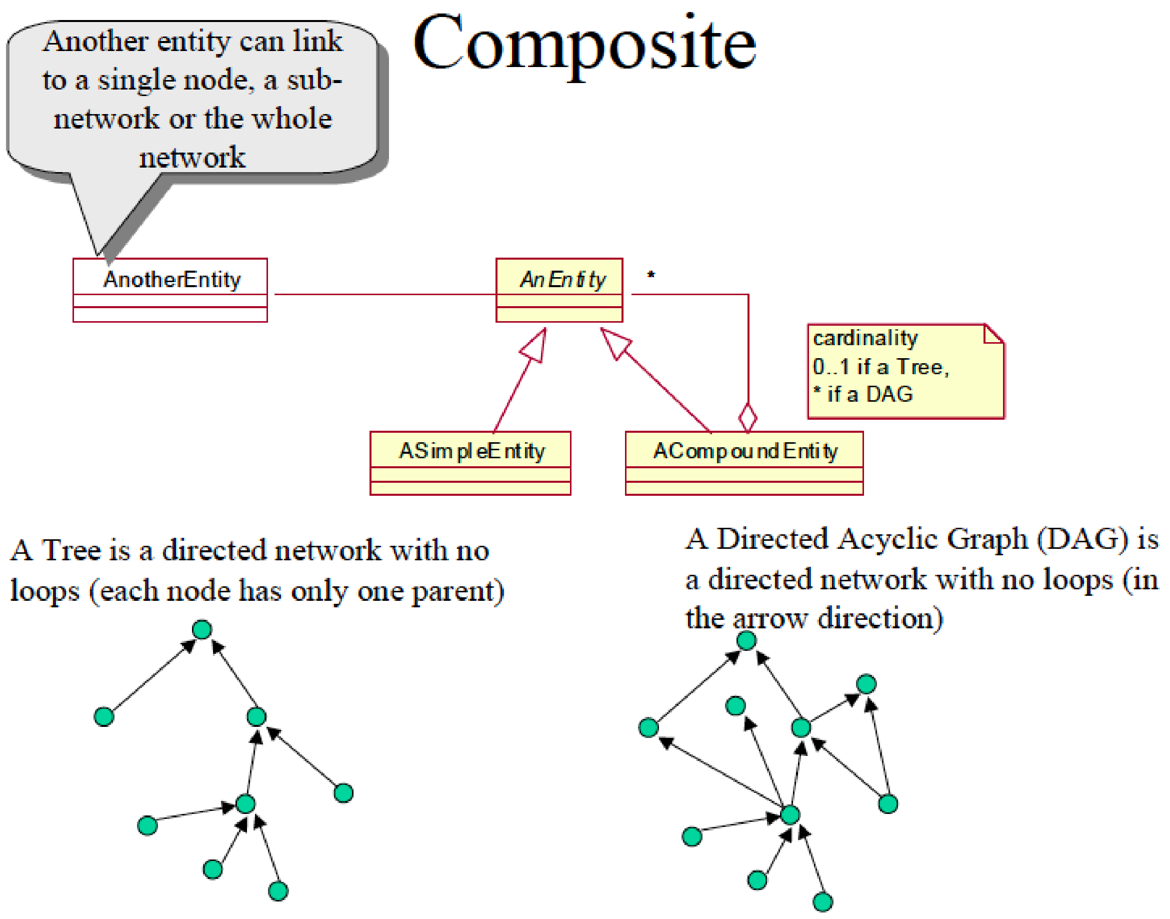

Composite
Another entity can link to a single node, a sub-network or the whole network
AnotherEntity
AnEntity
*
cardinality
0..1 if a Tree,
* if a DAG
ASimpleEntity
ACompoundEntity
A Tree is a directed network with no loops (each node has only one parent)
A Directed Acyclic Graph (DAG) is a directed network with no loops (in the arrow direction)

Figure 16. SID composite

Role Entity – The Role Entity pattern facilitates the representation of behaviour with respect to a given context. For instance, "A person who is a witness in the context of a legal trial". The use of roles is a fundamental pattern that helps simplify a model, and make it more closely represent the real world. Intrinsic attributes are those that a thing always has. Contextual attributes are those that relate to a thing in certain situations.

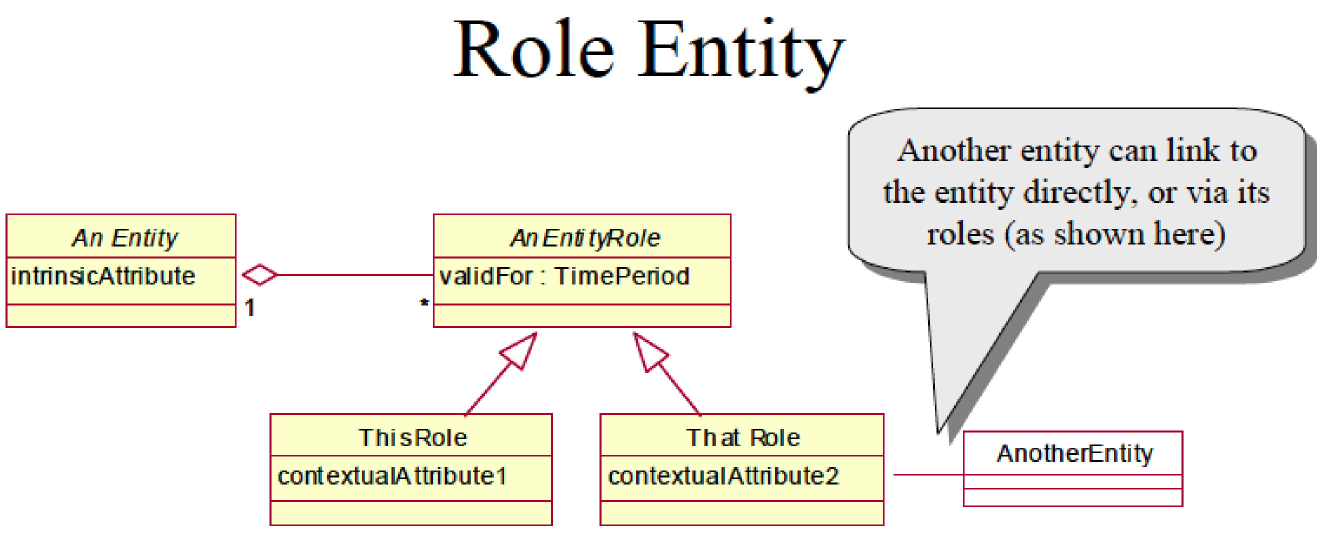


Figure 17. SID role entity

Temporal State Entity – This pattern is used to show the different states of an entity, the attributes for each state and the temporal or lifecycle aspects of an Entity. An Entity's state will change over time and it may be useful to keep only the current state or a complete history, depending on the business requirements. Separating the characteristics that need to be monitored over time in a separate entity makes it clearer than if it was shown as attributes in the entity.

Temporal State Entity

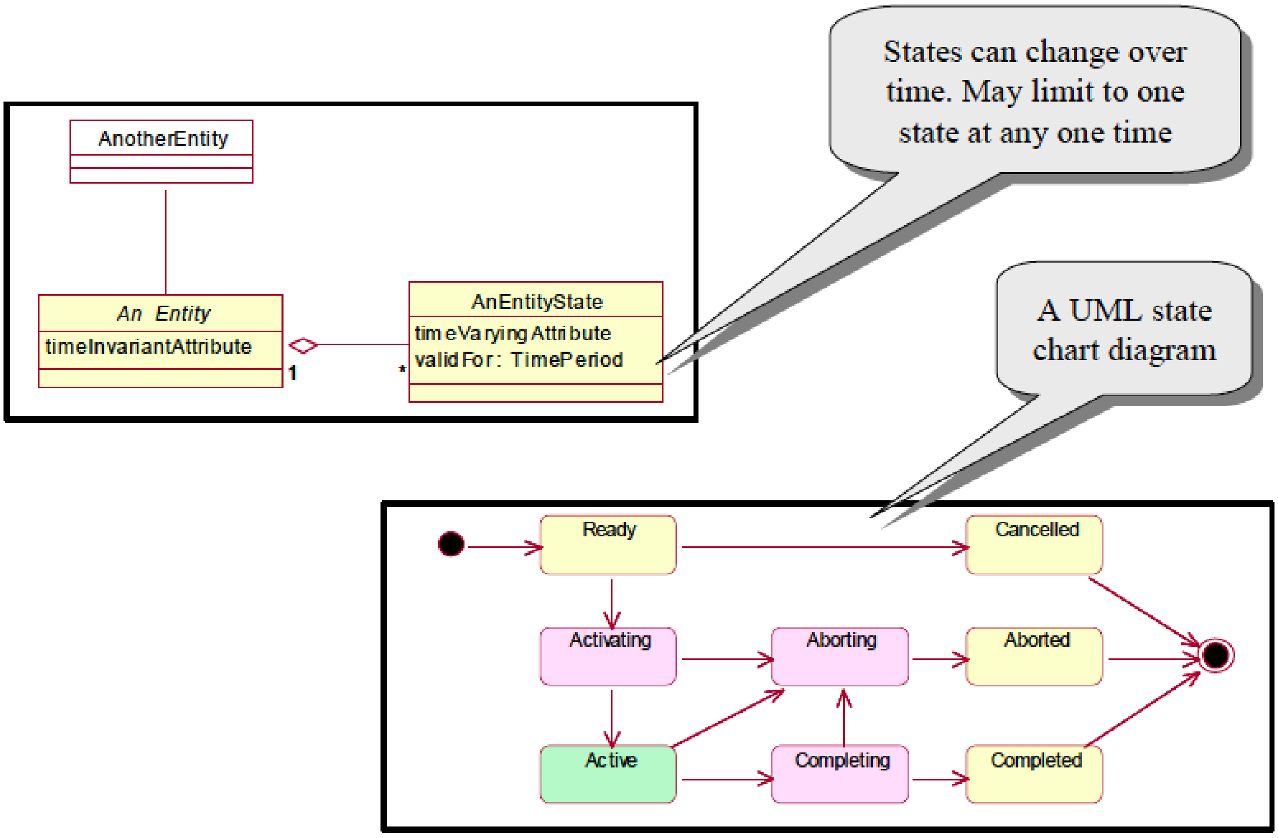


Figure 18. SID temporal state entity

Self Relationship – The Self Relationship pattern is used when an instance of an entity may have a relationship to other instances of the same entity. For instance, a family tree could be formed by linking individuals to their parents.

Relationship Type	**Description**
Dependency	This is where the two Entities have some starting or finishing dependency. E.g. Activity 2 cannot start until Activity 1 is complete.
Succession	This is where one or more Entities are replaced by one or more Entities. This is an abstract relationship and one of the concrete types listed below must be used.

Substitution	This is a one for one replacement. E.g. Activity 1 is no longer valid and has been replace by Activity 1A.
Division	This is a one for many replacement. E.g. Activity 1 is no longer valid and has been replaced by Activities 1A, 1B and 1C.
Fusion	This is a many for one replacement. E.g. Activities 1A, 1B and 1C are no longer valid and have been replaced by Activity 1Z.

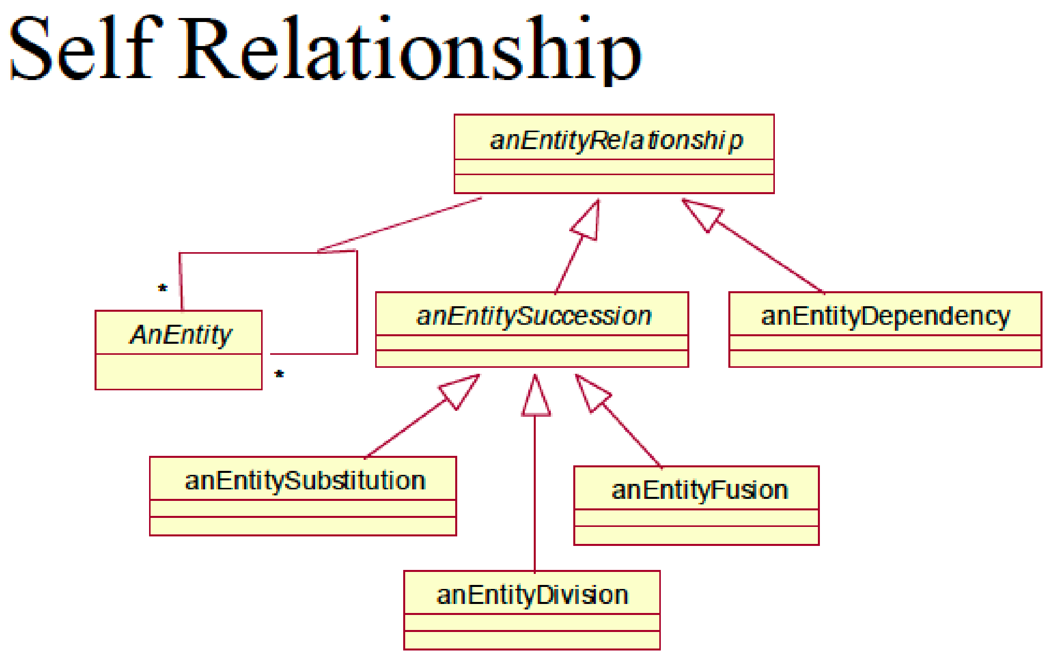


Figure 19. SID self-relationship

4.2.3. YANG

All YANG definitions are contained within "modules". This is exploited by YANG [rfc6020], to allow modelling of data in multiple hierarchies, where data may have more than one top-level node. In essence, a NETCONF server may implement a number of modules, allowing multiple views of the same data, or multiple views of disjoint subsections of the device's data.

Models that have multiple top-level nodes are sometimes convenient. However this should not be confused with multiple inheritance, as these node definitions are essentially distinct from each other, and no type inference can be made upon them. There is a proposal to add data-type inheritance to YANG [rfc6095].

Modules and sub-modules

A module contains three types of statements: module-header statements, revision statements, and definition statements. The module header statements describe the module and give information about the module itself. The revision statements give information about the history of the module and the definition statements are the body of the module where the data model is defined.

A module always includes a name-space declaration. A module may be divided into sub-modules. The "include" module header statement allows a module or sub-module to include material in sub-modules. The name-space of all types defined in the module, and any sub-modules is the including module name-space. The external view remains that of a single module, regardless of the presence or size of its sub-modules.

An "import" module header statement allows references to material defined in other modules. These imported types have the name-space of the module that defined them, but can be referred to in the importing module.

Types of nodes for data modelling

YANG defines four types of nodes for data modelling. In each of the following subsections, the example shows the YANG syntax as well as a corresponding NETCONF XML representation.

Leaf node. A leaf node contains simple data like an integer or a string. It has exactly one value of a particular type and no child nodes.

YANG Example:

```
        leaf host-name {
            type string;
            description "Host name for this system";
        }
```

NETCONF XML Example:

```
<host-name>my.example.com</host-name>
```

Leaf-list node. A leaf-list is a sequence of leaf nodes with exactly one value of a particular type per leaf.

YANG Example:

```
     leaf-list domain-search {
         type string;
         description "List of domain names to search";
     }
```

NETCONF XML Example:

```
<domain-search>high.example.com</domain-search>
<domain-search>low.example.com</domain-search>
<domain-search>everywhere.example.com</domain-search>
```

Container nodes. A container node is used to group related nodes in a sub-tree. A container has only child nodes and no value. A container may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).

YANG Example:

```
    container system {
         container login {
             leaf message {
                 type string;
                 description
                     "Message given at start of login session";
             }
         }
     }
```

NETCONF XML Example:

```
<system>
  <login>
    <message>Good morning</message>
  </login>
</system>
```

List nodes. A list defines a sequence of list entries. Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs. A list can define multiple key leafs and may contain any number of child nodes of any type (including leafs, lists, containers etc.).

YANG Example:

```
    list user {
        key "name";
        leaf name {
            type string;
        }
        leaf full-name {
            type string;
        }
        leaf class {
            type string;
        }
    }
```

NETCONF XML Example:

```
<user>
  <name>glocks</name>
  <full-name>Goldie Locks</full-name>
  <class>intruder</class>
</user>
<user>
  <name>snowey</name>
  <full-name>Snow White</full-name>
  <class>free-loader</class>
</user>
<user>
  <name>rzell</name>
  <full-name>Rapun Zell</full-name>
  <class>tower</class>
</user>
```

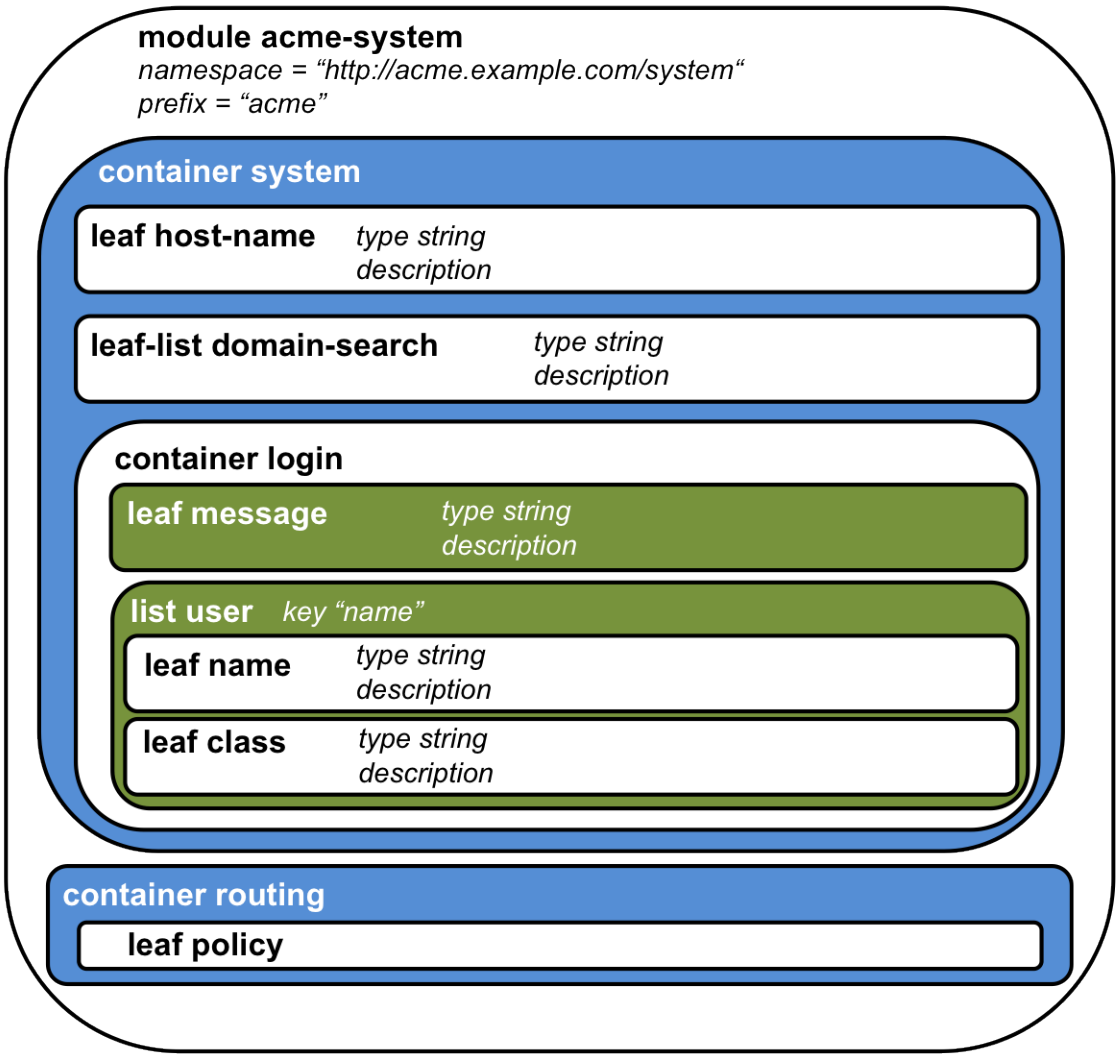


Figure 20. Example of YANG information model

4.3. Managed Object languages

In order, to generate a managed object model, a modelling language is used to describe the model, attributes of the classes, allowed relationships and known notifications. This section attempts to summarise some of the key features of the leading network management modelling languages.

4.3.1. GDMO (OSI, ITU x.700)

The [x722] specification (Guidelines for Defining Managed Objects, GDMO) defines a set of templates for the representation of various aspects of a managed object class definition and its associated naming structure.

The start of a template consists of a *template-label* and a *TEMPLATE-NAME*. A template contains one or more constructs, each of which is named by a *CONSTRUCT-NAME* and each may have a *construct-argument*. The *construct-argument* may in turn consist of a number of elements, as called for by the definition of the particular construct. Each instance of use of a template declares a unique *template-label* by which that instance may be referenced from other templates, and if the *REGISTERED AS* construct is present, assigns a value of an ASN.1 object identifier under which the instance has been registered. The semicolon character is used to mark the end of each construct (except REGISTERED AS and DEFINED AS) and to mark the end of a template.

Syntax overview

Managed Object Class template

The Managed Object Class template forms the basis of the formal definition of a managed object. Elements in the template allow the class to be placed at the appropriate node of the inheritance tree, the various characteristics of the class to be specified, and the behaviour of the class to be defined. The major elements of the definition are shown below.

```
<class-label> MANAGED OBJECT CLASS
[DERIVED FROM <class-label> [,<class-label>]* ;
]
[CHARACTERIZED BY <package-label> [,<package-label>]* ;
]
[CONDITIONAL PACKAGES <package-label> PRESENT IF condition-definition
[,<package-label> PRESENT IF condition-definition]* ;
]
REGISTERED AS object-identifier ;
supporting productions
condition-definition -> delimited-string
```

Package template

This template allows a package consisting of a combination of behaviour definitions, attributes, attribute groups, operations, notifications and parameters to be defined for subsequent insertion into a Managed Object Class template under the CHARACTERIZED BY or CONDITIONAL PACKAGES constructs. The major elements of the definition are shown below.

```
<package-label> PACKAGE
```

```
[BEHAVIOUR <behaviour-definition-label> [,<behaviour-definition-label>]* ;
]
[ATTRIBUTES <attribute-label> propertylist [<parameter-label>]*
[,<attribute-label> propertylist [<parameter-label>]*]* ;
]
[ATTRIBUTE GROUPS <group-label> [<attribute-label>]* [,<group-label>
[<attribute-label>]*]* ;
]
[ACTIONS <action-label> [<parameter-label>]* [,<action-label>
[<parameter-label>]*]* ;
]
[NOTIFICATIONS <notification-label> [<parameter-label>]* [,<notification-
label>
[<parameter-label>]*]* ;
]
[REGISTERED AS object-identifier] ;
supporting productions
propertylist -> [REPLACE-WITH-DEFAULT]
[DEFAULT VALUE value-specifier]
[INITIAL VALUE value-specifier]
[PERMITTED VALUES type-reference]
[REQUIRED VALUES type-reference]
[get-replace]
[add-remove]
value-specifier -> value-reference | DERIVATION RULE <behaviour-
definition-label>
get-replace-> GET | REPLACE | GET-REPLACE
add-remove -> ADD | REMOVE | ADD-REMOVE
```

Parameter template This template permits the specification and registration of parameter syntaxes and associated behaviour that may be associated with particular attributes, operations and notifications within the Package, Attribute, Action and Notification templates. The type specified in a Parameter template is used to fill in an ANY DEFINED BY x construct in a management PDU, where x is a field in the PDU that carries the object identifier assigned to the parameter. This mechanism is, for example, applicable to the definition of:

- processing failures;
- parameters of action requests/responses;
- parameters of notification requests/responses.

The major elements of the definition are shown below.

```
<parameter-label> PARAMETER
CONTEXT context-type ;
syntax-or-attribute-choice ;
[BEHAVIOUR <behaviour-definition-label>
[,<behaviour-definition-label>]* ;
]
[REGISTERED AS object-identifier] ;
supporting productions
context-type -> context-keyword |
ACTION-INFO |
ACTION-REPLY |
EVENT-INFO |
EVENT-REPLY |
SPECIFIC-ERROR
context-keyword -> type-reference.<identifier>
syntax-or-attribute-choice -> WITH SYNTAX type-reference |
ATTRIBUTE <attribute-label>
```

Name binding template

This template allows alternative naming structures to be defined for managed objects of a given managed object class by means of name bindings. A name binding allows an attribute to be selected as the naming attribute that shall be used when a subordinate object which is an instance of a specified managed object class is named by a superior object which is an instance of a specified managed object class or other object class, such as a Directory object class.

If a given name binding is used, the attribute identified as the naming attribute shall be present in the subordinate object. The naming attribute is used to construct the Relative Distinguished Name (RDN) of subordinate objects of that class. An RDN is constructed from the object identifier assigned to that attribute type and the value of the instance of the attribute. The Distinguished Name of the subordinate object is obtained by appending its RDN to the Distinguished Name of its superior object. Name bindings are not considered to be part of the definition of either of the classes that they reference. A given subordinate managed object class may have more than one name binding associated with it. The set of name bindings defines the set of possible naming relationships with superior objects and the set of managed object classes from which subordinate objects may be instantiated.

A name binding may also be defined to apply to all subclasses of the specified superior object class or all subclasses of the specified subordinate object class, or both.

```
<name-binding-label> NAME BINDING
SUBORDINATE OBJECT CLASS <class-label> [AND SUBCLASSES];
NAMED BY SUPERIOR OBJECT CLASS <class-label> [AND SUBCLASSES];
WITH ATTRIBUTE <attribute-label> ;
[BEHAVIOUR <behaviour-definition-label>
[,<behaviour-definition-label>]* ;
]
[CREATE [create-modifier [,create-modifier]]
[<parameter-label>]* ;
]
[DELETE [delete-modifier] [<parameter-label>]* ;
]
REGISTERED AS object-identifier ;
supporting productions
create-modifier -> WITH-REFERENCE-OBJECT |
WITH-AUTOMATIC-INSTANCE-NAMING
delete-modifier -> ONLY-IF-NO-CONTAINED-OBJECTS |
DELETES-CONTAINED-OBJECTS
```

Attribute template

This template is used to define individual attribute types. These definitions may be further combined by the attribute group template where attribute groups are required. The major elements of the definition are shown below.

```
<attribute-label> ATTRIBUTE
derived-or-with-syntax-choice ;
[MATCHES FOR qualifier [, qualifier]* ;
]
[BEHAVIOUR <behaviour-definition-label> [,<behaviour-definition-label>]* ;
]
[PARAMETERS <parameter-label> [,<parameter-label>]* ;
]
[REGISTERED AS object-identifier] ;
supporting productions
qualifier -> EQUALITY | ORDERING | SUBSTRINGS |
SET-COMPARISON | SET-INTERSECTION
derived-or-with-syntax-choice -> DERIVED FROM <attribute-label> |
WITH ATTRIBUTE SYNTAX type-reference
```

Attribute group template

This template allows attribute groupings to be defined; such groupings are applicable to situations where it is desirable to operate upon the collection of attributes that are members of the group. The behaviour definitions for a given managed object class define the meaning of Get attribute value and Replace with default value operations when applied to attribute groups. Each member of the group shall itself be defined as a single- or set-valued attribute type.

```
<group-label> ATTRIBUTE GROUP
[GROUP ELEMENTS <attribute-label> [,<attribute-label>]* ;
]
[FIXED ;
]
[DESCRIPTION delimited-string ;
]
REGISTERED AS object-identifier ;
```

Behaviour template

This template is used to define behavioural aspects of managed object classes, name bindings, parameters and attribute, action and notification types. The Behaviour template is intended to permit extension provisions, but behaviour specifications shall not change the semantics of previously-defined information. If information is left undefined, the behaviour definition shall be explicit about what is undefined.

```
<action-label> ACTION
[BEHAVIOUR <behaviour-definition-label>
[,<behaviour-definition-label>]* ;
]
[MODE CONFIRMED ;
]
[PARAMETERS <parameter-label> [,<parameter-label>]* ;
]
[WITH INFORMATION SYNTAX type-reference ;
]
[WITH REPLY SYNTAX type-reference ;
]
REGISTERED AS object-identifier ;
```

Notification template This template is used to define the behaviour and syntax associated with a particular Notification type. Notification types defined by means of this template may be carried in event reports by the M-EVENT REPORT service. The major elements of the definition are shown below.

```
<notification-label> NOTIFICATION
[BEHAVIOUR <behaviour-definition-label> [,<behaviourdefinition-
label>]* ;
]
[PARAMETERS <parameter-label> [,<parameter-label>]* ;
]
[WITH INFORMATION SYNTAX type-reference
[AND ATTRIBUTE IDS <field-name> <attribute-label>
[,<field-name> <attribute-label>]*
] ;
]
[WITH REPLY SYNTAX type-reference ;
]
REGISTERED AS object-identifier ;
```

Data types

GDMO uses ASN.1 as the data-type definition language. An overview is included in a separate section below.

4.3.2. SID

The SID is an analysis model which is focused on representing real world objects which are of interest to business. An analysis model includes things in which the business is interested (domain entities), how they are related to one another (associations), and key details about those things which help to define them unambiguously (domain-level-attributes). Using analysis techniques can provide a more detailed understanding of business concepts and aid in defining business processes more precisely. The SID should be used:

- As a starting point for internal modelling work, applications and messages between software components or database schemas
- To help in defining a common business terminology, e.g. for integration activities
- To help in understanding business concepts and their relationships

The SID is an information model and a data model, so designing applications using the SID eases integration between heterogeneous applications. SID facilitates the identification of domain objects and their associated relationships when designing an NGOSS solution. SID can reduce project analysis and design phases as it provides a readily available domain model based on best practice that can be directly applied to new projects without requiring significant modifications. The SID framework provides extensive documentation to describe not only the structure, but also the behaviour of domain-managed business entities. Business domain-managed entities in SID are modelled using UML diagrams and design patterns that can be easily imported to UML-based tools.

Information models are abstract models that describe the internal characteristics of a system that can be described as the 'business view' perspective. Data models specified by an enterprise architect are applicable to a specific enterprise and are a technology and application independent means of representing a 'system view' perspective of the logical information model. In SID, the data model aspect is not an obvious modelling artefact. There is a requirement to extend the SID model, identify and associate mappings once a processes elements are decomposed. It may prove difficult to maintain the eTOM to SID mappings at more detailed levels at process element decomposition /SID extensions. There are a number of limitations to the SID model in regards to modelling network environments. Firstly, in the SID model business-to-network translation (and vice versa) is not available in any palpable form. Secondly, required concepts such as context are not available in the model. Finally, state machines are not provided to model the behaviour of entities. There are currently no freely available open source tools available to support the conversion of abstract SID domain models into deployable concrete syntaxes.

Syntax overview

In the SID model there are three principal types of ManagedEntities, these are Product, Resource, and Service shown in the figure below. They are each types of managed entities, and exist in close support of each other. A Product can be implemented by zero or more CustomerFacingServices and zero or more PhysicalResources. Note that all three have business uses and, more importantly, either perform management functions and/ or collect management information.

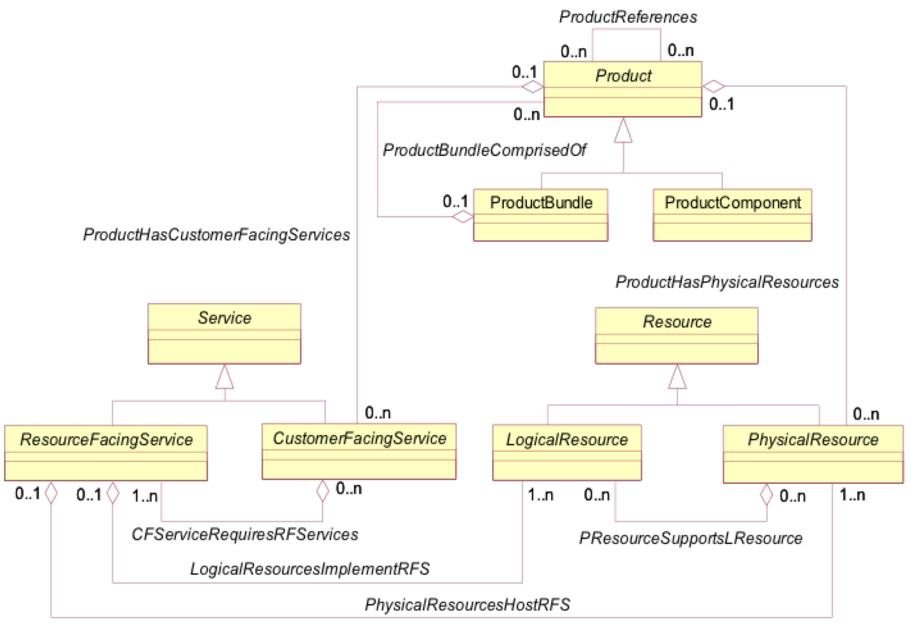


Figure 21. Relationship among Product, Service, and Resource domain business entities.

Product - In SID, Products are things (tangible or intangible) which enterprises, such as service providers, market, sell or lease to customers to create profit. Business entities in the Product domain have a close relationship with business entities in the Service and Resource domains. While Product business entities represent what the market sees of a provider's offerings, Service and Resource business entities represent the realisation of the offerings from a provider's perspective. For example, a Broadband Internet ProductOffering is realised by two CustomerFacingServices, one is a bandwidth connectivity service to the provider's network; the second is a virtual connectivity service to the Internet Service Provider.

Resources - A Resource may be defined as an entity that is inherently manageable and makes up a Product. Network entities, like routers are complex and can be divided into its physical and logical aspects to ease complexity. Thus, the chassis, cards, and cables of a router (among other things) are all physical entities, whereas the services and protocols that a router is running, or the number of processes that it has defined, are logical entities. Therefore, the Resource domain facilitates two types of abstractions, these are PhysicalResource and LogicalResource.

PhysicalResource - A PhysicalResource has two main purposes: (1) to collect common attributes and relationships for all hardware, and (2) to provide a convenient, single point where relationships with other managed objects can be defined. PhysicalResource has two main subclasses, PhysicalDevice and Hardware. A PhysicalDevice represents hardware devices that can be managed. Examples of this class include routers and switches, computers, and other end-devices that are managed. Hardware represents any type of physical entity that has a distinct physical identity and exists as an atomic unit. Hardware consists of Equipment (e.g., a LineCard that performs routing), EquipmentHolders (e.g., a Chassis or some other managed entity whose purpose is to “hold” Equipment), and AuxiliaryComponents (physical components that are required by the “Device” to operate correctly, but whose individual purposes are orthogonal to the main purpose of the device).

LogicalResource - LogicalResources require a PhysicalResources to be hosted on. Conceptually, a LogicalDevice represents the intelligence embedded in a particular Resource. This intelligence governs how a particular Resource (or even set of Resources behaves). A LogicalDevice entity is used to contain the different entities that collectively provide intelligence to a Resource. A LogicalDevice is an abstract base class that describes different logical aspects of devices (e.g., services that are running, or processes that are instantiated) that constitute a Product. It has two main purposes: (1) to collect common attributes and relationships for all logical entities, and (2) to provide a convenient, single point where relationships with other managed objects can be defined.

Service - Services need one or more resources to support them. Services are split into CustomerFacingServices and ResourceFacingServices. A CustomerFacingService is an abstraction that defines the characteristics and behaviour of a particular Service as seen by the customer. This means that the customer purchases, leases, uses and/or is otherwise directly aware of this type of Service. A ResourceFacingService is an abstraction that defines the characteristics and behaviour of a particular Service that is not directly seen or purchased by the customer. ResourceFacingServices are “internal” Services that are required to support a CustomerFacingService. The customer purchases CustomerFacingServices, and is unaware of the ResourceFacingServices which support the CustomerFacingService(s) that is purchased directly by the customer.

CustomerFacingService - A CustomerFacingService is an abstraction that defines the characteristics and behaviour of a particular Service as seen by the customer. This means that a customer purchases and/or is directly aware of this type of Service. This is in direct contrast to the definition of a ResourceFacingService, which supports a CustomerFacingService, but is not seen or purchased directly by the customer. For example, a VPN is an example of a CustomerFacingService, while the sub-services that perform different types of routing between network devices making up the VPN are examples of ResourceFacingServices. This is because a customer can order a VPN, but cannot order the services that are used to realise the VPN (e.g., MPLS, BGP, etc.).

ResourceFacingService - The ResourceFacingServices are hosted by one or more PhysicalResources and are implemented by one or more LogicalResources. All Services, regardless of whether they are CustomerFacingServices or ResourceFacingServices, are made up of changeable as well as invariant attributes, methods, relationships, and constraints. A ServiceSpecification defines the invariant characteristics and behaviour of a Service. It can be conceptually thought of as a template that different Service instances can be instantiated from. Each of these Service instances will have the same invariant characteristics. However, the other characteristics of the instantiated Service will be specific to each instance.

Data types

Table 1. SID Data types

Attribute Name	Description	Data Type	Required/ Optional
ID	A unique identifier for the CharacteristicSpecification.	String	Required
name	A word, term, or phrase by which a CharacteristicSpecification is known and distinguished from other CharacteristicSpecifications.	String	Required
description	A narrative that explains in detail what the CharacteristicSpecification is.	String	Optional
unique	An indicator that specifies if a value is unique for the specification. Possible values are;	String	Optional

Attribute Name	Description	Data Type	Required/ Optional
	"unique while value is in effect" and "unique whether value is in effect or not"		
valueType	A kind of value that the characteristic can take on, such as numeric, text, and so forth.	String	Required, if the specification is not a composite
minCardinality	The minimum number of instances a CharacteristicValue can take on. For example, zero to five phone numbers in a group calling plan, where zero is the value for the minCardinality.	Integer	Optional
maxCardinality	The maximum number of instances a CharacteristicValue can take on. For example, zero to five phone numbers in a group calling plan, where five is the value for the maxCardinality.	Integer	Optional
extensible	An indicator that specifies that the values for the characteristic can be extended by adding new values when instantiating a characteristic for an Entity.	Boolean	Optional
derivation Formula	A rule or principle represented in symbols, numbers, or letters, often in the form of an equation used to derive the value of a characteristic value.	String	Optional
validFor	The period of time for which a CharacteristicSpecification is applicable.	Time Period	Optional

4.3.3. YANG (RFC 6020)

The following section gives some overview examples of YANG [rfc6020] and its features.

Syntax overview

Every YANG module has a header part. The following listing shows a typical module header declaration, including the name-space declaration, revision info, and contact details. This module imports (i.e. references types declared in another module *ietf-inet-types*. Note that imported modules data types are referred to with a name-space prefix.

```
module pristine-dns-resolver {
   namespace "http://www.ict-pristine.eu/yang/pristine-dns-resolver/1.0"
   prefix "pristine-res"

   import "ietf-inet-types" { prefix "inet"; }

   organisation "Pristine consortium";
   contact "no-reply@ict-pristine.eu";
   description "A YANG DNS resolver for PRISTINE"

   revision "2014-06-25" {
     description "Example revision"
   }
```

Another interesting aspect is the feature capability. The following listing shows a trivial example, where an additional leaf attribute is added to an existing YANG model. A "feature" is a facility to mark a portion of the overall model as optional. Supported features (from a device) are listed when a session connection is established. In the following example, a status attribute is added to the existing yang container *dns/resolver_nameserver* (perhaps declared in another module).

```
feature "pristine-status" {
  description "An extension to add a special PRISTINE status"
}

typedef server-status {
  type enumeration {
    enum unknown;
```

```
      enum answering;
      enum failed;
    }
  }

  augment "/dns/resolver/nameserver" {
    leaf status {
      type server-status;
      config false;
      if-feature "pristine-status";
    }
  }
```

The additional leaf node (attribute) is only available if the device supports "pristine-status" capability. The attribute is marked as an enumeration type, that is not part of the configuration information.

YANG also allows additional restrictions to be imposed on defined types as shown in the following snippet. Here, a default value is applied so as the port number defaults to 53. This can be argued as support for **single inheritance**, where the specialised type has additional constraints applied to it, or can have an additional leaf node in the derived type.

```
  leaf nameserver {
    uses server-address {
      refine port { default 53; }
    }
  }
```

Data types

YANG has a set of built-in types, similar to those of many programming languages, but with some differences due to special requirements from the management information model. Additional types may be defined, derived from those built-in types or from other derived types. Derived types may use sub-typing to formally restrict the set of possible values. The different built-in types and their derived types allow different kinds of sub-typing, namely length and regular expression restrictions of strings and range restrictions of numeric types.

Table 2. List of YANG built-in types

Category	Types	Description
Integer	{u,}int {8,16, 32,64}	Signed and unsigned integers of different sizes or a integer range.
Decimal	decimal64	The subset of the real numbers, which can be represented by decimal numerals.
String	string	Human-readable strings in YANG. Allows legal characters of Unicode and ISO/IEC 10646
Boolean	boolean	A boolean value true or false
Enumeration	enumeration	The enumeration built-in type represents values from a set of assigned names (strings)
Bits	bits	Represents a bit set/mask. It is a set of flags identified by small integer position numbers starting at 0. Each bit number has an assigned name
Binary	binary	Represents any binary data, i.e., a sequence of octets.
Reference	leaf-ref	The leaf-ref type is used to reference a particular leaf instance in the data tree.
Reference	identity-ref	The identity-ref type is used to reference an existing identity
Reference	instance-identifier	The instance-identifier built-in type is used to uniquely identify a particular instance node in the data tree
Other	empty	The empty built-in type represents a leaf that does not have any value, it conveys information by its presence or absence
Other	union	The union built-in type represents a value that corresponds to one of its member types

YANG can define derived types from base types using the "typedef" statement. A base type can be either a built-in type or a derived type, allowing a hierarchy of derived types. A derived type can be used as the argument for the "type" statement.

4.3.4. Abstract Syntax Notation 1

Abstract Syntax Notation One (ASN.1) [x690] is a standard and notation that describes rules and structures for representing, encoding, transmitting, and decoding data in telecommunications and computer networking. The formal rules enable representation of objects that are independent of machine-specific encoding techniques.

Its formal notation style makes it possible to automate the task of validating whether a specific instance of data representation abides by the specifications.

Syntax overview

ASN.1 definitions are contained within "modules".

The basic types are described in the next section, however more complex types can defined by the following constructs:

- Sequence: ordered collection of variables of different type
- Sequence of: ordered collection of variables of the same type
- Set: unordered collection of variables of different types
- Set of: unordered collection of variables of the same type
- Choice: collection of distinct types from where only one can be present at a time

The following example gives the ASN.1 notation for the type *RDNSequence* as defined in [x501], which is used to identify entities in a X.500 directory. The *RDNSequence* type gives a path through an X.500 directory tree starting at the root. *RDNSequence* is a SEQUENCE OF type consisting of zero or more occurrences of *RelativeDistinguishedName*.

A *RelativeDistinguishedName* is a SET OF type consisting of zero or more occurrences of *AttributeValueAssertion*. The *AttributeValueAssertion* type assigns a value to some attribute of a relative distinguished name, such as country name or common name. AttributeValueAssertion is a

SEQUENCE type consisting of two components, an *AttributeType* type and an *AttributeValue* type. The *AttributeType* and_AttributeValue_ types are simple ASN.1 types, OBJECT IDENTIFIER and ANY respectively.

In essence, it gives gives a unique name to an object relative to the object superior, by declaring what a specific attribute value should be.

```
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::=
  SET OF AttributeValueAssertion

AttributeValueAssertion ::= SEQUENCE {
   AttributeType,
   AttributeValue }
```

Data types

The following table lists the basic types in ASN.1. Some ASN.1 types are avoided, for example *UTCTime*, as *UTCTimes* uses 2 two digit year format. Thus DATE-TIME is preferred over other time-stamp types. Some additional guidance on ASN.1 types are offered online [3].

Table 3. List of ASN.1 built-in types

Category	Types	Description
Integer	INTEGER	Signed and unsigned integers, many have an associated legal range.
Decimal	REAL	Values have 2 possible forms: decimal format such as 245.34 or sequence format such as {mantissa 2.4534, base 10, exponent 2}
String	IA5String, UTF8String, VisibleString	Human readable strings either ASCII, or UTF8 with control characters.
Boolean	BOOLEAN	A boolean value either true or false
Enumeration	ENUMER-ATED	Represents values from a set of assigned names (strings)

[3] See http://www.oss.com/asn1/resources/asn1-made-simple/types.html

Category	Types	Description
Bits	BIT-STRING	Can have 3 possible forms: binary - '011'B, hex - '6'H, and named - windowOpen, fanOn.
Binary	OCTET-STRING	Represents any binary data, i.e., a sequence of octets.
Reference	OBJECT-IDENTIFIER	Globally unique identifier for something.
Other	NULL	The empty built-in type
Other	ANY	Any defined type

4.3.5. Google Protocol Buffers

The Google Protocol Buffers (GPB) [gpb] is a language- and platform-neutral mechanism for serialising and de-serialising structured data. It essentially involves a) an interface description language [4] - that must be used to describe the data subject to serialisation / de-serialisation - and b) a tool - that translates these descriptions into source-code that are used for serialising / de-serialising the corresponding messages, on both sender and recipient sides.

Messages (i.e. data structures) and services can be described in one or more files - also called .proto files - and translated into source-code with an ad-hoc tool, the protoc compiler. This "compiler" produces programming language code [5] for Java, Python and C++ as follows:

- C++: protoc generates a .h and .cc file from each .proto, with a class for each message type.
- Java: protoc generates a .java file with a class for each message type, as well as a special Builder class for creating message class instances.

[4] An interface description/definition language (IDL), is a specification language used to describe a software component's interface. IDLs describe an interface in a language-independent way, enabling communication between software components that do not share a language – for example, between components written in C++ and components written in Java.

[5] Various other language implementations are also available, refer to http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns for further information

- Python: protoc generates a module with a static descriptor of each message type, which is then used with a meta-class - in order to create the necessary (Python) data access class at runtime.

Syntax overview

A full overview of the GPB syntax is given at [gpb-syntax]. The subset of EBNF grammar for the interface description language (.proto), extracted from the source code, is as follows:

```
proto ::= ( message | extend | enum | import | package | option | ";" )*

import ::= "import" strLit ";"
package ::= "package" ident ( "." ident )* ";"
option ::= "option" optionBody ";"
optionBody ::= ident ( "." ident )* "=" constant

message ::= "message" ident messageBody
extend ::= "extend" userType "{" ( field | group | ";" )* "}"

enum ::= "enum" ident "{" ( option | enumField | ";" )* "}"
enumField ::= ident "=" intLit ";"

service ::= "service" ident "{" ( option | rpc | ";" )* "}"
rpc ::= "rpc" ident "(" userType ")" "returns" "(" userType ")" ";"

messageBody ::= "{" ( field | enum | message | extend | extensions |
                      group | option | ":" )* "}"

group ::= label "group" camelIdent "=" intLit messageBody
field ::= label type ident "=" intLit
        ( "[" fieldOption ( "," fieldOption )* "]" )? ";"
fieldOption ::= optionBody | "default" "=" constant

extensions ::= "extensions" extension ( "," extension )* ";"
extension ::= intLit ( "to" ( intLit | "max" ) )?

label ::= "required" | "optional" | "repeated"
```

Messages

Messages are specified - within one or more .proto files - with the `message` keyword, as in the following example:

```
package test;

message Contact {
  required string first_name = 1;
  required string last_name = 2;
  required string email = 3;
}

message Contacts {
  repeated Contact person = 1;
}
```

The example defines messages that would be used for exchanging contacts information between remote applications, using RPCs. The `Contacts` message holds a set of `Contact` entries, each of them binding together all the fields that define a single contact (i.e. her last-, first- names and associated email address).

The source-code can be automatically generated by simply feeding the aforementioned definition into the protoc tool. The obtained source-files will contain both serialisation and de-serialisation functionalities as well as accessors that have to be used to handle the serialised data (e.g. `has_email()`, `set_email()`, `clear_email()`).

Services

The serialisation/de-serialisation code produced by protoc can be used with third-parties RPC (Remote Procedure Call) systems/frameworks. Upon detecting a service definition - identified by the "service" keyword - the protoc compiler automatically generates interface code and stubs in the target language. E.g. The RPC service `ThisIsAService` exposing a method `ThisIsAMethod` that takes an `InputMessageType` message as input and returns a `OutputMessageType` message as output, that is defined as follows:

```
service ThisIsAService {
  rpc ThisIsAMethod (InputMessageType) returns (OutputMessageType);
}
```

In this case, the protoc tool will generate the code for the service interface as well as some stubs. The stub code forwards all calls to an abstract class that the user will have to extend in order to define it terms of the RPC system in-use. This way, the generated code provides the mechanisms (boilerplate code) without locking into any particular RPC implementation.

Data types

A scalar message field can have one of the types listed in the following table:

Table 4. List of GPB scalar types

Type	Notes	C++ Type	Java Type	Python Type [a]
double		double	double	float
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long [b]
uint32	Uses variable-length encoding.	uint32	int [c]	int/long [b]
uint64	Uses variable-length encoding.	uint64	long [c]	int/long [b]
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long [b]

Type	Notes	C++ Type	Java Type	Python Type [a]
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 228.	uint32	int [c]	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 256.	uint64	long [c]	int/long [b]
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long [b]
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/ unicode [d]
bytes	May contain any arbitrary sequence of bytes.	string	Byte String	str

[a] In all cases, setting values to a field will perform type checking to make sure it is valid.
[b] 64-bit or unsigned 32-bit integers are always represented as long when decoded, but can be an int if an int is given when setting the field. In all cases, the value must fit in the type represented when set.
[c] In Java, unsigned 32-bit and 64-bit integers are represented using their signed counterparts, with the top bit simply being stored in the sign bit.
[d] Python strings are represented as unicode on decode but can be str if an ASCII string is given (this is subject to change).

Encoding and decoding

A (serialised) message is essentially a series of key-value pairs. The binary version of a message just uses the field's number as the key – the name and declared type for each field can only be determined on the decoding end by referencing the message type's definition (i.e. the .proto file). When a message is serialised, the keys and values are concatenated into a byte stream while when it is de-serialised, the parser skips fields that it does not recognise. This way, new fields can be added to a message without breaking old programs [6].

4.4. Comparison

This section outlines a comparison criteria used to evaluate the various SoTA managed object models and languages.

4.4.1. RINA Demo Managed Object Model reuse

The following figure outlines a proposed RIB structure as outlined in the RIB-implementation-notes. These notes, while not a formal RINA specification, were used to derive a potential map of the suggested managed object model for RINA. This "demo" model was used to validate the RINA concept in working prototypes.

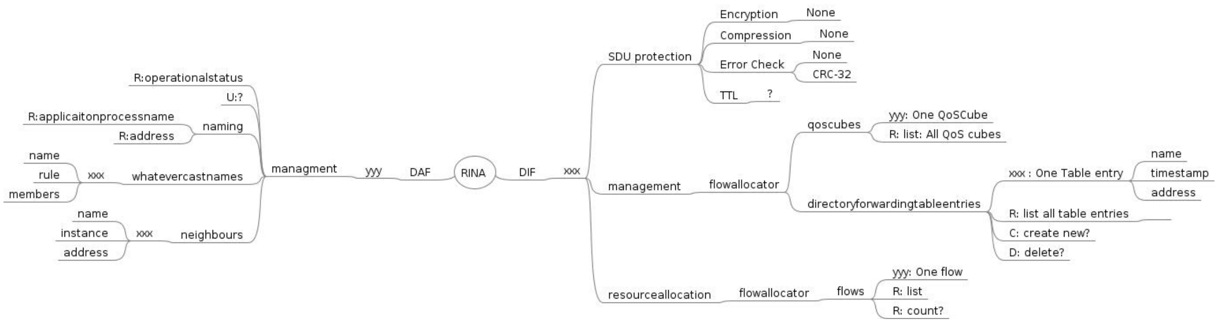


Figure 22. RINA Demo Managed object model mind map

As this demo RIB was specifically used for demonstration purpose it has some failings:

1. It is not a complete specification. It contains just enough of the RIB to allow working prototypes to be developed.
2. It was generated from a bottom-up approach. Thus it contains additional implementation detail, which may prove completely unnecessary in a more formal principle or concept driven constructions.

Therefore a more formal, and principle driven approach was used to derive a more complete RIB for RINA. This involved a wider review of existing

[6] To this end, the "key" for each pair in a wire-format message is actually two values – the field number from the .proto file, plus a wire type that provides just enough information to find the length of the following value.

network management work, and a selection of the best concepts and ideas from these specifications for inclusion in the proposed RINA RIB.

4.4.2. Language Selection Criteria

The RIB is a Management Information Base in the traditional network management system. As such the concepts that need to captured within the Managed object model are used as the basis for the comparison

Attributes
: Does the language support the concept of individually modifiable attributes? Does it support basic types only, or basic types and arbitrary constructs?

Notifications
: Does the language offer any explicit support for delivering notifications (or asynchronous events)? This is a mandatory requirement of the language.

Inheritance
: Does the supported modelling language support concepts of inheritance? For example, with *multiple* inheritance multiple managed object classes can be used to derive new class definitions. With *single* inheritance only a single object class can be used.

Containment
: Does the supported modelling language support a containment tree?

Behaviour
: How is behaviour specified? Is it a textual description or are more formal semantics applied? Does the language support arbitrary "operations" to be invoked?

Open-source tools
: Are there open-source tools available for the modelling language?

4.4.3. Language Comparison

Table 5. Language comparison

Body	Attributes	Notifications	Inheritance	Containment	Behaviour	Open-source tools
OSI - CMIP	Basic types	Yes	Multiple	Yes	Text	Yes

Body	Attributes	Notifications	Inheritance	Containment	Behaviour	Open-source tools
and X700	and arbitrary constructs					
IETF - NetConf and YANG	Basic types and arbitrary constructs	Yes	Single	Yes	Text	Yes
TM-Forum - SID	Basic types and arbitrary constructs	Event pattern	Multiple	Yes	Text	No
ASN.1	N/A	N/A	N/A	N/A	Yes	Yes
GPB	N/A	N/A	N/A	N/A	N/A	Yes

YANG only supports single inheritance. It allows multiple "modules" to represent an underlying resource; however this fact is not captured in the type system and therefore cannot be deduced when applying a protocol operation.

YANG’s extensibility with regard to adding extra Remote Procedure Call (RPC) can be seen as a double edged sword. It is convenient, however, in the current competitive environment it may be used to add vendor specific protocol operations. This will decrease the commonality of the model over time.

All the languages support a free text representation of the behaviour of the managed object. For more autonomic management, a stronger specification of the behaviour is desirable.

On the basis of these comparison factors, the OSIs GDMO language comes out as the front runner, followed by YANG as SID has no freely available open source tooling.

4.4.4. Supported encodings

This section offers a comparison of the supported encodings available for each of the languages. Given the variety of encodings some additional clarifications are necessary

SID is defined in terms of XML schemas, or XSDs, that can be used to facilitate the creation of an integration framework. Messages that are interchanged between applications are defined using the SID model components (business entities, attributes, and their associations) together with application-specific extensions to the SID. Extensions can include objects such as additional business entities, attributes and/or associations.

A mention is made of the way GPB uses "key-value" mechanisms in defining structured data, which allows easier extension, for example, adding an additional field afterwards. Each field is numbered so additional fields can be added and still allow backwards compatibility with the older version.

Table 6. Language encoding comparison

Data Language	Available encodings
OSI - GDMO	Uses ASN.1 for encodings
TM-Forum - SID	Text based XML schema (ecore)
IETF - YANG	Text based XML (Default), JSON ([netmod] Internet Draft)
OSI - ASN.1	Binary based (BER [x690], PER [x691]) and text based XML ([x693])
Google - GPB	Binary based

The supported encodings are seen as a secondary criteria after open source tools because the tools generate the code to perform the encoding and decoding operations.

4.4.5. Managed Object concepts reuse

The following is a list of recommendations of which concepts to re-use within the RIB managed object model.

- From YANG, the clear separation between "Configuration" data and "operational" state as outlined is highly **desirable**. It allows configuration to be backed up, or replicated without including unnecessary managed object state information.

- From YANG, the "feature" and "augment" capabilities offer equivalent functionality to GDMO’s conditional packages. This allows some aspects of the managed object to be optional, and supported only on certain versions of the RIB.
- Also from NETCONF, the explicit create-subscription and cancel-subscription offer a convenient way to create and cancel notification subscriptions. The create operation can take additional optional arguments, for example, a time window for which notifications are required, a filter expression, and a stream name.
- From SID, GDMO and GPB the object orientation of the managed object class specifications is highly **desirable**.
- From GPB, the use of name-value pairs for complex types is **desirable** as it allows extension without breaking backwards compatibility.

5. Part II

The following sections are are implementation specific.

Implementation specific sections

Proposed managed object modelling language

Proposed inheritance / type hierarchy for the managed objects.

Proposed containment hierarchy for the RIB itself.

Proposed architecture for implementing the DMS

6. Proposed RIB

The following section outlines the proposed Resource Information Base for the PRISTINE project.

The first step is to outline the language used to specify Managed Objects. This is followed by an overview of the RIB itself. The RIB is constructed with two object hierarchies:

Object *inheritance* hierarchy
The various utility managed object (MO) definitions required to represent the information and data manipulated during the operation of a DMS. This information is essentially static in the sense it is defined by the managed object language and can only be extended by adding additional managed object definitions at runtime.

Object *containment* hierarchy
This hierarchy captures the runtime information and instance related configuration and state of the DMS. This information contains all the necessary detail required to represent both DIFs and DAFs, their current state, and inter-relations.

6.1. Managed Object Model language

6.1.1. Introduction

The managed object model of RINA specifies the characteristics of RINA managed objects, their relationships and how they are named. It also provides tools to describe the attributes and behaviour of these objects, as well as discusses different options for object definition languages and its encodings.

All instances of managed objects of the same type are described by a *managed object class* definition. A managed object class definition defines, for an instance of the class:

- the properties or characteristics visible at the managed object boundary - these are called *attributes* and each property has a value.
- the *management operations* that may be applied (which are the same for all the objects: create, delete, read, write, start and stop).
- the *behaviour* the object exhibits in response to management operations.

- the *notifications* the object can produce.
    - NOTE: CDAP does not explicitly define notification operations (as opposed to CMIP), but they are an essential property of the managed object model and have to be modelled using the 6 operations available in CDAP (see below).
- its *position in the inheritance hierarchy* of the managed object class.
- all the possible *name bindings* of the managed object class (see below).

Managed objects, their attributes and their properties

A managed object is defined in terms of attributes it possesses, operations that may be performed upon it, notifications that it may issue and its relationships with other managed objects [x700]. An attribute of a managed object has an associated value, represented by a simple or a complex data type. Some attributes may be "public" i.e. directly accessible at the CDAP protocol level. Public attributes are individually-addressable attributes of the object, modelled as contained object classes.

Attributes types

The following basic types are defined:

- Integer number (signed and unsigned)
- Real number (signed and unsigned)
- String
- Enumeration
- Byte string
- Date
- Time

More complex types, can defined by the following:

- Sequence: ordered collection of variables of different type
- Sequence of: ordered collection of variables from same type
- Set: unordered collection of variables of different types [7]

[7] Due to the performance implications of Set when encoding and decoding, Set of and Sequence are preferred representations.

- Set of: unordered collection of variables of the same type
- Choice: collection of distinct types from where to choose a type

Inheritance, naming of object classes and attributes

Inheritance is the name given to a relationship between managed object classes that is used to produce clear and consistent managed object definitions and to encourage the controlled re-use of pieces of specification. It will be often desirable to define one object class as an extension of another. This process is termed specialization. The new class is said to be a subclass of the old class and the old class is said to be a superclass of the new class; the subclass is also said to inherit the characteristics of the superclass.

Managed object classes form an inheritance hierarchy, partially ordered by the subclass-superclass relationship. A single class is defined to be at the top of the hierarchy and so to be the superclass of all other classes. It is called **Top**. Any attributes defined for Top are called properties in that they are mandatory and common to all managed object classes. The following properties are currently defined for the Top object class:

- **objectClass** (string): The name of the managed object class.
- **objectName** (nameBinding, see later): The name binding used to instantiate the object
- **objectInstance** (integer): An integer that uniquely identifies the object instance within the RIB
- **publicAttributes** (sequence of integers): a set of Ids for objects that are synonyms for individually-addressable attributes of the object. For each, the model should state what operations are allowed, and what side-effects (if any) that access creates. (This will usually be pretty short — most properties would allow only a few operations at all, and there aren't usually side-effects for anything but start and possibly write, and those may have already been described in the description of the object itself.)

Each managed object class is assigned a name that is at least unambiguous in the name-space defined for RINA Management (Network and Layer Management) identifiers. Attributes can be named after the name of the object class they are part of; for example: given an object class called "objectclass1" and a property called "property1", the name of the property can be "objectclass2/property1".

Modelling notifications

The CDAP specification [cdap] highlight the fact that there is no explicit message type for notifications, for example, as can be found in CMIP. The general approach for implementing notifications is to model them as an attribute read or write. In order to capture Notifications as attributes a return type for the attribute must be decided upon, and this type is called *Event*. A proposal is made below for an inheritance hierarchy for *Event*.

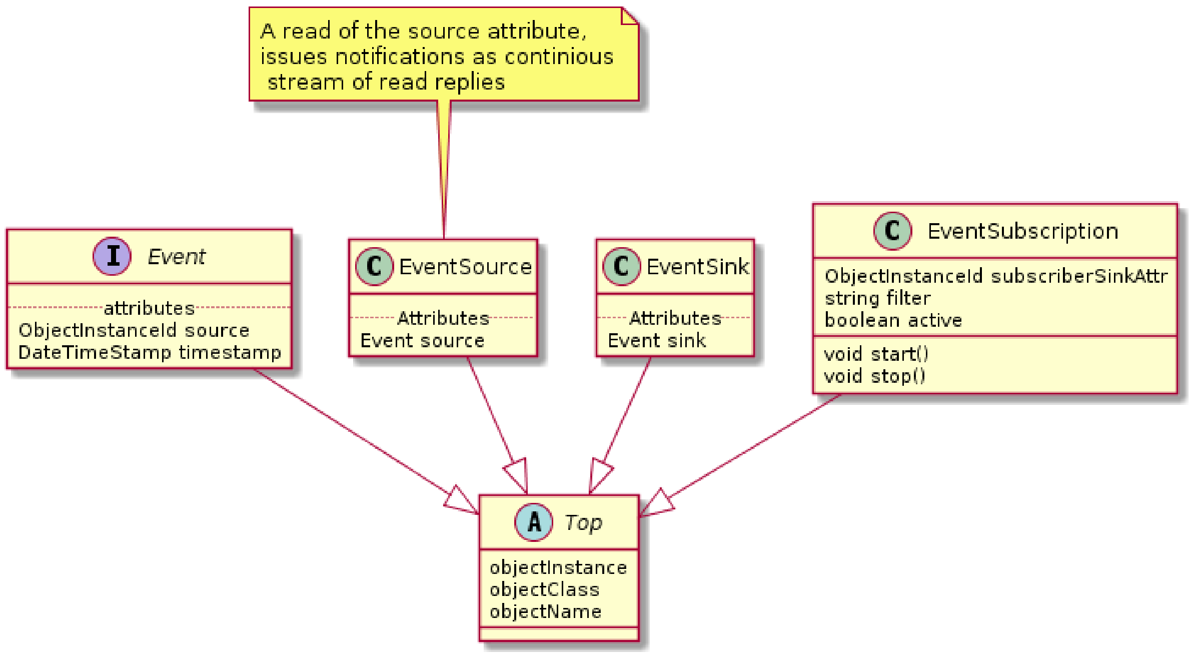


Figure 23. Notification inheritance hierarchy

From the figure it can be seen that Event is the base type for all notifications. Notifications can be generated for RIB object, creation, deletion, state change (1 or more attributes changed) or a custom triggered Event. An custom triggered event is defined as an arbitrary event generated when a specific trigger condition is satisfied. It refers to the object instance that generated the event, as well as local severity estimation.

Each approach to the notification subscription mechanism is described in the following paragraphs.

Modelling subscription as an attribute read

According to a CDAP specification, an attribute read can generate multiple replies. These multiple replies can be sent when a notification is triggered, where each reply corresponds to one notification. Conceptually, the attribute returns a Notification stream captured as an Event Managed Object or a subclass.

A disadvantage is RINA does not allow multiple inheritance or mix-ins. There it is not possible for a managed class to inherit from ManagedElement and EventSource at the same time. A workaround is to have a subclass in the containment tree capturing notification generation. This is depicted in the following sequence diagram.

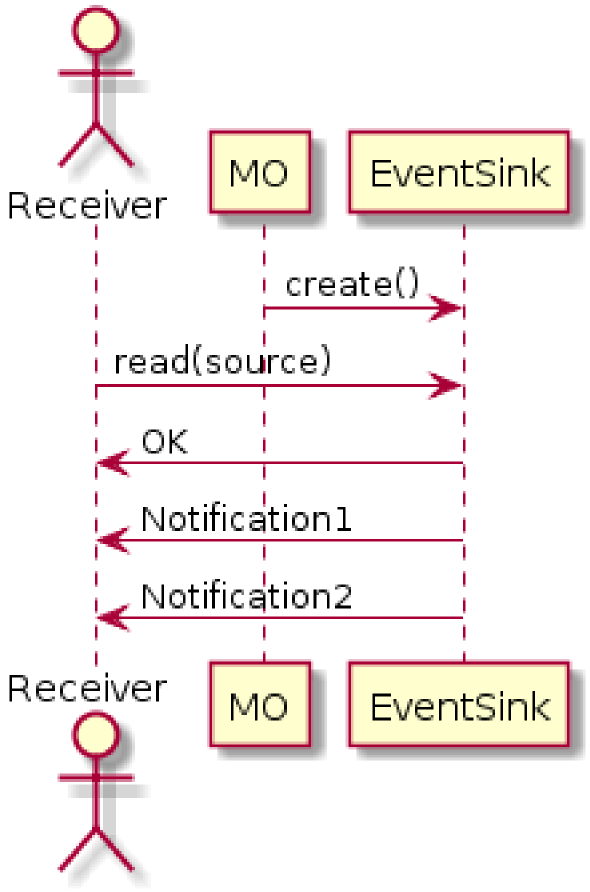


Figure 24. Subscribing for notifications

Modelling subscription as an attribute write

A second approach is to model the subscription for notifications as a CDAP attribute write. Specifically, a write to a specified attribute at the receiver side. From the receivers point of view the attribute contains the last "notification" from the sender. The receiver sends the object instance id (for notifications) to the sender, and the sender when a notification is generated, writes to the "sink" attribute of that object instance.

However, where multiple notification sources are sending notifications at the same time, it is possible to have the attribute value overwritten before the local manager can respond. Remember, the precise notification is the attribute value.

A recommendation is to add an "EventStream" datatype, where an *EventStream* is an *Event*, with a configurable length buffer and queue semantics.

A hybrid model allowing subscriptions

It is possible to improve on the notification semantics a little, by adding the concept of a subscription object. This object allows adding a "Filter" and a minimal interval to pre-discard notifications prior to sending. This managed object is shown in the following diagram

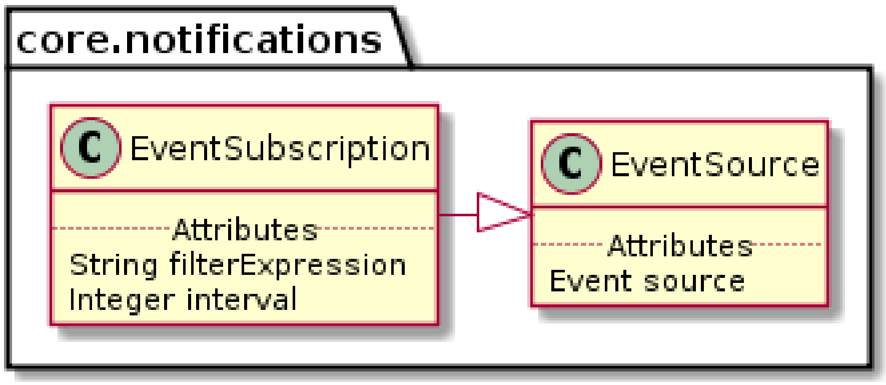


Figure 25. Subscription managed object

The corresponding sequence of events is then modified to take advantage of the Subscription object. Please note that the start and stop operations perform no function on a subscription object and are omitted.

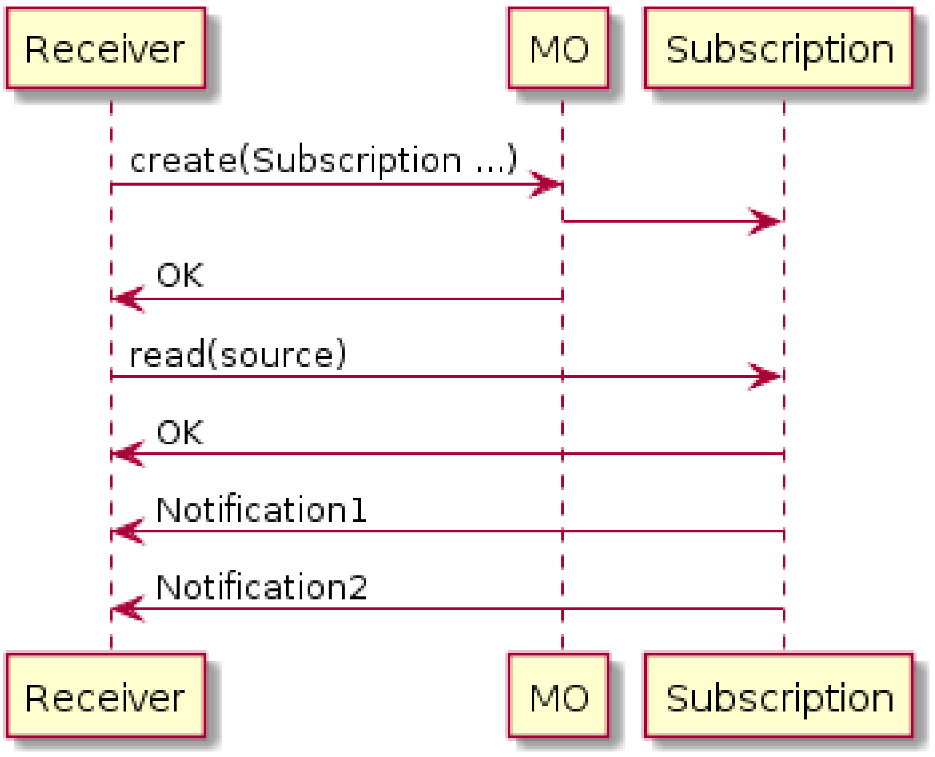


Figure 26. Subscribing for notifications with filtering

The receiver creates a Subscription object on the Managed Object of interest, in this case MO. The MO creates a child in its containment tree, with the attributes supplied in the create request. On receipt of the acknowledgement, the Receiver sends a read request for the source attribute on the Subscription managed object. This is acknowledged, and as the notifications are generated, they are passed to the receiver as multiple read responses.

A receiver can unsubscribe by sending a delete request to Subscription object.

Containment, object instance naming and name bindings

The naming of managed object instances is based on the idea that managed objects are contained within other managed objects. For example, an IPC Process is executing in a system, in order to refer to the IPC Process Managed object corresponding to the IPC Process is natural to say that it is contained in a system managed object. The main purpose of containment relationships is to give a naming structure, and so they may be set up in whatever way is more natural and convenient. A managed object has only one immediately containing object, and so the containment structure of managed objects is a tree.

The only consequence of the containment structure other than on names is that there is an existence relationship such that a contained managed object cannot continue to exist if the managing object containing it disappears.

Each managed object instance must have a name, so that it can be referred to in the protocol. The name is assigned when the managed object is created, whether by local action or as a result of the management *CREATE* operation.

The name of a managed object is constructed by going step-by-step down the containment tree, starting from the root of the tree. A single step, from containing object to contained object, gives a component of the name termed a *Relative Distinguished Name* (RDN). The fully qualified form is constructed by concatenating the relative RDNs. Within a given containing managed object, a contained managed object is identified by quoting the value of an attribute that is used to name it, by saying that it is the one which has a particular attribute with a particular value. Thus the RDN consists of a *attribute value assertion* (AVA) that names an attribute and quotes the value it must have (name="value"). Note that the RDN is a string, whose value is assigned to the *objectName* property of the object instance.

Name bindings

The specification that a particular managed object is contained within another managed object and is identified by a particular attribute is called a **name binding**. Name bindings may also contain other information, such as:

- Rules to be applied when creating and deleting managed objects, which may differ depending on their location in the containment tree.
- Behaviour specific to location in the containment tree

For example a name binding for an IPC Process managed object would specify that its container class is the System managed object, and that it can use the attribute *ipcProcessID* to name it.

6.1.2. Templates for describing managed objects

The following information is required to describe an object class:

- Name of the class
- Name of superclass

- Name and description of attributes
    - Including the definition of object classes of "public" attributes
- Description of all possible name bindings for the class
- Explanation of object behaviour for the create/delete/read/write/start/stop operations (what is allowed/not allowed, under what circumstances, outputs and side effects of the operations, …)
- Explanation of the notifications the object can generate, and under what conditions

There is a need for a set of templates that allow RIB architects to define their objects. Templates should have an unambiguous identity and should be able to refer to each other for importing managed object class definitions from one template to another.

As a working assumption (for now), templates will be written in natural language, some simple, graphical representation of the inheritance and the containment tree in order to help visualizing the structure.

6.2. Inheritance tree

The following section outlines the MO classes that help define the interface and behaviour of the Distributed Management System.

6.2.1. General inheritance tree

All the objects in the containment tree are also in the inheritance tree. However for clarification purposes, we have only drawn those that have specific importance in this tree. In other words, we have avoided those objects that are inheriting from Top but have no object inheriting from them. All super classes of objects are mentioned in their specific description.

An *ApplicationProcess* is the representation of a program executing in a processing system intended to accomplish some purpose. An Application Process contains one or more tasks or Application-Entities, as well as functions for managing the resources (processor, storage, and IPC) allocated to this program.

An *IPCProcess* represents a task within a processing system which uses IPC.

DAFManagement is responsible of DAF enrolment and overall management of the DMS. It makes use of *DIFManagement* to accomplish its management activities, i.e. create flows to send management operations and receive notifications.

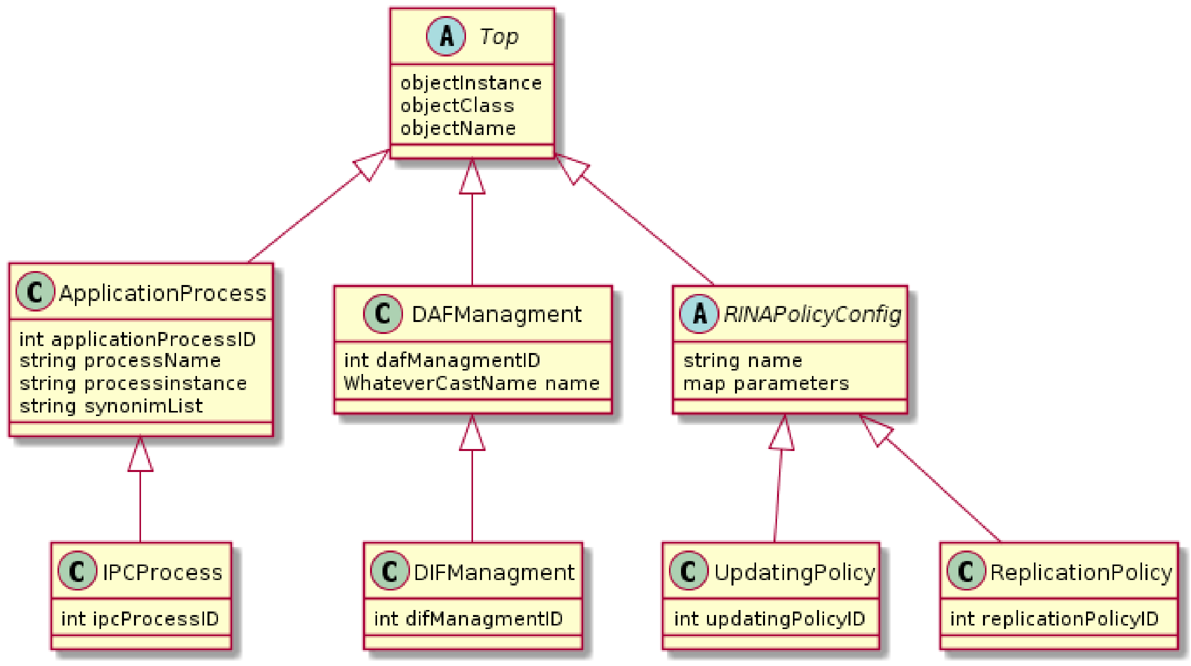


Figure 27. General DAF/DIF objects

RINAPolicyConfig is the base class for all policies, it inherits from Top and adds a *name* attribute containing a description and unique policy name. It also adds a parameters attribute containing a set of name-value pairs. These parameters can be used to configure the policies with runtime information. For example, an Encryption policy using pre-shared keys may require the key to be supplied when the policy instance is instantiated.

Two examples, of management policies that apply to the RIB daemon are given. An *UpdatingPolicy* is the policy that determines when, what and how to update the objects of the RIB. This object is part of a catalogue of all the possible *UpdatingPolicy* policies that the RIB Daemon can select. A *ReplicationPolicy* represents the policy which determines the replication of the information maintained in the RIB to other *IPC Process*. This object is also part of a catalogue.

6.2.2. SDU protection and forwarding table policies

SDU protection allows application data to be protected from compromise, and provides any required protection for SDUs. Any data corruption protection over the data and PCI including life-time guards (TTL, hop count) and/or encryption are performed by these policies. SDU Protection policies may be different on each allocated flow.

IntegrityCheckPolicy controls which integrity checks are performed on the SDUs. *EncryptionPolicy* controls which encryption algorithms are applied, if any. The *CompressionPolicy* determines if any message compression is applied and the *TTLPolicy* determines how many hops a SDU can take before being discarded.

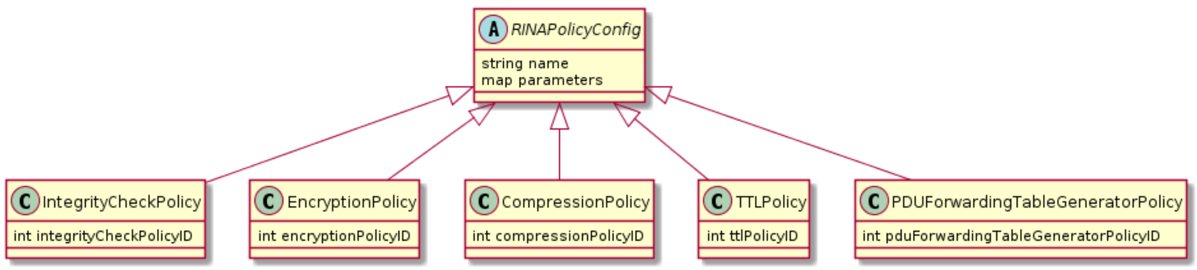


Figure 28. SDU protection policies

The *PDUForwardingGeneratorPolicy* represents the policy used for the generation of the routing tables. Specifically, populating the next hop table and the PDU forwarding table.

6.2.3. Security (Authentication and Access control)

The security policies control the security parameters of the DIF. It covers authentication, access control and how credentials are managed.

An *EnrollmentPolicy* determines how and when another IPC process can join the DIF.

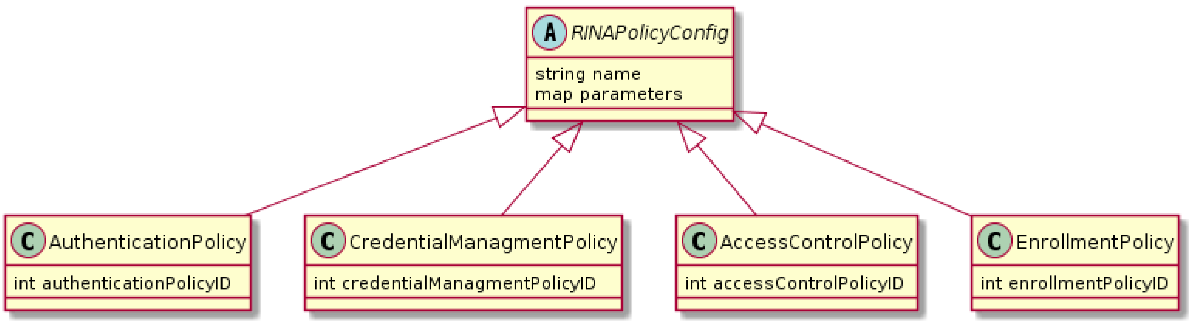


Figure 29. Security related policies

An *AccessControlPolicy* is used to determine who has access to a DIF, and can request flows on it. An *AuthenticationPolicy* is the policy that the application processes use to authenticate each other. It can range from none, to user/

password, Public Key Infrastructure (PKI) - based authentication, etc. The valid lifetime of these access and authentication credentials is governed by a *CredentialManagementPolicy*.

6.2.4. Routing and Resource Allocation

The figure below shows routing and resource allocation related policies.

The *RMTSchedulingPolicy* is one of the more important policies. This is the scheduling algorithm that determines the order input and output queues are serviced. *RMTQMonitorPolicy* controls what parameters are stored to manage the routing queues. This policy can be invoked whenever a PDU is placed in a queue and may keep additional variables that may be of use to the decision process of the RMT-Scheduling Policy and the MaxQPolicy. A *MaxQPolicy* is invoked when a routing queue reaches or crosses the maximum queue length allowed. For example, it may allow extra space to be allocated to the queue, flag a warning to the DMS, etc.

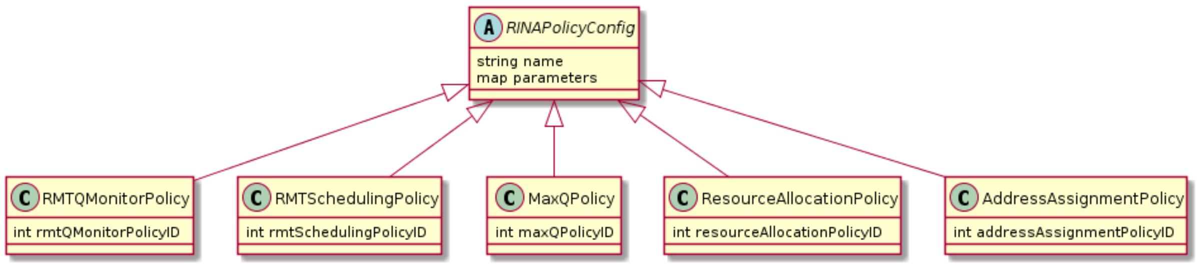

A RINAPolicyConfig
string name
map parameters
C RMTQMonitorPolicy
int rmtQMonitorPolicyID
C RMTSchedulingPolicy
int rmtSchedulingPolicyID
C MaxQPolicy
int maxQPolicyID
C ResourceAllocationPolicy
int resourceAllocationPolicyID
C AddressAssignmentPolicy
int addressAssignmentPolicyID

Figure 30. Resource allocation and flow related policies

ResourceAllocationPolicy represents the policy which determines how to distribute the resources between different members. Resources may be distributed evenly or assigned on a priority basis. An *AddressAssignmentPolicy* determines when to assign an address to an IPCProcess. For example, a lazy policy would wait until there are active flows, before assigning an address.

6.2.5. Flow related policies

The figure below shows policies governing Flow Allocation, and the behaviour when flows arrive.

An *UnknownFlowPolicy* is consulted when a PDU arrives that should be delivered in this IPC process but there is no corresponding active flow. The *NewFlowRequestPolicy* is used to convert an Allocate Request is into a create_flow request. Its primary task is to translate the request into the proper QoSclass-set, flow set, and access control capabilities.

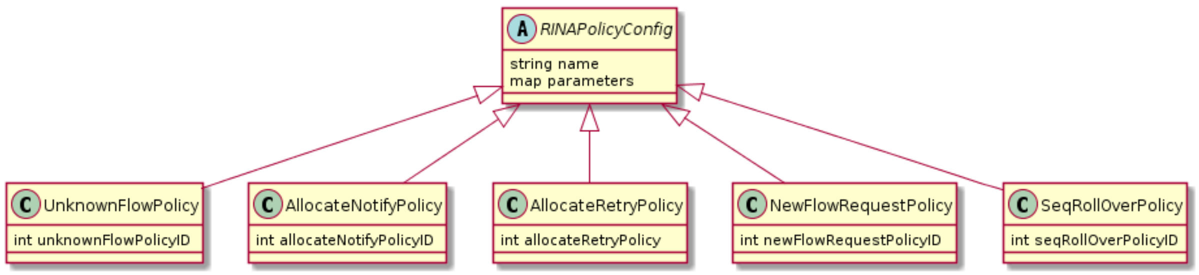


Figure 31. Resource allocation and flow related policies

An *AllocateRetryPolicy* is used when the destination has refused the create a flow, and the local flow allocator wishes to request creation again. In governs how many retries are made, and how long between retries. The *AllocateNotifyPolicy* determines when the requesting application is given an Allocate Response. In general, the choices are: once the request is determined to be well-formed and a create flow request has been sent, or with-held until a create flow response has been received or the maximum number of retries has been exhausted.

SeqRollOverPolicy is invoked when the roll-over occurs on the sequence numbers. Some house-keeping actions may be required by the Flow Allocator to modify the bindings between connection-endpoint-ids and port-ids.

6.2.6. Performance utility classes

The main focus of the performance monitoring utility classes is to define managed objects that can capture notifications. A key performance management class is the *Event* managed object. It is expected that specific performance events are defined as derived types of this class.

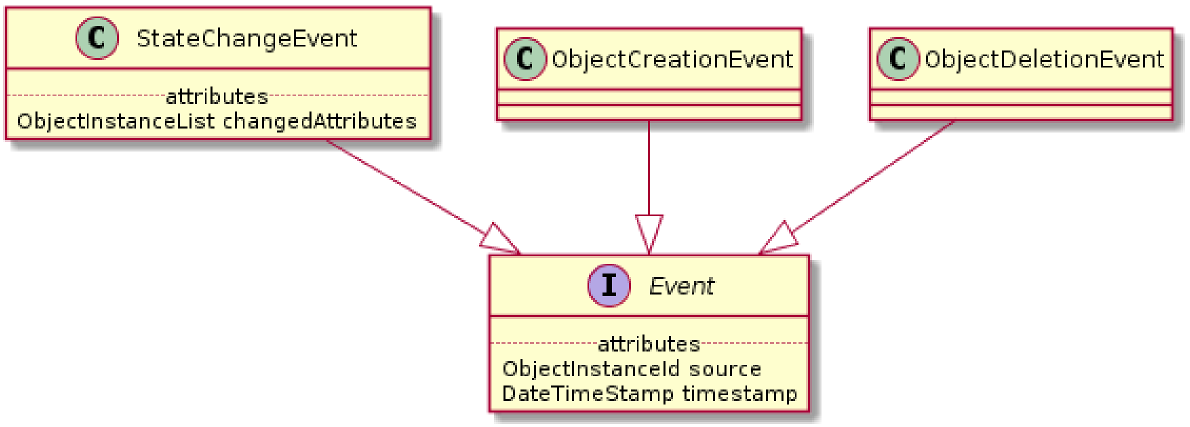


Figure 32. Notification inheritance hierarchy

An *Event* originates from a specific source and occurs at a given instance in time. Derived types may be defined where the reason for the event is caused by more than one managed object, i.e. cant be represented as a *StateChangeEvent*. This would allow more complex performance events to be defined, an capture the object instance identities of the triggering managed objects.

6.3. Containment tree

The following section outlines the MO classes that help define the structure of the DMS.

6.3.1. RINA containment tree

In this section we will show the containment tree of the common objects defined in PRISTINE for RINA. The containment tree is a tree of RIB objects which is following the concept of "has a" relationship. This means that objects in top level contain the lower level objects linked to them.

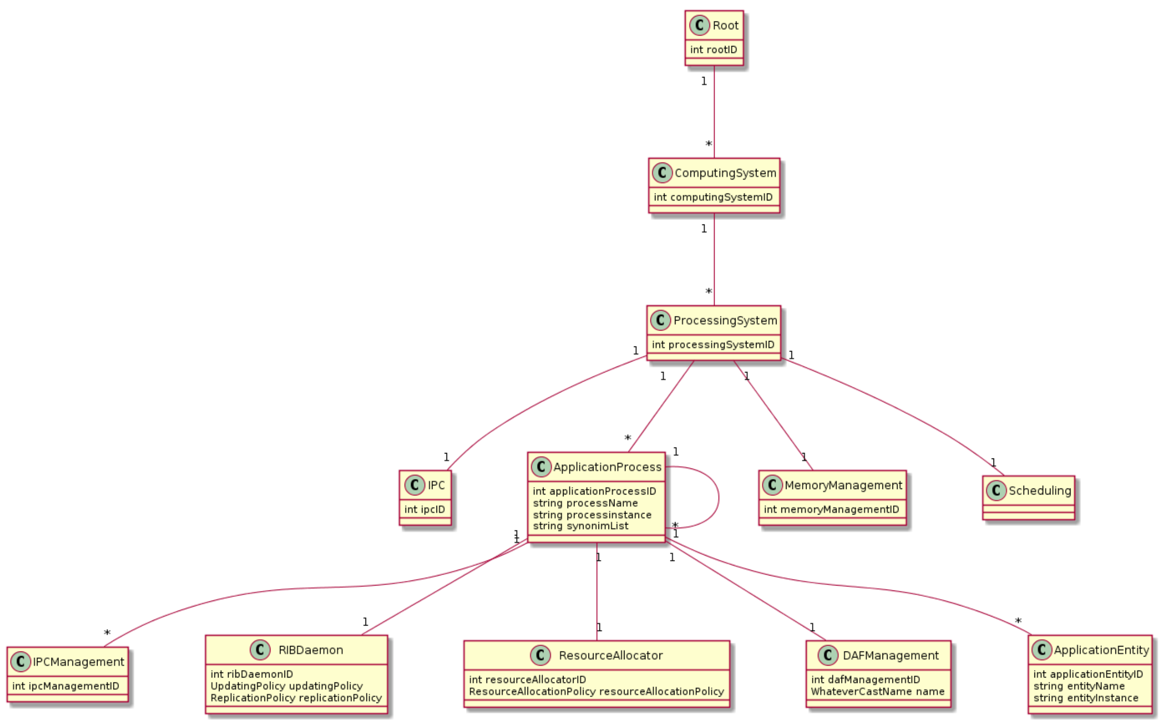


Figure 33. Containment tree: Common structure

This shows the common structure for the containment tree. Most aspects of this structure are common between DIF specific sections and DAF specific sections. For DAF management all objects are contained under a specific *ApplicationProcess*. Further details for each Managed Object can be found in Annex A.

DAF containment tree

This diagram shows the object model which must be common in any DAF object model.

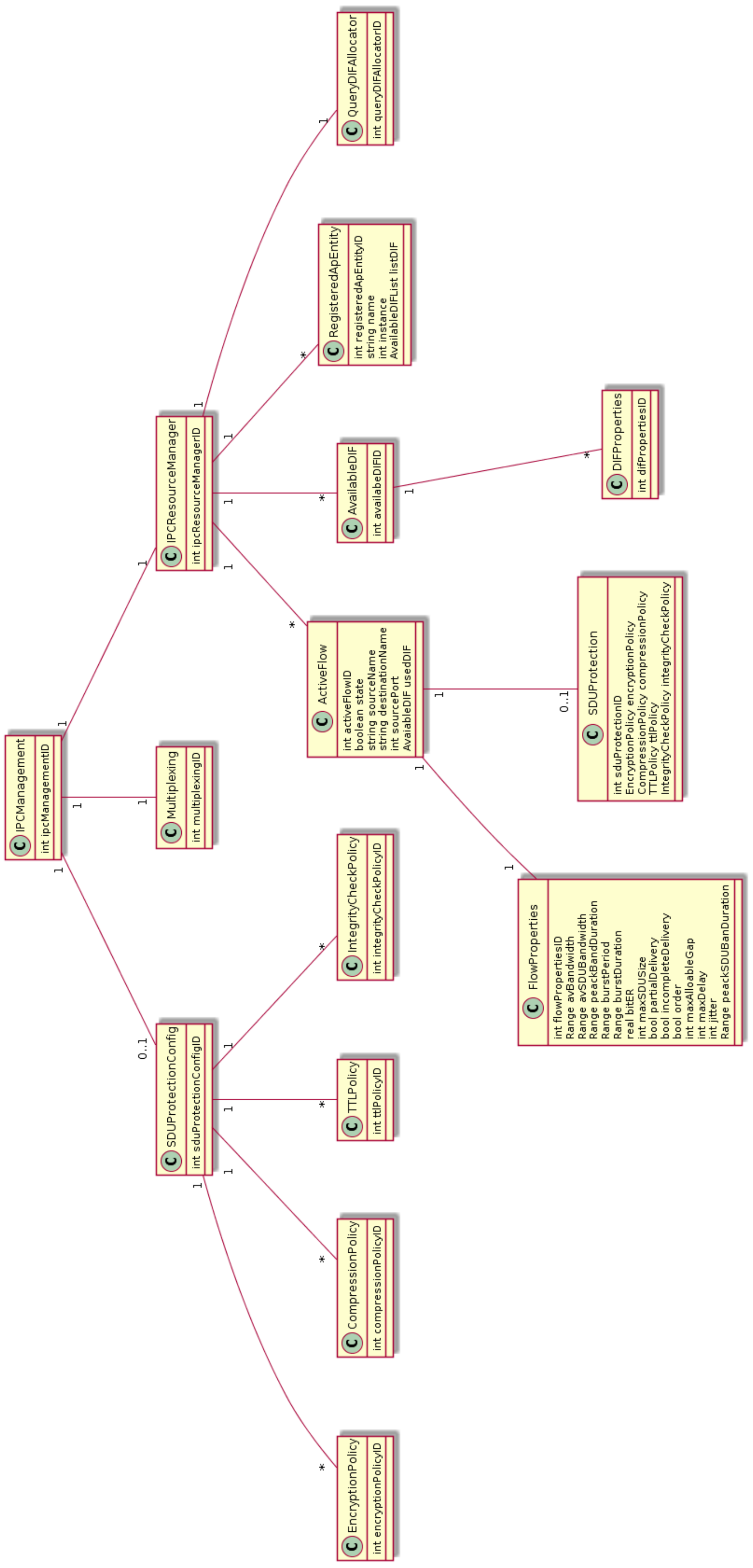


Figure 34. Containment tree: IPC management branch

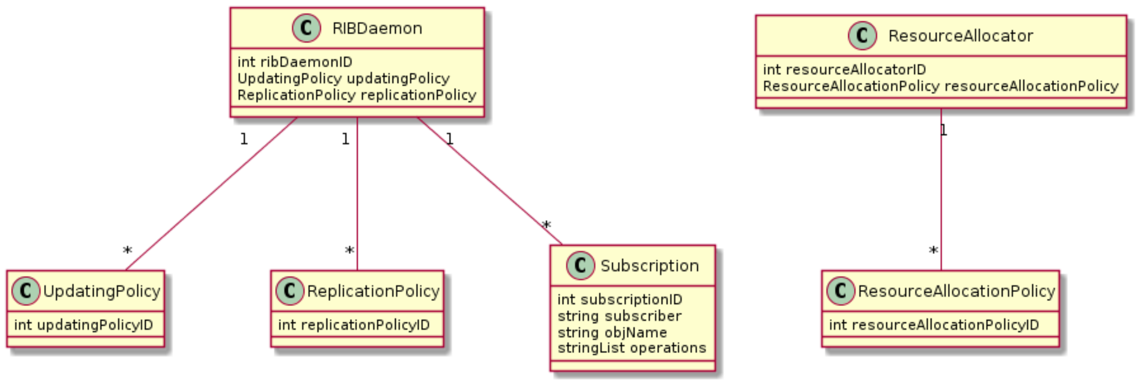


Figure 35. Containment tree: RIB Daemon and Resource management branches

The RIB Daemon managed object serves as a repository of available updating and replication behaviours. It contains a list of subscriptions for notifications when a specific Managed Object in the RIB is updated. The *ResourceAllocator* contains a catalogue of available/known *ResourceAllocationPolicy* instances within the DAF.

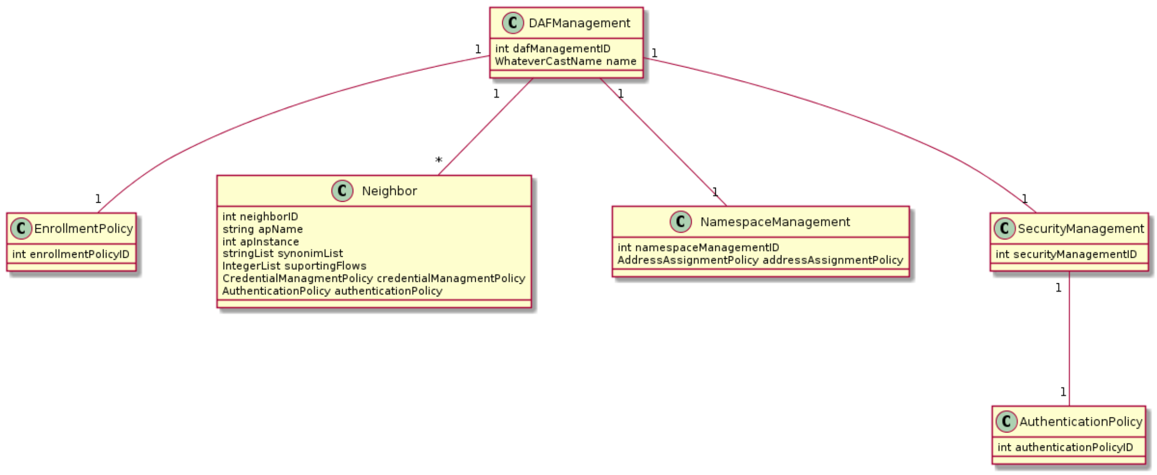


Figure 36. Containment tree: DAF management branch

The DAF management branch contains a list of "peer" nodes or neighbours. It contains a policy for managing the DAF name-space, and one covering security management namely the authentication policy for the DAF.

DIF containment tree

This diagram shows the object model which must be common in any DIF object model.

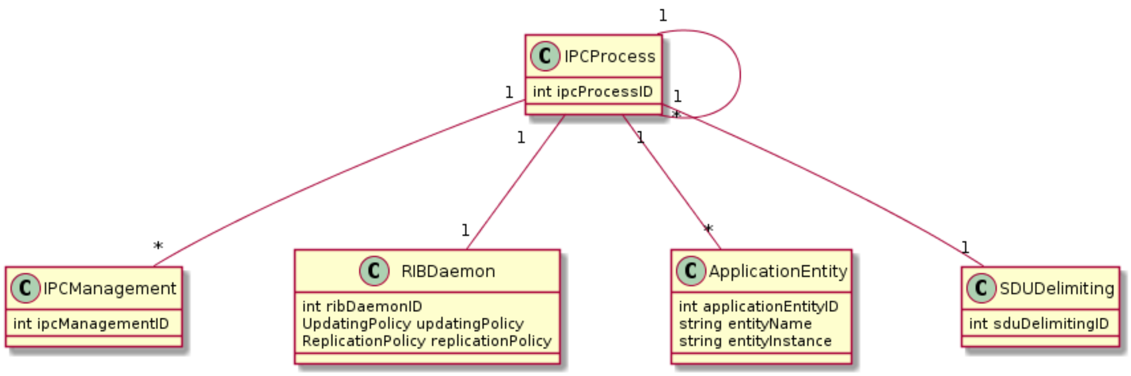


Figure 37. Containment tree structure: DIF

First the root of the DIF containment tree is rooted on an *IPCProcess*. The *IPCManagement* and *RIBDaemon* branches have a similar structure to those as presented in the DAF sections. The *ApplicationEntity* has no contained objects, so just records the associated application entity to the *IPCProcess*.

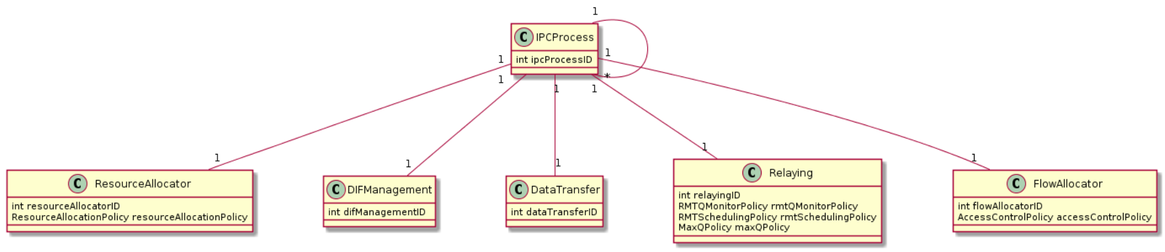


Figure 38. Containment tree structure: DIF part 2

The structure of these branches is more complex and shown in subsequent diagrams.

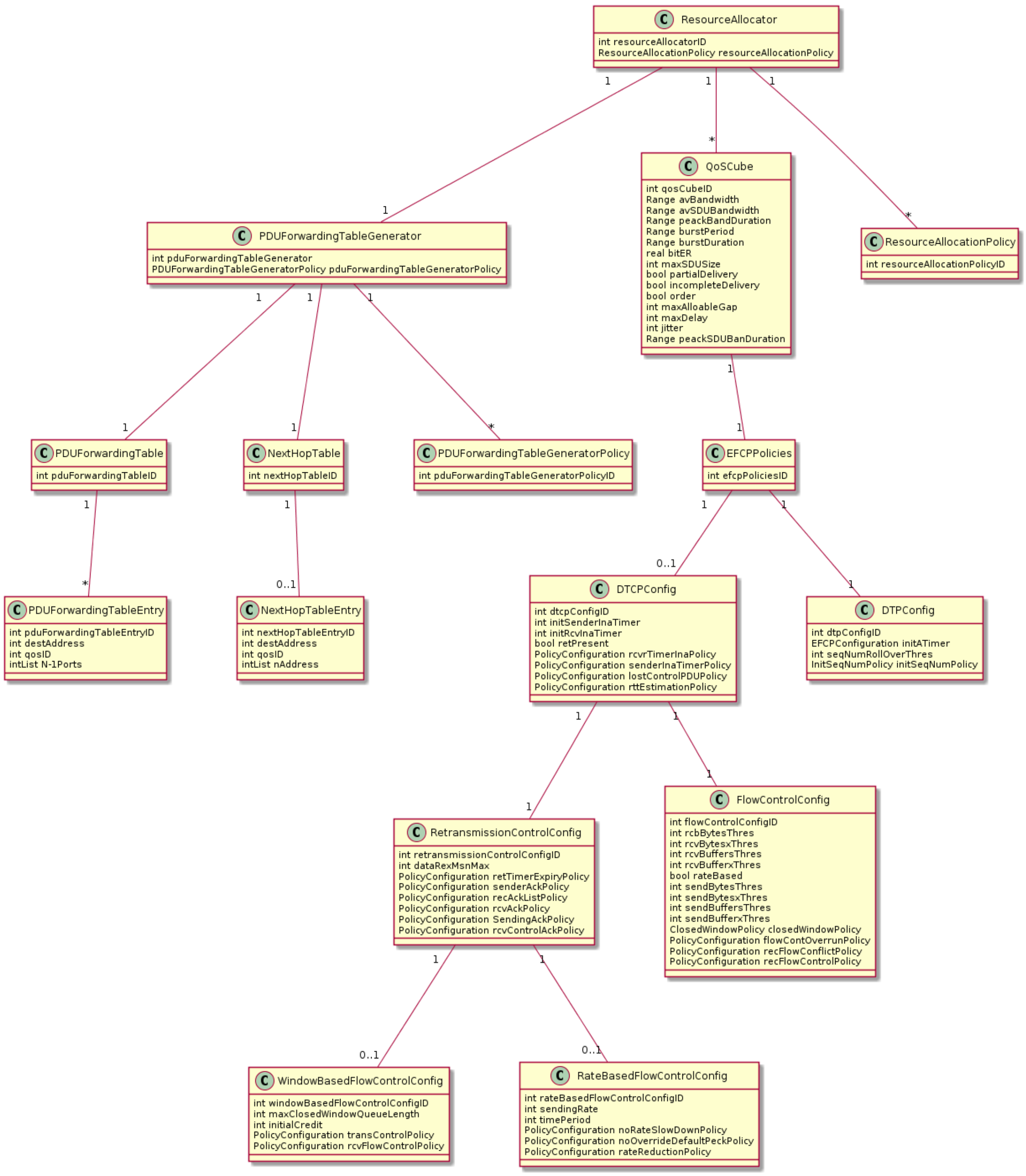
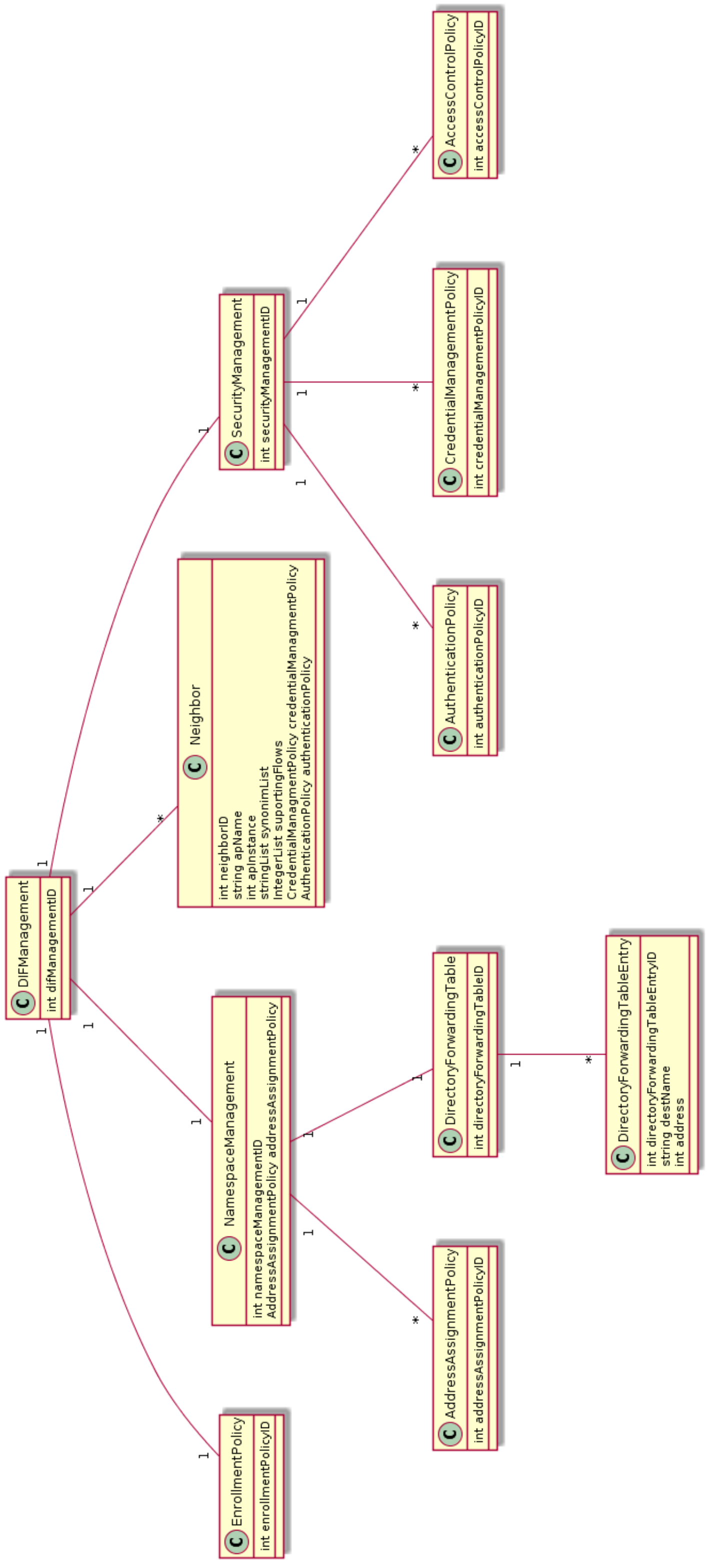

ResourceAllocator
int resourceAllocatorID
ResourceAllocationPolicy resourceAllocationPolicy
QoSCube
int qosCubeID
Range avBandwidth
Range avSDUBandwidth
Range peackBandDuration
Range burstPeriod
Range burstDuration
real bitER
int maxSDUSize
bool partialDelivery
bool incompleteDelivery
bool order
int maxAlloableGap
int maxDelay
int jitter
Range peackSDUBanDuration
ResourceAllocationPolicy
int resourceAllocationPolicyID
PDUForwardingTableGenerator
int pduForwardingTableGenerator
PDUForwardingTableGeneratorPolicy pduForwardingTableGeneratorPolicy
PDUForwardingTable
int pduForwardingTableID
NextHopTable
int nextHopTableID
PDUForwardingTableGeneratorPolicy
int pduForwardingTableGeneratorPolicyID
EFCPPolicies
int efcpPoliciesID
PDUForwardingTableEntry
int pduForwardingTableEntryID
int destAddress
int qosID
intList N-1Ports
NextHopTableEntry
int nextHopTableEntryID
int destAddress
int qosID
intList nAddress
DTCPConfig
int dtcpConfigID
int initSenderInaTimer
int initRcvInaTimer
bool retPresent
PolicyConfiguration rcvrTimerInaPolicy
PolicyConfiguration senderInaTimerPolicy
PolicyConfiguration lostControlPDUPolicy
PolicyConfiguration rttEstimationPolicy
DTPConfig
int dtpConfigID
EFCPConfiguration initATimer
int seqNumRollOverThres
InitSeqNumPolicy initSeqNumPolicy
FlowControlConfig
int flowControlConfigID
int rcbBytesThres
int rcvBytesxThres
int rcvBuffersThres
int rcvBufferxThres
bool rateBased
int sendBytesThres
int sendBytesxThres
int sendBuffersThres
int sendBufferxThres
ClosedWindowPolicy closedWindowPolicy
PolicyConfiguration flowContOverrunPolicy
PolicyConfiguration recFlowConflictPolicy
PolicyConfiguration recFlowControlPolicy
RetransmissionControlConfig
int retransmissionControlConfigID
int dataRexMsnMax
PolicyConfiguration retTimerExpiryPolicy
PolicyConfiguration senderAckPolicy
PolicyConfiguration recAckListPolicy
PolicyConfiguration rcvAckPolicy
PolicyConfiguration SendingAckPolicy
PolicyConfiguration rcvControlAckPolicy
WindowBasedFlowControlConfig
int windowBasedFlowControlConfigID
int maxClosedWindowQueueLength
int initialCredit
PolicyConfiguration transControlPolicy
PolicyConfiguration rcvFlowControlPolicy
RateBasedFlowControlConfig
int rateBasedFlowControlConfigID
int sendingRate
int timePeriod
PolicyConfiguration noRateSlowDownPolicy
PolicyConfiguration noOverrideDefaultPeckPolicy
PolicyConfiguration rateReductionPolicy

Figure 39. Containment tree: DIF - Resource Management



DIFManagement
int difManagementID
Neighbor
int neighborID
string apName
int apInstance
stringList synonimList
IntegerList suportingFlows
CredentialManagmentPolicy credentialManagmentPolicy
AuthenticationPolicy authenticationPolicy
SecurityManagement
int securityManagementID
AccessControlPolicy
int accessControlPolicyID
CredentialManagementPolicy
int credentialManagementPolicyID
AuthenticationPolicy
int authenticationPolicyID
NamespaceManagement
int namespaceManagementID
AddressAssignmentPolicy addressAssignmentPolicy
DirectoryForwardingTable
int directoryForwardingTableID
DirectoryForwardingTableEntry
int directoryForwardingTableEntryID
string destName
int address
AddressAssignmentPolicy
int addressAssignmentPolicyID
EnrollmentPolicy
int enrollmentPolicyID

Figure 40. Containment tree: DIF - Management

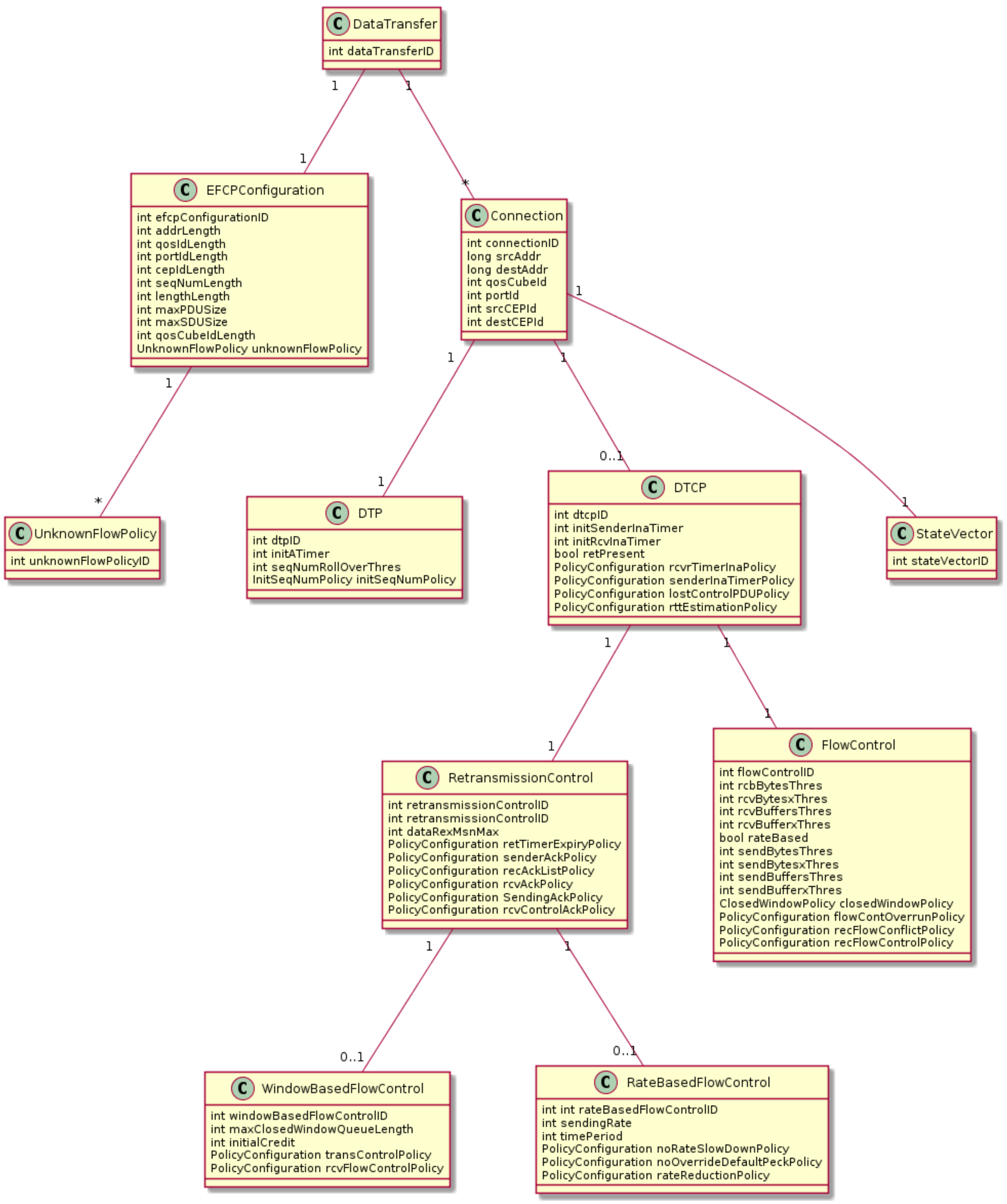

DataTransfer
int dataTransferID
1
1
1
*
EFCPConfiguration
int efcpConfigurationID
int addrLength
int qosIdLength
int portIdLength
int cepIdLength
int seqNumLength
int lengthLength
int maxPDUSize
int maxSDUSize
int qosCubeIdLength
UnknownFlowPolicy unknownFlowPolicy
Connection
int connectionID
long srcAddr
long destAddr
int qosCubeId
int portId
int srcCEPId
int destCEPId
1
1
*
UnknownFlowPolicy
int unknownFlowPolicyID
1
1
DTP
int dtpID
int initATimer
int seqNumRollOverThres
InitSeqNumPolicy initSeqNumPolicy
1
0..1
DTCP
int dtcpID
int initSenderInaTimer
int initRcvInaTimer
bool retPresent
PolicyConfiguration rcvrTimerInaPolicy
PolicyConfiguration senderInaTimerPolicy
PolicyConfiguration lostControlPDUPolicy
PolicyConfiguration rttEstimationPolicy
1
1
StateVector
int stateVectorID
1
1
1
1
RetransmissionControl
int retransmissionControlID
int retransmissionControlID
int dataRexMsnMax
PolicyConfiguration retTimerExpiryPolicy
PolicyConfiguration senderAckPolicy
PolicyConfiguration recAckListPolicy
PolicyConfiguration rcvAckPolicy
PolicyConfiguration SendingAckPolicy
PolicyConfiguration rcvControlAckPolicy
FlowControl
int flowControlID
int rcbBytesThres
int rcvBytesxThres
int rcvBuffersThres
int rcvBufferxThres
bool rateBased
int sendBytesThres
int sendBytesxThres
int sendBuffersThres
int sendBufferxThres
ClosedWindowPolicy closedWindowPolicy
PolicyConfiguration flowContOverrunPolicy
PolicyConfiguration recFlowConflictPolicy
PolicyConfiguration recFlowControlPolicy
1
1
0..1
0..1
WindowBasedFlowControl
int windowBasedFlowControlID
int maxClosedWindowQueueLength
int initialCredit
PolicyConfiguration transControlPolicy
PolicyConfiguration rcvFlowControlPolicy
RateBasedFlowControl
int int rateBasedFlowControlID
int sendingRate
int timePeriod
PolicyConfiguration noRateSlowDownPolicy
PolicyConfiguration noOverrideDefaultPeckPolicy
PolicyConfiguration rateReductionPolicy

Figure 41. Containment tree: DIF - Data Transfer

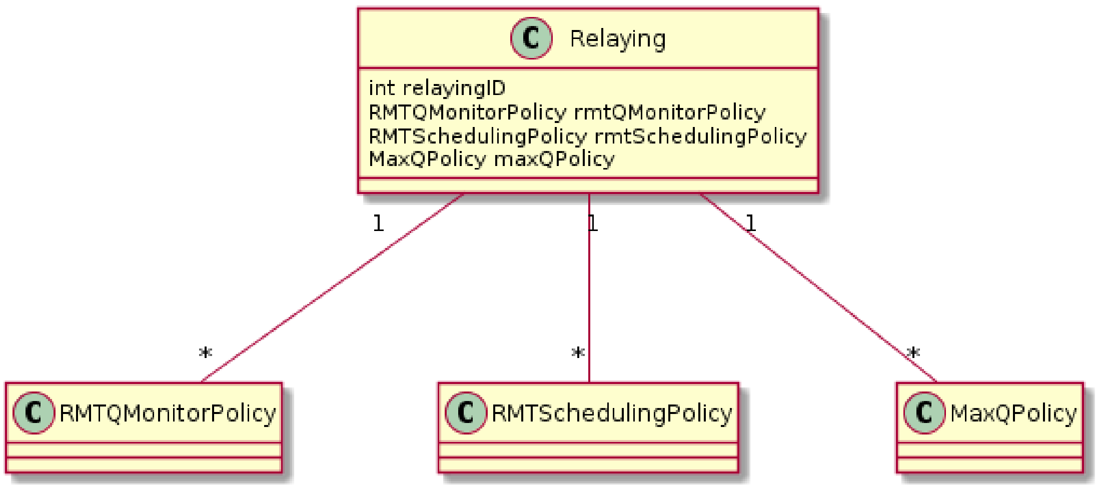


Figure 42. Containment tree: DIF - Relaying

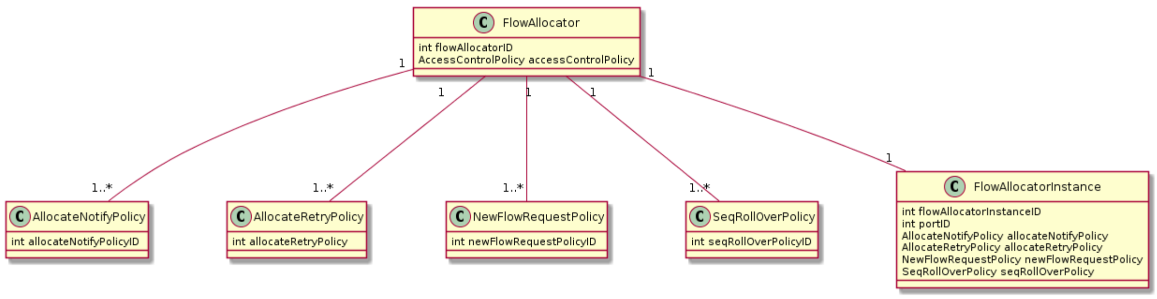

FlowAllocator
int flowAllocatorID
AccessControlPolicy accessControlPolicy
1
1
1
1
1
1..*
1..*
1..*
1..*
1
AllocateNotifyPolicy
int allocateNotifyPolicyID
AllocateRetryPolicy
int allocateRetryPolicy
NewFlowRequestPolicy
int newFlowRequestPolicyID
SeqRollOverPolicy
int seqRollOverPolicyID
FlowAllocatorInstance
int flowAllocatorInstanceID
int portID
AllocateNotifyPolicy allocateNotifyPolicy
AllocateRetryPolicy allocateRetryPolicy
NewFlowRequestPolicy newFlowRequestPolicy
SeqRollOverPolicy seqRollOverPolicy

Figure 43. Containment tree: DIF - Flow Allocation

7. Manager Architecture

The Management Architecture needs to support the design, installation and configuration of RINA management systems. The peculiarities of system management need to be recognized in the development process to optimize control, administration and maintenance. Objects of the management can act in different roles. Figure 44, “Manager-agent network structure” below shows a typical scenario in which managers (general or local) invoke operations on agents (general or sub-agents). Agents in turn direct the operations towards managed objects they control. The managed objects are then responsible to run the operation on the resources they do control. Communication in the other direction (i.e. from the managed objects back to the managers) is realised by notifications.

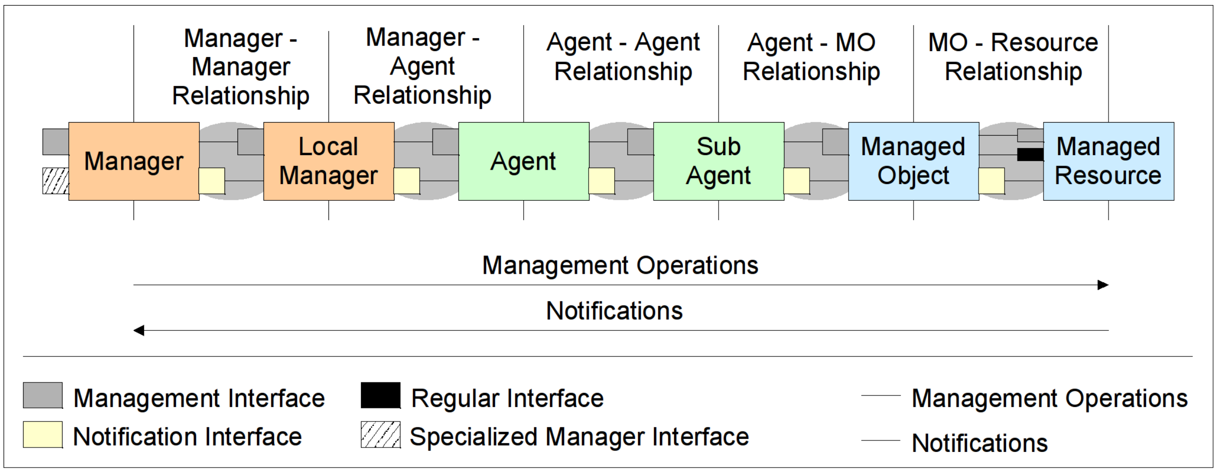


Figure 44. Manager-agent network structure

Managers, agents and managed objects belong to the system’s management, i.e. the RINA management system developed by PRISTINE. They are applied with management specific interfaces. The managed resource is either a logical resource in form of an application object or a physical resource in form of a device. Managed resources are not bound to the communication paradigm of operation/notification and need not to specify management-specific interfaces.

In a classical management system, notifications are exchanged via special notification interfaces. PRISTINE can offer an event service or use specific notification calls in object interfaces. However, the system designer should be aware of the special relationships between all roles. These relationships model the functionality of each role. Managers are provided with information about the whole system (or parts in case of local managers) that has to be managed. They generate complex management operations, often using high-level policies, and offer them to superior management systems or human operators. Agents have only knowledge on specific parts of the system. They are able to split management operations to the managed objects they control. A managed object maps a simple management operation to appropriate operations of the managed resource. The Core Model already declares a type definition that covers all relevant management roles. Each application object can be accompanied with this information in order to characterize it as manager, agent, or managed object.

The next sections introduce the specific principles and concepts that need to be applied in the Distributed Management System (DMS) for RINA.

7.1. DMS specific principles

The following section outlines the additional supports the DMS should provide to enable the correct processing of CDAP management operations.

7.1.1. CDAP protocol support

The CDAP protocol allows the targeting of a CDAP management operation to a single or set of managed objects. These managed objects form a containment hierarchy or management hierarchy and this structure is exploited. Additionally, objects could be filtered and scoped, and operations can be forwarded via a set of objects until the finally addressed object is reached. CDAP shares some of its behavioural mechanisms with CMIP, thus some of the implementation techniques outlined in [vanDeMeer] can be reused.

The management hierarchy consists of objects fulfilling different roles, as introduced above. In a peer to peer network those objects might offer distributed lookup and discovery services, persistence, and resource management.

Objects can invoke operations on other objects and they can receive notifications on occurred events. Both types of communication can be either synchronous or asynchronous. High-level management policies can be applied for the forwarding of operation calls and notifications. The following subsections introduce the application of the protocol for addressing objects within hierarchies taking a management system as example. The mechanisms can be also employed by other applications, such as peer to peer networks or intelligent agents.

Management operation requests are forwarded along the hierarchical structure of the management tree to the agents or the managed objects, on which these operations should be performed. Therefore, the address of an object consists of a description of the path to this object and a unique object identification. The path description is composed of one or more unique object identifications of agents, which are responsible for the addressed agent or managed object.

An address may refer to single or multiple agents or managed objects (e.g. all objects of an agent or even all objects of a specific sub-tree). Addresses and lists of addresses are assigned to a management operation request. The protocol needs to offer flags in the option parameter of the operation that describe the execution policy of operations. For better understanding, the following abstract operation is used instead of the complete operation. The field list of entities contains all information regarding the addresses of objects.

```
Op('list of entities', 'flag', ...)
```

A hierarchy consists of nodes and leafs. Each node and each leaf represent an object of a distributed application with a special functionality. Nodes can forward operation calls to other nodes or leafs. Nodes can also execute operations locally. Leafs receive operation calls and execute them. Both, leafs and nodes, can send notifications to other objects in the hierarchy. In a management system, nodes are called agents and leafs are called managed objects.

7.1.2. Multi-node addressing

Each node in the hierarchy of a management system is provided with the information that it belongs to the group of agents. Addressing a node in the hierarchy now implies to set the flag entityType to agent. The management protocol can evaluate this flag at each node to invoke the proper operation or to discard the operation completely. The two other flags of the option field of the protocol have the following meaning:

- When the *recursivelyFlag* is set, the operation is forwarded from each node to all other subordinate nodes. No action regarding a forward is done when this flag is not set.
- When the *localExecutionFlag* is set, the operation is executed by the node itself. No action regarding a local execution is done when this flag is not set.

The two flags do not affect each other. In the worst case, the operation is neither executed locally nor forwarded to other nodes. Furthermore, the two flags allow three variants:

1. An operation is executed locally and not forwarded to other nodes.
2. An operation is not executed locally but forwarded to other nodes.
3. An operation is executed locally and forwarded to other nodes.

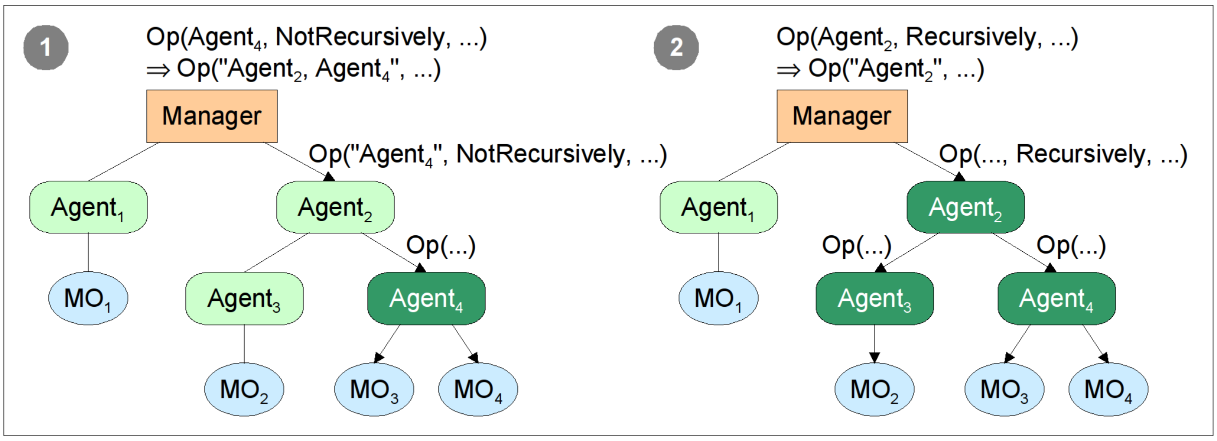

1
Op(Agent4, NotRecursively, ...)
⇒ Op("Agent2, Agent4", ...)
Manager
Op("Agent4", NotRecursively, ...)
Agent1
Agent2
Op(...)
MO1
Agent3
Agent4
MO2
MO3
MO4
2
Op(Agent2, Recursively, ...)
⇒ Op("Agent2", ...)
Manager
Op(..., Recursively, ...)
Agent1
Agent2
Op(...)
Op(...)
MO1
Agent3
Agent4
MO2
MO3
MO4

Figure 45. Addressing nodes

The figure above shows the effects of the *recursivelyFlag*. In case 1, a single node is addressed directly. The operation is initiated by the object Manager and forwarded via the object Agent2 to the object Agent4. In case an operation is called by the object Manager on the object Agent2 with an activated *recursivelyFlag*. The object Agent2 automatically forwards this operation call to the objects Agent3 and Agent4. The *localExecutionFlag* is used to ensure that the operation is executed on the addressed objects (Agent4 in case 1, Agent3 and Agent4 in case 2) only or also on the objects that forward the operation (Agent2 in both cases).

The usage of these two flags can be combined with the addressing of objects supported by the management protocol. The address field of the protocols allows for multiple addresses and object paths. This means, the object Manager is able to invoke the same operations not only on the nodes Agent2, Agent3, and Agent4 but with the same call on the object Agent1.

Addressing leaf objects

Each leaf in the hierarchy of a management system is provided with the information that it belongs to the group of managed objects. Addressing a leaf in the hierarchy implies to set the flag entityType to managed object. The management protocol can evaluate this flag at each node to invoke the proper operation in the managed object itself. The option *recursivelyFlag* has no special meaning and will not be interpreted by leaf objects. The flag *localExecutionFlag* is evaluated and the operation is invoked locally when set. Otherwise the requested operation is forwarded up to the addressed leaf object.

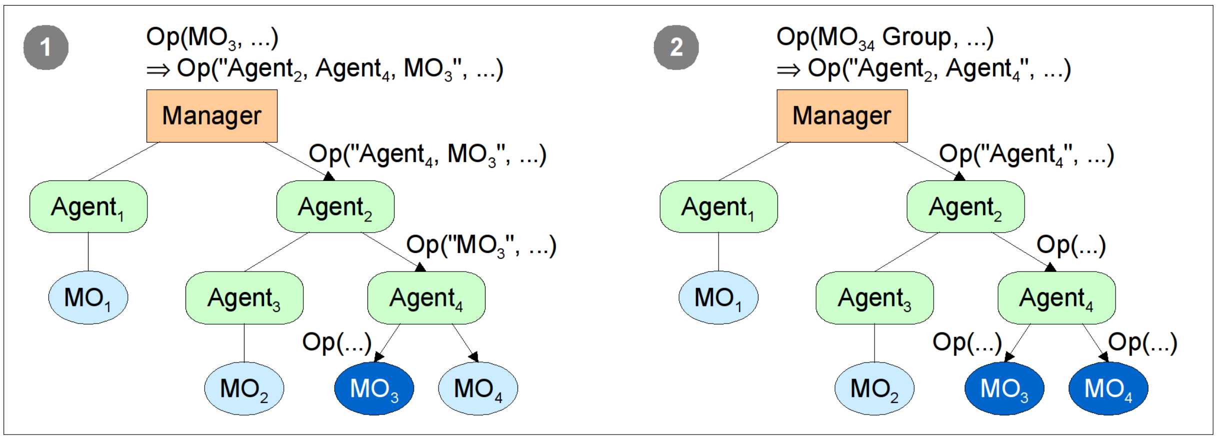


Figure 46. Addressing leaf nodes

The figure above shows the two possibilities for addressing leaf objects within hierarchies. In case 1, the object Manager addresses the object MO3 directly. All objects in between Manager and MO3 forward the operation request. Case 2 depicts the addressing of two leaf objects with a group call. The object Manager addresses the objects MO3 and MO4 with one function call. This kind of addressing can be achieved by two different ways. First, the address filed can include the address of the object Agent4 and the options field has an activated flag *recursivelyFlag* and a deactivated flag *localExecutionFlag*. The second possibility is to address both objects, MO3 and MO4, directly with an activated flag *localExecutionFlag*. The flag entityType must be set to managed object for this scenario.

7.1.3. Transaction support

The management protocol should also support transactional operations. The necessity of the transaction concept refers to operations changing the state of multiple objects within the system. If the states of multiple objects depend on one another, the performance of operations on these objects may be only sensible when they are performed successfully on all affected objects. If the operation could not be performed on one or more objects, the system is in an inconsistent state. This may influence the system's runtime behaviour negatively and has to be avoided. Again, we are borrowing some detail from the discussion in [vanDeMeer].

A simple commit protocol to use is the two-phase commit protocol. Additionally, transactions are combined with the support for hierarchies so that transactional operations cannot only be provided for peer to peer object communication but also for hierarchical object communication.

The protocol applies transaction processing to an operation call when the transactionFlag in the options filed is activated. The management API should be in the position to offer a configuration function that activates this flag for all operation calls or for the communication with a certain object group, or for the communication with a certain object.

When the flag transactionFlag is activated, the protocol generates a Transaction Identifier (TID) for the related operation call. The protocol must be supplied with information, how the requested operation can be rolled back, that is how the object can be returned to the state it had before the transactional operation was executed. This information, usually a function call on the application object, is stored together with the TID. This is the mechanism to remember an undo operation as long as the transaction is active. The application object has to take care that the altered data cannot be changed until the end of the transaction, e.g. applying locking mechanisms.

When the transactional operation is performed on multiple objects, each individual call is treated as a single transaction. Node objects must store all TIDs of subordinate objects and the roll-back mechanism for each operation call. When node or leaf object do not support transactional operations (e.g. when they are not implementing transaction processing), the superior objects can simulate a transaction by interpreting return values and restore the object's pre-transactional state through specific operations.

The figures in the following two subsections show the protocol handling of a successful and a non-successful transaction in four steps. The initialization of the transaction is identical for both. The object Manager requests an operation that should be executed on the objects MO1, MO2, MO3, and MO4 as a single transaction. It generates two TIDs, one for the operation call to the object Agent1 and one for the operation call to the object Agent2. Now, it requests the forwarding of the operation. Subordinate objects (in the example Agent3) perform the same actions.

Successful Transaction

The first step is finalized by the successful execution of the requested operation on all leaf objects. In step 2 of a successful transaction (case two in the following figure), all leaf objects have executed the requested operation successfully and return the TID to notify the object Manager.

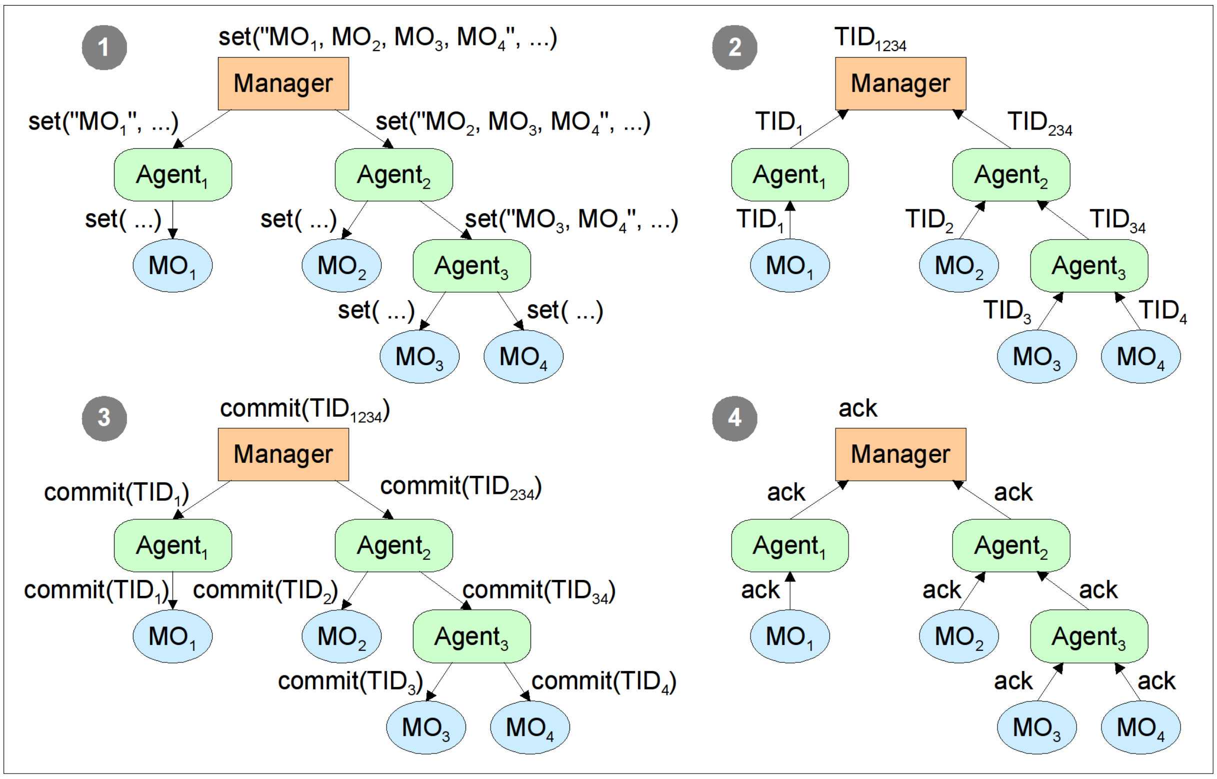


Figure 47. Flow for a successful transaction

Since each operation is treated as a single transaction, all node object and the object Manager follow the same procedure. When they have received all TIDs from subordinate objects, they return the TID related to the communication with their own superior object to that very object.

Finally, the object Manager receives the two TIDs it has generated by itself from the node objects Agent1 and Agent2. Now, the object Manager invokes step 3. It sends a commit message to all objects to indicate that the transaction was completely successful and that no rollback mechanism has to be performed. This commit message is forwarded up to the leaf objects. The final step number 4 comprises the emitting of an acknowledgement message from all involved objects. At the same time, all information related to the transactions is removed by the objects. The objects receiving a commit message release the locks of its data and the transaction related information. The system has reached its final state and is ready for further transactions.

Unsuccessful Transaction

In a non-successful transaction, the first step could not be performed on all objects successfully. The second step of a non-successful transaction is shown by case 2 in the following figure. In the given example, the requested operation could not be executed on the leaf object MO4. This event changes the steps 2, 3, and 4. In step 2, the leaf object MO4 returns an abort message to the superior object. The node object Agent3 receives one commit message and the abort message. It stores the TID from the leaf object MO3 to remember that the operation was successfully executed there, and the abort message for MO4. Now, it sends an abort message to its own superior object. At the end of the chain, the object Manager receives one commit message (from Agent1) and one abort message (from Agent2).

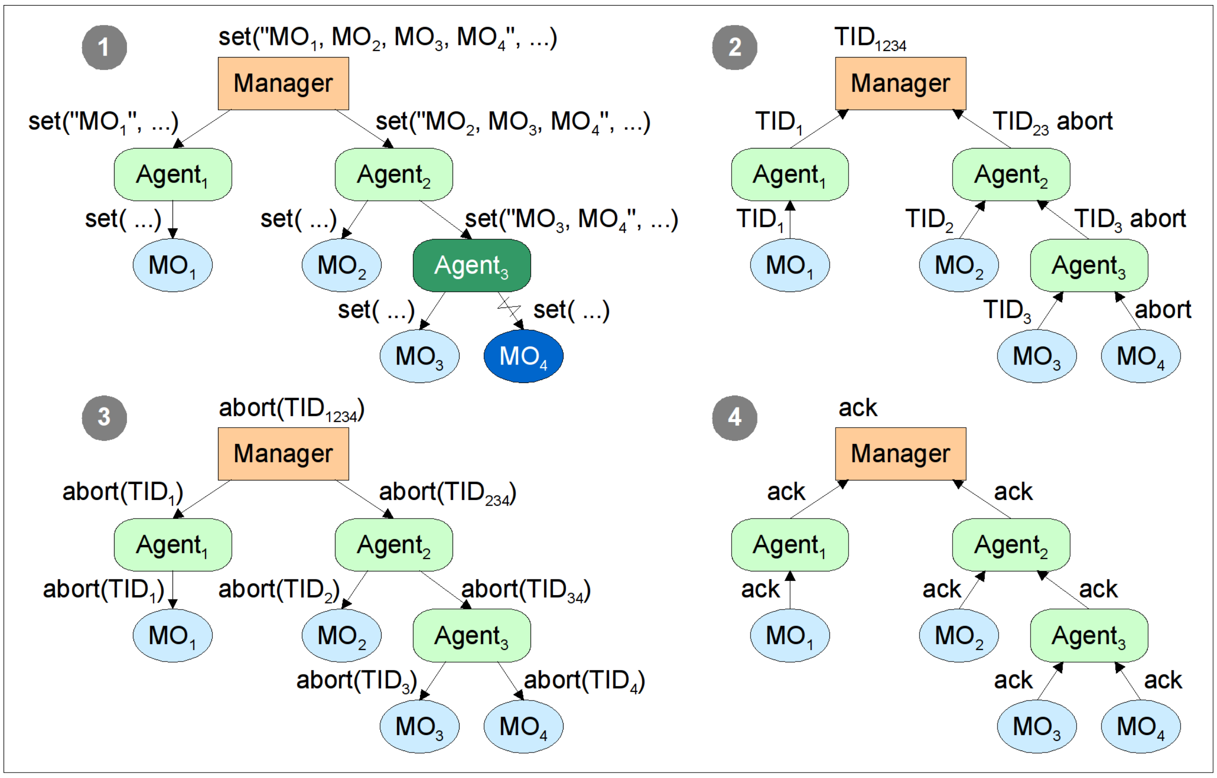


Figure 48. Flow for an unsuccessful transaction

In step 3, the manager sends an abort command to all objects. All objects that receive the abort command invoke the proper actions to roll-back the already performed operation. Step 4 shows that all objects that successfully realized the undo actions send an acknowledge message. When the object Manager has received all acknowledgements, the system is in the same state as prior to the transaction.

Additional issues

The two-phase commit protocol has several disadvantages. The commit, abort, and acknowledge messages need to be transmitted successfully in order to realize the transaction. In the case that these messages are not received by the addressed objects, the system might run into a deadlock awaiting messages and locking other transactions or operations. For this reason, the Three Phase Commit (3PC) protocol has been developed. This protocol provides a secure handling of transactions, but increases the protocol overhead and the number of communications during a transaction. The three phase commit (3PC) protocol was tested and this overhead prevented the general adoption of this protocol.

PRISTINE might apply a more simple mechanism to avoid deadlocks. Each transaction is accompanied with a time-out. This parameter specifies that objects should invoke a roll-back automatically when they have not received messages within a certain time slot. A transaction is no longer valid after the time out has been reached. The PRISTINE API should offer the configuration of this time out.

7.2. Common Principles

This section includes the **general design principles**, patterns and concepts that are used to guide the architecture and development of the DMS. They are applied to assist in the efficient and scalable design of DMS, but are not DMS specific, in the sense they can be applied to general information processing systems.

7.2.1. CQRS Pattern

Command Query Responsibility Segregation (CQRS) - is a design pattern [spl] based on Bertrand Meyers Command Query Separation (CQS) [cqs]. A detailed clarification can be found in [cqrs], while a description of related software artefacts can be found in [cqrs2]

In short the pattern says that: a message either returns state or changes state but should not do both. Substitute state with data to generalise. This means that there is a query model (return state) and a command model (change state). The pattern assumes that the connecting element are events, thus one can still build event-sourced systems using the pattern. As a consequence, the resulting system is no longer using a (simple) Create,Read,Update,Delete (CRUD) data store.

7.2.2. Strategy Pattern

The strategy pattern is one of the standard Gang of Four (GoF) behavioural design patterns [spl], also known as the Policy Pattern. The pattern alows algorithms to vary depending on context (usually the client using it). In other words, it defines policies for mechanisms. In the RINA sense, the mechanism can be "checksum" while the policy (strategy) applied for a particular client can be MD5, SHA-1, CRC, or whatever else is required.

7.3. DMS behaviour

In order to gain a better understanding of the typical operation of the DMS, it is useful to consider a concrete use-case, for example what happens when a given CDAP operation is processed. CDAPs M_START or M_STOP operations are conceptually is quite similar to CMIP M_ACTION, so we reuse an explanatory diagram from [vanDeMeer].

For each CDAP operation (M_CREATE, M_READ, M_WRITE, M_START, M_STOP, M_DESTROY) that may applied to an object, the protocol implementation is responsible for the validation of the operation. In this case a M_START operation is being applied. The protocol implementation searches for an appropriate M_START operation in its local database, i.e. ensures that M_START is defined for the managed object. If the operation was not found, an exception is generated and returned to the calling object. Otherwise, further processing can be applied.

The next step is to filter the affected objects of the operation call. As introduced in the protocol specifications, an operation call can be applied with two flags that indicate local execution and forwarding. An additional flag identifies types of objects the operation should be forwarded to. The protocol analyses the flags to decide whether the operation should be executed locally, forwarded to specific types of objects, or forwarded in form of a broadcast to all objects connected to the application.

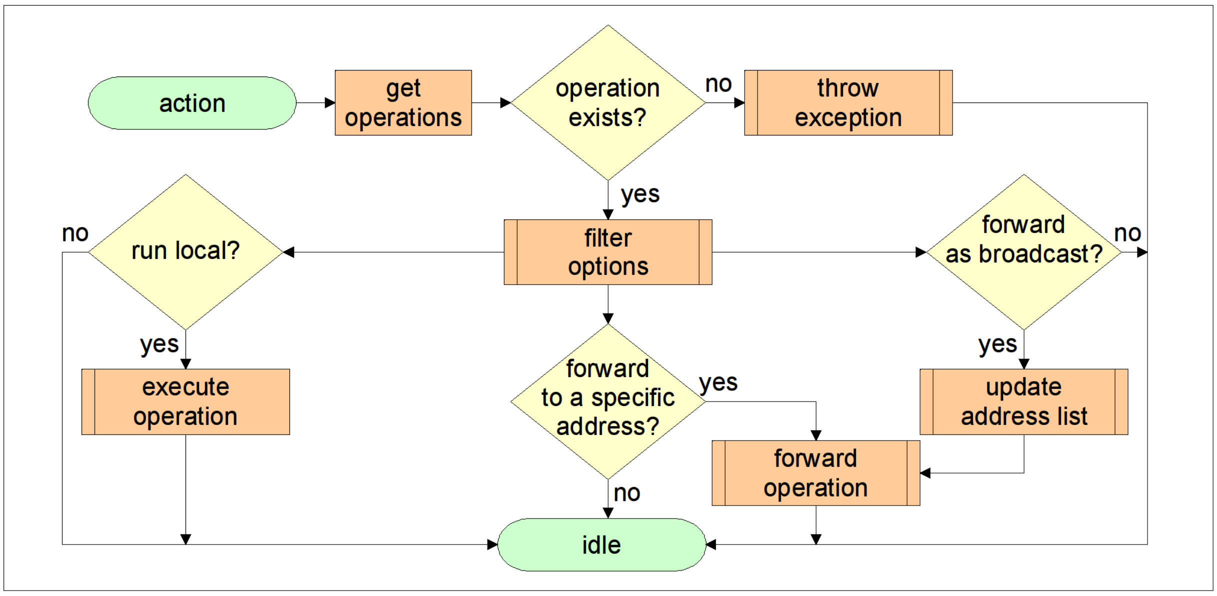


Figure 49. DMS CDAP operation processing behaviour

The local execution and the forwarding can be combined so that an operation is executed locally and forwarded. After the evaluation of the filter parameter, the protocol invokes the proper operation action, transmits possible return values, and returns to the state idle awaiting new CDAP operation calls.

7.4. Architecture

This section outlines a high level architectural view of the DMS.

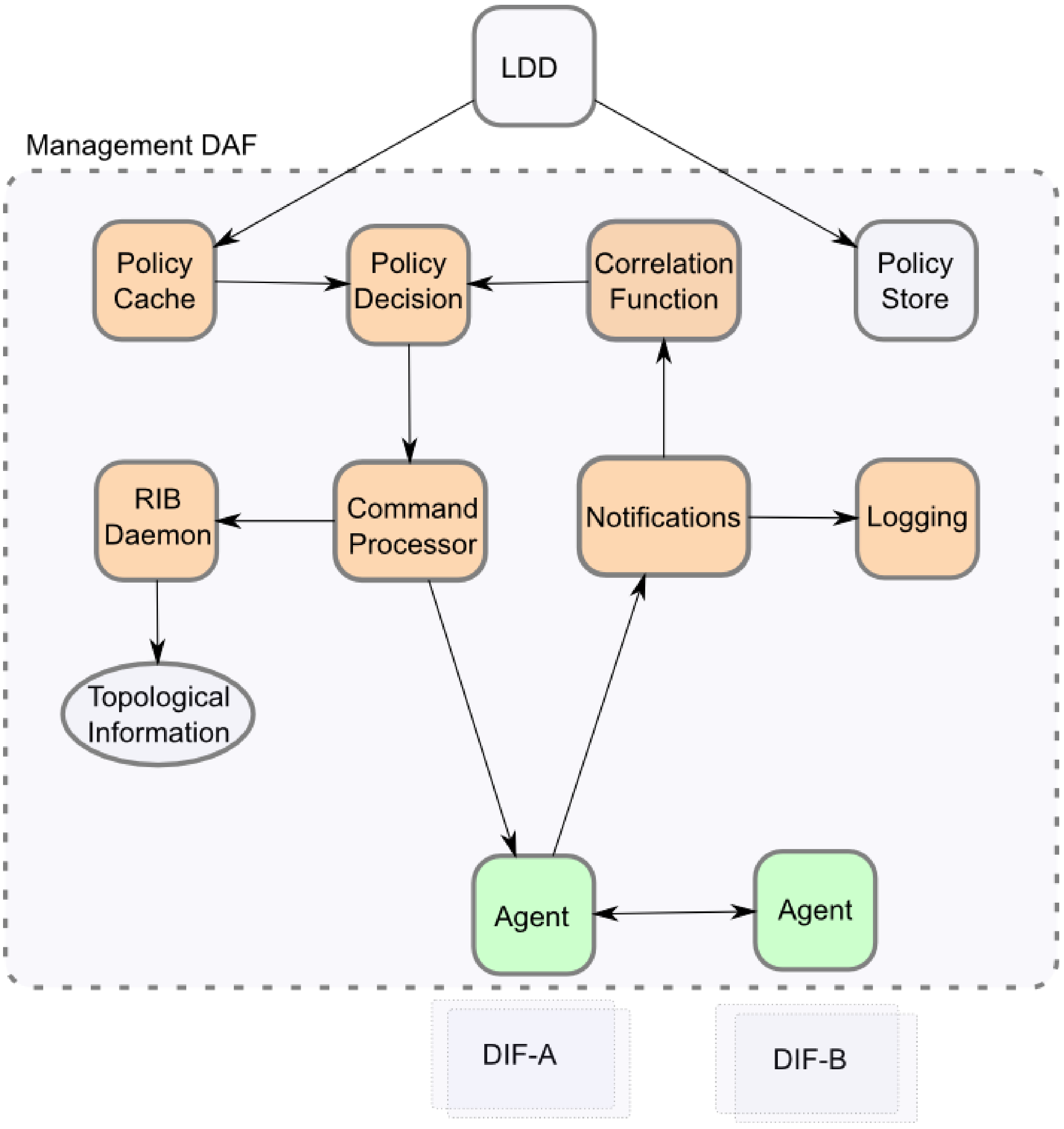


Figure 50. DMS High level Architecture

7.4.1. Component descriptions

The architecture is made up from the following components.

LDD

Language Driven Development (LDD). This is the policy creation and validation environment. It allows the DMS operator to re-use published policies from a policy store (or set of policy stores), augment them with custom logic and either republish them to the Policy store, or push to the a specified DMS for immediate use by the DMS. It is expected the LDD, will include some policy analysis (to ensure the policy is valid) and may include some simulation aspects (to simulate the effect the policy may have on a simulated network).

Policy Store
The Policy Store is a repository for obtaining policies for use in the DMS. There may be a single policy store for all policies, however large enterprises may chose to operate there own policy store, in addition to the publicly accessible one.

Policy Cache
This is the subset of policies in use in the DMS. It is reasonable to expect some of these policies are "selection" policies, whose sole purpose is to diagnose/detect network conditions, gather additional information, and come to a decision as to which of the available polices (in the cache) should apply. Other policies will be more directly involved in applying policy actions to the network. [8]

Command Processor
The command processor processes CDAP commands, and ensures they are forwarded to the correct management agents for further processing. The command processor makes use of the management DAF to communicate with these agents. It also uses topological information from the RIB daemon, to work out the correct containment hierarchy from processing the command.

Management Agent
The management agent is responsible for processing commands from the DMS manager. It also can generate notifications of events that have occurred (or been detected) at that agent. The management agent is covered in greater detail in a following section.

[8] These are referred to as *management policies* to distinguish them from RINA policies, as defined in the RINA specification. For example, what how to detect a node failure (⇒ multiple QoS alarms from Flows in different DIFs) and a decision or action to take when a node fails (⇒ notify operator joe.smith via SMS on weekends, or via IM during the working week)

RIB Daemon
: The RIB daemon captures the containment tree structure and in this case is primarily used for scoping and filtering to identify the appropriate management agents.

Notifications
: The notification processor accepts notifications from any or all management agents. Its first duty is to log the notification event, so as there is a record for auditing purposes. The second responsibility is to forward the event for correlation processing.

Logging
: The logging process is responsible for logging all events. Logs of events are stored so that they can be searched and analysed at a later time for creating graph metrics for auditing purposes if required.

Correlation
: This process runs with the intent of aggregating notifications into higher level notifications. For example, for repeated notifications performing de-duplication from two or more agents, or aggregating different notifications (A, B) into a composite notification (AB). Many different algorithms exist for performing correlation, and a set of algorithms may be applied to achieve the desired outcome.

Policy Decision
: The final component is the Policy decision component. Its purpose is to look at the notification events, determine which policies apply, gather any additional information necessary to evaluate those policies and select a set of actions that are applicable and apply them. This could result in management commands being sent to the command processor.

It should be noted that addition instances of the notification and correlation functions may be created to accommodate the DMS scale. For example, a large DMS will have multiple correlation components operating in parallel. Depending on the configuration, each one could be operating on a different segment of the overall network, or processing notification events in a first-come first-serve or worker queue basis. Therefore, the number of instances of the above components that will depend on the demand on the DMS and its peak workload.

8. Management Agent

The Management Agent (MA) is a functional entity mainly in charge of managing RINA related resources in a processing system. It maintains the dialogues with the DMS by reacting to its requests - for control, administration and maintenance as described in Figure 44, “Manager-agent network structure” - while hiding/abstracting the processing system details - e.g. such as those OS related - in order to lower down the overall system complexity - i.e. the complexity of the DMS, of its related MAs and their inter-communications.

Depending on the network design and the management constraints, the processing system may have more that one MA instances, each of them managing a portion of the IPC resources - eventually belonging to different administrative domains or stakeholders. These MA instances can be categorized by the nature of their operations, such as:

- local agents, where the agent is able to manage (e.g. bootstrap) the IPC resources without the interaction of a remote AP (e.g. the NM-DMS manager), for instance by reading a static configuration from the local file-system.
- remote agents belong to one or more NM-DMS DAFs and interact a) with one or more managers (centralized approach), b) with other MAs alike (autonomic approach) or c) a combination of both,in order to configure the system’s IPC resources.

However, given the scope of the project, PRISTINE will focus its efforts on a centralized and single administrative domain scenario. Moreover, the architecture hereby presented is functional and thus - in order to not introduce details which could be unnecessary at the moment - a single MA instance per (processing) system is assumed in the following sections.

8.1. Functionalities

As part of the NM-DMS DAF, the RINA reference model defines the main functions that an agent shall implement. These functions can be remotely accessed from an AP - a Manager AP - to remotely change the configuration of the (processing) system, via CDAP operations over the RIB. In this regard, the main functionalities exposed by the MA can be further categorized in the following areas:

NM-DMS related:

- IPC provisioning (e.g. control the creation and destruction of IPC processes).
- IPC configuration (e.g. such as policies configuration).
- Monitoring and fault management (e.g. monitoring the state of an IPC process or the state of a - physical - link)
- Inter-layer management (i.e. inter-layer optimizations and configurations - such as Access Control Lists between DIF-N and DIF-(N-1)).

OS-DMS related:

- Status retrieval:
    - Hardware resources, such as:
        - CPU (e.g. architecture, available capacity, current load).
        - Memory (e.g. available memory, used memory).
        - Storage (e.g. available storage, used storage).
        - Network (e.g. available NICs, NICs used by shim IPC Processes).
    - Software "resources", such as:
        - Management Agent information (e.g. name, description, localization, agent software version).
        - PRISTINE SDK information (e.g. SDK version).
        - Policies catalogue (i.e. policies already available in the processing system, see. next section).
        - OS related information (e.g. OS name, version)

Apart the previously mentioned functionalities, the MA could also implement autonomic behaviours. Such behaviours - e.g. switch-overs, fail-backs or quorum/election in high-availability related procedures - generally impact on the other MAs - e.g. clustered peers that are switching-over/failing-back have to keep their states synchronised - and therefore require additional procedures which are out of the scope of the present deliverable.

8.1.1. Policies

Policies are a fundamental architectural piece within RINA, since they allow to customise the operation of the DIFs according to the needs of the network/system designer. In such context, the MA is in charge of reacting to DMS commands and manage the policies in the system accordingly.

From the perspective of the MA, the policies have the following - DMS driven - life-cycle:

1. Deployment: upon request (from the DMS), the policies are a) retrieved (downloaded) from an external catalogue, b) installed into the processing system and c) made available in the MA local catalogue for instantiation. The MA local catalogue will be available for queries by the DMS - such as check-for-presence/directory like operations - as well as for all the other operations stated in this list. It is foreseen that policies, once installed in the MA catalogue, get unique handles (aliases) that should be used in order to ease their reference.
2. Instantiation: policies in the MA catalogue are instantiated in the context of an IPC. Since one deployed policy may be instantiated multiple times, the system is in charge of keeping track of their instances count (i.e. through techniques such as reference-counting).
3. Un-instantiation: this operation is the counterpart of the previous one. Once a policy instance is no more of use - such as in the case that it gets replaced or an IPC is destroyed - its corresponding resources are released, and its references count decreased accordingly. Once all its instances are deleted from the system, it can be safely unloaded from the running system.
4. Un-deployment: once a policy is un-instantiated from all the IPCs in the system, there are no more bindings between the running system and the policy (code), and therefore it can be safely removed. Finally, the policy in the MA policies catalogue is removed.

The MA hides the low level details of the aforementioned phases from the DMS, transforming the DMS (high-level) requests into MA (low-level) procedures. For example, for step 1 it is foreseen that the DMS requests will contain only the indication of the policies that should be deployed within the processing system, the MA will translate them into the set of software packages that should be retrieved from one or more external catalogues (repositories), download and finally install them - following the particular procedures of the OS present in the processing system.

A particular implementation (e.g. a Debian based one) might translate these procedures as in the following:

1. The DMS requests the deployment of policy `p` from catalogue `g`.
2. The MA (e.g. one having Debian Wheezy as OS and PRISTINE SDK version x.y) fulfils the request by:
   a. Translating `p` into a package name (e.g. `p.deb`) and `g` in a valid external reference (e.g. ftp.debian.org).
   b. Fetching the package from the external reference (e.g. downloads ftp://ftp.debian/org/p-X.deb for Wheezy).
   c. Installing it (e.g. issues and `apt-get install p`).
   d. Finally notifying back the DMS about the success of the operation.

8.2. Bootstrapping requirements

The MA requires at least to contact or to be contacted by the DMS during the very initial bootstrapping of the system. As such, there is a need of a pre-configured (existing) DIF facility and a DAF that can support the connection between the MA and the DMS.

The minimum requirements for supporting this connection are, from top to bottom:

- A MA AP that runs in the management DAF (MA-DAF) where the DMS AP runs, as well as the credentials and all the other parameters required to contact the DMS.
- A management DIF (MA-DIF).
- At least 1 shim DIF [9] that allows the MA and DMS IPCs in the MA-DIF to establish a connection, either directly or via other IPCs within the MA-DIF.

The details on fulfilment of these requirements - from both MA and DMS point of views - are out of the scope of this document. However, one possibility could be that these tasks are performed relying on a explicit

[9] A name given to wrapping a non RINA compliant network component with a DIF API.

configuration - e.g. retrieved from a configuration file or from a local database - and then accessing the local IPC manager, on both DMS and MA sides.

8.3. High level architecture

The MA can be decomposed into the following main functional blocks or modules, providing the low-level functionalities needed to fulfil the requirements stated in the previous sections:

- DMS/MA Interface (DMS-MA-IF): this module is in charge of exposing the system's RIB to the DMS via CDAP, and represents the interface that the DMS has to access in order to interact with the MA. The module primarily fetches the information from the other - internal - modules and/or triggers the commands corresponding to the requests received from the DMS.
- NM Management (NM-mgmt): the NM-mgmt module is in charge of talking with the IPC Manager and performs all the network management related functions. Within NM-mgmt there are the following sub-modules:
    - IPC management (IPC-mgmt): it controls the IPC Processes in the (processing) system, through the IPC Manager.
    - IPC monitoring (IPC-mon): monitors the IPCs state in the system and eventually generates CDAP events towards the DMS (e.g. in case of over/under-threshold values).
    - Inter-DIF management (Inter-DIF-mgmt): in charge of performing local inter-DIF monitoring and optimization.
    - Policies management (Policy-mgmt): is in charge of maintaining the (local) policies catalogue as well as solving (e.g. automatically satisfying) eventual inter-policy constraints.
- OS Management (OS-mgmt): the OS-mgmt module deals with the low-level OS details. Its main functionality is to collect OS related information, such as: the OS name, version, PRISTINE SDK version, the available resources (CPU, memory, storage space) including the available network interfaces, the current load metrics etc. It order to accomplish its task, it may make use of services provided by the underlying OS - e.g. OS daemons in UNIX based systems.

The following figure depicts the MA high level architecture. It is worth noticing that all the functional blocks depicted in the figure have the scope of the processing system, e.g. the Policy-mgmt block handles the local policies.

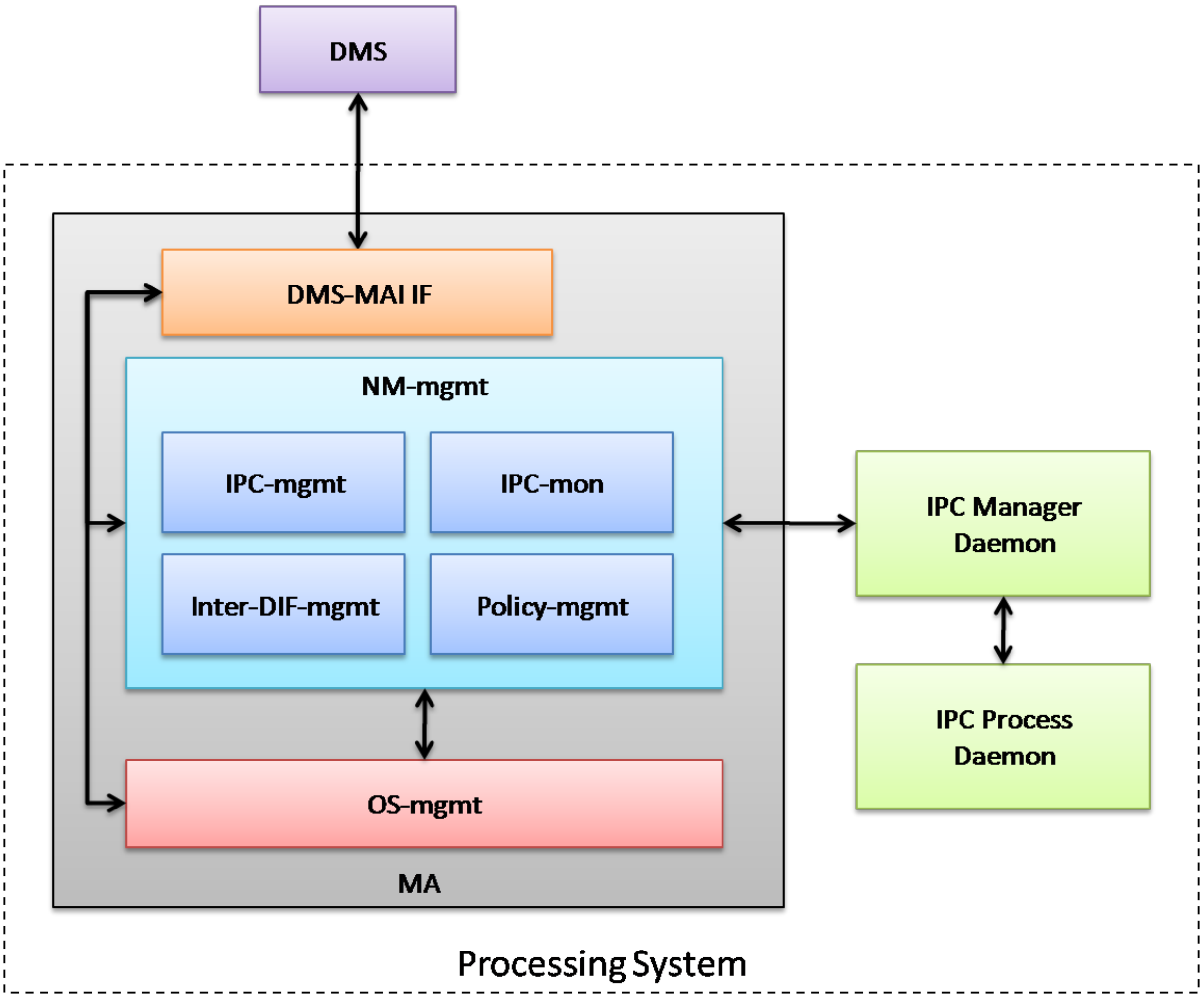


Figure 51. Management Agent high level architecture

8.4. Implementation plans

It is foreseen that the MA will be implemented as an AP and all its modules and sub-modules will be relying on a common framework. That framework will provide basic functionalities, such as logging facilities, persistent configuration, event handling related functionalities.

9. Next steps

This document represents a draft specification of the common elements of the DIF Management System (DMS). It focused on the common elements between configuration, performance and security management. This commonality included the protocol (CDAP), the Managed Object model (proposed RIB) and an initial high level architecture upon which higher level management functions can be provided.

Some of design decisions are documented. However the absence of an "ideal" Managed Object language, constraints the ability to make more decisions without some further work. This is discussed further in the following sections.

9.1. Managed object language selection

From the SoTA work, it can be seen that no one language fits all the criteria necessary for RINA. GDMO comes the closest in terms of features. However, GDMO comes with additional undesired features:

Notifications
: GDMO has built in support for notifications, however the implementation of these requires explicit protocol support (M_NOTIFICATION) message which is not present in CDAP.

Arbitrary operations
: GDMO comes with the capacity to define arbitrary operations. Worse GDMO allows arbitrary syntax to be applied to these operations (which results in a future maintenance headache) as operation parameters cannot be upgraded without breaking older RIB versions.

Multiple inheritance
: GDMO supports multiple object class inheritance. However it is possible to inherit a naming conflict. For example, the same attribute name declared in both parents. Theoretically, both attributes should exist however that is problematic to implement correctly.

Conditional packages
: GDMO comes with conditional packages, i.e. optional attributes or operations can be included based on a specified test expression (the condition). GDMO allows conditional package inclusion, based

on any condition. This adds complexity. The mix-ins or traits [10] concept is exploited in GDMO through package inclusion. However, its conditional packages also add complexity, as the condition needs to be evaluated at object instance access time i.e. at runtime on a per-operation (M_READ/M_WRITE/M_START/M_STOP) processing basis.

Ideally, conditional expressions should by limited to RIB "version" constraints, that can be resolved as the **object instance is created**, and remain in place for the duration of the object instances lifetime.

As the Managed Object concept will also be used by applications (performing Inter-Process Communication), then it is desirable to have trait or mix-in concepts available, to make application objects flexible, so they can consume notification events, and also add application specific attributes and operations.

9.2. RIB tooling selection

A key aspect not discussed in the SoTA is the availability of open source tools, for verifying and compiling the definitions into something more usable, in the form of executable code. The emphasis here is on *open source* to ensure RINA RIBs can be created and altered with a free tool-set.

Table 7. Available open source tooling

Language	Name	Licence	Source code	Compliance
YANG	pyang	New BSD	python	
YANG	jYang	GPL	Java	RFC 6020
GDMO - ASN.1	pyasn1	BSD	python	BER / CER / DER codecs
GDMO - ASN.1	asn1c	BSD	c,c++	BER/DER/XER/PER codecs
SID	none	n/a	UML	proprietary solutions only

[10] Technically, these have slightly different semantics when a name conflict occurs. Traits require explicit resolution while mix-ins implicitly resolve to the first declared

Language	Name	Licence	Source code	Compliance
GPB	gpb	New BSD	c,c++,java	Wide variety of languages supported.

It is clear that there is no ideal open source tool available. Therefore a custom tool will be need created based on a modified version of an open source tool. Finally, a concrete syntax needs to be chosen eg. XML, JSON, or another binary form to encode and decode messages.

The ideal language here is a custom version of GDMO supporting GPB as a data representation language. This allows the wide programming language support of the open source GPB tooling to be exploited, while retaining an object oriented, network management language feel.

9.3. RIB validation

The RIB will need some additional Managed Objects to support individual management functional areas. Validation of the RIB is necessary through further prototyping work. Both the proposed inheritance tree and the containment tree will need prototyping work to verify that they are sound, i.e. have no contradictions, and can be used to implement a working DMS.

Ideally, this work will be used to update the RINA specifications. Alternative implementations could then be "baked-off" against one another to ensure the managed object language captures enough semantics to enable inter-operability between different RIB implementations.

List of definitions

Application Process (AP)
The instantiation of a program executing in a processing system intended to accomplish some purpose. An Application Process contains one or more tasks or Application-Entities, as well as functions for managing the resources (processor, storage, and IPC) allocated to this AP.

Common Application Connection Establishment Phase (CACEP)
CACEP allows to Application Processes to establish an application connection. During the application connection establishment phase, the APs exchange naming information, optionally authenticate each other, and agree in the abstract and concrete syntaxes of CDAP to be used in the connection, as well as in the version of the RIB. It is also possible to use CACEP connection establishment with another protocol in the data transfer phase (for example, HTTP).

Common Distributed Application Protocol (CDAP)
CDAP enables distributed applications to deal with communications at an object level, rather than forcing applications to explicitly deal with serialization and input/output operations. CDAP provides the application protocol component of a Distributed Application Facility (DAF) that can be used to construct arbitrary distributed applications, of which the DIF is an example. CDAP provides a straightforward and unifying approach to sharing data over a network without having to create specialized protocols.

Distributed Application Facility (DAF)
A collection of two or more cooperating APs in one or more processing systems, which exchange information using IPC and maintain shared state.

Distributed-IPC-Facility (DIF)
A collection of two or more Application Processes cooperating to provide Interprocess Communication (IPC). A DIF is a DAF that does IPC. The DIF provides IPC services to Applications via a set of API primitives that are used to exchange information with the Application's peer.

Inter-DIF Directory (IDD)
The IDD is a component of DAP IPC Management that takes Application Naming Information and access control information and returns a list of DIF-names and their supporting DIF-names where the application may be found.

IPC-Process
An Application-Process, which is a member of a DIF and implement locally the functionality to support and manage IPC using multiple sub-tasks.

Layer
See DIF.

(N)-DIF
The DIF from whose point of view a description is written.

(N+1)-DIF
A DIF that uses a (N)-DIF. A (N)-DIF may only know the application process names of IPC-Processes in a (N+1)-DIF. Depending on the degree of trust between adjacent DIFs, (N)-DIF management may share other information with a (N-1)-DIF.

(N-1)-DIF
A DIF used by a (N)-DIF. The IPC Processes on the (N)-DIF appear as ordinary Application Processes to a (N-1)-DIF. Depending on the degree of trust between adjacent DIFs, (N)-DIF management may share other information with a (N-1)-DIF.

PDU
Protocol Data Unit, The string of octets exchanged among the Protocol Machines (PM). PDUs contain two parts: the PCI, which is understood and interpreted by the DIF, and User-Data, that is incomprehensible to this PM and is passed to its user

Processing system
The hardware and software capable of executing programs instantiated as Application Processes that can coordinate with the equivalent of a “test and set” instruction, i.e. the tasks can all atomically reference the same memory.

Resource Information Base (RIB)
The logical representation of information held by the IPC Process for the operation of the DIF.

1. Acronym list

AE, Application Entity (AP), AP	Application Process (AP)
API	Application Programming Interface
ASN.1	Abstract Syntax Notation.1
AVA	Attribute Value Assertion
BGP	Border Gateway Protocol
BSD	Berkeley Software Distribution
CACEP	Common Application Connection Establishment Phase
CDAP	Common Distributed Application Protocol
CIM	Common Information Model
CMIP	Common Management Information Protocol
CPU	Central Processing Unit
CQRS	Command Query Responsibility Segregation
CRC	Cyclic redundancy Check
CRUD	{Create, Read, Update, Delete}
DAF	Distributed Application Facility

DAP	Distributed Application Process
DCN , DIF	Distributed-IPC-Facility
DMS	DIF Management System
DNS	Domain Name Service
DMTF	Desktop Management Task Force
EBNF	Extended Backus–Naur Form
ECMA	European Computer Manufacturers Association
GDMO	Guidelines for the Definition of Managed Objects
GPB	Google Protocol Buffers
GPL	GNU Public Licence
HTTP	Hyper-Text Transport Protocol
IDD	Inter-DIF Directory
IDL	Interface Definition/Description Language
IEC	International Electro-technical Commission
IETF	Internet Engineering Task Force
IM	Instant Message
IPC	Inter-Process Communication

ISO	International Standards Organisation
IT	Information Technology
ITU	International Telecommunications Union
JSON	JavaScript Object Notation
LDD	Language Driven Development
MIB	Managed object Information Base
MO	Managed Object
MPLS	Multi-Protocol Label Switching
NFV	Network Function Virtualisation
NGOSS	New Generation Operations Systems and Software
NM	Network Management
OS	Operating System
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PKI	Public Key Infra-structure
RDN	Relative Distinguished Name

RIB	Resource Information Base
RINA	Recursive Inter-Network Architecture
RMT	Routing and Multiplexing Task
RPC	Remote Procedure Call
SDK	Software Development Kit
SDN	Software Defined Networking
SDU	Service Data Unit
SHA	Secure Hash
SID	Shared Information/Data Model
SMI	Structure for Managment Information
SMS	Short Messaging Service
SNMP	Simple Network Management Protocol
TM-Forum	Tele-Management Forum
TMN	Telecommunications Management Network
TTL	Time To Live
UML	Unified Modelling Language

UTF
Unicode Transformation Formats

VPN
Virtual Private Network

WBEM
Web Based Enterprise Management

WG
Working Group

References

- [asn1] ITU-T. *X.680-X.693 : Information Technology - Abstract Syntax Notation One (ASN.1) & ASN.1 encoding rules*. November 2008. Available online[11].
- [clem2006] Alexander Clemm, *Network Management Fundamentals*, Cisco Press, 2006.
- [cdap] ITU-T. *CDAP – Common Distributed Application Protocol Reference*. December 2010. Available upon request.
- [cmip] ITU-T. *X.711 : Information technology - Open Systems Interconnection - Common Management Information Protocol: Specification*. October 1997. Available online[12].
- [cosign] COSIGN. *COSIGN Project Website* 2014, 23 Feb. 2014, Available online[13].
- [cqs] Bertrand Meyers. *Command Query Separation* 23 Feb. 2014, Available online[14].
- [cqrs] Udi Dahan. *Clarified CQRS* 23 Feb. 2014, Available online[15].
- [cqrs2] Kanasz Rober. *Introduction to CQRS*, 21 Mar. 2013, Available online[16].
- [gpb] Google. *Google Protocol Buffers developer guide*. Available online[17].
- [gpb-syntax] Google. *Google Protocol Buffers: Language guide*. Available online[18].
- [greenicn] GreenICN. *GreenICN Project Website* 2013, 23 Feb. 2014, Available online[19].
- [json] Ecma International. *The JSON Data Interchange Format*. Standard ECMA-404, October 2013. Available online[20]

[11] http://www.itu.int/rec/T-REC-X.680-X.693-200811-I/en
[12] https://www.itu.int/rec/T-REC-X.711-199710-I/en
[13] http://www.fp7-cosign.eu/
[14] http://martinfowler.com/bliki/CommandQuerySeparation.html
[15] http://www.udidahan.com/2009/12/09/clarified-cqrs/
[16] http://www.codeproject.com/Articles/555855/Introduction-to-CQRS
[17] https://developers.google.com/protocol-buffers/
[18] https://google-developers.appspot.com/protocol-buffers/docs/proto?hl=es
[19] http://www.greenicn.org//
[20] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

- [leone] LEONE. *LEONE Project Website* 2013, 23 Feb. 2014, Available online[21].
- [m3010] ITU-T. *M.3010 : Principles for a telecommunications management network*. February 2000. Available online[22].
- [m3400] ITU-T. *M.3400 : TMN management functions*. February 2000. Available online[23].
- [netide] NetIDE. *NetIDE Project Website* 2014, 11 Jul. 2014, Available online[24].
- [netmod] Internet Engineering Task Force (IETF). *JSON Encoding of Data Modelled with YANG*. IETF Draft, April 2014. available online[25]
- [rfc3410] Internet Engineering Task Force (IETF). *Introduction and Applicability Statements for Internet Standard Management Framework*. IETF RFC 3410, December 2002.
- [rfc3535] Internet Engineering Task Force (IETF). *Overview of the 2002 IAB Network Management Workshop*. IETF RFC 3535, December 2002.
- [rfc4741] Internet Engineering Task Force (IETF). *NETCONF Configuration Protocol*. IETF RFC 4741, November 2006.
- [rfc6020] Internet Engineering Task Force (IETF). *YANG A Data modelling language for NETCONF*. IETF RFC 6020, October 2010.
- [rfc6095] Internet Engineering Task Force (IETF). *Extending YANG with Language Abstractions*. IETF RFC 6095, March 2011.
- [sid] TMForum, Information Framework (SID) Model - Concepts and Principles. GB922, Information Framework Suite Release 13.5, December, 2013.
- [spl] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
- [strauss] STRAUSS. *STRAUSS Project Website* 2013, 23 Feb. 2014, Available online[26].

[21] http://www.leone-project.eu/drupal/home/
[22] https://www.itu.int/rec/T-REC-M.3010/en
[23] https://www.itu.int/rec/T-REC-M.3400/en
[24] http://www.netide.eu//
[25] http://tools.ietf.org/html/draft-ietf-netmod-yang-json-00
[26] http://www.ict-strauss.eu/en//

- [tnova] T-NOVA. *T-NOVA Project Website* 2014, 23 Feb. 2014, Available online[27].
- [trilogy2] Trilogy 2. *Trilogy 2 Project Website* 2013, 23 Feb. 2014, Available online[28].
- [unify] UNIFY. *UNIFY Project Website* 2013, 23 Feb. 2014, Available online[29].
- [vanDeMeer] Sven van der Meer, *Middleware and Application Management Architecture*. PhD Thesis, Berlin Institute of Technology (TUB), October 2002, available online[30]
- [xml] World Wide Web Consortium (W3c). *Extensible Markup Language 1.0, Fifth Edition*. W3C Recommendation, November 2008.
- [x501] ITU-T. *X.501 : Information technology - Open Systems Interconnection - The Directory: Models*. October 2012. Available online[31]
- [x690] ITU-T. *X.690 : Information technology - ASN.1 encoding rules: SN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. November 2008. Available online[32]
- [x691] ITU-T. *X.691 : Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*. November 2008. Available online[33]
- [x693] ITU-T. *X.693 : Information technology - ASN.1 encoding rules: XML Encoding Rules (XER)*. November 2008. Available online[34]
- [x700] ITU-T. *X.710 : X.700 : Management framework for Open Systems Interconnection (OSI) for CCITT applications*. September 1992. Available online[35].
- [x710] ITU-T. *X.710 : Information technology - Open Systems Interconnection - Common Management Information Service*. October 1997. Available online[36].

[27] http://www.t-nova.eu//
[28] http://www.trilogy2.it.uc3m.es//
[29] http://www.fp7-unify.eu//
[30] http://opus4.kobv.de/opus4-tuberlin/frontdoor/index/index/docId/502
[31] https://www.itu.int/rec/T-REC-X.501-201210-I/en
[32] https://www.itu.int/rec/T-REC-X.690-200811-I/en
[33] https://www.itu.int/rec/T-REC-X.691-200811-I/en
[34] https://www.itu.int/rec/T-REC-X.693-200811-I/en
[35] https://www.itu.int/rec/T-REC-X.700/en
[36] https://www.itu.int/rec/T-REC-X.710/en

- [x711] ITU-T. *X.711 : Information technology - Open Systems Interconnection - Common Management Information Protocol*. October 1997. Available online[37].
- [x720] ITU-T. *X.720 : Information technology - Open Systems Interconnection - Structure of management information: Management Information Model*. 1993. Available online[38].
- [x722] ITU-T. *X.722 : Information technology - Open Systems Interconnection - Structure of management information: Guidelines for the definition of managed objects*. February 2000. Available online[39].

[37] https://www.itu.int/rec/T-REC-X.711/en
[38] https://www.itu.int/rec/T-REC-X.720/en
[39] https://www.itu.int/rec/T-REC-X.722/en

A. Annex: Managed Object reference

A.1. RIB Objects description

This section provides an initial version of the templates describing the objects of the RINA Resource Information Base proposed by PRISTINE. The class definitions will be refined and improved as the project evolves, probably with the inclusion of new classes required by the configuration, performance or security management tasks.

A.2. AccessControlPolicy

A.2.1. Description

Policy to determine the access control to a DIF. Determine how and when another IPC process can join the DIF. This object is part of a catalogue of all the possible AccessControlPolicy policies that the FlowAllocator can select.

A.2.2. Name Binding

{SecurityManagement, accessControlPolicyID = <value>}

A.2.3. Super Class

RINAPolicyConfig

A.2.4. Attributes

- int accessControlPolicyID: Id of the instance.

A.3. ActiveFlow

A.3.1. Description

Contains the information of an allocated flow.

A.3.2. Name Binding

{IPCResourceManager, activeFlowID = <value>}

A.3.3. Super Class

Top

A.3.4. Attributes

- int activeFlowID: ID of the instance.
- boolean state: State of the flow.
- string sourceName: Name of the source application/IPC process .
- string destinationName: Name of the destination application/IPC process.
- int sourcePort: Port used by the flow.
- AvailableDIF usedDIF: DIF used by the flow.

A.4. AddressAssignmentPolicy

A.4.1. Description

Policy that determines when to assign an address to an IPCProcess. This object is part of a catalogue of all the possible AddressAssignmentPolicy policies that the NamespaceManagement can select.

A.4.2. Name Binding

{NamespaceManagement, addressAssignmentPolicyID = <value>} Fupdate

A.4.3. Super Class

RINAPolicyConfig

A.4.4. Attributes

- int addressAssignmentPolicyID: ID of the instance.

A.5. Alarm

A.5.1. Description

Abstract object used to be inherited by specific alarms.

A.5.2. Name Binding

A.5.3. Super Class

Top

A.5.4. Attributes

- int alarmID: ID of the instance.

A.6. AllocateNotifyPolicy

A.6.1. Description

This policy determines when the requesting application is given an Allocate_Response primitive. In general, the choices are once the request is determined to be well-formed and a create_flow request has been sent, or withheld until a create_flow response has been received and MaxCreateRetires has been exhausted. This object is part of a catalogue of all the possible allocateNotifyPolicy policies that the FlowAllocatorInstance can select.

A.6.2. Name Binding

{FlowAllocator, allocateNotifyPolicyID = <value>}

A.6.3. Super Class

RINAPolicyConfig

A.6.4. Attributes

- int allocateNotifyPolicyID: ID of the instance.

A.7. AllocateRetryPolicy

A.7.1. Description

This policy is used when the destination has refused the create_flow request, and the FAI can overcome the cause for refusal and try again. This policy should re-formulate the request. This policy should formulate the contents of the reply. This object is part of a catalogue of all the possible AllocateRetryPolicy policies that the FlowAllocatorInstance can select.

A.7.2. Name Binding

{FlowAllocator, allocateRetryPolicy = <value>}

A.7.3. Super Class

RINAPolicyConfig

A.7.4. Attributes

- int allocateRetryPolicy: ID of the instance.

A.8. ApplicationEntity

A.8.1. Description

A task within an application process directly involved with exchanging application information with other APs.

A.8.2. Name Binding

{ApplicationProcess, applicationEntityID = <value>}

{IPCProcess, applicationEntityID = <value>}

A.8.3. Super Class

Top

A.8.4. Attributes

- int applicationEntityID: ID of the instance.
- string entityName: Name of the application entity.
- strin entityInstance: Instance of the application entity.

A.9. ApplicationProcess

A.9.1. Description

The instantiation of a program executing in a processing system intended to accomplish some purpose. An Application Process contains one or more tasks or Application-Entities, as well as functions for managing the resources (processor, storage, and IPC) allocated to this AP.

A.9.2. Name Binding

{ProcessingSystem, applicationProcessID = <value>}

A.9.3. Super Class

Top

A.9.4. Attributes

- int applicationProcessID = ID of the instance.
- string processName: Name of the application process.
- string processInstance: Instance of the application process.
- string synonimList: List of possible syonims of the application process.

A.10. AuthenticationPolicy

A.10.1. Description

Policy that the application processes use to authenticate each other. It can range from none, to user/password, Public Key Infrastructure (PKI) - based authentication, etc. This object is part of a catalogue of all the possible AuthenticationPolicy policies that the Neighbour can select.

A.10.2. Name Binding

{SecurityManagement, authenticationPolicyID = <value>}

A.10.3. Super Class

RINAPolicyConfig

A.10.4. Attributes

- int authenticationPolicyID: ID of the instance.

A.11. AvailabeDIF

A.11.1. Description

Represents a DIF that can be joined by the IPCProcess.

A.11.2. Name Binding

{IPCResourceManager, availabeDIFID = <value>}

A.11.3. Super Class

Top

A.11.4. Attributes

- int availabeDIFID: ID of the instance.

A.12. CompressionPolicy

A.12.1. Description

Policy that determines the type of compression that can be applyed to a SDU. This object is part of a catalogue of all the possible CompressionPolicy policies that the SDUProtection can select.

A.12.2. Name Binding

{SDUProtectionConfig, compressionPolicyID = <value>}

A.12.3. Super Class

RINAPolicyConfig

A.12.4. Attributes

- int compressionPolicyID: ID of the instance.

A.13. ComputingSystem

A.13.1. Description

The collection of all processing systems (some specialized) under the same management domain with no restrictions of their connectivity.

A.13.2. Name Binding

{Root, computingSystemID = <value>}

A.13.3. Super Class

Top

A.13.4. Attributes

- int computingSystemID: Id of the instance.

A.14. Connection

A.14.1. Description

Represents a connection between two IPC process.

A.14.2. Name Binding

{DataTransfer, connectionID = <value>}

A.14.3. Super Class

Top

A.14.4. Attributes

- int connectionID: ID of the instance.
- long srcAddr: Address of this IPC process
- long destAddr: Address of the neighbor IPCProcess
- int qosCubeId: Identification of the specific QoS cube that is being used.
- int portId: Port indentification of this connection.
- int srcCEPId: TODO: add

- int destCEPId: TODO add.

A.15. CredentialManagementPolicy

A.15.1. Description

Policy that determines the behaivor of the credentials within a DIF. Concepts like revocation and updating will be addressed by this policy. This object is part of a catalogue of all the possible CredentialManagementPolicy policies that the Neighbor can select.

A.15.2. Name Binding

{SecurityManagement, credentialManagementPolicyID = <value>}

A.15.3. Super Class

RINAPolicyConfig

A.15.4. Attributes

- int credentialManagementPolicyID: ID of the instance.

A.16. DAFManagement

A.16.1. Description

DAFManagement is responsible of DAF enrollment and overall management.

A.16.2. Name Binding

{ApplicationProcess, dafManagementID = <value>}

A.16.3. Super Class

Top

A.16.4. Attributes

- int dafManagementID: ID of the instance.
- WhateverCastName whateverCastName: name of the DAF.

A.17. DataTransfer

A.17.1. Description

Controls the behaviour of the transmittion of the data.

A.17.2. Name Binding

{IPCProcess, dataTransferID = <value>}

A.17.3. Super Class

Top

A.17.4. Attributes

- int dataTransferID: ID of the instance.

A.18. DIFManagement

A.18.1. Description

Object responsible of the managment of the DIFs.

A.18.2. Name Binding

{IPCP, difManagementID = <value>}

A.18.3. Super Class

DAFManagement

A.18.4. Attributes

- int difManagementID: ID of the instance.

A.19. DIFProperties

A.19.1. Description

Object containing one or more properties of a DIF.

A.19.2. Name Binding

{AvailableDIF, difPropertiesID = <value>

A.19.3. Super Class

Top

A.19.4. Attributes

- int difPropertiesID: ID of the instance.

A.20. DirectoryForwardingTableEntry

A.20.1. Description

Entry of the DirectoryForwardingTable. Contains the Destination IPC process Name and its address.

A.20.2. Name Binding

{DirectoryForwardingTable, directoryForwardingTableEntryID = <value>}

A.20.3. Super Class

Top

A.20.4. Attributes

- string destName: Name of the application.
- int address: Address of the application.

A.21. DirectoryForwardingTable

A.21.1. Description

Table that contains all the known Application Names and their respective addresses.

A.21.2. Name Binding

{NamespaceManagement, directoryForwardingTableID = <value>}

A.21.3. Super Class

Top

A.21.4. Attributes

- int directoryForwardingTableID: ID of the instance.

A.22. DTCPConfig

A.22.1. Description

This object determines the parameters of the DTCP. This object is part of the catalogue of all the possible QoS configurations accepted by the DIF. The selected configuration is in the object DTCP.

A.22.2. Name Binding

{EFCPPolicies, dtcpConfigID = <value>}

A.22.3. Super Class

Top

A.22.4. Attributes

- int dtcpConfigID: ID of the instance.
- int initSenderInaTimer. Should be approximately 2Δt. This must be bounded. A DIF specification may want to specify a maximum value.
- int initRcvInaTimer. Should be approximately 3Δt. This must be bounded. A DIF specification may want to specify a maximum value.
- bool retPresent. Indicates whether Retransmission Control (potentially with gaps) is in use.
- PolicyConfiguration rcvrTimerInaPolicy. This policy is used when DTCP is in use. If no PDUs arrive in this time period, the receiver should expect a DRF in the next Transfer PDU. If not, something is very wrong. The timeout value should generally be set to 3(MPL+R+A).
- PolicyConfiguration senderInaTimerPolicy. This policy is used when DTCP is in use. This timer is used to detect long periods of no traffic,

indicating that a DRF should be sent. If not, something is very wrong. The timeout value should generally be set to 2(MPL+R+A)

- PolicyConfiguration lostControlPDUPolicy. This policy determines what action to take when the PM detects that a control PDU (Ack or Flow Control) may have been lost. If this procedure returns True, then the PM will send a Control Ack and an empty Transfer PDU. If it returns False, then any action is determined by the policy.
- PolicyConfiguration rttEstimationPolicy. This policy is executed by the sender to estimate the duration of the retransmission timer. This policy will be based on an estimate of round-trip time and the Ack or Ack List policy in use.

A.23. DTCP

A.23.1. Description

Object that represents the DTCP which provides loosely bound mechanisms.

A.23.2. Name Binding

{Connection, dtcpID = <value>}

A.23.3. Super Class

Top

A.23.4. Attributes

- int dtcpID: ID of the instance.
- int initSenderInaTimer. Should be approximately 2Δt. This must be bounded. A DIF specification may want to specify a maximum value.
- int initRcvInaTimer. Should be approximately 3Δt. This must be bounded. A DIF specification may want to specify a maximum value.
- bool retPresent. Indicates whether Retransmission Control (potentially with gaps) is in use.
- PolicyConfiguration rcvrTimerInaPolicy. This policy is used when DTCP is in use. If no PDUs arrive in this time period, the receiver should expect a DRF in the next Transfer PDU. If not, something is very wrong. The timeout value should generally be set to 3(MPL+R+A).

- PolicyConfiguration senderInaTimerPolicy. This policy is used when DTCP is in use. This timer is used to detect long periods of no traffic, indicating that a DRF should be sent. If not, something is very wrong. The timeout value should generally be set to 2(MPL+R+A)
- PolicyConfiguration lostControlPDUPolicy. This policy determines what action to take when the PM detects that a control PDU (Ack or Flow Control) may have been lost. If this procedure returns True, then the PM will send a Control Ack and an empty Transfer PDU. If it returns False, then any action is determined by the policy.
- PolicyConfiguration rttEstimationPolicy. This policy is executed by the sender to estimate the duration of the retransmission timer. This policy will be based on an estimate of round-trip time and the Ack or Ack List policy in use.

A.24. DTPConfig

A.24.1. Description

This object determines the parameters of the DTP. This object is part of the catalogue of all the possible QoS configurations accepted by the DIF. The specific configuration is in the object DTP.

A.24.2. Name Binding

{EFCPPolicies, dtpConfigID = <value>}

A.24.3. Super Class

Top

A.24.4. Attributes

- int dtpConfigID: ID of the instance.
- int initATimer: Assigned per flow that indicates the maximum time that a receiver will wait before sending an ACK. Some DIFs may wish to set a maximum value for the DIF. If 0 means immediate acking.
- int seqNumRollOverThres: When the sequence number is increasing beyond this value, the sequence number space is close to rolling over, a new connection should be instantiated and bound to the same port-ids, so that new PDUs can be sent on the new connection.

- InitSeqNumPolicy initSeqNumPolicy: This policy allows some discretion in selecting the initial sequence number, when DRF is going to be sent.

A.25. DTP

A.25.1. Description

Object that represents the DTP which provides tightly bound mechanisms.

A.25.2. Name Binding

{Connection, dtpID = <value> }

A.25.3. Super Class

Top

A.25.4. Attributes

- int dtpID: ID of the instance.
- int initATimer. Assigned per flow that indicates the maximum time that a receiver will wait before sending an ACK. Some DIFs may wish to set a maximum value for the DIF. If 0 means immediate acking.
- int seqNumRollOverThres. When the sequence number is increasing beyond this value, the sequence number space is close to rolling over, a new connection should be instantiated and bound to the same port-ids, so that new PDUs can be sent on the new connection.
- InitSeqNumPolicy initSeqNumPolicy. This policy allows some discretion in selecting the initial sequence number, when DRF is going to be sent.

A.26. EFCPConfiguration

A.26.1. Description

Object that define a syntax of EFCP.

A.26.2. Name Binding

{DataTransfer, efcpConfigurationID = <value>}

A.26.3. Super Class

Top

A.26.4. Attributes

- int efcpConfigurationID: ID of the instance.
- unsigned int addrLength: The length of an address in bits: (4, 8, 16, 32, 64?)
- unsigned int qoSIdLength: The length of a QoS-id in bits: (4, 8)
- unsigned int portIdLength: The length of a Port-id in bits. (4, 8, 12, 16)
- unsigned int cepIdLength: The length of a CEP-id in bits.(4, 8, 12, 16)
- unisgned int seqNumLength: The length of a SequenceNumber in bits. (4, 8, 16, 32, 64)
- unsigned int lengthLength: The length of a PDU in bits. (4,8,16,32?)
- unsigned int qosCubeIdLength: The length of the QoSCube.
- unsigned int maxPDUSize: The maximum size allowed for a SDU written to/ read from this DIF. This parameter may be restricted by specific QoS cubes or even specific flows to a smaller value but not to a larger value.
- unsigned int maxSDUSize: The maximum size allowed for a PDU in this DIF.
- UnknownFlowPolicy unknownFlowPolicy. Specific selected UnknownFlowPolicy.

A.27. EFCPPolicies

A.27.1. Description

This object determines the policies that can be used by the EFCP. This object is part of the catalogue of all the possible QoS configurations accepted by the DIF.

A.27.2. Name Binding

{QoSCube, efcpPoliciesID = <value>}

A.27.3. Super Class

Top

A.27.4. Attributes

- int efcpPoliciesID: ID of the instance.

A.28. EncryptionPolicy

A.28.1. Description

Policy that determines the type of encryption that can be applyed to a SDU. This object is part of a catalogue of all the possible EncryptionPolicy policies that the SDUProtection can select.

A.28.2. Name Binding

{SDUProtectionConfig, encryptionPolicyID = <value>}

A.28.3. Super Class

RINAPolicyConfig

A.28.4. Attributes

- int encryptionPolicyID: ID of the instance.

A.29. EnrollmentPolicy

A.29.1. Description

Determines the enrollment policy of this DIF.

A.29.2. Name Binding

{DAFManagement, enrollmentPolicyID = <value>}

{DIFManagement, enrollmentPolicyID = <value>}

A.29.3. Super Class

RINAPolicyConfig

A.29.4. Attributes

- int enrollmentPolicyID: ID of the instance.

A.30. FlowAllocatorInstance

A.30.1. Description

The object corresponds to an instance of the flow allocator for an specific flow. The Flow Allocator-Instance determines which policies will be used to provide the characteristics requested in the Allocate.

A.30.2. Name Binding

{FlowAllocator,flowAllocatorInstanceID = <value>}

A.30.3. Super Class

Top

A.30.4. Attributes

- int flowAllocatorInstanceID. Port used by the flow.
- AllocateNotifyPolicy allocateNotifyPolicy. This policy determines when the requesting application is given an Allocate_Response primitive. In general, the choices are once the request is determined to be well-formed and a create_flow request has been sent, or withheld until a create_flow response has been received and MaxCreateRetires has been exhausted.
- AllocateRetryPolicy allocateRetryPolicy. This policy is used when the destination has refused the create_flow request, and the FAI can overcome the cause for refusal and try again. This policy should re-formulate the request. This policy should formulate the contents of the reply.
- NewFlowRequestPolicy newFlowRequestPolicy. This policy is used to convert an Allocate Request is into a create_flow request. Its primary task is to translate the request into the proper QoSclass-set, flow set, and access control capabilities.
- SeqRollOverPolicy seqRollOverPolicy. This policy is used when the SeqRollOverThres event occurs and action may be required by the Flow

Allocator to modify the bindings between connection-endpoint-ids and port-ids.

A.31. FlowAllocator

A.31.1. Description

The Flow Allocator is responsible for creating and managing an instance of IPC.

A.31.2. Name Binding

{ProcessingSystem, flowAllocatorID = <value>}

A.31.3. Super Class

Top

A.31.4. Attributes

- int flowAllocatorID: ID of the instance.
- AccessControlPolicy accessControlPolicy

A.32. FlowControlConfig

A.32.1. Description

This object determines the parameters of the flow control. This object is part of the catalogue of all the possible QoS configurations accepted by the DIF. The specific configuration is determined by the object FlowControl.

A.32.2. Name Binding

{DTCPConfig, flowControlConfigID = <value>}

A.32.3. Super Class

Top

A.32.4. Attributes

- int flowControlConfigID: ID of the instance.

- int rcbBytesThres. The number of free bytes below which flow control does not move or decreases the amount the Right Window Edge is moved.
- int rcvBytesxThres.The percent of free bytes below which flow control does not move or decreases the amount the Right Window Edge is moved.
- int rcvBuffersThres. The number of free buffers at which flow control does not advance or decreases the amount the Right Window Edge is moved.
- int rcvBufferxThres.The percent of free buffers below which flow control should not advance or decreases the amount the Right Window Edge is moved.
- bool rateBased. Indicates whether rate-based flow control is in use.
- int sendBytesThres. The number of free bytes below which flow control should slow or block the user from doing any more Writes.
- int sendBytesxThres. The percent of free bytes below, which flow control should slow or block the user from doing any more Writes.
- int sendBuffersThres. The number of free buffers below which flow control should slow or block the user from doing any more Writes.
- int sendBufferxThres. The percent of free buffers below which flow control should slow or block the user from doing any more Writes.
- ClosedWindowPolicy closedWindowPolicy. This policy is used with flow control to determine the action to be taken when the receiver has not extended more credit to allow the sender to send more PDUs. Typically, the action will be to queue the PDUs until credit is extended. This action is taken by DTCP, not DTP.
- PolicyConfiguration flowContOverrunPolicy. This policy determines what action to take if the receiver receives PDUs but the credit or rate has been exceeded.
- PolicyConfiguration recFlowConflictPolicy. This policy is invoked when both Credit and Rate based flow control are in use and they disagree on whether the PM can send or receive data.
- PolicyConfiguration recFlowControlPolicy. This policy allows some discretion in when to send a Flow Control PDU when there is no Retransmission Control.

A.33. FlowControl

A.33.1. Description

Controls the flow of data.

A.33.2. Name Binding

{DTCP, flowControlID = <value>}

A.33.3. Super Class

Top

A.33.4. Attributes

- int flowControlID: ID of the instance.
- int rcbBytesThres. The number of free bytes below which flow control does not move or decreases the amount the Right Window Edge is moved.
- int rcvBytesxThres.The percent of free bytes below which flow control does not move or decreases the amount the Right Window Edge is moved.
- int rcvBuffersThres. The number of free buffers at which flow control does not advance or decreases the amount the Right Window Edge is moved.
- int rcvBufferxThres.The percent of free buffers below which flow control should not advance or decreases the amount the Right Window Edge is moved.
- bool rateBased. Indicates whether rate-based flow control is in use.
- int sendBytesThres. The number of free bytes below which flow control should slow or block the user from doing any more Writes.
- int sendBytesxThres. The percent of free bytes below, which flow control should slow or block the user from doing any more Writes.
- int sendBuffersThres. The number of free buffers below which flow control should slow or block the user from doing any more Writes.
- int sendBufferxThres. The percent of free buffers below which flow control should slow or block the user from doing any more Writes.

- ClosedWindowPolicy closedWindowPolicy. This policy is used with flow control to determine the action to be taken when the receiver has not extended more credit to allow the sender to send more PDUs. Typically, the action will be to queue the PDUs until credit is extended. This action is taken by DTCP, not DTP.
- PolicyConfiguration flowContOverrunPolicy. This policy determines what action to take if the receiver receives PDUs but the credit or rate has been exceeded.
- PolicyConfiguration recFlowConflictPolicy. This policy is invoked when both Credit and Rate based flow control are in use and they disagree on whether the PM can send or receive data.
- PolicyConfiguration recFlowControlPolicy. This policy allows some discretion in when to send a Flow Control PDU when there is no Retransmission Control.

A.34. FlowProperties

A.34.1. Description

This object defines the characteristics of a flow. It is a selection of an specefic QoSCube.

A.34.2. Name Binding

{ActiveFlow, flowPropertiesID = <value>}

A.34.3. Super Class

Top

A.34.4. Attributes

- int flowPropertiesID: ID of the instance.
- Range avBandwidth: Average bandwidth in bytes/s.
- Range avSDUBandwidth: Average bandwidth in SDUs/s.
- Range peackBandDuration: Duration of the peack bandwidth.
- Range peackSDUBanDuration: Duration of the SDU peack.
- Range burstPeriod: Period of the burst.

- Range burstDuration: Duration of the bursts.
- real bitER: Bit error rate.
- int maxSDUSize: The maximum SDU size for the flow.
- bool partialDelivery: Indicates if partial delivery of SDUs is allowed or not.
- bool incompleteDelivery: Indicates if incomplete delivery is allowed.
- bool order: Indicates if SDUs have to be delivered in order.
- int maxAllowableGap: Indicates the maximum gap allowed among SDUs, a gap of N SDUs is considered the same as all SDUs delivered.
- int maxDelay: Indicates the maximum delay allowed in this flow.
- int jitter: Indicates the maximum jitter allowed in this flow

A.35. IntegrityCheckPolicy

A.35.1. Description

Policy that determines the type of integrity check that can be applied to a SDU. This object is part of a catalogue of all the possible EncryptionPolicy policies that the SDUProtection can select.

A.35.2. Name Binding

{SDUProtectionConfig, integrityCheckPolicyID = <value>}

A.35.3. Super Class

RINAPolicyConfig

A.35.4. Attributes

- int integrityCheckPolicyID: ID of the instance.

A.36. IPC

A.36.1. Description

IPC manages the communication between processing systems.

A.36.2. Name Binding

{ProcessingSystem, ipcID = <value>}

A.36.3. Super Class

Top

A.36.4. Attributes

- int ipcID: ID of the instance.

A.37. IPCManagement

A.37.1. Description

IPCManagement manages communication for the tasks ensuring that tasks have flows with given quality of service. IPC Management balances the communication requirements of the tasks with the efficiency requirements of the DAF.

A.37.2. Name Binding

{ApplicationProcess, ipcManagementID = <value>}

{IPCProcess, ipcManagementID = <value>}

A.37.3. Super Class

Top

A.37.4. Attributes

- int ipcManagementID: ID of the instance.

A.38. IPCProcess

A.38.1. Description

A task within a processing system which uses IPC.

A.38.2. Name Binding

{ProcessingSystem, ipcProcessID = <value>}

A.38.3. Super Class

Top

A.38.4. Attributes

- int ipcProcessID: ID of the instance.

A.39. IPCResourceManager

A.39.1. Description

The IPC Resource Manager provides the policy coordination among the elements of IPC Management of a DAF. The primary role of the IRM is managing the use of supporting DIFs and in some cases, participate in creating new DIFs.

A.39.2. Name Binding

{IPCManagement, ipcResourceManagerID = <value>}

A.39.3. Super Class

Top

A.39.4. Attributes

- int ipcResourceManagerID: ID of the instance.

A.40. MaxQPolicy

A.40.1. Description

This policy is invoked when a queue reaches or crosses the threshold or maximum queue lengths allowed for this queue. Note that maximum length may be exceeded. This object is part of a catalogue of all the possible MaxQPolicy policies that the Relaying can select.

A.40.2. Name Binding

{Relaying, maxQPolicyID = <value>}

A.40.3. Super Class

RINAPolicyConfig

A.40.4. Attributes

- int maxQPolicyID: ID of the instance.

A.41. MemoryManagement

A.41.1. Description

Manages the memory resources within a Processing System of the different Application Processes.

A.41.2. Name Binding

{ProcessingSystem, memoryManagementID = <value>}

A.41.3. Super Class

Top

A.41.4. Attributes

- int memoryManagementID: ID of the instance.

A.42. Multiplexing

A.42.1. Description

Multiplexing manages the multiplexing of multiple allocation requests onto the supporting DIF, maintaining pools of flows, etc. It is also possible for the DAF to support reverse multiplexing, i.e. combining several incoming flows into a single flow. This may require a policy for marks to be inserted in the data stream for synchronizing.

A.42.2. Name Binding

{IPCManagement, multiplexingID = <value>}

A.42.3. Super Class

Top

A.42.4. Attributes

- int multiplexingID: ID of the instance.

A.43. Namespace-Management

A.43.1. Description

Object responsible of the management of the name-space of the DIF.

A.43.2. Name Binding

{DAFManagement, namespaceManagementID = <value>}

{DIFManagement, namespaceManagementID = <value>}

A.43.3. Super Class

Top

A.43.4. Attributes

- int namespaceManagementID: ID of the instance.
- AddressAssignmentPolicy addressAssignmentPolicy: Specific AddressAssignmentPolicy used.

A.44. Neighbor

A.44.1. Description

Object that represents one neighbor of the DIF.

A.44.2. Name Binding

{DAFManagement, neighborID = <value>} {DIFManagement, neighborID = <value>}

A.44.3. Super Class

Top

A.44.4. Attributes

- int neighborID: ID of the instance.
- string apName: Name of the neighbor.
- int apIntance: Instance used.
- stringList synonimList: List of sysnonims for this apName.
- intList: supportingFlows: List of flows supported by this application/IPC
- CredentialManagementPolicy credentialManagementPolicy: Selected CredentialManagementPolicy
- AuthenticationPolicy authenticationPolicy: Selected AuthenticationPolicy.

A.45. NextHopTableEntry

A.45.1. Description

Entry of a NextHopTable.

A.45.2. Name Binding

{NextHopTable, nextHopTableEntryID = <value>}

A.45.3. Super Class

Top

A.45.4. Attributes

- int nextHopTableEntryID: ID of the instance.
- int destAddress: Address of the destination IPCProcess.

- int qosID: Selected QoS.
- intList nAddress: List of the next hop addresses that can be used to reach the destination.

A.46. NewFlowRequestPolicy

A.46.1. Description

This policy is used to convert an Allocate Request is into a create_flow request. Its primary task is to translate the request into the proper QoSclass-set, flow set, and access control capabilities. This object is part of a catalogue of all the possible NewFlowRequestPolicy policies that the FlowAllocatorInstance can select.

A.46.2. Name Binding

{FlowAllocator, newFlowRequestPolicyID = <value>}

A.46.3. Super Class

RINAPolicyConfig

A.46.4. Attributes

- int newFlowRequestPolicyID: ID of the instance.

A.47. NextHopTable

A.47.1. Description

Table that represents the next hopes that can be used to a destination in the DIF.

A.47.2. Name Binding

{PDUForwardingTableGenerator, nextHopTableID = <value>}

A.47.3. Super Class

Top

A.47.4. Attributes

- int nextHopTableID: ID of the instance.

A.48. PDUForwardingTableEntry

A.48.1. Description

Entry of the PDUForwardingTable.

A.48.2. Name Binding

{PDUForwardingTable, pduForwardingTableEntryID = <value>}

A.48.3. Super Class

Top

A.48.4. Attributes

- int pduForwardingTableEntryID: ID of the instance.
- int destAddress: Address of the destination IPCProcess.
- int qosID: Selected QoS.
- intList n-1Port: List of the n-1 ports that can be used to reach the destination.

A.49. PDUForwardingTableGenerator

A.49.1. Description

This object represents the generation of the routing tables.

A.49.2. Name Binding

{Resourceallocator, pduForwardingTableGenerator = <value>}

A.49.3. Super Class

Top

A.49.4. Attributes

- int pduForwardingTableGenerator: ID of the instance.
- PDUForwardingTableGeneratorPolicy pduForwardingGeneratorPolicy: policy used to determine the how to generate the routing tables, the next hop table and the PDU forwarding table.

A.50. PDUForwardingTableGeneratorPolicy

A.50.1. Description

Policy used to determine the how to generate the routing tables, the next hop table and the PDU forwarding table. This object is part of a catalogue of all the possible PDUForwardingTableGeneratorPolicy policies that the PDUForwardingTableGenerator can select.

A.50.2. Name Binding

{PDUForwardingTableGenerator,
pduForwardingTableGeneratorPolicyID = <value>}

A.50.3. Super Class

RINAPolicyConfig

A.50.4. Attributes

- int pduForwardingTableGeneratorPolicyID: ID of the instance.

A.51. PDUForwardingTable

A.51.1. Description

Table that represents the mapping between next hops and n-1 ports i.e which ports can be used to reach a given next hop.

A.51.2. Name Binding

{PDUForwardingTableGenerator, pduForwardingTableID = <value>}

A.51.3. Super Class

Top

A.51.4. Attributes

- int pduForwardingTableID: ID of the instance.

A.52. Processing System

A.52.1. Description

The hardware and software capable of executing programs instantiated as Application Processes that can coordinate with the equivalent of a “test and set” instruction, i.e. the tasks can all atomically reference the same memory.

A.52.2. Name Binding

{ComputingSystem, processingSystemID = <value>}

A.52.3. Super Class

Top

A.52.4. Attributes

- int processingSystemID: ID of the instance.

A.53. QoSCube

A.53.1. Description

Determine the set of parameters that define an specific quality of service for a flow. This class is a catalogue of the possible configurations accepted by the DIF. The specific configuration is determined by the object FlowProperties.

A.53.2. Name Binding

{Resourceallocator, qosCubeID = <value>}

A.53.3. Super Class

Top

A.53.4. Attributes

- int qosCubeID: ID of the instance.
- Range avBandwidth: Average bandwidth in bytes/s.
- Range avSDUBandwidth: Average bandwidth in SDUs/s.
- Range peackBandDuration:
- Range peackSDUBanDuration
- Range burstPeriod
- Range burstDuration
- real bitER
- int maxSDUSize: he maximum SDU size for the flow.
- bool partialDelivery: Indicates if partial delivery of SDUs is allowed or not.
- bool incompleteDelivery
- bool order: Indicates if SDUs have to be delivered in order.
- int maxAlloableGap: Indicates the maximum gap allowed among SDUs, a gap of N SDUs is considered the same as all SDUs delivered.
- int maxDelay: Indicates the maximum delay allowed in this flow.
- int jitter: Indicates the maximum jitter allowed in this flow

A.54. QueryDIFAllocator

A.54.1. Description

Object responsible of controlling the Allocation of a new DIF given a request by the IPCM to connect with one Application.

A.54.2. Name Binding

{IPCResourceManager, queryDIFAllocatorID = <value>}

A.54.3. Super Class

Top

A.54.4. Attributes

- int queryDIFAllocatorID: ID of the instance.

A.55. RateBasedFlowControlConfig

A.55.1. Description

This object determines the parameters of the RateBasedFlowControl. This object is part of the catalogue of all the possible QoS configurations accepted by the DIF. The specific configuration is determined by the object RateBasedFlowControl.

A.55.2. Name Binding

{RetransmissionControlConfig, rateBasedFlowControlConfigID = <value>}

A.55.3. Super Class

Top

A.55.4. Attributes

- int rateBasedFlowControlConfigID: ID of the instance.
- int sendingRate. Indicates the number of PDUs that may be sent in a TimePeriod. Used with rate-based flow control.
- int timePeriod. Indicates the length of time in microseconds for pacing rate-based flow control.
- PolicyConfiguration noRateSlowDownPolicy. This policy is used to momentarily lower the send rate below the rate allowed.
- PolicyConfiguration noOverrideDefaultPeckPolicy.This policy allows rate-based flow control to exceed its nominal rate. Presumably this would be for short periods and policies should enforce this.
- PolicyConfiguration rateReductionPolicy. This policy allows an alternate action when using rate-based flow control and the number of free buffers is getting low.

A.56. RateBasedFlowControl

A.56.1. Description

Controls the data sent through a flow using a rate flow control. This object is selected from the catalogue given by all the RateBasedFlowControlConfig objects.

A.56.2. Name Binding

{RetransmissionControl, rateBasedFlowControlID = <value>}

A.56.3. Super Class

Top

A.56.4. Attributes

- int rateBasedFlowControlID: ID of the instance.
- int sendingRate. Indicates the number of PDUs that may be sent in a TimePeriod. Used with rate-based flow control.
- int timePeriod. Indicates the length of time in microseconds for pacing rate-based flow control.
- PolicyConfiguration noRateSlowDownPolicy. This policy is used to momentarily lower the send rate below the rate allowed.
- PolicyConfiguration noOverrideDefaultPeckPolicy.This policy allows rate-based flow control to exceed its nominal rate. Presumably this would be for short periods and policies should enforce this.
- PolicyConfiguration rateReductionPolicy. This policy allows an alternate action when using rate-based flow control and the number of free buffers is getting low.

A.57. RegisteredApEntity

A.57.1. Description

Contains the information about a registered application: its name and the DIFs where it is registered.

A.57.2. Name Binding

{IPCResourceManager, registredApEntityID = <value>}

A.57.3. Super Class

Top

A.57.4. Attributes

- int registredApEntityID: ID of the instance.
- string name: Name of the application entity.
- int instance: Instance of the application entity.
- AvilableDIFList listDIF: The list of one or more DIFs in which the application is registered.

A.58. Relaying

A.58.1. Description

This object controls the outgoing and incoming data and decides which path will it follow.

A.58.2. Name Binding

{IPCProcess, relayingID = <value>}

A.58.3. Super Class

Top

A.58.4. Attributes

- int relayingID: ID of the instance.
- RMTQMonitorPolicy rmtQMonitorPolicy. Specific selected RMTQMonitorPolicy.
- RMTSchedulingPolicy rmtSchedulingPolicy. Specific selected RMTSchedulingPolicy.
- MaxQPolicy maxQPolicy. Specific selected MaxQPolicy.

A.59. ReplicationPolicy

A.59.1. Description

Represents the policy which determines the replication of the information maintained in the RIB to other IPCProcess. This object is part of a catalogue of all the possible ReplicationPolicy policies that the RIBDaemon can select.

A.59.2. Name Binding

{RIBDaemon, replicationPolicyID = <value>}

A.59.3. Super Class

RINAPolicyConfig

A.59.4. Attributes

- int replicationPolicyID: ID of the instance.

A.60. ResourceAllocationPolicy

A.60.1. Description

Represents the policy which determines how to distribute the resources between different members. This object is part of a catalogue of all the possible ResourceAllocationPolicy policies that the ResourceAllocator can select.

A.60.2. Name Binding

{Resourceallocator, resourceAllocationPolicyID = <value>}

A.60.3. Super Class

RINAPolicyConfig

A.60.4. Attributes

- int resourceAllocationPolicyID: ID of the instance.

A.61. ResourceAllocator

A.61.1. Description

ResourceAllocator may send work to different members for execution, either because they have unique capabilities, e.g. print a document, or for load balancing. In a DIF, routing and resource allocation are Task Scheduling policies.

A.61.2. Name Binding

{ApplicationProcess, resourceAllocatorID = <value>}

{IPCProcess, resourceAllocatorID = <value>}

A.61.3. Super Class

Top

A.61.4. Attributes

- int resourceAllocatorID: ID of the instance.
- ResourceAllocationPolicy resourceAllocationPolicy: selected ResourceAllocationPolicy.

A.62. RetransmissionControlConfig

A.62.1. Description

This object determines the parameters of the RetransmissionControl. This object is part of the catalogue of all the possible QoS configurations accepted by the DIF. The specific configuration is determined by the object RetransmissionControl.

A.62.2. Name Binding

{DTCPConfig, retransmissionControlConfigID = <value>}

A.62.3. Super Class

Top

A.62.4. Attributes

- int retransmissionControlConfigID: ID of the instance.
- int dataRexMsnMax. Indicates the number of times the retransmission of a PDU will be attempted before some other action must be taken.
- PolicyConfiguration retTimerExpiryPolicy. This policy is executed by the sender when a Retransmission Timer Expires. If this policy returns True, then all PDUs with sequence number less than or equal to the sequence number of the PDU associated with this timeout are retransmitted; otherwise the procedure must determine what action to take. This policy must be executed in less than the maximum time to Ack.
- PolicyConfiguration senderAckPolicy. This policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This is useful for multicast and similar situations where one might want to delay discarding PDUs from the retransmission queue.
- PolicyConfiguration recAckListPolicy. This policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This policy is used in conjunction with the selective acknowledgement aspects of the mechanism and may be useful for multicast and similar situations where there may be a requirement to delay discarding PDUs from the retransmission queue.
- PolicyConfiguration rcvAckPolicy.This policy is executed by the receiver of the PDU and provides some discretion in the action taken. The default action is to either Ack immediately or to start the A-Timer and Ack the LeftWindowEdge when it expires.
- PolicyConfiguration SendingAckPolicy. This policy allows an alternate action when the A-Timer expires when DTCP is present.
- PolicyConfiguration rcvControlAckPolicy. This policy allows an alternate action when a Control Ack PDU is received.

A.63. RetransmissionControl

A.63.1. Description

Controls the retransmission of the data.

A.63.2. Name Binding

{DTCP, retransmissionControlID = <value>}

A.63.3. Super Class

Top

A.63.4. Attributes

- int retransmissionControlID: ID of the instance.
- int dataRexMsnMax. Indicates the number of times the retransmission of a PDU will be attempted before some other action must be taken.
- PolicyConfiguration retTimerExpiryPolicy. This policy is executed by the sender when a Retransmission Timer Expires. If this policy returns True, then all PDUs with sequence number less than or equal to the sequence number of the PDU associated with this timeout are retransmitted; otherwise the procedure must determine what action to take. This policy must be executed in less than the maximum time to Ack.
- PolicyConfiguration senderAckPolicy. This policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This is useful for multicast and similar situations where one might want to delay discarding PDUs from the retransmission queue.
- PolicyConfiguration recAckListPolicy. This policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This policy is used in conjunction with the selective acknowledgement aspects of the mechanism and may be useful for multicast and similar situations where there may be a requirement to delay discarding PDUs from the retransmission queue.
- PolicyConfiguration rcvAckPolicy.This policy is executed by the receiver of the PDU and provides some discretion in the action taken. The default action is to either Ack immediately or to start the A-Timer and Ack the LeftWindowEdge when it expires.
- PolicyConfiguration SendingAckPolicy. This policy allows an alternate action when the A-Timer expires when DTCP is present.

- PolicyConfiguration rcvControlAckPolicy. This policy allows an alternate action when a Control Ack PDU is received.

A.64. RIBDaemon

A.64.1. Description

RIBDaemon has the dual role of ensuring information is available to tasks in a timely manner and ensuring that the distributed data of the DAF is managed. The former may entail periodic retrieval and/or distribution of information, or on specific events and maintaining a given level of replication, synchronization, etc. In networking terms, this combines what has traditionally been known as routing update and event management.

A.64.2. Name Binding

{ApplicationProcess, ribDaemonID = <value>}

{IPCProcess, ribDaemonID = <value>}

A.64.3. Super Class

Top

A.64.4. Attributes

- int ribDaemonID: ID of the instance.
- UpdatingPolicy updatingPolicy: Selected UpdateingPolicy.
- ReplicationPolicy replicationPolicy. Selected ReplicationPolicy.

A.65. RMTQMonitorPolicy

A.65.1. Description

Three parameters are provided to monitor the queues. This policy can be invoked whenever a PDU is placed in a queue and may keep additional variables that may be of use to the decision process of the RMT-Scheduling Policy and the MaxQPolicy. This object is part of a catalogue of all the possible MaxQPolicy policies that the Relaying can select.

A.65.2. Name Binding

{Relaying, rmtQMonitorPolicyID = <value>}

A.65.3. Super Class

RINAPolicyConfig

A.65.4. Attributes

- int rmtQMonitorPolicyID: ID of the instance.

A.66. RMTSchedulingPolicy

A.66.1. Description

This is the meat of the RMT. This is the scheduling algorithm that determines the order input and output queues are serviced. We have not distinguished inbound from outbound. That is left to the policy. To do otherwise, would impose a policy. This policy may implement any of the standard scheduling algorithms, FCFS, LIFO, longestQfirst, priorities, etc. This object is part of a catalogue of all the possible MaxQPolicy policies that the Relaying can select.

A.66.2. Name Binding

{Relaying, rmtSchedulingPolicyID = <value>}

A.66.3. Super Class

RINAPolicyConfig

A.66.4. Attributes

- int rmtSchedulingPolicyID: ID of the instance.

A.67. Root

A.67.1. Description

Base object of the containment model. Is a object that contains all the other objects.

A.67.2. Name Binding

{rootID = <value>}

A.67.3. Super Class

Top

A.67.4. Attributes

- int rootID: ID of the instance.

A.68. Scheduling

A.68.1. Description

Manages the resources within a Processing System for the different Application Processes.

A.68.2. Name Binding

{ProcessingSystem, schedulingID = <value>}

A.68.3. Super Class

Top

A.68.4. Attributes

- int schedulingID: ID of the instance.

A.69. SDUDelimiting

A.69.1. Description

Object that controls the delimiting of an SDU, so that the DIF can ensure being able to deliver the SDU to its recipient.

A.69.2. Name Binding

{IPCProcess, sduDelimitingID = <value>}

A.69.3. Super Class

Top

A.69.4. Attributes

- int sduDelimitingID: ID of the instance.

A.70. SDUProtectionConfig

A.70.1. Description

This object determines the parameters of the SDUProtection. This object is part of the catalogue of all the possible QoS configurations accepted by the DAF/DIF. The specific configuration is determined by the object SDUProtection.

A.70.2. Name Binding

{IPCManagement, sduProtectionConfigID = <value>}

A.70.3. Super Class

Top

A.70.4. Attributes

- int sduProtectionConfigID: ID of the instance.

A.71. SDUProtection

A.71.1. Description

SDUProtection appears in the DAF/DIF so that application data can be protected from compromise by theToplevel DIF. The tasks of a DAP may request multiple connections of different QoS cubes. This module provides any required protection for the SDUs this distributed application may pass to an IPC facility, distributed or otherwise. The last function performed on SDUs before they are delivered to the layer below is taking necessary precautions to safeguard their integrity. Any data corruption protection over the data and IPC including life-time guards (TTL, hop count) and/or encryption are performed here. SDU Protection may be different on each allocated flow.

A.71.2. Name Binding

{ActiveFlow, sduProtectionID = <value>}

A.71.3. Super Class

Top

A.71.4. Attributes

- int sduProtectionID: ID of the instance.
- EncryptionPolicy encryptionPolicy: Selected EncryptionPolicy.
- CompressionPolicy compressionPolicy: Selecte CompressionPolicy.
- TTLPolicy ttlPolicy. Selected TTLPolicy;
- IntegrityCheckPolicy integrityCheckPolicy: Selected IntegrityCheckPolicy.

A.72. SecurityManagement

A.72.1. Description

Controls the security policies and parameters of the DIF.

A.72.2. Name Binding

{DAFManagement, securityManagementID = <value>}

{DIFManagement, securityManagementID = <value>}

A.72.3. Super Class

Top

A.72.4. Attributes

- int securityManagementID: ID of the instance.

A.73. SeqRollOverPolicy

A.73.1. Description

This policy is used when the SeqRollOverThres event occurs and action may be required by the Flow Allocator to modify the bindings between connection-endpoint-ids and port-ids. This object is part of a catalogue of all the possible allocateNotifyPolicy policies that the FlowAllocatorInstance can select.

A.73.2. Name Binding

{Flowallocator, seqRollOverPolicyID = <value>}

A.73.3. Super Class

RINAPolicyConfig

A.73.4. Attributes

- int seqRollOverPolicyID: ID of the instance.

A.74. StateVector

A.74.1. Description

Loosely couples the two state machines of the EFCP, th DTP and the DTCP.

A.74.2. Name Binding

{Connection, stateVectorID = <value>}

A.74.3. Super Class

Top

A.74.4. Attributes

- int stateVectorID: ID of the instance.

A.75. Subscription

A.75.1. Description

Manages a subscription to the RIB.

A.75.2. Name Binding

{RIBDaemon, subscriptionID = <value>}

A.75.3. Super Class

Top

A.75.4. Attributes

- int subscriptionID: ID of the instance.
- string subscriber: name of the subscriber.
- string objName: Object subscribed to.
- stringList: operations

A.76. Top

A.76.1. Description

Top object of the Inheritance tree.

A.76.2. Attributes

A.77. TTLPolicy

A.77.1. Description

Policy that determines the type of TTL that can be applyed to a SDU. This object is part of a catalogue of all the possible EncryptionPolicy policies that the SDUProtection can select.

A.77.2. Name Binding

{SDUProtectionConfig, ttlPolicyID = <value>}

A.77.3. Super Class

RINAPolicyConfig

A.77.4. Attributes

- int ttlPolicyID: ID of the instance.

A.78. UnknownFlowPolicy

A.78.1. Description

When a PDU arrives for a Data Transfer Flow terminating in this IPC-Process and there is no active DTSV, this policy consults the ResourceAllocator to determine what to do. This object is part of a catalogue of all the possible UnknownFlowPolicy policies that the EFCPConfiguration can select.

A.78.2. Name Binding

{EFCPConfiguration, unknownFlowPolicyID = <value>}

A.78.3. Super Class

RINAPolicyConfig

A.78.4. Attributes

- int unknownFlowPolicyID: ID of the instance.

A.79. UpdatingPolicy

A.79.1. Description

Policy that determines when, what and how to update the objects of the RIB. This object is part of a catalogue of all the possible UpdatingPolicy policies that the RIBDaemon can select.

A.79.2. Name Binding

{RIBDaemon, updatingPolicyID = <value>}

A.79.3. Super Class

RINAPolicyConfig

A.79.4. Attributes

- int updatingPolicyID: ID of the instance.

A.80. WindowBasedFlowControlConfig

A.80.1. Description

Determine the set of parameters that define an specific window based flow control. This class is a catalogue of the possible configurations accepted by the DIF. The specific configuration is determined by the object WindowBasedFlowControl.

A.80.2. Name Binding

{RetransmissionControlConfig, windowBasedFlowControlConfigID = <value>}

A.80.3. Super Class

Top

A.80.4. Attributes

- int windowBasedFlowControlConfigID: ID of the instance.
- int maxClosedWindowQueueLength. The number PDUs that can be put on the ClosedWindowQueue before something must be done.
- int initialCredit. Added to the initial sequence number to get right window edge.
- PolicyConfiguration transControlPolicy. This policy is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control, e.g. the ECN bit is set.
- PolicyConfiguration rcvFlowControlPolicy. This policy is invoked when a Transfer PDU is received to give the receiving PM an opportunity to update the flow control allocations.

A.81. WindowBasedFlowControl

A.81.1. Description

Controls the data sent through a flow using a window flow control.

A.81.2. Name Binding

{RetransmissionControl, windowBasedFlowControlID = <value>}

A.81.3. Super Class

Top

A.81.4. Attributes

- int windowBasedFlowControlID: ID of the instance.
- int maxClosedWindowQueueLength. The number PDUs that can be put on the ClosedWindowQueue before something must be done.
- int initialCredit. Added to the initial sequence number to get right window edge.
- PolicyConfiguration transControlPolicy. This policy is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control, e.g. the ECN bit is set.
- PolicyConfiguration rcvFlowControlPolicy. This policy is invoked when a Transfer PDU is received to give the receiving PM an opportunity to update the flow control allocations.