Deliverable-5.3

Proof of concept of DIF Management System

Deliverable Editor: Micheal Crotty, TSSG

Publication date:	30-April-2015
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, proof-of-concept

Synopsis: This document describes the proof of concept of DIF Management System for the first iteration of PRISTINE DMS. This proof of concept covers design and implementation aspects of the DMS Manager, required RINA policies and interaction with the Management Agent. It also presents a brief overview of how to write PRISTINE specific management strategies.

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.

List of Contributors

Deliverable Editor: Micheal Crotty, TSSG
TSSG: Jason Barron
i2CAT: Bernat Gaston
NXW: Vincenzo Maffione
ATOS: Miguel Angel Puente
BISDN: Marc Sune
LMI: Sven van der Meer

Executive Summary

This deliverable describes the proof of concept of DIF Management System for the first iteration of PRISTINE DIF Management System (DMS). Specifically, it contains the design of the DMS manager and documents its current implementation.

It begins with a general introduction to OSI management, the DMS and some realistic deployment options and migration strategies.

The next section describes the DMS Manager in greater detail. It outlines best practices for modular software design and presents a layered design that allows deployment (and technology) flexibility. This is followed by a detailed description of the components that form each layer in the DMS.

This is followed by a worked example of writing a "management strategy". This details the java interfaces to use, and how implementations of these interfaces may be combined to provide desired results.

An update on the current design of the Management Agent (MA) is then presented. It outlines the improvements and the trade-offs of implementing the design. This is followed by the current implementation status and a benchmark against the requirements defined in earlier deliverables.

The validation section attempts to document the "validation" environment. It outlines the default (RINA) policy set that is required in the IRATI stack, the validation scenario with three nodes and the management strategies required to support the validation.

The final section attempts to summarise the work left to be done. For manager and management agent, additional implementation work is needed to support the monitoring scenarios as defined in [D52]. This includes the next steps to be undertaken and the (RINA) stack policies that will need to be implemented. The Manager also has some additional work needed to support a more complete (RINA) stack integration.

Table of Contents

List of Figures

List of acronyms

3PP	Third Party Provider
AMQP	Advanced Message Queuing Protocol
CACE	Common Application Connection Establishment
CDAP	Common Distributed Application Protocol
CLI	Command Line Interface
DAF	Distributed Application Facility
DAP	Distributed Application Process
DIF	Distributed-IPC-Facility
DMS	Distributed Management System
DSL	Domain Specific Language
DTCP	Data Transfer and Control Protocol
EFCP	Error and Flow Control Protocol
ES	Event System
FCAPS	Fault, Configuration, Accounting, Performance and Security
IPC	Inter-Process Communication
IPCP	Inter-Process Communication Process
IPCM	Inter-Process Communication Manager
NM	Network Management
NMS	Network Management Service
OODA	Observe, Orient, Decide, and Act
OSS	Operation Support System
PDU	Protocol Data Unit
RIB	Resource Information Base
RINA	Recursive Inter-Network Architecture
SDU	Service Data Unit
VLAN	Virtual LAN
VM	Virtual Machine

1. Introduction[1]

RINA is proposed as a clean state network architecture designed specifically to tackle the complexity inherent in many of todays' networks. 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.

1.1. Scope[2]

This deliverable focuses specifically on the work conducted to implement a proof of concept DIF management system. The DIF Management system follows an OSI (manager/agent) model. The first sections describe the design and implementation details of the DMS Manager components as it currently stands. A key feature of the DMS manager is the flexibility it offers in specifying management strategies. For example, for network configuration it allows a set of management strategies to define a declarative contract based specification for a DIF and provides a mechanism to verify DIF configuration via the RIB. The current implementation status of the management agent parts is also included.

1.2. Classic Management Paradigm[3]

The classic management paradigm, often referred to as Manager-Agent Paradigm, builds a management system out of Managers and Agents supported by Managed Objects (MO) and Management Information Bases (MIB). A manager realises management functionality, in classic terms grouped functionally as fault, configuration, performance, accounting and security management (also called the FCAPS). An agent controls

[1] file:///work/pristine/pristine-rawwiki/wp5/d53/1-introduction.asciidoc:1
[2] file:///work/pristine/pristine-rawwiki/wp5/d53/1-introduction.asciidoc:5
[3] file:///work/pristine/pristine-rawwiki/wp5/d53/11-arch.asciidoc:3

real resources, which can either be physical (network nodes, e.g. switches, routers) or logical (e.g. elements of a protocol stack or other non-physical artefacts). To simplify management instrumentation, an agent does not operate directly on the resource but uses MOs, which represent an abstraction of a resource for management purposes including communication, functional and information modelling aspects. MOs are collected in a MIB, which serves two purposes:

a. for a specification it contains all related specifications of MOs and

b. at runtime it provides the means to identify an MO (or a set of MOs) and to communicate with it.

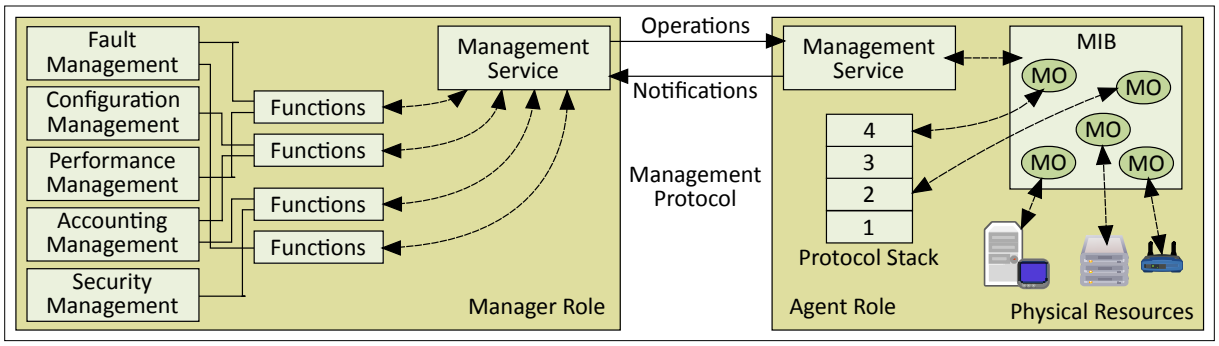

Figure 1. Manager Agent Paradigm

The communication between Manager and Agent is realised by a Management Service defining the management protocol and the information exchanged using it. The communication between Manager and Agent is often standardised in terms of notifications and (management) operations. An Agent sends notifications issued by MOs to the manager and the manager send (management) operations to the Agent, which in turn issues them to the respective MOs. This paradigm is well established in network management and has been adopted by ITU's OSI Management [X700], ITU's Telecommunication Management Network (TMN) [M3010] and IETF's Simple Network Management Protocol (SNMP)[rfc3410].

1.3. PRISTINE DIF Management System[4]

In PRISTINE, we adopt the classic Manager-Agent Paradigm introduced above to realise the PRISTINE DIF Management System (DMS). We consider three different domains:

[4] file:///work/pristine/pristine-rawwiki/wp5/d53/11-arch.asciidoc:18

a. RINA as the managed network,
b. the DMS as the PRISTINE contribution to a RINA management system and
c. an outside domain to demonstrate how other systems (e.g. legacy management systems, vendor-specific solutions) can be connected to the DMS.

The main purpose of this document is to describe the DMS, followed by how it integrates with RINA and finalised by a discussion on how external systems can be connected to the DMS.

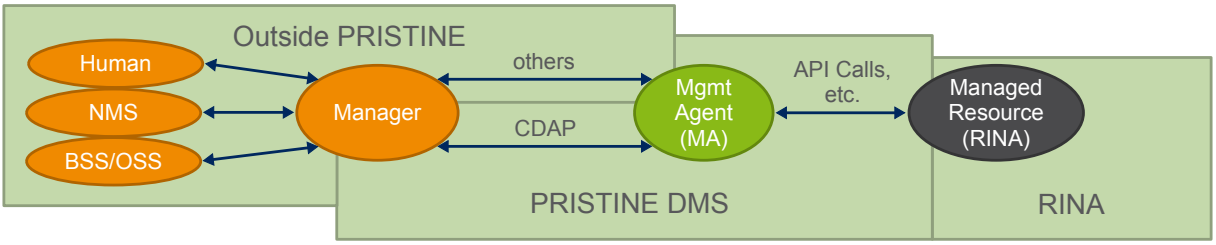

Figure 2. DMS Paradigm

In the PRISTINE DMS, the Agent is closely related to RINA. Here, we do not use any of the standardised MIBs to control MOs as such but the RINA Resource Information Base (RIB) and the resource object it contains. Thus, the DMS Management Agent (MA) maintains a RIB of the parts of a RINA network it controls. Communication between the MA and the RINA network is realised by the Common Distributed Application Protocol (CDAP), which is part of the RINA specification. CDAP provides all means to realise the Manager-Agent communication in a RINA-specific way, so there is no need to define a separate management protocol for the DMS.

The information exchanged between the DMS Manager and the MA is also based in the RINA RIB. The notifications the MA can send are defined in the RIB. Management operations the Manager can issue are the CDAP defined operations executed on RIB objects. CDAP provides for six operations: create, delete, start, stop, read and write. Thus the manager is able to create and delete RIB objects, read and alter information, and also start and stop them. These operations can be issued on a single RIB object for simple management or in a transaction covering multiple RIB objects for more complex management operations.

The manager can realise any management function (as per the introduced FCAPS but also additional functions if required) by decomposing a

complex management function into one or more CDAP operations on specific RIB objects. This eliminates the need for a more complex communication interface, as can be found in other management approaches and systems. In turn, this allows for a rather simple management middleware covering the basic communication (CDAP), functional (FCAPS) and information modelling (RIB) aspects of the RIB. Furthermore, the RINA Distributed IPC Facility (DIF) concept already implements a full domain model for management of flows with a particular DIF. This means that even the (management) domain model of the DMS can be inherited from RINA, including the addressing of any RINA resource.

The primary activity of the DMS can now be described as follows: a RINA system uses policies to configure the behaviours of a DIF and its components. The policies and the related resources are modelled in the RIB. The DMS MA provides access the RIB for the DMS Manager, which in turn uses management strategies to realise specific management functionality. We are using the term “management strategies” to distinguish DMS management from RINA control policies. In classic management terminology the DMS management strategies are management policies. In other words, the DMS manager maintains strategies and configuration templates, uses them to issue operations on RIB objects via the MA. In turn, the MA monitors RIB objects and issues notifications to the DMS Manager, triggering the strategies and the evaluation of configuration profiles.

This drastically simplified management middleware allows DMS users to focus on the actual management functionality expressed in DMS strategies. In RINA terms there are four different management points to be considered:

a. managing the underlying Operating System (OS-DMS),
b. managing a particular DIF (NM-DMS),
c. managing a collection of DIFs (AM-DMS) and finally
d. managing a RINA name space (NSM-DMS).

The DMS can provide for a standard set of strategies for each of these management points while allowing a DMS user to customise and/or extend this set. The required instrumentation for these tasks is built into the DMS middleware. The result is a simple, distributed, yet very powerful

management solution based on RINA principles and standards ready to manage RINA networks.

1.4. Deployment Options and Migration Strategies[5]

The PRISTINE DMS has been built to accommodate several deployment options and migration strategies. This includes mechanisms to connect outside systems, e.g. legacy systems such an existing Operation Support System (OSS) or an existing Network Management system (NMS), to the DMS. To be RINA compliant, the DMS will need to use the Common Application Connection Establishment (CACE) with an appropriate communication module. This CACE can then be mapped to CDAP (for native RINA communication) or to any other protocol effectively wrapping this very protocol – e.g. HTTP – with CDAP. The DMS is built to support a CACE using any communication protocol, ideally CDAP. This allows for multiple deployment options and migration strategies.

The starting point here is the DMS MA which is developed to communicate with a RINA network using CDAP and offers a CDAP interface for the DMS Manager. However, an initial deployment as well as the initial development of the DMS Manager might not (quite often cannot) enjoy a full CDAP implementation. For those deployments, the DMS has developed a CDAP connector which translates CDAP messages from and to the MA into any DMS specific communication protocol realised by a DMS Messaging System (DMS-MS).

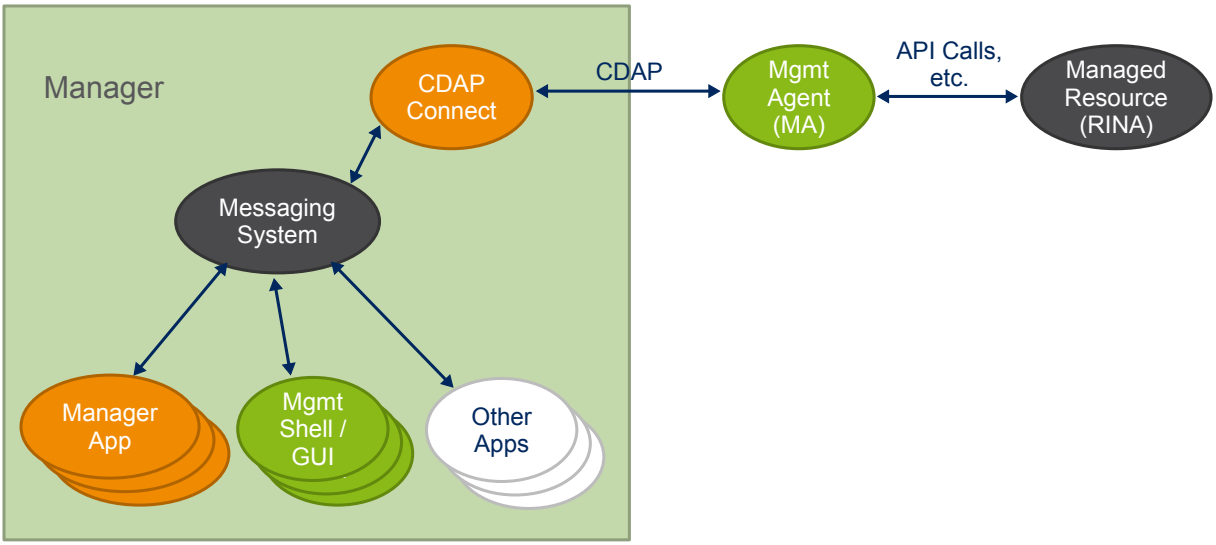

Figure 3. Deployment Option 1

[5] file:///work/pristine/pristine-rawwiki/wp5/d53/11-arch.asciidoc:52

This DMS-MS can use any underlying communication protocol and message encoding, as long as the actual messages are a direct representation (syntactical and semantical equivalent) to the original CDAP messages the MA understands. This allows management applications (distributed DMS managers), Command-Line Interfaces (CLI, e.g. management shells), Graphical User Interfaces (GUI, e.g. in a web browser using HTML5) and other applications to build the above introduced management strategies operating on RIB objects using the defined CDAP operations. On the other hand, this allows deployment options that can use (for multiple reasons) any underlying communication protocol and concrete messaging encoding.

In the example of the developed DMS, we are using a W3C Websocket [rfc6455] implementation for communication and a JSON schema [json-s] for message encoding. The CDAP Connector simply receives a JSON message on a Websocket and directly translates it into a CDAP protocol unit sent to the MA. Vice versa, the CDAP Connector receives CDAP protocol units from the MA and directly translates them into JSON encoded Websocket protocol units.

In case where a native CDAP implementation is available to the DMS Manager, the deployment option changes. Now, the DMS Manager components (manager applications, CLI, GUI, other applications) can directly use CDAP to communicate with the MA, eliminating the use of the CDAP Connector.

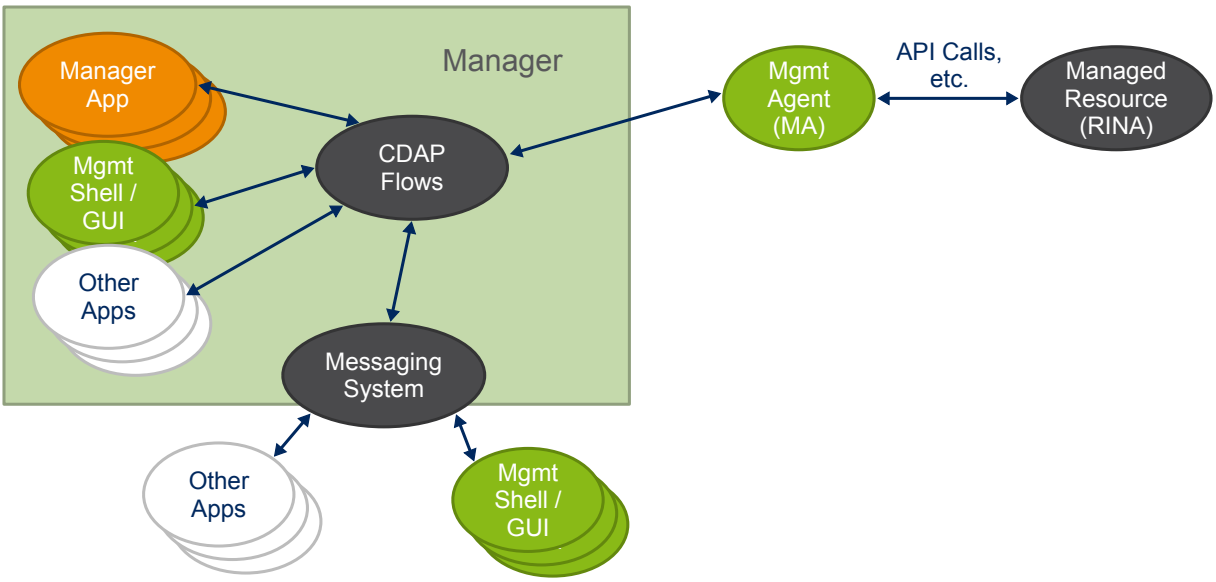

Figure 4. Deployment Option 2

However, legacy applications can still be connected to the DMS Manager using the already implemented DMS-MS. Several intermediate steps between the first deployment option (using a CDAP connector) and the second deployment option (using a native CDAP implementation and a messaging system for external components) are possible. These can be used to define and execute a migration strategy for the DMS Manager.

These migration strategies are directly supported by the current DMS implementation. The DMS design (and the actual implementation) realise a simple interface for management applications to manipulate RIB objects, regardless of the underlying protocol, be that a combination of Websocket/ JSON, native CDAP or something else. For each selected deployment option and identified migration strategy, the only requirement is to link the DMS Manager applications to a version of the DMS implementation supporting the required communication stack. No change in the developed management functions or applications is required. Designed this way, the DMS can support virtually any deployment option and migration strategy in a future-proof way.

1.5. Community-driven Management Strategies[6]

Managing the details of a RINA network can require a significant understanding of the underlying design principles, the RINA standards (e.g. for DIFs, CDAP, Naming and Address Architecture) and particular RINA implementations (e.g. the IRATI RINA implementation). The implied learning curve can impede an uptake of RINA as an alternative networking architecture especially because the management of a RINA network is a fundamental task for its successful operation. While the DMS is designed and built to support flexible management, not all RINA users will be willing to or able to develop their own management strategies.

The DMS addresses this issue by facilitating an active exchange of DMS Management Strategies among RINA users and of course among managers of RINA networks. First, the DMS platform described in this document provides a generic state machine that in turn allows building management strategies in many different flavours. We provide an OODA based strategy implementation, but one can easily use action policies (or Event-Condition-Action (ECA) policies), goal policies, utility functions or

[6] file:///work/pristine/pristine-rawwiki/wp5/d53/11-arch.asciidoc:76

even more advanced mechanisms such as Bayesian networks. The generic state machine also allows using a combination of the above.

Second, this flexible mechanism to use virtually any implementation of a strategy (as long as it is using the provided generic state machine as underlying execution environment) can be used to create a market and exchange place for DMS Management Strategies. This market place can supply standard strategies approved by RINA experts (or the Pouzin Society as the RINA guardian) as well as any strategy of any user, user group, organisation or company. The idea of this market place is very similar to the well-known app stores for mobile phone operating systems and platforms.

2. DMS Manager[7]

The platform for the DMS manager is designed to facilitate the flexibility necessary to manage a RINA deployment. The design decisions that guided the development of the DMS Manager are listed below.

First, all parts of the DMS Manager communicate by exchanging messages. This leads to an event-sourced [fowler05] design in which all interfaces are defined by events. An event can be sent, received, manipulated, logged, stored and archived. Events that have been stored can be re-run in the DMS for testing, debugging or demonstration purposes. Complete management scenarios can be explored in any detail using those re-runs.

Second, the DMS platform is designed as a *distributed* management system. A messaging system connects all DMS components. Any number of components can be connected, limited only by performance and scale limitations of the actually used messaging system. Different messaging systems can be interconnected to join different worlds, for instance RINA CDAP with AMQP, distributed (hash-) maps and Websockets. This allows for integration of enterprise messaging systems (AMQP) with web applications (Websocket) and RINA networks. A distributed system requires configuration management for components and their connections. Currently, the platform uses Apache Zookeeper for distributed configuration management. Zookeeper can be easily replaced by any other equivalent system.

Third, the DMS platform uses a layered design focusing on a modular implementation. Layers define the platform's functional areas. Modules implement a layer's functionality or parts of it. Modules will have a direct, acyclic dependency graph from the top layer to the bottom layer realising a complete deployable stack. Changing modules in the stack allows migrating from one deployment option to another. These changes can be done for all or only a few management applications at a time.

Fourth, the DMS platform is using a shared event taxonomy and a Domain Specific Language (DSL) approach to define events. The event taxonomy defines all elements of a platform event. The DSL approach then allows defining complete event languages and dialects. The main DSL used in

[7] file:///work/pristine/pristine-rawwiki/wp5/d53/2-manager-design.asciidoc:1

the platform is a DSL realising CDAP. Other DLSs provided are for the control of management applications and legacy integration as well as for specific demonstrations. The DSL approach has built-in mechanisms for automated translation between different DSLs and different DSL dialects.

Fifth, the DMS platform supports integration with legacy (management) systems. The combination of different messaging systems with the DSLs approach (automated translation) allows mapping communication and information being translated automatically between very different technology worlds.

Sixth, the DMS platform is designed to generate most runtime code from specifications. For instance, a DSL automatically provides an interface (Application Programming Interface – API) to create, send, receive and access events for that DSL. The connection to a particular messaging system only requires calling the respective module implementing the access point. Tools are provided for defining, maintaining, using the event DSLs, including runtime tools for distributed logging, archiving and re-running event sets. All management applications using a defined event DSL can use an automatically generated Command Line Interface (CLI) for control. This CLI also allows for scripting single or complete management and control tasks. This in turn allows a RINA system administrator to use their tool of choice (e.g. bash scripts, configuration profiles) to run otherwise tedious management tasks.

2.1. Layered, Modular Platform Design[8]

The platform design comprises four layers each with well-defined scope and interactions. The scope of the layers goes from platform generic (management application) down to runtime specific (runtime platform). Top layers use functionality from lower layers (without cross-layer interactions). The use of functionality can be contractual and policy controlled if required. Bottom layers provide functionality to higher layers. The provisioning of functionality can be done by an API, a library, a protocol or a service access point, to name a few options.

[8] file:///work/pristine/pristine-rawwiki/wp5/d53/2-manager-design.asciidoc:17

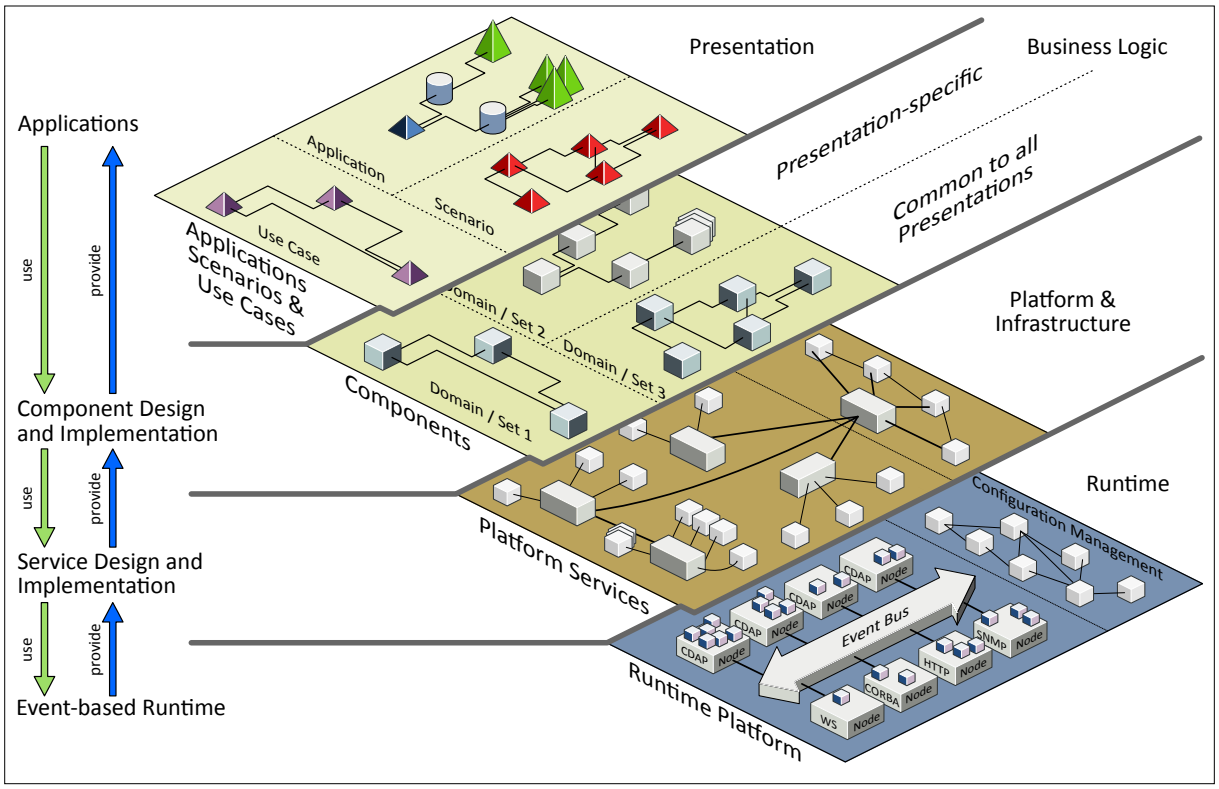

Figure 5. Layered, Modular Platform Design

Layers classify the parts of the DMS platform and determine levels of abstractions with four layers being identified: presentation (for demonstrations, prototypes and management solutions), business logic (with two sub-layers), platform and infrastructure (for communication and common services), and runtime (for off-the-shelve, open source or 3rd party runtime solutions).

The presentation layer provides a view for applications, scenarios and use cases. Specific requirements of those are defined and realised here. They can be developed in isolation or with stronger interactions. The main scope of this layer is to present the applications usually to a human being.

The business logic layer provides for a component view. Its main scope is to implement the required business logic for the presentation layer in form of components as the unit of deployment (deployable software artefacts). If required, it can be subdivided into a part focusing on specific logic for specific presentations (called presentation-facing business logic) and a part focusing on more generic business logic supporting two or more specific presentations (called common business logic). Business logic components can be grouped in sets or domains supporting different application deployment options and strategies.

The platform and infrastructure layer builds and deploys the actual distributed platform using runtime services. Services can be grouped if required. This layer also needs to provide a means for distributed configuration management of the platform. Ideally it must allow for an automated deployment or at least for very flexible scripting of the deployment, i.e. there should be no single deployment scenario being *mandated* by this layer.

The runtime layer provides the runtime platform realising the DMS distributed event system and the configuration management components. Here we find the off-the-shelve, open source, 3rd party or self-developed runtime components for the messaging system, all tools for DSL processing, component deployment and configuration management.

2.2. Layered, Modular Implementation Design[9]

The design of the implementation directly follows the platform design. On the presentation layer we find the management demonstrations and applications. The business logic layer is sub-divided as discussed above. The presentation-facing part builds the DMS Manager applications. The common logic part builds the DMS Management Strategies (currently using OODA state machine), the CDAP DSL and the management application DSLs (for communicating with the manager) and a RIB implementation supporting all management applications. The platform and infrastructure layer provides the messaging system (current) and CDAP (future) which is then realised by the runtime layer.

2.3. Layered, Modular Implementation Packages[10]

The implementation of the DMS platform provides for a number of modules with a directed, acyclic dependency graph. The naming of the packages links them to the respective layer of the design. Packages starting with 5- are in the presentation layer. Packages starting with 4- are presentation-facing business logic components. Packages starting with 3- are common business logic components. Finally, packages starting with 2- are platform and infrastructure components.

[9] file:///work/pristine/pristine-rawwiki/wp5/d53/2-manager-design.asciidoc:35
[10] file:///work/pristine/pristine-rawwiki/wp5/d53/2-manager-design.asciidoc:40

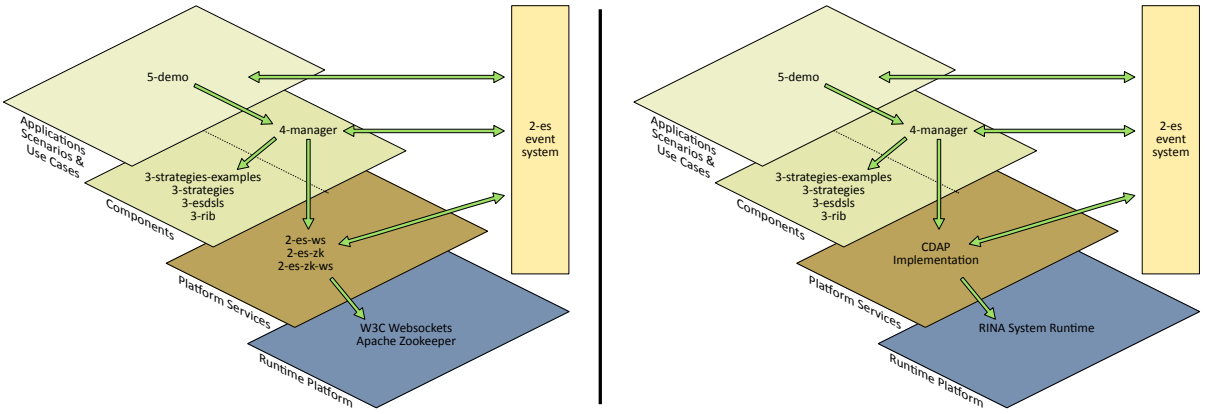

Figure 6. Layered, Modular Implementation Packages

The presentation layer has currently one implementation 5-Demo to demonstrate DMS Manager strategies. It will be extended in the PRISTINE project with the implementations of the use cases. The business logic layer contains the manager (4-Manager) and several common components. 3-RIB implements a RIB view for the strategies. The package 3-EsDSLs defines the event DSLs for the DMS, including the CDAP representation, a trigger language for simulating network events and the control language for managers and strategies. Based on those two, the package 3-Strategies implements the generic OODA strategy and all required tools to define and deploy an OODA strategy. Finally, the package 3-Strategies-Examples provides a number of example and testing strategies. Beside the examples, all packages should require very little change once they mature.

For platform and infrastructure the DMS Manager package (4-Manager) can select between multiple options to use, each providing messaging systems in different constellations. Ideally, the DMS platform ships with a native CDAP implementation, as planned for the future. As a starting point, the platform implements two different messaging packages: 2-ES-WS and 2-ES-ZK-WS. The first realises a Websocket based messaging system with manual configuration of connections. Here, a server component must be started and then all other components must be manually connected to it (this can be scripted but the DMS platform does not provide for further automated deployment). The second one provides the same messaging connector (Websocket) but with an automated configuration of connections using Apache Zookeeper. For deployment, one only needs to provide access to a Zookeeper cluster and the implementation will gather all other configuration automatically, including all client configuration. The package 2-ES-ZK implements the

required functionality to communicate with Zookeeper, including creation of ephemeral configuration objects and re-connectors for automated re-creation of connections in case of network (or Websocket) failure.

One package, uniquely, provides functionality across all platform layers: 2-ES. This package implements the complete distributed event system including the shared taxonomy, the used DSL framework, event and DSL APIs, messaging system connector interfaces, runtime tools and components, automated generation of CLI and event processing. In addition, this package implements the generic state machine used by the management strategies. Last but not least, the package provides a complete visualisation framework for events using the event taxonomy. This visualisation framework is realised in HTML5 and can be run in any Websocket-enabled HTML5 web browser. The event visualizer can be flexibly configured for online event viewing (live runtime events) or for offline event analysis (including multiple options for visual event correlation).

2.4. DMS Manager Architecture[11]

This design guides in the DMS Manager software architecture. DMS applications can run in any execution environment, e.g. inside an OpenStack compute node, in a Docker container or on a native operating system. For communication, it can use multiple messaging systems, e.g. Websockets, AMQP messaging solutions or native CDAP. On top of those messaging systems, we build communication adaptors which simplify the configuration and usage of those messaging systems. This means that a DMS application does not need to be concerned with any details of the messaging system.

[11] file:///work/pristine/pristine-rawwiki/wp5/d53/22-manager-arch.asciidoc:1

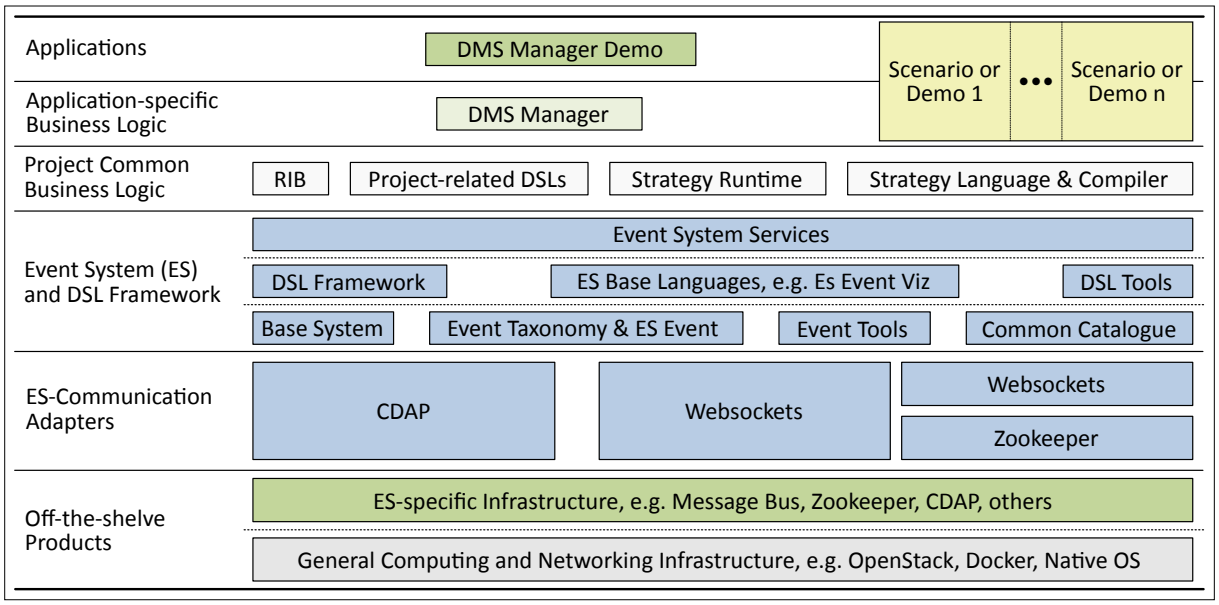

Figure 7. DMS Manager Software Architecture

The lowest level of abstraction is the DMS Event System (ES). It implements the base system (utility classes); the event taxonomy and the infrastructure for creating, sending, receiving and decomposing events; standard event tools and a common catalogue. The DSL framework allows for an easy definition of events and their content for a particular domain [12]. This includes automated syntax and semantic checks and automated event processing against a specified DSL. ES services then provide the simple access to that functionality.

With this underlying infrastructure, the actual management system and its applications can focus on building their specific business logic. In case of the DMS, this is a RIB view, management related DSLs, and an implementation of management strategies with related tools to create and deploy them into a DSM. Finally, we can build any number of management applications, demonstrations or scenario specific solutions.

[12] In PRISTINE's case, that domain is management of RINA DIFs, and DAFs.

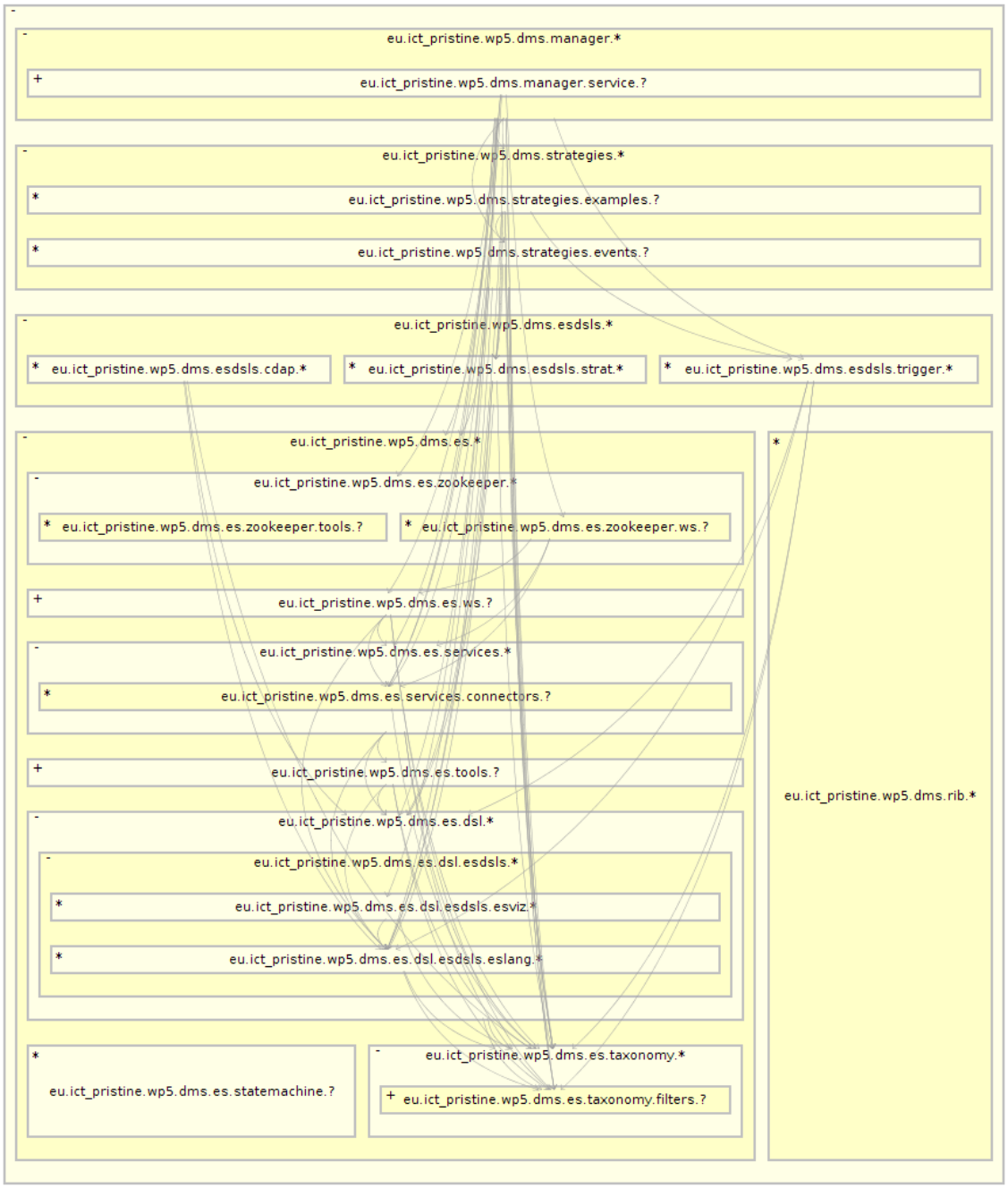

Figure 8. DMS Implementation: Architecture View

The implementation architecture directly follows the described software architecture.

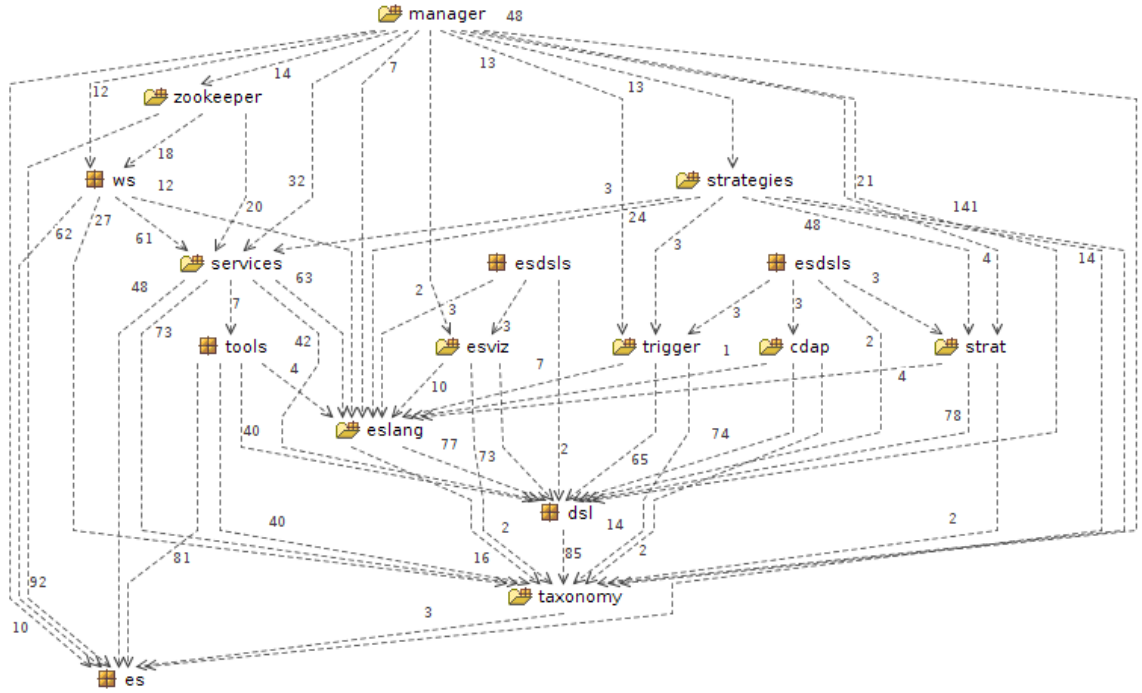

Figure 9. DMS Architecture Package and Dependency View

The introduced packages' dependencies represent a directed, acyclic graph. The dependencies with weight factor are shown in Figure 9.

2.5. Detailed Description – 2-ES[13]

This section details the underlying Event System (ES). Each part of this package is discussed in detail. Structure 101 [S101] is used to generate a package architecture figure (with dependencies) and package composition view(s) are provided to show how the different parts of this package link together.

[13] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:3

2.5.1. ES Event Taxonomy and ES Event[14]

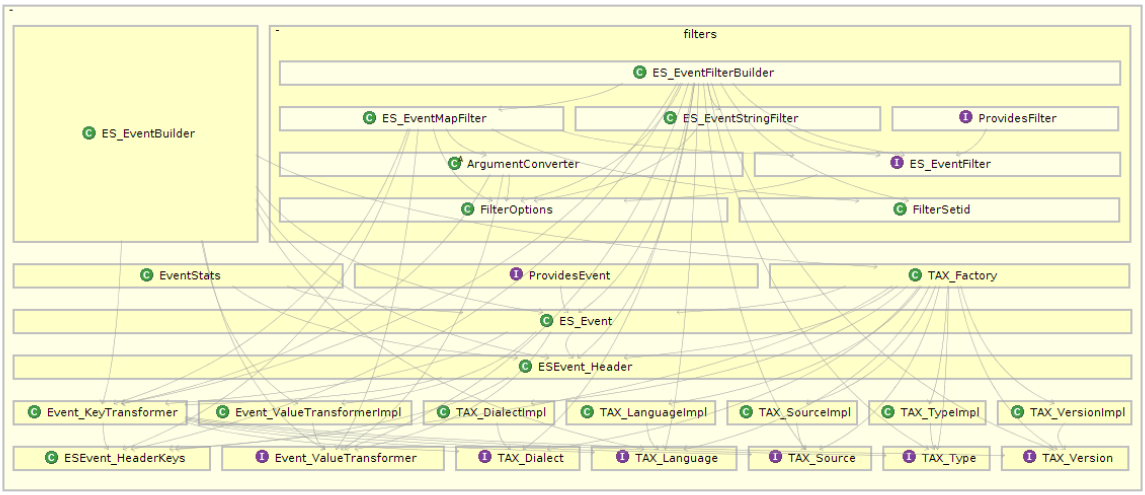

Figure 10. ES Taxonomy and ES Event Architecture View

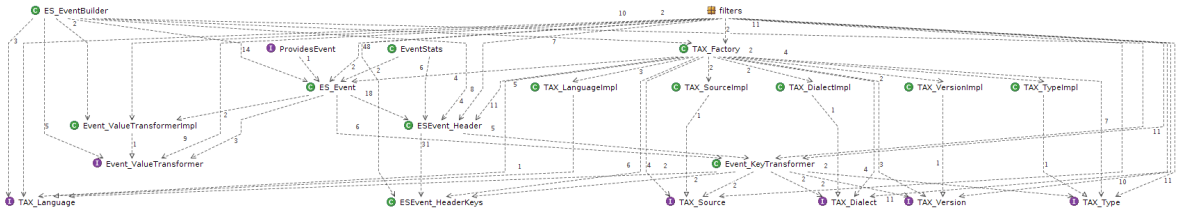

Figure 11. ES Taxonomy and ES Event Composition View

ES Taxonomy[15]

The event taxonomy defines events exchanged between components. In the taxonomy, an event contains a fixed header part and a flexible payload part. The header is used to automatically process events, integrate them with a DSL and route events in the messaging system.

An event header is defined by five primitives: type, source, language, dialect and version. The event type characterises an event and can be used for categorisation, for example using the FCAPS management functions. The event type is arbitrary but must be understood by sender and receiver. Standard event types can be found in the Event DSL described below.

The event source indicates the sender of an event. This can be done in more generic terms (e.g. categorising an application as OSS or NMS or Manager

[14] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:8
[15] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:19

or MA) or in more specific terms (e.g. providing also an address or specific name of the sender). The event source is arbitrary and can be specific to a given event DSL.

The event language specifies which event DSL being used for the payload part. This must be specified using the actual DSL identifier, it is not arbitrary. For automated event processing the language identifier is used to request the DSL for syntax and semantic checks on incoming events or events to be sent. It is also used in a similar way in the visualizer to process otherwise arbitrary events.

The event (language) dialect indicates a particular dialect of the event language used in the given event. This identifier must specify an existing dialect defined in the event DSL. It is then used to further refine the automated event processing described above for the event language.

The event (language) version indicates a specific version of the event DSL the event is using. This header primitive is introduced to simultaneously support multiple versions of the same event DSL yet being able to monitor and debug the components of the system.

All header primitives are defined as interfaces (prefixed by TAX_) and supported by a default implementation. Since they are standardised in the event taxonomy, there should be little need to extend these default implementations. However, for integrating the headers into a higher-level concept (such as the DSL framework), some minor specialisations might be necessary.

The event header also contains some primitives that are automatically completed when an event is generated. They supply identifier and time information. The identifier and a hash code for that identifier are created using an ES standard method [16]. This ensures that all events have a unique identifier. The second automatically generated primitive is a time stamp for event creation. The time will be stored as a Java epoch time using the standard second time scale and a time string which uses the following format: yyyy-MM-dd HH:mm:ss.SSS.

[16] The current standard method uses Java VM and process IDs for the generation. Other, non-Java based, methods could use for instance UUIDs or GUIDs for unique identifiers.

ES Event[17]

The ES Event implements an event with header and content maps plus transformers. Both header and content map are key/value based hash maps. The header map is immutable. The content map is mutable.

An event builder is provided to simplify event creation. It generates the event header and initialises an event. The event builder is the only way to create events. It performs a complete validity check on all given input and only creates an event if it is valid, i.e. it conforms to the event taxonomy. Otherwise no event is created.

Connecting event creation and processing to specific external or internal data source is realised using transformers. One can transform keys and/or values. A standard key transformer is provided for the header primitives. All other transformers are domain specific and cannot be generalised.

ES Event Filters[18]

A standard task in event systems is filtering. The ES provides a filter framework for simple (string-based) and complex (full event map) filters. Using this framework, one can easily implement any type of specific event filter. A filter builder is provided to simplify the creation of filters. The builder does provide basic constraint primitives such as: may contain, must contain, and must not contain. All of them can be applied to key and values separately. Event taxonomy constraint primitives are supported as well (e.g. of-type and of-version). The filter builder also allows for complete event or event header filtering.

ES Event Tools[19]

An ES event compiler takes the current event taxonomy and definition and generates specific target compilations. Currently supported are targets for JavaScript and [asciidoc] documentation (in tables or plain text). Other targets can easily be added if required.

An ES log reader can read logged or archived ES events (in JSON format) and re-create the original events as a sorted list (using time for sorting

[17] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:39
[18] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:47
[19] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:51

order). This log reader is used by other packages to create tools for re-running complete event sets through a messaging system or visualizer.

2.5.2. DSL Framework[20]

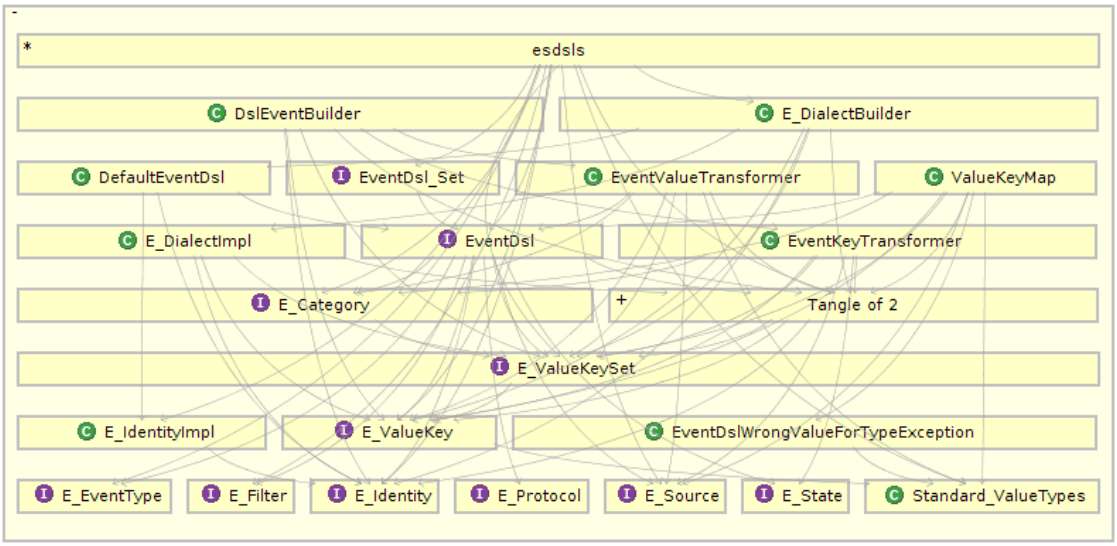

Figure 12. DSL Framework Architecture View

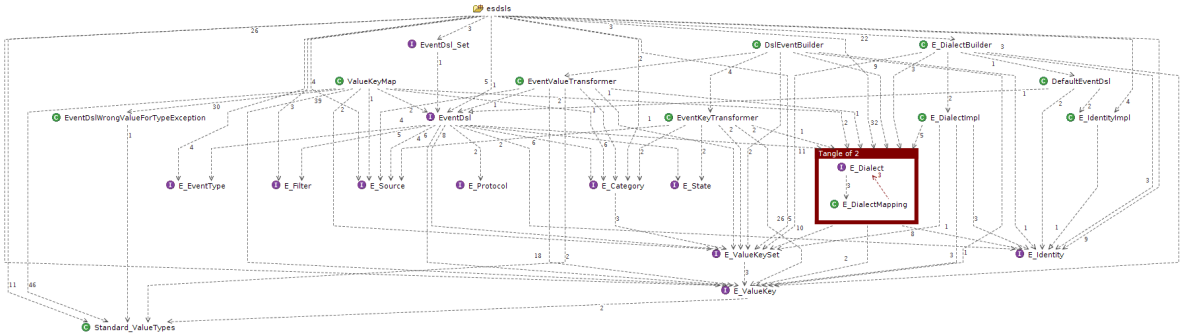

Figure 13. DSL Framework Composition View

The DSL framework links the Event System with a domain language. It does this by facilitating the simple creation of Domain-specific Languages with automated integration of the Event System for event processing. Additionally, DSLs from the DSL framework can be directly used to create command line interfaces and remote control procedures.

A DSL is defined by: an identifier (the language name), dialects, event types, categories, value keys, value key sets, states, filters and (experimental)

[20] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:58

protocols. Only the identifier is a mandatory DSL element. All other elements can be added as required.

Identifier[21]

The DSL identifier contains multiple keys. As well as the actual identifier (called name), it also provides a version (the language version) and a date (the date the language version was finalised and published). This triple is used to process DSLs. The identifier and the version keys are automatically mapped to the event taxonomy keys for event language and event version. A specialised handling could add the date to the version if required.

In addition, the identifier also provides for a (human friendly) display name and description. These two keys are used for generating documentation.

```
public static E_Identity IDENTITY = new E_IdentityImpl(
    "EDSL_ESEvent",
    "ES Event DSL",
    "1.0.0",
    "2014-Jan-29",
    "A language for standard ES events"
);
```

The example above shows the generation of a DSL identifier for a language called “EDSL_ESEVENT” with version 1.0.0 from January 29th, 2014.

Value Keys[22]

A value key is a combination of a key (as identifier), a value type (type information for the value) and a description. Value keys are used as keys in event content maps. The value type information enables automated mapping and processing of key values. This mechanism allows for cross-programming-language shared semantics once the DSL definition is shared.

The example below shows the definition of a value key “interval” of type integer.

[21] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:71
[22] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:89

```
/** An interval, e.g. for a timer or timeout. */
interval(
 Standard_ValueTypes.Integer,
 "An interval, e.g. for a timer or timeout"
),
```

The DSL framework automatically deals with the following standard value types:

Type	Description
String	- String value type
Integer	- Integer value type
Double	- Double value type
Boolean	- Boolean value type
TAX_Type	- ES taxonomy event type
TAX_Language	- ES taxonomy event language
TAX_Source	- ES taxonomy event source
DSL_Category	- ES DSL category
DSL_Source	- ES DSL source
DSL_Dialect	- ES DSL dialect
DSL_ValueKey	- ES DSL value key
DSL_ValueKeySet	- ES DSL value key set
EventDsl	- ES DSL object
Map	- Standard map
String_List	- List of strings
Object	- Any object

Domain specific value types can be added for each DSL.

Value Key Sets[23]

A value key set is the combination of a key (as identifier), a list of value keys (see above) and a description. In programming languages, it can be compared to the definition a method or function call (they key is the call name and the set of value keys are the parameters). The example below shows a value key set defined as enumerate with value keys “id” and “reason”:

[23] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:128

```
/** A system shutdown request with reason. */
systemShutdown(
 new E_ValueKey[]{ESD_ValueKeys.id, ESD_ValueKeys.reason},
 "system shut down request by a server or client"
),
```

Since value key sets are specific to a domain, no standard sets are defined. However, the ES internal DSLs define their own sets as enumerates, which means they can be easily reused elsewhere.

Dialects[24]

A dialect represents a collection of value key sets. The collection can contain sets from a single DSL or other DSLs. The collection can be the complete set of value key sets or a partial one. The identifier for the dialect is automatically mapped to the event dialect in the event taxonomy. The following example shows the definition of a dialect “ES_EVENTS” as enumerate with display name, collection of value key sets and a description:

```
ES_EVENTS(
  "ES Server and System Dialect",
  new E_ValueKeySet[]{
    ESD_ValueKeySets.systemUpdateConnect,
    ESD_ValueKeySets.systemUpdateDisconnect,
    ESD_ValueKeySets.systemShutdown,
    ESD_ValueKeySets.serverReply,
  },
  "commands and information events for the ES core system"
),
```

The dialect provides the following additional keys for automated processing:

- A flag to indicate if the dialect should only be used in an automated or if it is also allowed to be used on the command line (empty means not automated)
- A default event type to indicate what type an auto-generated event should have if no type is provided to the builder

[24] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:142

- A default value key to indicate what value key should be added by default to an event generated for the dialect (empty means none)
- Mappings from and to other value key sets (and thus dialects) to allow for an automated mapping between dialects. This mapping can for instance define that an incoming alarm can be automatically mapped to a configuration request and how this mapping should happen.

Categories[25]

Categories are introduced into the DSL to auto-generate other views on the language's value key sets. Those views are called categories. They are, for instance, used by the DSL tools to build "categories" of commands. Categories only provide a collection of value key sets (plus display name and description) and no further information.

```
VIZESV(
  "viz-esv",
  new E_ValueKeySet[]{
    EEV_ValueKeySets.createEsv,
    EEV_ValueKeySets.startEsv,
    EEV_ValueKeySets.stopEsv,
    EEV_ValueKeySets.deleteEsv,
  },
  "Commands for an event stream GUI"
),
```

The example above shows the definition of a category "VIZESV" as an enumerate, combining commands for manipulating the event visualizer GUI. Other commands for the GUI can be put in different categories, thus simplifying the user interface.

States[26]

For state-based domains, the DSL should contain the definition of those states. This is important for the DSL defining the DMS strategies, as they are state based. A state in the DSL definition only provides the signature of the state, it cannot be used by itself to define a complete state machine. States only contain a key, a display name and a description.

[25] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:166
[26] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:185

```
INACTIVE( "inactive",
   "strategy is inactive, only possible next state is active"),
ACTIVE( "active",
   "strategy is active, sudo state only seen when things go wrong"),
TERMINATE( "terminate",
   "strategy is terminated and will be removed asap"),
UNKNOWN( "unknown", "unknown state"),
```

The example above shows the life-cycle states of DMS strategies defined as enumerates.

Filters[27]

Each DSL can define a set of standard filters based on the ES filter framework. Standard filters will simplify event processing, for example in a policy or strategy system. The example below shows the creation of a new filter in a DSL as an enumerate using the filter builder in the enumerate constructor:

```
/** Filter for the ES shutdown event. */
SHUTDOWN_EVENT(new ES_EventFilterBuilder()
 .withStandardOptions()
 .withKeyMembers("f-shutdown", "ST filter", "filter for ST events")
 .ofType(ESD_Types.ES_SHUTDOWN)
 .fromSource(ESD_Sources.SYSTEM)
 .inLanguage(EDSL_ESEvent_1_0_0_Identity.IDENTITY)
 .getMapFilter()
),
```

Note: Filters are immutable. This means that the DSL can only provide standard filters. If a filter needs to be altered later, it can be used with the filter builder to create a new, specialised filter.

The actual filter builder is not used in the DSL framework to avoid a tight coupling to that particular implementation.

[27] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:202

Protocols[28]

Protocols are currently an experimental feature of a DSL. The intent is to add a formal definition of protocols to a DSL, which can be useful in network related DSLs.

Programmatic DSL Tools[29]

The framework provides a DSL event builder and transformers. The builder can be used to create an ES event from a DSL. It does all syntax and semantic checks automatically and only creates an event if it conforms to a DSL definition (and ultimately to the ES taxonomy with automated mapping of DSL keys to taxonomy keys).

The framework also provides a set of key and value transformers as a specialisation of the event transformers introduced above. These transformers allow for an automated translation of data from external source, such as databases, into correct value types for any DSL.

Standard DSLs[30]

Two standard DSLs are provided:

a. one for events and commands to control the ES messaging system and
b. one with commands to remotely control the event visualizer GUI.

These two DSLs can be used as examples. They use most of the features the framework defines. They also provide for number of standard event types and value types. Those can be directly reused in other languages.

DSL CLI Tools[31]

The framework also builds a number of command line tools for processing DSL specifications. Those tools allow for automated compilation of a single DSL or a complete DSL set. Current supported targets for the compilation, are JavaScript and [asciidoc] (for table-oriented or plain text documentation). Other targets can easily be added to the compiler.

[28] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:223
[29] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:226
[30] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:231
[31] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:240

2.5.3. Services[32]

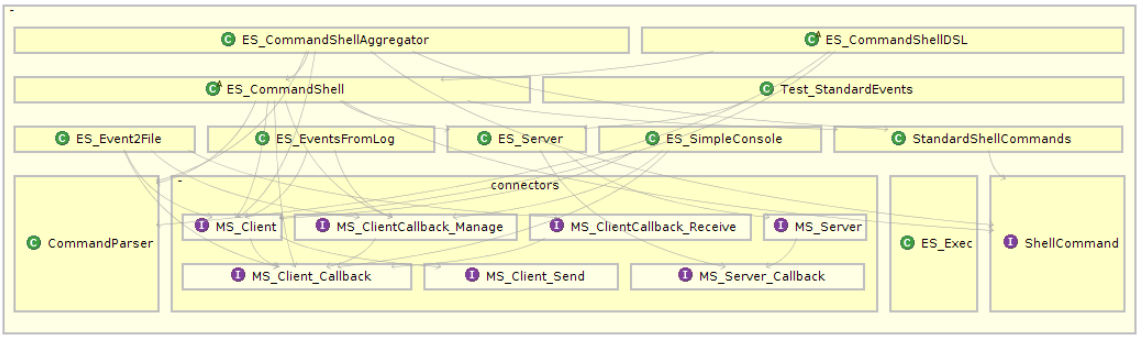

Figure 14. ES Services Architecture View

The service package provides a simple service execution framework, a set of classes to generate CLIs for servers (including a simple console), a standard server implementation (the container component for all applications) and some tools for event processing.

The service execution framework allows for flexible scripting of deployments without further code changes (even moving/renaming classes only requires replicating those name changes in deployment scripts). The CLI classes come with a set of predefined commands, full console handling and also automated DSL processing. The latter feature allows registration of a set of DSLs with a console and let the console take care of the CLI generated from the DSL. No specific CLI, console or server code is required. An aggregator shell automatically collects all generated DSL shells to create an ueber-shell.

Two tools for event processing are implemented: events-to-file and events-from-log. Events-to-file connects to the messaging system, receives (currently all) events and stores them in a file (for archiving or further off-line processing). Events-from-log reads events from a file and sends them into the messaging system.

[32] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:244

Service Connectors[33]

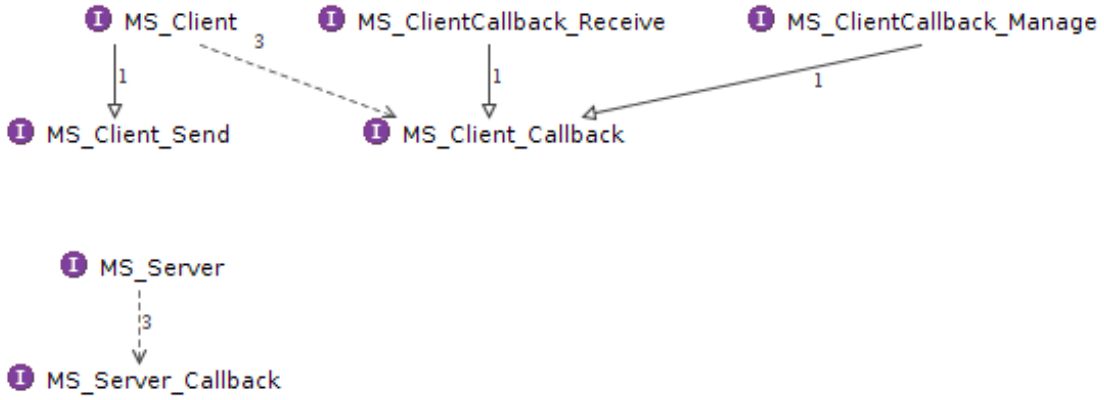

Figure 15. ES Service Connectors Composition View

The service connectors are the interfaces to the underlying messaging system. These interfaces differ between server and client of the messaging system (i.e. not the service package of the ES). Servers and clients use call backs (back to the application). Client call backs can be restricted to event reception only or allow also managing the client (i.e. event driven).

2.5.4. Event-Viz[34]

The EventViz (Event Visualizer) is a fully configurable visualisation tool for ES event streams using HTML5 technology. The core implementation is done in JavaScript using SVG graphics on an HTML5 canvas as the front-end. Connectivity to the rest of the DMS is realised using Websockets, which are now a built-in feature of all modern browsers. To run the EventViz component one only needs a HTML5 capable browser with Websocket support.

Parts of the core implementation are supported by 3PPs. Dependency management between JavaScript modules is managed using Require.js. DOM tree manipulation is realised using jQuery and jQuery-UI. Internal objects are implemented using Prototype.js. Time information is processed using Moment.js. Finally, the actual visualisation is realised using Highchart and Highstock. All 3PPs are available as open source for academic and

[33] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:256
[34] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:265

commercial use, except Highchart and Highstock which need a commercial license for commercial use.

The EventViz can be run in online and offline mode. In online mode, all its features can be controlled remotely once the HTML5 backend is loaded and initialised. Once a visualisation has been started and connected to the DMS event system, it will automatically receive all events and show them according to the loaded configuration. In offline mode one can use a menu structure to manually load different configurations, which in turn provide customised visualisations of stored events. The remote control for online mode is defined by the DSL *EDSL_ESEventViz* using the above described DSL framework. The standard ES shell is providing CLI access to the language and initiates the communication with the EventViz. The language defined functions for:

- Create a visualization, including all its HTML5 elements. Creation can be parameterised with a link to a set of arguments for customised visualizations.
- Delete a visualization, including all its HTML5 elements.
- Start/stop a visualisation, which means that once connected one can remotely start and stop the online visualisation of events.

A configuration for the offline mode allows showing all events in a time-sorted manner (the same as for the online mode). It also allows correlating events or information from events to show specific aspects of the event stream.

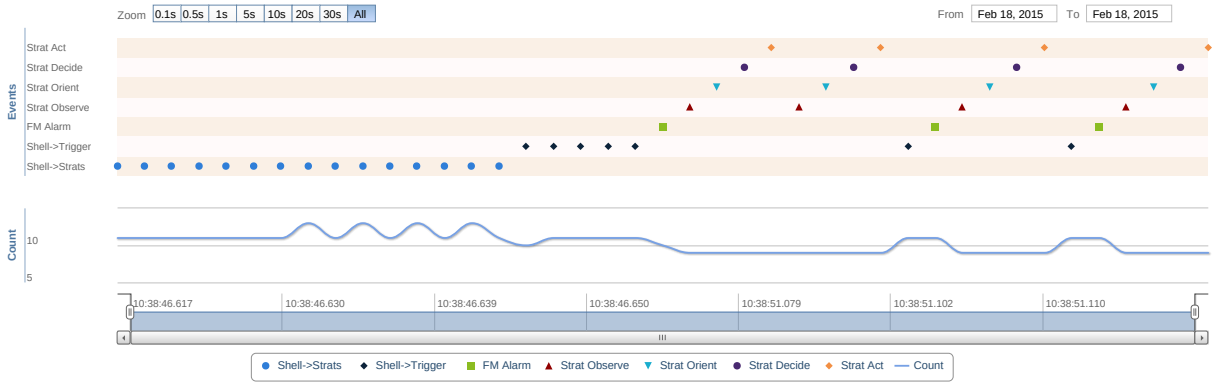

Figure 16. Event Visualiser with Offline Event Stream

Figure 16 shows an offline example of an event stream. On the top, one can see the events time sorted and normalised using same visual distance

between the events regardless of their actual timestamp distance. This normalisation is done to emphasise the correlated visualisation on the bottom. The correlation shown here is the count of key/value pairs in each event of the event stream. A time line and slider on the bottom allows zooming in and out of a section of the event stream. Other correlations can be added, as there is no limit to the number of correlation visualisations.

The EventViz will read a configuration file, which can be used to customise it in virtually any aspect of the visualisation. The events for the event stream can be specified (including the behaviour of the X and Y axis) and event filters can be applied, just to name a few options. Most configurable options are forwarded to Highchart and Highstock applications, their respective documentation can be consulted for all options. Skins and themes are defined separate of a configuration. At the moment, a standard theme is provided. This can be customised for any deployment of the visualizer.

Based on Highchart / Highstock features an event visualisation can be exported as PNG, JPEG, PDF or SVG. This enables to reuse offline and online visualisations in documentation. The current implementation only supports the export as manual feature. Future versions will support that with remote control as well. The visualisation in Figure 16 was manually edited SVG export of an offline mode visualisation.

2.5.5. Generic State Machine[35]

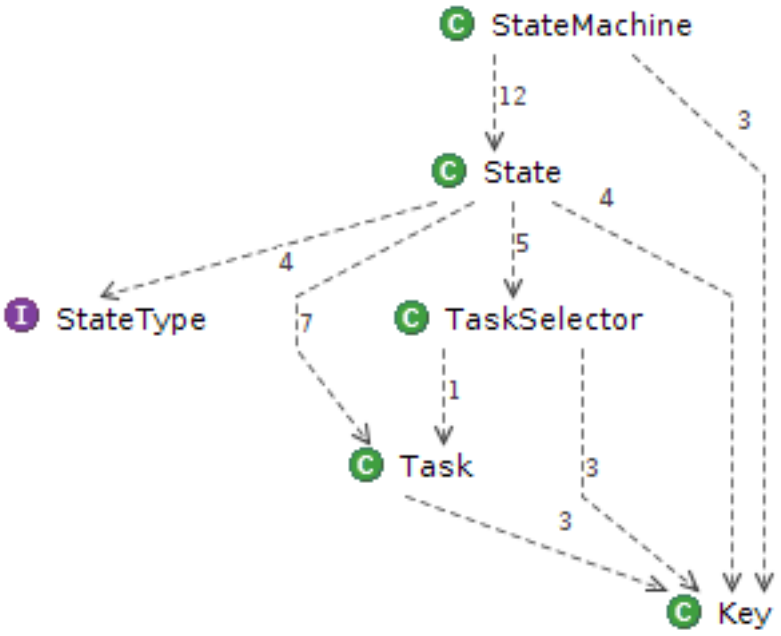

Figure 17. Generic State Machine Classes

The ES provides a generic state machine implementation. This implementation can be used to build any Finite State Machine (FSM). Each state in the state machine can have a number of tasks executing the state logic. The state also has a task selector, which selects a tasks logic at runtime based on context. Furthermore, a state knows its potential next state, which can be either static (if there is only one possible next state) or dynamic (if there are multiple option, selected based on the result of the task logic). Each state has a type, which permits a strongly-typed set of states constituting a state machine.

The actual state machine then only provides a method *execute* which executes the initial state and then every next state until there is no next state provided (meaning that the last executed state was the final state of the state machine).

The input and output of each state (and the overall state machine) are events. Events are strongly- typed so that they can be used as the interfaces of states correlated to the state types. For instance, a state of type A will

[35] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:291

accept an event of type EA and produce an event of type EB which is accepted by a state of type B. The input event of the state machine is called trigger (since it triggers the execution of the state machine). The output event of the state machine is called action since it contains the results of the executed state.

This state machine can be used anywhere. It is used to build DMS strategies and provides the execution environment for them. All that an application using the state machine needs to add is a threading model (if multiple state machines have to be executed in parallel) and a trigger/action mechanism.

2.5.6. Overview of ES Tools[36]

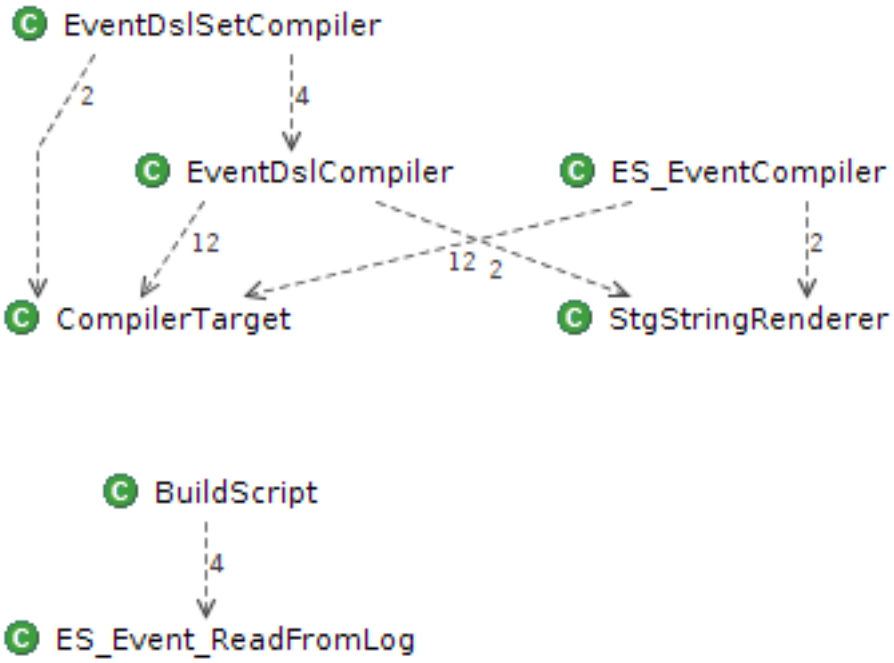

Figure 18. ES Tools

The set of tools provided by the ES cover event handling and DSL processing. The event handling tools are base implementations to read events from logs or archives and to build and send event streams using the original event's headers, including the time stamps.

The DSL processing tools are compilers for DSL sets (e.g. all DSLs in a package or a hierarchy of all DSLs and all of their versions in a

[36] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:304

package) and single event DSLs. The tools also compile the underlying event taxonomy, to provide documentation and for generation of shared semantics for the EventViz. Thus, the compile targets are [asciidoc] with tables (documentation), asciidoc using plain text (documentation) and JavaScript (EventViz or other JavaScript-based applications). Generation of HTML documentation can be done using the generated asciidoc. More targets can easily be created as required in either the ES package or as specialisation of the existing tools.

2.5.7. Backend – Automated Execution and standard CLI[37]

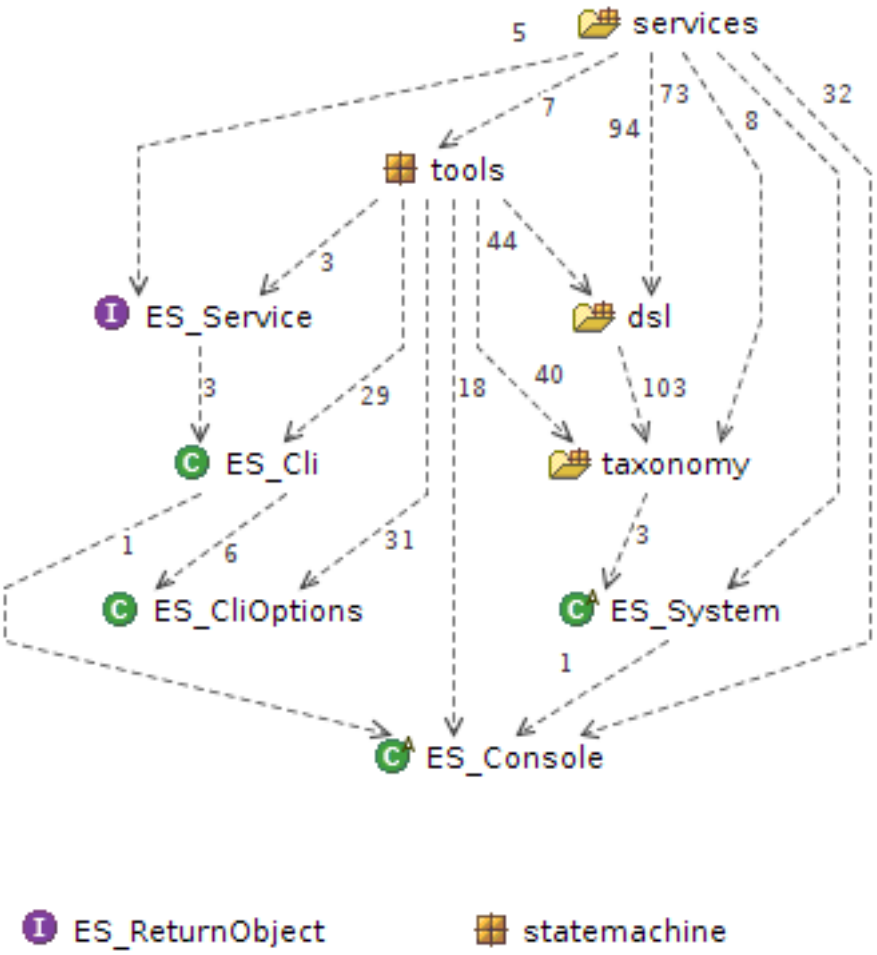

Figure 19. ES Backend with Service Execution and Standard CLI

The ES backend system provides a number of common functions and classes.

[37] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:314

- *ES_Service* provides an interface that ES services can use for easy execution. Combined with *ES_Exec*, this interface allows to find executable ES services at runtime, query their CLI and invoke them by fixed identifier (application synonym) or dynamically by their package and class name. This facilitates DevOps style scripting to run a set of services independent of the internal implementation or execution environment.
- *ES_Cli* and *ES_CliOptions* provide a standard CLI parser and standard options that can be reused by all ES services.
- *ES_System* provides standard functions e.g. for generation of identifiers, handling of system properties and standard console handling.
- *ES_Console* provides access to standard in and standard out using a logging service. This allows to write servers and console applications without direct access to standard consoles, configurable at execution time to either use them and/or other output mechanism.

A generic return object is also provided that allows defining functions that can return a status of execution along with an actual return value. This eliminates the need to return nulls from functions and all ES applications should use *ES_ReturnObject* instead.

2.5.8. Standard Event DSLs[38]

The ES defines a set of standard DSLs using the DSL framework. The standard ES Language is used to, internally, control executed servers and clients of an ES event system. It provides the means for the servers/clients to build synchronous communication (request/reply) over otherwise asynchronous event systems, handle client/server registrations and to initiate a controlled system shutdown. The EventViz language contains all functions to remotely control the Event Visualisation component.

[38] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:332

2.6. Detailed Description – 2-ES-WS[39]

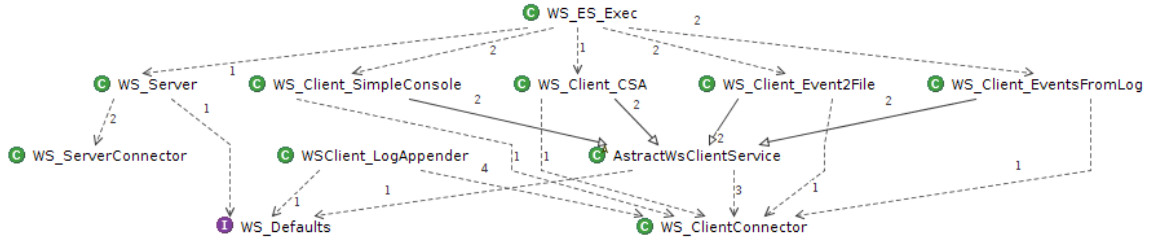

Figure 20. ES Websocket Connectors

The package 2-ES-WS implements Websocket connectors for the DMS-ES. These connectors are essentially implementations of the ES service connector interfaces using Websockets, plus a Websocket specific implementation of the ES tools for event processing. The package comes with a specialisation of the ES execution service to allow for runtime access to all implemented connectors (and standard servers and clients).

The Websocket server will listen on the default port (8887) or a specified port. The clients will use the default host (localhost) and the default port (8887) for connections if not instructed otherwise. The default logging port is 8889.

[39] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:337

2.7. Detailed Description – 2-ES-ZK[40]

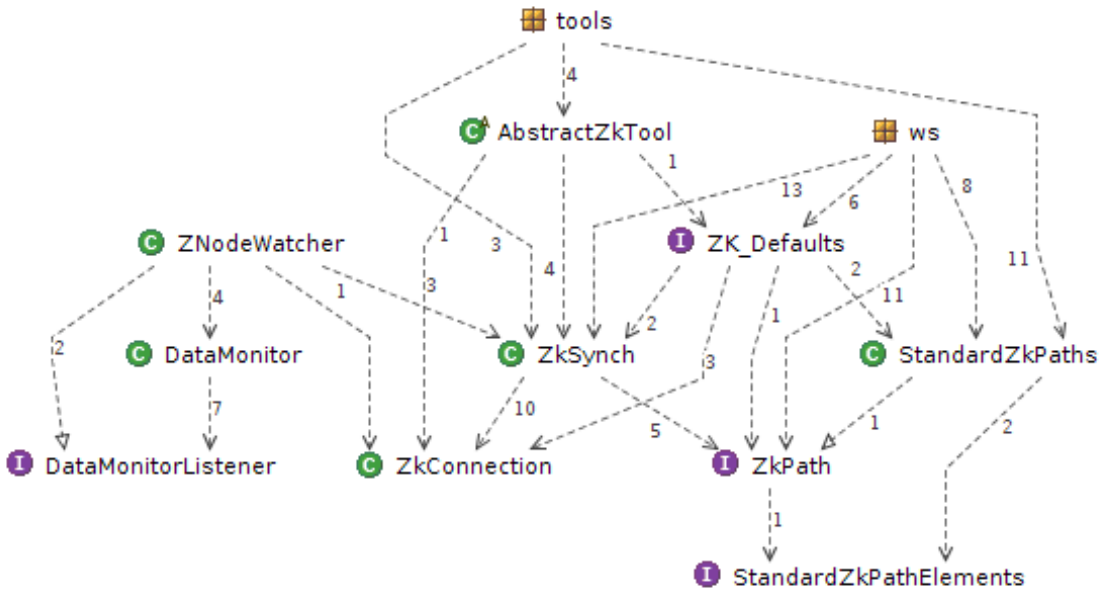

Figure 21. ES Zookeeper Connector

The ES Zookeeper connector (ES-ZK) implements a set of classes that provide access to an Apache Zookeeper cluster for distributed configuration management. The implementation allows for automated creation of permanent and ephemeral ZK nodes. Paths for the nodes can be pre-defined with configuration options and ZK tools can be used to setup the nodes in the cluster.

Furthermore, the classes provide all instrumentation for Service Connectors to store and retrieve information from a Zookeeper cluster. This means that, for instance, a Websocket connector can automatically retrieve server information for the connection and then automatically connect to a Websocket server. A special watcher class implements the functionality required to monitor Zookeeper nodes in case a server terminates abnormally and then re-establish a connection once the server is started again.

- *ZK_Defaults* maintains the default connection parameters for Zookeeper. They are used if no parameters are given programmatically or via CLI.

[40] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:348

- *ZkSync* is a higher level object maintaining information on a Zookeeper node. It is initialised with a *ZkConnection* object for the initial connection to a cluster. It can also be used with a watcher for monitoring of changes in the node, e.g. removal of nodes, changes of node content or appearance of nodes. A message system server for instance can create an ephemeral node with connection parameters and a *ZkSync* object can "watch" this node and change the connection to that server when the parameters change.
- Standard paths and *ZkPaths* realise standard path assembly based on path elements. A path can be configured as static or ephemeral, allowing CLI tools to create initial cluster configurations. The default connection details are maintained by *Zk_Defaults*.
- *AbstractZkTool* provides the skeleton for building tools with Zookeeper connection. The implemented tools are *ZkTool_Init* and *ZkTool_Reset*. Here, "init" means to create all paths on a cluster and "reset" means to remove all paths from the cluster. Paths refers to the standard paths in *StandardZkPaths*, which are constructed from path elements in *StandardZkPathElements*.

Using *ZkDefaults* the default host is localhost (127.0.0.1). The default port is 2181. The default timeout for a connection setup is 5000 ms. The default logger port is 2138.

2.8. Detailed Description – 2-ES-WS-ZK[41]

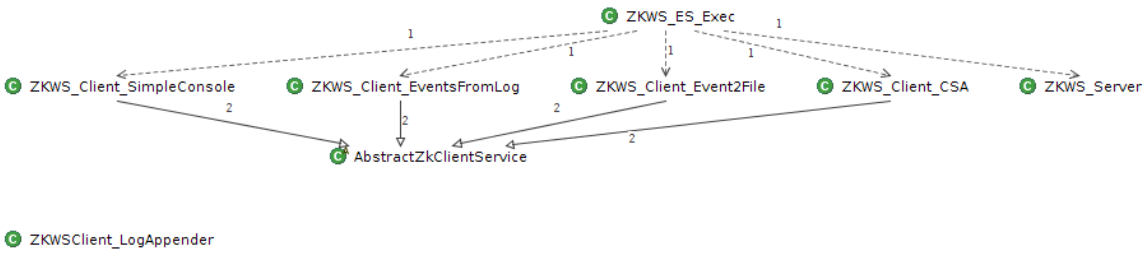

Figure 22. ES Websocket-Zookeeper Connectors

The package 2-ES-WS-ZK combines the Websocket connector with the Zookeeper connector. It extends the original 2-ES-WS tools with automated runtime configuration using Zookeeper. Using this connector,

[41] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:368

the CLI applications (servers and clients) can connect automatically to a Zookeeper cluster (using either default parameters or given CLI parameters) and read all relevant Websocket parameters from there. No further configuration on the applications is required.

This package allows for (almost, if no special ZK parameters are required) zero configuration of a complete DMS deployment.

2.9. Detailed Description – 3-ES-DSLs[42]

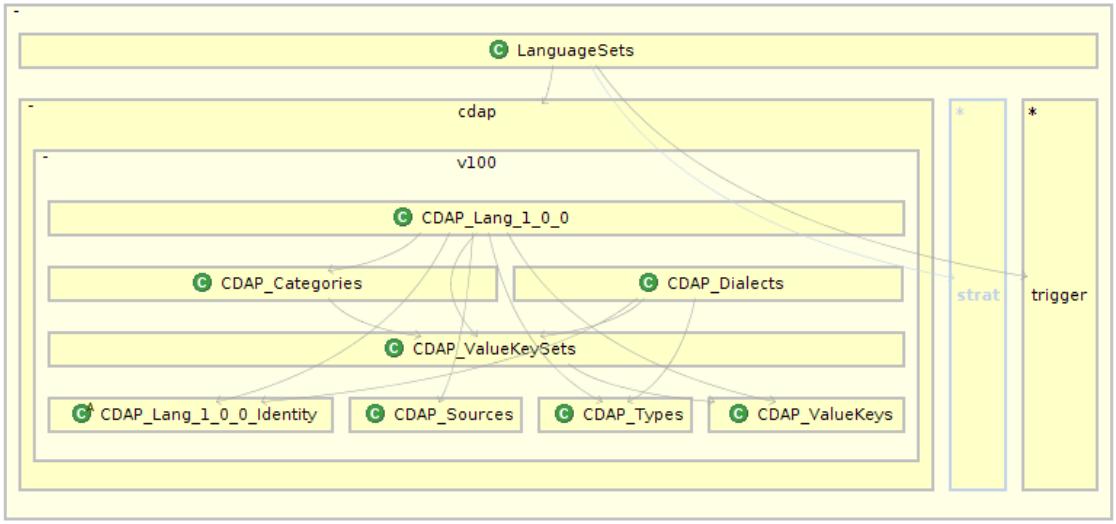

Figure 23. DMS DSLs Architecture View

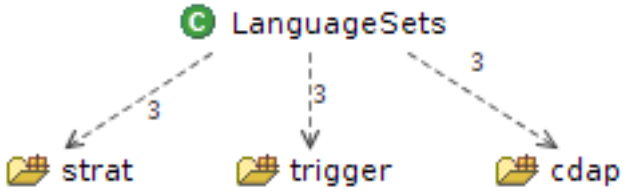

Figure 24. DMS DSLs Implementation View

The current DMS implementation comes with three pre-defined DSLs: cdap, strat and trigger. The “cdap” language *CDAP_Lang* defines a DSL view on the CDAP protocol to build a DMS CACE. The “strat” language

[42] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:378

Strat_Lang defines a set of functions to deploy, load, start, change and terminate strategies on a DMS manager via a remote interface, usually a CLI. The "trigger" language *Trigger_Lang* defines an experimental language to issue triggers into the DMS for demonstration or testing purposes. All DSLs are defined using the ES DSL framework.

2.9.1. CDAP_Lang[43]

The CDAP language represents a 1-to-1 mapping of the CDAP protocol standard into an ES DSL. The value keys are taken from the PRISTINE CDAP implementation (see Section 4.2.3). The value key sets are the six basic operations that CDAP uses (create/delete, start/stop, read/write) in their request and response versions, plus the operation cancel-read. The current CDAP language is built April 20, 2015 as version 1.0.0.

The language only defines one type: ES_CDAP as a CDAP message. Other types might be added in the future for instance to tag notifications differently from other CDAP messages for faster filtering and processing. The DMS Manager and the DMS Agent are the only Event sources within the DMS. Other source can be added if required, for instance ,when legacy software is integrated into a DMS system.

The language uses two different dialects: one for requests and one for responses. The request dialect is non-automatic; it can be used in any application context. The response dialect is automatic; it can only be used as a response to a request but not for instance from a CLI.

The language defines a number of categories to group CDAP operations in the CLI or other interfaces: *OP* (standard operation), *OP_R* (a response), *CONN* (connection handling operations), *CONN_R* (the response of a CONN) and *ADD* (additional operations, currently cancel read request and response).

2.9.2. Strat_Lang[44]

The Strategy language defines all means to remotely deploy, start, stop and alter strategies on a DMS manager. The current language is built January 20, 2015 as version 1.0.0.

[43] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:388
[44] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:398

The language defines two value keys, one to define what triggers a strategy can have and one to define what trigger context a strategy is able to process. All other value keys are taken from the ESD_ValueKeys defined in the ES.

The language defines value key sets for all remote operations of strategies:

- deployStrategy - deploys an existing strategy using class name, identifier and trigger
- activateStrategy - activates a particular strategy using its identifier
- deactivateStrategy - deactivates a particular strategy using its identifier
- terminateStrategy - terminates a particular strategy using its identifier
- listStrategies - lists all deployed strategies
- registerStrategyClass - registers a new strategy class using a class name
- deRegisterStrategyClass - de-registers a strategy class using a class name
- listStrategyClasses - lists all registered strategy classes
- registerTriggerClass - registers a new trigger (filter) class using a class name
- deRegisterTriggerClass - de-registers a trigger(filter) class using a class name
- listTriggerClasses - lists all registered trigger (filter) classes

The language defines a set of states that strategies as a whole can be in:

Table 1. Strategy states

State	Description
INACTIVE	Strategy is inactive, only possible next state is active
ACTIVE	Strategy is active
TERMINATE	Strategy is terminated and will be removed asap
UNKNOWN	Unknown state

The language uses the event sources defined in the following table.

Table 2. Strategy event sources and types

Source (DMS_STRATEGY_)	Event type (ES_STRAT_)	Description
DMS_SHELL	n/a	Events from the DMS shell

Source (DMS_STRATEGY_)	Event type (ES_STRAT_)	Description
OBSERVE	OBSERVE	Events send by a DMS strategy (observe state)
ORIENT	ORIENT	Events send by a DMS strategy (orient state)
DECIDE	DECIDE	Events send by a DMS strategy (decide state)
ACT	ACT	Events send by a DMS strategy (act state)

The language defines two dialects: *STRAT__SM_CMD* for CLI or otherwise issued commands to control strategies (non-automatic dialect) and *STRAT_STATES* for events that are issued by particular active states of a strategy for logging or visualisation purposes (automatic dialect).

The language defines a set of categories for CLI and other user interfaces: *STRATEGY* for strategies, *TRIGGER* for triggers and *STRATEGY_CLASSES* for everything related to strategy classes.

There are no other types defined. However, future use of the language might lead to the definition of new types to allow for a fine-grained filtering of the different operations for strategies once more complex DMS deployments require it.

2.9.3. Trigger_Lang[45]

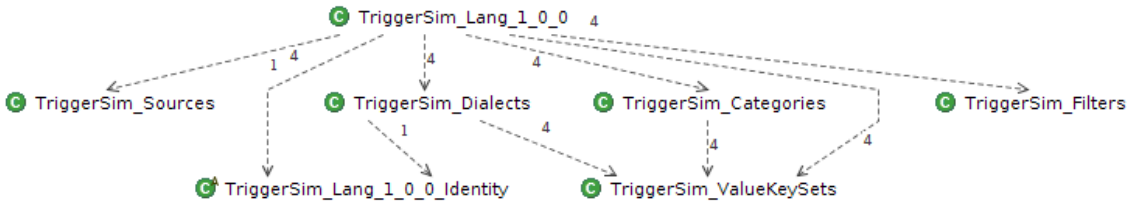

Figure 25. Trigger DSL implementation

The trigger language is an experimental language used to send triggers into the DMS for testing or demonstration purposes. It defines functions for listing, explaining and sending triggers. It also provides a function to register a trigger provider as an experimental trigger source for a DMS. The current language is built February 17, 2015 as version 1.0.0.

[45] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:451

The language comes with one dialect and one category covering all defined functions (value key sets). Additionally, the language defines a standard filter for triggers, which can be used by strategies to test or demo manually send triggers. The only defined source is trigger shell for manually sending triggers into the DMS.

2.10. Detailed Description – 3-Strategies[46]

The DMS strategies are built on top of the generic state machine of the Event System (ES). This allows building virtually any type of management policy. For the current version of the DMS, we have implemented strategies using a four state Observe-Orient-Decide-Act(OODA) loop [boyd96].

For the DMS strategies, we are using these states in the following way.

Observe

Monitors an event stream (including notifications, which are special cases of events). The task of the monitoring is to detect an event (or a set of events) that will trigger the strategy. While we can define a complex set of events representing a trigger, the recommended way to realise the Observe state is to define a single event as trigger and use the content of the event as contextual information about the trigger. This means that we can move the complex event processing outside of the strategy, and then make use of existing Complex Event Processing (CEP) systems to do the heavy lifting.

Orient

Takes the incoming event (trigger) and its payload (context) and shapes the way the strategy can observe, decide and act. It is the most important state in the strategy. Using the trigger and generic heritage (e.g. a business goal for the DMS), cultural traditions (e.g. the way a particular department manages a network or a DIF), new information (external data sources that can be accessed by the strategy to collect additional information), previous experience (historical data on previous executions of the strategy or the wider DMS if possible), and analysis and synthesis (e.g. means to describe the observed situation using all the above data and information). Note: the situation that Orient can describe can (i.e. will) vary from trigger to trigger and thus from

[46] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:461

execution to execution. This means that with an intelligent Orient state, the strategy can become a very powerful tool of the DMS.

Decide

Based on the realised orientation, this state decides what to do about it. This decision (or this set of decisions), following Boyd, describes a hypothesis, i.e. it is not necessarily a final decision as we see in Action Policies (or ECA Policies). Rather it states what should be done and why it should be done to deal with the observation and the orientation.

Act

Is taking the decision and finding the best way to realise it. This "best way" can be dependent on the decision, time, location, or other contextual information. It most certainly depends on the availability of Management Agents and other resources that can be instructed to realise the decision. This also means that this state translates the decision of the strategy into an action for the Management Agent (in RINA a configuration change on a RIB object).

According to Boyd, any taken action results in unfolding interactions with the environment. This means that by changing a configuration, a strategy can create new events that can be picked up by it or other strategies. This mechanism used carefully, can realise chaining of strategies. It also can be used to provide powerful strategies that can understand the impact of their decisions.

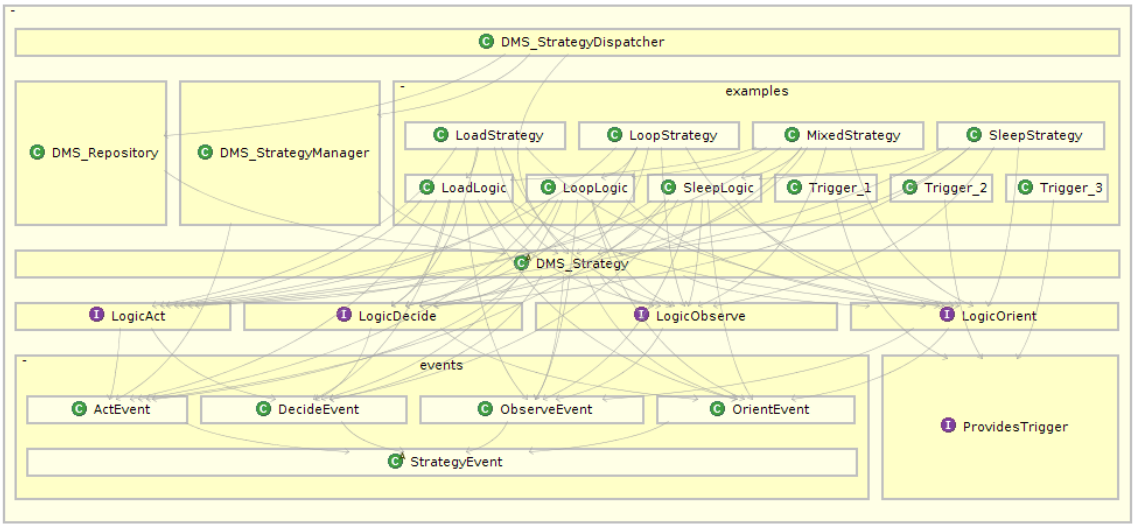

Figure 26. Strategies Software Architecture

The software architecture realising OODA strategies, their triggering and their execution for the DMS uses the same event-driven mechanism as the DMS-ES. All information passed into the strategy as well as between states are modelled as events. Events have a specific type depending on the state sending them (*ObserveEvent* as trigger, *OrientEvent* between orient and decide, *DecideEvent* between decide and act, *ActEvent* as action to the MA).

Each state then implements its logic. Since this logic implementation is specific to where and how the DMS is used, the framework here only provides interfaces that are linked into the implemented OODA state machine (*LogicObserve*, *LogicOrient*, *LogicDecide*, and *LogicAct*). The implementation logic for observe and act states can be generalised for many strategies.

To store strategy information, the framework provides DMS repository (storing all artefacts needed to create a strategy) and a DMS Strategy Manager (which will load, execute, trigger, unload strategies). A dispatcher is provided to realise multiple trigger mechanisms and to allow remote control of the DMS Strategy Manager via CLI or events. This allows for automation of strategy management.

Finally, a few examples for implemented strategy logic are provided. They show how a state can sleep (do nothing for a period of time), loop (go through a simple loop to simulate light activity) and load (go through a complex loop to simulate heavy activity).

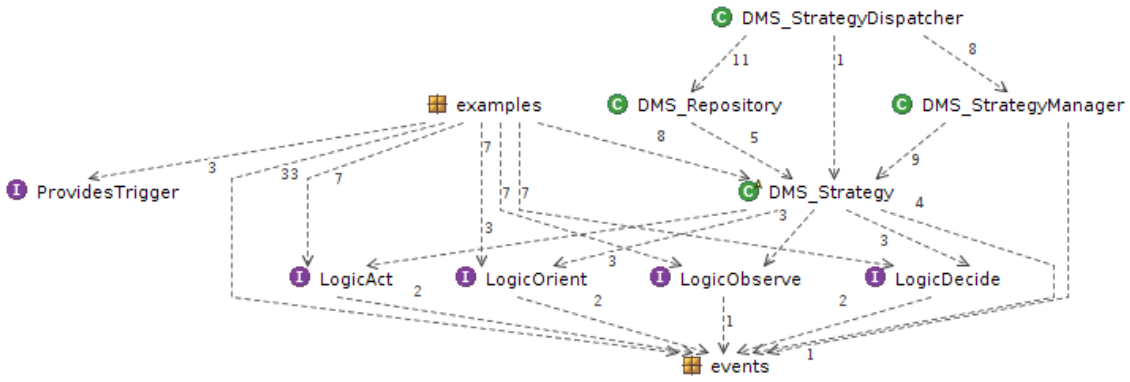

Figure 27. Strategies Dependencies

The dependencies between the classes show how strategies are assembled and how they are executed. The assembly of a strategy associates logic implementations with states. The strategy state machine (fixed) will then

execute the logic when it reaches the associated state. Specialised events can be created and consumed by the logic implementation, the state machine ensures that they are exchanged between the appropriate states.

The DMS repository picks up strategies (complete sets) as well as logic implementations (to allow for runtime assembly of strategies). Once they are deployed in a repository at runtime, the DMS Strategy Manager can be instructed to use them. The dispatcher provides the runtime interface for doing that remotely using the Strategy DSL.

An additional interface, *ProvidesTrigger*, can be used to generate triggers for demonstration and testing purposes. An implementation of this interface will realise a trigger that a strategy is looking for in the observe state. The Trigger DSL contains all required functions and a specific shell can be used to control triggers.

2.11. Detailed Description – 3-RIB[47]

The RIB implementation for the DMS consists of two parts:

a. The RIB-Model package defines the model in an abstract syntax and provides tools to generate code from that model. .The RIB package is the target for the generated code, i.e. it contains all auto-generated Java classes generated from the RIB-Model.

The model currently defined and used for code generation is the PRISTINE RIB model. However, the mechanism described here can be extended to other RIB models, virtually any number of them. For instance, one can define a WiFi RIB model or a USB RIB model and generate the code for them using the RIB tools.

The definition of the RIB model is separated into four different parts, each contained in a separate file. The abstract syntax being used throughout is JSON.

2.11.1. Attribute Types (attribute-types.json)[48]

This is a collection of all types that can be used for attributes in this RIB Model. Types are essentially synonyms to a particular behaviour and

[47] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:503
[48] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:516

allowed values. For instance, a type “Boolean” can have two distinct values 0 or 1 (or in different representations: true or false). A type “list” represents an object that contains a (not further qualified) list of values. Further specialisations of list can indicate a list holding only a given type or lists with particular characteristics for insertion, deletion and sorting order.

```
{"id":"APP_NAMING_INFO",  "description":"Type for an App Naming Info"},
{"id":"AVAILABLEDIF",  "description":"Type for an Available DIF"},
{"id":"BOOLEAN",   "description":"Type for a Boolean"},
```

2.11.2. Attributes (attributes.json)[49]

Attributes consists of an identifier and a type. The identifier must be unique in the RIB model. The type must be one of the attribute types defined in the model.

```
{"id":"address",   "type":"INTEGER",  "description":"###"},
{"id":"addressLength",  "type":"INTEGER",  "description":""},
{"id":"allocateNotifyPolicy", "type":"POLICY_CONFIG", "description":""},
```

2.11.3. Nodes (nodes.json)[50]

Nodes are objects in the RIB model tree that may contain attributes. A node has an identifier and an attribute list. This list can be empty (no attributes) or contain any attribute defined in the model. Each attribute can only be contained once, e.g. a node “DTCP” can contain one (and only one) attribute “flowControl”.

```
{"id":"SecurityManagement",
 "attributes":
["auditingPolicy", "credentialManagementPolicy"], "description":""},
{"id":"Neighbor",
 "attributes":["processName", "processInstance", "address",
  "underlayingDIFs", "underlayingFlows", "authenticationPolicy",
  "numberOfEnrollmentAttempts"], "description":""},
{"id":"Neighbors",
 "attributes":[], "description":""},
```

[49] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:526
[50] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:536

```
{"id":"Enrollment",
 "attributes":
["enrollmentPolicy", "newMemberAccessControlPolicy"], "description":""},
```

2.11.4. Edge Containment (edges-containment.json)[51]

Edges define a containment relationship for nodes. An edge has a source (the container) and a target (the containment). Source and target can be any node defined in the model. An edge can furthermore define a cardinality for the targets, effectively realising lists (or arrays) of targets. A cardinality of -1 means any number. By defining the edges (and the nodes) one can create the tree that the RIB model represents.

```
{"source":"ComputingSystem", "target":"ProcessingSystems", "cardinality":-1},
{"source":"ComputingSystems", "target":"ComputingSystem", "cardinality":-1},
{"source":"Connection", "target":"DTCP", "cardinality":-1},
{"source":"DAF", "target":"ManagementAgents", "cardinality":-1},
{"source":"DAFs", "target":"DAF", "cardinality":-1},
```

2.11.5. Code Generation[52]

The code generator (CodeGen) takes the RIB model definitions described above and creates Java classes implementing the attribute types and the nodes with the given containment. Type class names start with *T_*. Node class names start with *RO_* (RIB object). For the nodes, there attributes are generated as private members with automated default initialisation and getter/setter methods on the node class.

The code generator also generates Java enumerates for each of the four definition files to provide easy, programmatic access to the original definition.

The current target for the code generator is Java (Java classes). However, the code generator is implemented to be easily extend-able for other targets, e.g. [asciidoc] for documentation, JavaScript or other programming language. The code generation is driven by StringTemplates and all information required to activate and use a particular template is maintained in the code generation target description (*CodeGenTarget*). This

[51] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:552
[52] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:566

is the same mechanism as used for the Event DSL code generator of the DMS ES.

2.12. Detailed Description – 4-Manager[53]

The manager provides a container for strategies. It uses the Event DSLs described above for strategy management. It uses the underlying DMS Event system for all communication, effectively providing an CACE for DMS management applications. The implementation of the manager does not employ the RIB model, since that will be in the scope of specific strategies one can implement using the DMS.

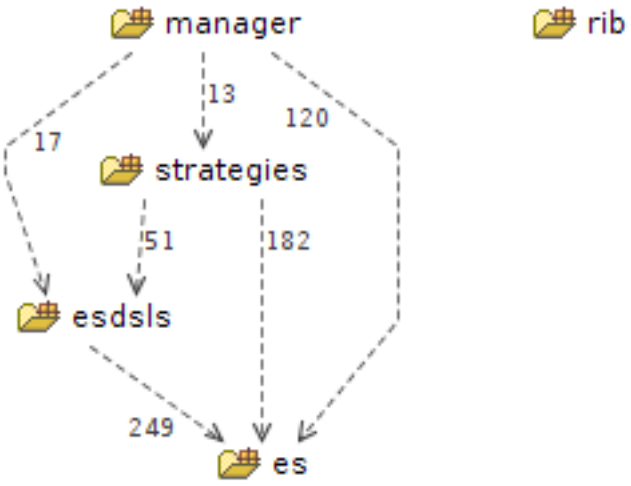

Figure 28. DMS 4-Manager packages and dependencies

The software architecture of the manager shows how three different applications are implemented and then the ES service connectors are used to link them to the underlying event system.

The first application is the DMS Manager. It maintains and executes strategies. The functionality of the manager is defined in the Strategy DSL and implemented by the Strategy Dispatcher. The manager application simply provides a container for a controlled deployment and execution of the strategies. Thus the implementation is very simple.

The second component is the DMS Shell. It takes the Strategy DSL and provides a CLI interface to remotely control a DMS Manager (using

[53] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:574

the event system). The DMS ES already implements a shell that can take an Event DSL and auto-generate a CLI for it. Thus the DMS shell implementation takes the Strategy DSL and uses the ES shell to build the CLI at runtime. It then forwards console input to the auto-generated CLI and prints out return messages to the console.

The third component is a Trigger Shell, which uses the Trigger DSL for sending test and demo triggers to the DMS Manager to test strategy behaviour without an actual managed system being present. It basically implements logic for all commands from the Trigger DSL.

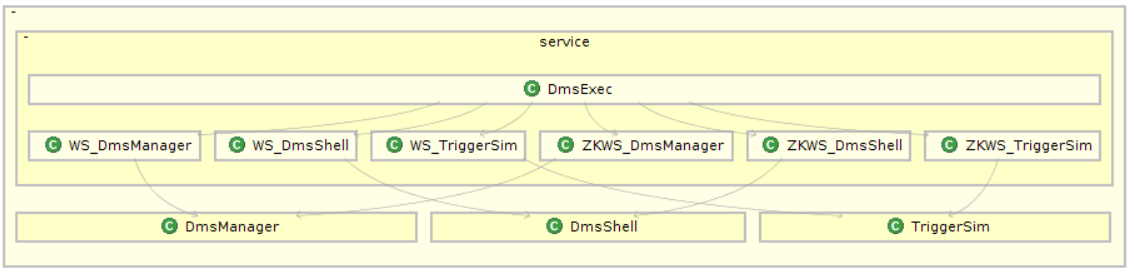

Figure 29. DMS 4-Manager application architecture

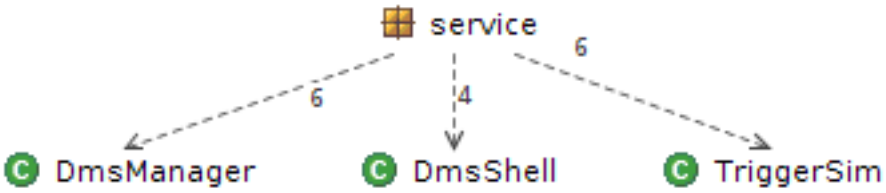

Figure 30. DMS 4-Manager application packages

The three applications are then linked to the service connectors for plain Websockets (*WS_ *) and the Websocket/Zookeeper connector (*ZKWS_ *). The applications can then be deployed in any of the two messaging system. Other connectors can be added once they become available in the DMS ES.

2.13. External Dependencies[54]

All code for the DMS was written from scratch. One of the goals for the implementation was to keep dependencies to external libraries and 3PPs

[54] file:///work/pristine/pristine-rawwiki/wp5/d53/24-manager-components.asciidoc:599

to a minimum. The following dependencies do exist in the code (they are all open source projects):

- SLF4J – used in the DMS ES for logging as well as for console output. The latter one is implemented as a specific case of logging. This means that the behaviour of all console outputs (prints) can be externally configured the same way that logging can be configured. Since SLF4J is only a logging framework, several logging backends can be added at compile time and then configured at runtime. The standard backend used in the DMS ES is logback. SLF4J documentation can be found here: http://www.slf4j.org/
- Jackson – all conversions from and to JSON are using the Jackson libraries. They provide for fast and proven JSON infrastructure. Documentation on Jackson can be found here: https://github.com/FasterXML/jackson
- Stringtemplate – this is a library that allows defining templates for text generation, which can then be included into a compiler or code generator. The template language is simple yet very powerful. By externalising the code generation into templates one can change the target output without changing actual compiler code. All code generators (Event DSL, RIB) use Stringtemplates. Documentation on the library and the template language can be found here: http://www.stringtemplate.org/
- Websocket – this is a communication standard maintained by W3C. The standard is about to be final. There exist many different implementations with various levels of functionality. The implementation used in the DMS ES can be found here: http://java-websocket.org/
- Apache Zookeeper – is used for external, distributed configuration management. The ES-ZK and ES-WS-ZK packages are implementing Zookeeper functionality. To use Zookeeper, one will need to download the software package for a host operating system, install and configure it. Documentation for Zookeeper (API and system) can be found here: https://zookeeper.apache.org
- Apache Commons – the commons packages provide proven solutions for common programing situations. The DMS ES is using the CLI implementation (https://commons.apache.org/proper/commons-

cli) and the common language tools (https://commons.apache.org/proper/commons-lang).

- Asciidoc – is a markup language and compiler for generating documents using simple ASCII text documents[asciidoc]. Asciidoc is used in the DMS ES to generate documentation as well as a target for Event DSL compilers. The generated text files are then processed using a tool to generate HTML files. The DMS ES is using the asciidoc Maven plugin[adoctor15] for automated processing.

The EventViz component uses a number of JavaScript libraries as external dependencies. Those libraries are only required if the EventViz component is used. The external libraries need to be loaded into the Browser, which is done automatically by the deployed manager component.

- Require – is a package that helps to maintain dependencies and to load dependencies. Details can be found here: http://requirejs.org/
- Prototype – is used to define classes and inheritance for JavaScript objects in the EventViz. Documentation can be found here: http://prototypejs.org/
- jQuery – is used for direct DOM manipulations to load graphic DIV elements and to manipulate graphic and text for the visualiser. Documentation can be found here: https://jquery.com/
- jQueryUI – is used for several graphical elements. Documentation can be found here: https://jqueryui.com
- Moment – is used to deal with all time information. Documentation can be found here: http://momentjs.com
- Highcharts – the event stream visualisation uses Highcharts and Highstock as backend. These two libraries provide a large and extensively customisable set of graphs. The EventViz uses a number of those graphs with specific configurations. The external configuration of the EventViz is then mapped (in large parts) directly to the Highchart graphs. Documentation on the API and the graphs can be found here: http://www.highcharts.com/

3. Strategy design[55]

Using the DMS system as discussed above, the main task to manage a RINA network is to write strategy logic, assemble strategies and deploy them on the DMS Manager.

3.1. Writing Strategy Logic[56]

Currently, the DMS supports OODA strategies. Logic needs to be written for all of the four states of an OODA strategy. Each logic implements the logic interface of the state, which basically contains a single method. The four interfaces are:

```
public interface LogicObserve {
 ObserveEvent executeObserve(ES_Event trigger);
}
public interface LogicOrient {
 OrientEvent executeOrient(ObserveEvent fromObserve);
}
public interface LogicDecide {
 DecideEvent executeDecide(OrientEvent fromOrient);
}
public interface LogicAct {
 List<ActEvent> executeAct(DecideEvent fromDecide);
}
```

Each logic function receives a particular event, predefined in the DMS system. One can specialise those events for particular strategies. The predefined event contains a header and content, similar to the events from the DMS ES. The header will be filled automatically by the constructor, and the content has to be filled then by the logic implementation. Below is the definition of the DecideEvent with the constructor creating the header information automatically:

```
public class DecideEvent extends StrategyEvent {

 public DecideEvent(){
  super(Strat_Types.ES_STRAT_DECIDE,
```

[55] file:///work/pristine/pristine-rawwiki/wp5/d53/3-strategy-design.asciidoc:1
[56] file:///work/pristine/pristine-rawwiki/wp5/d53/3-strategy-design.asciidoc:5

```
   Strat_Sources.DMS_STRATEGY_DECIDE,
   Strat_Lang_1_0_0_Identity.IDENTITY,
   Strat_Dialects.STRAT_STATES_100);
  }
 }
```

Once the logic for each state is implemented, one can create a new Strategy class using those implementations. For example, we implement an observe logic that simply takes the incoming trigger event (a standard DMS ES event) and copies all its contents to an outgoing observe event (using the special *ObserveEvent* class). If the trigger contains a key called “trigger” with the contents “myTrigger”. We call the class D53_Observe and could be implemented as follows.

```
public class D53_Observe implements LogicObserve {
 @Override
 public ObserveEvent executeObserve(ES_Event trigger) {
  if(trigger.getString("trigger").equals("myTrigger")){
   ObserveEvent ret = new ObserveEvent();
   ret.getContents().putAll(trigger.getContent());
   return ret;
  }
  return null;
 }
}
```

Now we can implement the logic for the orient state which will receive the created *ObserveEvent*. The example below is kept simple: it takes the trigger from the observe event and copies it into the orient event. Then it adds a key “office” with either true (if triggered between 9am and 5pm) or false (if triggered between 5pm and 9am) value. This means that “myTrigger” has been received within or outside office ours. The orient logic also adds information about a key “system” set to “”XML-CONFIG”.

```
public class D53_Orient implements LogicOrient {
 @Override
 public OrientEvent executeOrient(ObserveEvent fromObserve) {
  OrientEvent ret = new OrientEvent();
  ret.getContents().put("trigger",
 fromObserve.getContents().get("trigger"));

  LocalDateTime timePoint = LocalDateTime.now();
```

```
   int h = timePoint.getHour();
   int m = timePoint.getMinute();
   if((h>9 && h<18) && (m>=00 && m<=59)){
    ret.getContents().put("office", true);
   }
   else{
    ret.getContents().put("office", false);
   }
   ret.getContents().put("system", "XML-CONFIG");
   return new OrientEvent();
  }
 }
```

Now we can implement the logic for the decide state. Again, the actual logic is kept simple and only selects an office configuration if we are in an office situation and a closed configuration if not. If the situation is undecided, the logic does not make any decision.

```
public class D53_Decide implements LogicDecide {
 @Override
 public DecideEvent executeDecide(OrientEvent fromOrient) {
  Boolean office = (Boolean)fromOrient.getContents().get("office");
  if(office==true){
   DecideEvent ret = new DecideEvent();
   ret.getContents().put("config", config.Office());
   ret.getContents().put("system",
 fromOrient.getContents().get("system"));
   return ret;
  }
  else if(office==false){
   DecideEvent ret = new DecideEvent();
   ret.getContents().put("config", config.OutOfOffice());
   ret.getContents().put("system",
 fromOrient.getContents().get("system"));
   return ret;
  }
  else{
   return null;
  }
 }
}
```

Finally, we can write the logic for the act state. We receive a decide event that states what configuration has to be used. Depending on the preferred

configuration system (from orient) and the availability of that preferred configuration system, the specific configuration file is then sent to the configuration systems.

```
public class D53_Act implements LogicAct {
 @Override
 public List<ActEvent> executeAct(DecideEvent fromDecide) {
  List<ActEvent> ret = new ArrayList<>();
  ActEvent retA = null;
  switch((String)fromDecide.getContents().get("system")){
   case "XML-CONFIG":
    if(xmlCfg.isAvailable){
     retA = new ActEvent();
     retA.getContents().put("for", "xmlConfig");
     retA.getContents().put("config",
 (Config)fromDecide.getContents().get("config").toXml());
    }
    break;
   case "JSON-CONFIG":
    if(jsonCfg.isAvailable){
     retA = new ActEvent();
     retA.getContents().put("for", "jsonConfig");
     retA.getContents().put("config",
 (Config)fromDecide.getContents().get("config").toJSON());
    }
    break;
   case "CHEF-CONFIG":
    if(chefCfg.isAvailable){
     retA = new ActEvent();
     retA.getContents().put("for", "jsonConfig");
     retA.getContents().put("config",
 (Config)fromDecide.getContents().get("config").toChef());
    }
    break;
   default:
    break;
  }
  if(retA==null){
   //create error about unavailable configuration system
  }
  ret.add(retA);
  return ret;
 }
}
```

All logic for all states is implemented now.

3.2. Assemble a Strategy[57]

Assembling already implemented logic simply means to create a new DMS Strategy class as follows.

```
public class D53_Strategy extends DMS_Strategy {

 public D53_Strategy(){
  this.setObserve(new D53_Observe());
  this.setOrient(new D53_Orient());
  this.setDecide(new D53_Decide());
  this.setAct(new D53_Act());
 }
}
```

This new strategy class (*D53_Strategy*) can now be compiled and the resulting jar added to a DMS Manager classpath. Once the jar is deployed, the strategy can be executed and added to the DMS Manager.

3.3. Implement a Strategy Trigger[58]

For each strategy, the DMS Manager needs an associate trigger. The trigger can be implemented using *ProvidesFilter* interface. Additionally, the interface *ProvidesTrigger* can be used to create test and demo triggers for the strategy. Below the implementation of a simple trigger for the strategy D53.

```
public class D53_Trigger implements ProvidesFilter, ProvidesTrigger {

 protected ES_EventFilter filter;

 public D53_Trigger() {
  this.filter = new ES_EventFilterBuilder()
   .withStandardOptions()
   .withKeyMembers(D53_Trigger.class.getName(),
D53_Trigger.class.getSimpleName(), "example policiy: simple filter for
D53 events")
```

[57] file:///work/pristine/pristine-rawwiki/wp5/d53/3-strategy-design.asciidoc:154
[58] file:///work/pristine/pristine-rawwiki/wp5/d53/3-strategy-design.asciidoc:173

```
    .ofType(ESD_Types.ES_EVENT_FM)
    .inLanguage("D53_EXAMPLES")
    .inDialect("D53_EXAMPLES Dialect")
    .ofVersion("1.0.0")
    .mustContain(ESD_ValueKeys.info, "D53 Test Trigger")
    .mustNotContainKey(Strat_ValueKeys.context)
    .getMapFilter()
   ;
  }

  @Override
  public ES_EventFilter getFilter(){
   return this.filter;
  }

  @Override
  public ES_Event triggerEvent() {
   Map<Object, Object> t1Map = new HashMap<>();
   t1Map.put(ESD_ValueKeys.info.getName(), "D53 Test Trigger");
   t1Map.put(ESD_ValueKeys.name.getName(), "myTrigger");
   t1Map.put(ESD_ValueKeys.reason.getName(), "test trigger");

   return new ES_EventBuilder()
    .setType(ESD_Types.ES_EVENT_FM)
    .setLanguage("D53_EXAMPLES")
    .setDialect("D53_EXAMPLES Dialect")
    .setVersion("1.0.0")
    .setSource(TriggerSim_Sources.TRIGGER_SHELL)
    .setContent(t1Map)
    .event();
  }
 }
```

The trigger class can now be compiled and the resulting jar added to the DMS Manager classpath.

3.4. Deploy and Activate a Strategy[59]

Using the DMS shell the deployment and activation of the newly created strategy D53_Strategy can be done with the following commands.

```
# sm registerStrategyClass
```

[59] file:///work/pristine/pristine-rawwiki/wp5/d53/3-strategy-design.asciidoc:223

```
 className:eu.ict_pristine.wp5.dms.strategies.examples.D53.D53_Strategy

# sm registerTriggerClass
 className: eu.ict_pristine.wp5.dms.strategies.examples.d53.D53_Trigger
# sm deployStrategy
 className: eu.ict_pristine.wp5.dms.strategies.examples.d53.D53_Strategy,
 id:d53,
 trigger:eu.ict_pristine.wp5.dms.strategies.examples.d53.D53_Trigger

# sm activateStrategy id:d53
```

Once the strategy is activated it will be triggered by the DMS Manager if an incoming event matches the filter of the associated trigger object.

4. Agent design[60]

In D5.2 section 5 [D52], we described the high level architecture of the Management Agent (MA) and we detailed some of its components. We also described the workflow of the MA and some of its requirements.

In this section we are going to explain the updates on the MA architecture, the implementation decisions that have been agreed between the partners and the scenarios that have been set to test the MA functionalities.

4.1. Architecture updates[61]

The development of the Management Agent (MA) has inspired a new vision on the foreseen architecture of the IPC manager. The IPC manager is the daemon in charge of controlling and managing various IPC processes, which means that it has to allow a proper communication within the system (between the IPC processes) and outside the system (between the IPC processes and the user). The main IPC manager responsibilities are:

a. To provide an interface for the management of various IPC Processes.
b. To effectively manage the IPC processes.

In the IRATI stack architecture, the management agent has always been modeled as a daemon (process) with the following requirements:

1. To provide a console to the end-user to control and monitor various IPC processes.
2. To be able to configure a DIF with different policies provided by the user.
3. To control an monitor the state of the managed IPC processes. Allowing creation, registration, assignment and destruction of IPC processes.

These three requirements has been achieved by the IRATI stack with a single software module and daemon called IPC manager. However, the development of a new functionality in PRISTINE, the MA, has arisen the need of an architectural change. Note that the MA requirements have many similarities with the requirements of the IPC manager, since the MA has to be able to provide an interface to the DMS Manager, has to be able to

[60] file:///work/pristine/pristine-rawwiki/wp5/d53/4-agent-design.asciidoc:1
[61] file:///work/pristine/pristine-rawwiki/wp5/d53/42-agent-architecture.asciidoc:1

configure a DIF according to the manager orders and has to control and monitor the state of the IPC processes.

We have realized that there is a logical separation between the functionality that controls and monitors the state of the managed IPC processes, which is composed basically by a method to communicate with the IPC processes, and the other functionalities, which provide a communication method with the end user. This interaction is done either by a configuration file (local configuration), a console (local control and monitoring), or using the MA (remote configuration, control and monitoring).

Following this reasoning, the IPC manager is now responsible only of providing a unique interface for communicating with the IPC processes and controlling the events generated by this interaction. Hence, the north bound interaction (with the user) functionalities (either local or remote) have been separated as add-ons of the IPC manager, which can be loaded (or not) depending on the characteristics of the system.

The following figure shows the new architecture of the IPC manager.

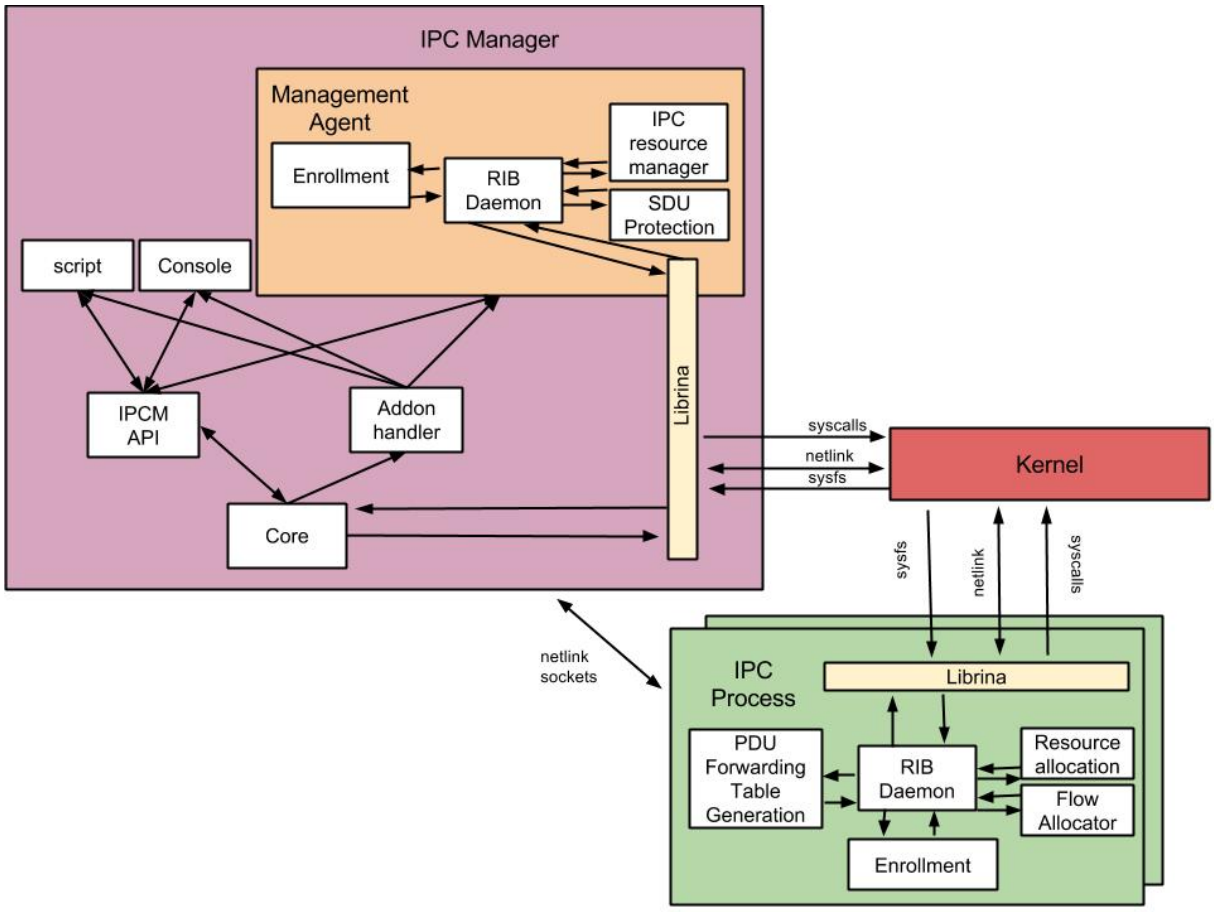

Figure 31. Management Agent architecture

4.2. Low level implementation[62]

Several implementation improvements and decisions have been carried out since D5.2. The most important criterion that have lead to these modifications are (ordered by priority):

1. Achievement of the requirements.
2. Accommodation in the RINA theory and in the developed specifications (D5.2)
3. Re-usability of the code for future functionalities.
4. Scalability.
5. Minimization of changes in the already developed modules of the IRATI stack (to maximise code reuse).

The most important implementation decisions are explained in the following subsections.

4.2.1. MA as an IPCM Addon[63]

There are two choices of direction regarding implementation Management Agent (MA):

- develop it as a separate daemon process,
- develop it as a module (an internal software component) of the IPC manager process.

Both options meet the first 4 criteria. However, there are still some significant differences between them.

If the MA is developed as a separate daemon, a method to exchange information between the IPC manager and the MA will need to be implemented. The preferred method for inter-process communications in the IRATI stack is Netlink[rfc3549], although there are other possibilities.

If the MA is developed as an IPCM module, the MA and the IPC manager share the same address space, so no inter-process communication method will be needed. On the other side, the development of the MA as a

[62] file:///work/pristine/pristine-rawwiki/wp5/d53/44-agent-implementation.asciidoc:1
[63] file:///work/pristine/pristine-rawwiki/wp5/d53/44-agent-implementation.asciidoc:13

module implies some minor changes in the architecture as explained in the previous subsection. It also implies that there is only one MA per node [64].

We have decided to implement the MA as a module of the IPC manager because of the many functionalities are in common between the two modules. To allow a proper initialization of IPCM modules (addons), a command line switch has been added to the IPC manager program to indicate which addons have to be loaded. Since the MA is still at its inception, the corresponding module is not loaded by default on IPC manager startup - it is only loaded if explicitly asked to do so. This default behaviour is likely to change in future WP5 releases, when the whole Management system will improve and gain functionality, superseding the IPC manager configuration file as the primary/preferred RINA node configuration infrastructure.

In addition to the MA addon, two already existing IPCM functionalities have been turned into addons, so that they can be made optional:

- The console component, which provides a CLI interface to the IPC manager functionalities. On some very simple systems, the IPC manager functions will be accessed by the MA only (and hence by the Manager). Consequently the console may not be needed. For the time being, however, the console addon is loaded by default and is there to support local testing/debugging.
- The script component is in charge of realising the configuration specified by the IPC manager configuration file. This addon is loaded by default. The IPC Manager needs certain local configuration to bootstrap itself with the bare minimum to enroll the MA to the management DIF.

4.2.2. Delegation of the RIB[65]

The RIB of the MA is composed by the aggregation of the RIBs of the managed IPC processes, plus the MA's own RIB (see [D52] section 5). To implement this delegation, there were two possibilities:

- The MA maintains and synchronize both its own RIB and the RIBs of the managed IPC processes. In this case the MA can respond immediately on

[64] The current IRATI/PRISTINE prototype only allows a single IPC Manager per node. However, RINA theory allows more than one IPC Manager per node.
[65] file:///work/pristine/pristine-rawwiki/wp5/d53/44-agent-implementation.asciidoc:37

the Manager when the latter accesses the RIB of a managed IPC process. On the other side, a synchronization method is needed between IPC Processes and MA, so that the MA can update the IPC processes' RIBs in order to reflect the updated internal state of the corresponding IPC process. In the IRATI stack the IPCPs are separate OS processes, each with its own virtual address space. This supports the implementation approach of using RIB delegation [66].

- The MA maintains and synchronize its own RIB, but delegates to the managed IPC processes all the RIB accesses destined to their RIBS. In this case the MA acts as a simple forwarder for the request/response RIB transaction. A drawback of this solution is that the MA cannot respond immediately to the Manager request, because it has to wait for the delegated IPC process to respond.
- The MA will support aggregated values that are a function of RIB data in more than one IPCP.

4.2.3. RIB and CDAP updates[67]

For PRISTINE, a new version of librina-RIB and librina-CDAP libraries have been developed. The old IRATI version, lacks a proper versioning mechanism and a scheme that allows multiple RIB versions to co-habitate in the same system.

However, things have improved with the inclusion of the MA, since an MA can communicate to different IPC processes in different DIFs. The RIBs for these different DIFs can evolve and potentially use different RIB versions. In this new enhanced version, multiple schemes (one for each RIB version) can be defined and filled with RIB objects.

We have also taken advantage of the need of a complete restructuring of the RIB library to separate the CDAP implementation from the RIB implementation. The manager, or any other application, can use another persistence technology [68] in its (DAF) RIB implementation if it desires, rather than the one selected for the MA.

[66] An alternative implementation may have a single RIB for (all IPCPs on) the processing system maintained by the MA.
[67] file:///work/pristine/pristine-rawwiki/wp5/d53/44-agent-implementation.asciidoc:51
[68] For example, using a clustered object database, or a persisted event stream.

The decoupled version also exposes less internal details on its interface (cleaner) and therefore it is easier to reuse. The re-factor has also given the possibility to fix some low level architectural problems of these two libraries.

4.2.4. Connecting the MA with the manager[69]

In PRISTINE's scope, a single manager is able to manage many MAs (1-to-N relationship). Two different flows must therefore be allocated between the Manager and **each** agent, one for management commands and one flow for reporting notifications [70]. In practice it must be decided who establishes these flows, e.g. which endpoint is the initiator and starts the flow allocation.

In order to reduce the MA/Manager configuration, it has been decided that the MA initiates the flow allocation (and CACE) procedure towards the Manager. However, the implementation has been designed to allow either endpoint to request a flow to the other one.

4.3. State of the implementation[71]

In order to improve the effectiveness and timeliness of the MA development process, we have partitioned the planned MA functionalities into 3 phases, each one composed of various sprints.

1. Phase 1: Creation of the MA. At the end of this phase the MA will be able to:
   a. allocate a flow an establish an application connection with the Manager.
   b. receive CDAP operations to its RIB.
   c. create an IPC process from a remote RIB operation.
   d. return the RIB state of an IPC process in response to a remote read request.
2. Phase 2: Delegation of the IPC processes sub-RIB. At the end of this phase, the MA will be able to:

[69] file:///work/pristine/pristine-rawwiki/wp5/d53/44-agent-implementation.asciidoc:64
[70] There is a significant difference in the QoS parameters used in these flows (one requires in-order - reliable delivery, the other tolerates out of order, unreliable delivery), justifying the need for both flows.
[71] file:///work/pristine/pristine-rawwiki/wp5/d53/46-agent-state.asciidoc:1

   a. Delegate any CDAP operation targeting an IPC process RIB object (except for creation and deletion) to the involved IPC process.
   b. Implement the functionalities defined in D5.2 required by the Manager.
3. Phase 3: Notification system. At the end of this phase, the MA will be able to to:
   a. Receive any modification or other events taking place in the IPC processes RIBs.
   b. Group and filter these events.
   c. Create notification reports and send to the Manager if needed.

Phase 1 has been successfully completed, and phase 2 is under active development. The next subsection report on the requirements met by the first phase.

4.3.1. Requirements achieved[72]

In [D52], the MA workflow, architecture and component description is exposed. From this description, several requirements have been extracted. The following list briefly explain how those requirements have been met by the current implementation.

Req-JOIN-1
The IPC Manager Daemon, at the behest of MA, needs to create all the IPC Processes required to allow the MA to join the DMS-DAF. Therefore it needs to know a minimal configuration of these IPC Processes.

Addressed by
The script addon of the IPC manager is able to load a configuration file with all the instructions and information needed to create these IPC Processes.

Req-JOIN-2
The MA needs to know the name of the Manager process

Addressed by
A new optional section has been created in the IPC manager configuration file to load the name of the Manager and the name of the DIF used to communicate with it.

[72] file:///work/pristine/pristine-rawwiki/wp5/d53/46-agent-state.asciidoc:23

Req-JOIN-3

The MA needs to be able to establish application connections with Manager AP(s), optionally authenticating (*CACEP component with authentication policy*). For a successful authentication the MA needs to have the proper credentials prior to contacting the Manager.

Addressed by

A CACEP component has been created in the MA which allows the MA and the Manager to establish an application connection. Currently, authentication has not been included. An authentication module is being developed in librina, under the WP2 scope, and will be reused in the MA.

Req-JOIN-4

The MA needs to be able to exchange CDAP messages with the Manager (*CDAP library component*), who will operate remotely on the MA RIB (*RIB and RIB Daemon components*)

Addressed by

CDAP librina library has been refactored so that it is possible to use it as a stand-alone library (decoupled from the RIB library) - by the Manager. On the other side the MA can seamlessly use the CDAP library via the RIB Daemon.

Req-JOIN-5

The MA needs to be able to successfully carry out the actions defined in the DMS-DAF Enrollment specification (*Enrollment Task component*)

Addressed by

The enrollment specification has been implemented in this version of the MA, which allows it to enroll with the Manager.

Req-JOIN-6

The MA may need to be able to encrypt the SDUs it sends through an N-1 flow (*SDU Protection component*)

Addressed by

In this version of the MA, encryption is still not implemented. SDU Protection module is being developed in the scope of WP2 and we plan to reuse its methods for the MA.

Req-PROCREQ-1

The MA needs to be able to request to the IPC Manager Daemon the creation and destruction of IPC Process Daemons, and to get the result of those operations

Addressed by

The Manager is able to send a create CDAP message to the MA targeting the IPCProcess RIB object which results in the creation of a new IPC process.

Req-PROCREQ-2

The MA needs to be able to request operations on the RIB of the IPC Process Daemons in the processing system, and to get the result of those operations

Addressed by

Currently, it is still not possible to request operations on the RIB of the IPC Process - the delegation of the IPC Processes' RIB is under active development. To be able to simulate a read on the IPC process RIB (an operation required in the phase 1), we have created a RIB object under every IPCProcess RIB object in the MA, which responds to a read CDAP operation with the information of the RIB of such IPC process (query RIB operation).

5. Validation[73]

The integration plan for the DMS manager and management agent (as outlined in [D61]) calls for a simple linear integration:

1. Manager (Strategy provision and test)
2. Manager + PRISTINE Strategies
3. Manager + Strategies + Management Agent

To facilitate some parallel testing a (testing) manager in C++ has be created to allow CDAP functionality testing of the agent.

1. Manager(testing) + Management Agent

The next sections outline the requirements on the default policy set, a set of management strategies, the scenario used and a procedure to validate the operation of DMS.

5.1. Default policy set[74]

For the most part the default set of policies is sufficient to support management traffic over the NMS-DAF. However, a least tow types of QoSCubes will need to be supported:

a. CDAP commands, where the flow is bidirectional (between Manager and Management Agent), ie. responses are expected for most commands, therefore the same QoS Cube applies in both directions.
   Key requirements are: reliable retransmission (EFCP policy), with a guaranteed bandwidth (Resource allocation, DTP policies) and disconnect notifications (DTCP policy)

b. CDAP notifications, where the flow is unidirectional (from Management Agent to Manager).
   Key requirements are: reliable retransmission (EFCP policy), with a looser set of bandwidth, latency and jitter parameters (Resource allocation: QoS policies), and ideally whatever-cast routing support (Addressing, NamespaceManagement policies).

[73] file:///work/pristine/pristine-rawwiki/wp5/d53/5-validation.asciidoc:1
[74] file:///work/pristine/pristine-rawwiki/wp5/d53/5-validation.asciidoc:15

The CDAP command QoS cube will also be used for RIB synchronisation, for example, new neighbour announcements.

5.2. Three node validation scenario[75]

The following figure shows the validation scenario used.

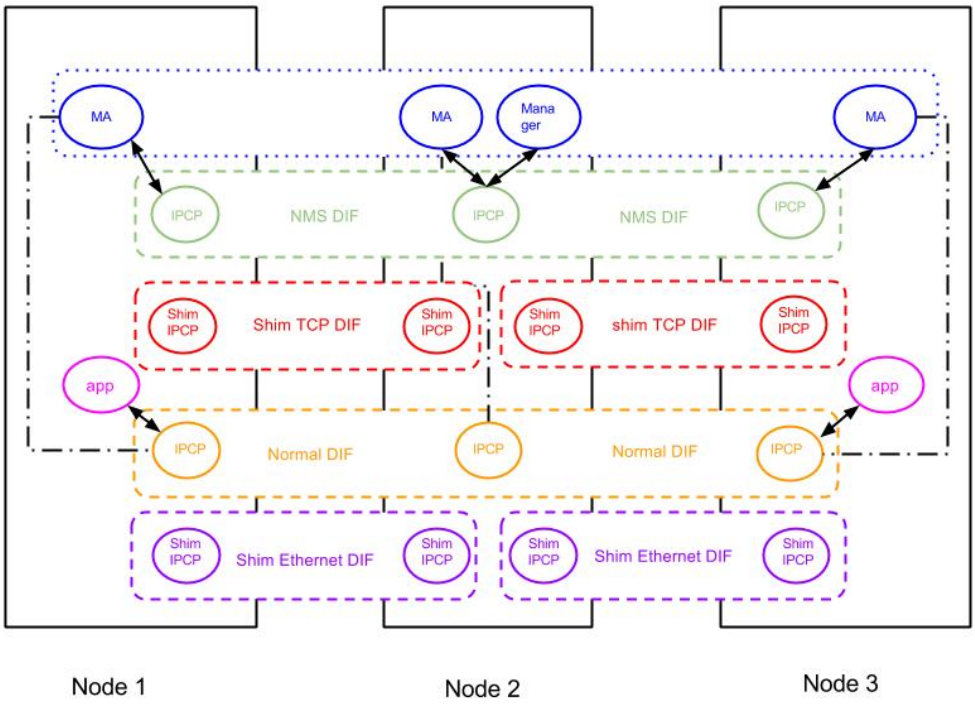

Figure 32. DMS validation scenario

The workflow used for this scenario is as follows:

1. The three nodes bootstrap and load all the needed DIF configurations from a configuration file. Therefore, the shim-TCP IPC process, the shim Ethernet IPC process and the normal NMS IPC process are created. All these operations are carried out by means of the scripting add-on and an appropriate configuration file.
2. The MAs of the three nodes are loaded and started waiting for the Manager registration.

[75] file:///work/pristine/pristine-rawwiki/wp5/d53/52-validation-scenario.asciidoc:1

3. The Manager registers to the NMS DIF, which causes all the MAs to allocate a flow using the NMS DIF and to establish an application connection to the Manager.
4. The Manager sends a CDAP create message targeting a new IPC process object in the RIB of node 2. In this single message, there is enough information for creating the IPC process, assigning it to the normal DIF and registering it to the shim Ethernet DIF.
5. The MA of node 2 receives this message, creates the IPCProcess RIB object, creates the IPC process using the IPC manager, assigns it to the normal DIF, registers it to the shim Ethernet DIF and responds to the Manager.
6. The manager reads the response, and if the creation of the IPC process has been successful, it does the same procedure with the node 1. In this case, the CDAP message contains enough information to create the IPC process and to enroll it to the normal DIF using the already created IPC process in the node 2.
7. If the creation and enrollment of the IPC process in the node 1 has been successful, the same procedure is repeated for the node 3.
8. After the reception of the response message of node 3, the Manager sends a CDAP read message over the RIBDaemon RIB object under the recently created IPC processes.
9. The MAs receive the read request and respond to the manager with the result of the IPC manager query rib operation, which is the information of the objects contained in the RIB of the IPC process.

A limited (mock-up) Manager have been developed only for testing purposes and has been to validate the achievement of the phase 1 agent objectives, as explained in Section 4.3.1.

5.3. Strategies used for validation[76]

The strategies below give an insight into the type of management activities 93-bibliography.xml[77] necessary for DMS operation. The following management strategies are needed, which correspond to generic use-cases for the DMS system.

[76] file:///work/pristine/pristine-rawwiki/wp5/d53/54-validation-strategies.asciidoc:2
[77] 93-bibliography.xml#D52

1. *shimipcp.CreationStrategy*. This strategy is unique as the shim IPCP provides no QoS classes, multiplexing support or EFCP mechanisms. A shim-IPC is a thin IPC layer over the existing VLAN or Ethernet functionalities.
2. *normalipcp.CreationStrategy*. This strategy is used to support two general types of flows, outlined above. In general, these are referred to a "CDAP command flow" and a "CDAP notification flow".
3. *normalipcp.DestructionStrategy*. This strategy is used to destroy normal-IPC Processes. However, from the manager viewpoint the CDAP commands are equivalent for shim-IPC Process destruction.
4. *managementagent.MonitoringStrategy*. This strategy is a delegation strategy where the CDAP actions are issued by the sub-strategies. For example, to monitor a threshold value several sub-strategies may be employed depending on the exact context:
   a. An EventForwardingDiscriminator(EFD) may need to be created
   ⇒ *maEFD.CreationStrategy* (CDAP READ (EFDs), CDAP CREATE (EFD))
   b. Or an existing EventForwardingDiscriminator adjusted
   ⇒ *maEFD.AdjustmentStrategy*, (CDAP READ (EFD old filter), CDAP STOP (EFD notifications), CDAP WRITE (new filter), CDAP START (EFD notifications))
   c. Or removed (as per adjusted unless the filter expression becomes empty)
   ⇒ *maEFD.DestructionStrategy* (CDAP STOP (EFD), CDAP DELETE (EFD))

6. Future plans[78]

In this section we present the next steps to be implemented and tested in the forthcoming work. These will be developed and tested (unit testing) within WP5 during the second phase of the project. Then, WP6 will integrate them in the use case software bundles and extract results from the respective trials.

6.1. Monitoring[79]

[D51] and [D52] describe the overview of the RINA approach with regards the monitoring strategies within the DIF Management System. The different monitoring strategies (reactive, proactive or a hybrid mix) are discussed in [D52].

Focusing on RINA, there are two information transfer domains that have to be addressed:

- Information transfer from the Kernel Space to the User Space within a system. I.e. making available low level kernel information (network stats, CPU usage, etc.) to the processes in the User Space, i.e. the Management Agent or other IPC processes.
- Information transfer from the Management Agent to the central Manager. By means of notifications, the Management Agent communicates to the Manager information with regards the corresponding events the Manager is subscribed to.

6.1.1. Steps toward the RINA monitoring approach[80]

To achieve a complete and efficient monitoring approach for PRISTINE, we will focus on the following steps to produce a valuable monitoring solution to be trialled in WP6.

Kernel Space to Management Agent

1. Determine the system by which the information is exposed from the Kernel Space to the User Space processes. The immediate approach

[78] file:///work/pristine/pristine-rawwiki/wp5/d53/8-conclusion.asciidoc:1
[79] file:///work/pristine/pristine-rawwiki/wp5/d53/8-conclusion.asciidoc:5
[80] file:///work/pristine/pristine-rawwiki/wp5/d53/8-conclusion.asciidoc:16

is to store the data in the corresponding IPCP RIB, but as the RIB is constantly evolving, this presents an issue when integrating the different implementations. To overcome this issue, an intermediate solution can be considered to ease the exposure of the Kernel data to the User Space. A practical approach is to consider the Sys File System (sysfs) provided by the Linux kernel, which is a virtual file system that exports information from kernel subsystems, hardware devices and associated device drivers to the user space. This solution shall be further explored.

2. As a follow-up step of the previous point, an approach to reflect the monitored Kernel information in the corresponding IPCP RIB shall be determined. Two possibilities are present:
   a. to take advantage of the previous solution and populate the RIB with the information from the Sys File System, or
   b. to develop a different solution to write the monitored Kernel information in the RIB (e.g. by means of System Calls).

Management Agent to the DMS Manager

1. Define an efficient and scalable monitoring approach for the notifications between the Management Agent and the Manager. To that end, an event-based monitoring approach is envisaged to be designed and implemented in both the Management Agent and the Manager. The monitoring accuracy is inversely proportional to the involved signaling load and a trade-off must be studied. In general terms, the Manager will configure the Manager Agent monitoring policies in a dynamic fashion to maintain a low network load while minimizing the accuracy loses. This shall also be further studied.

Data processing is also a relevant aspect to be considered. There is a careful balance required here between raw and processed data that needs to be considered. Processing data at the Agent has two negatives:

a. overhead, the primary job of the Network Element is not monitoring; and
b. the processed data may mask a problem from the manager.

Sending data that is more raw, the manager can always find out what is happening (volume overhead here vs putting a new policy in place during a crisis). In contrast, sending computed metrics may mask what is really

going on. Also, an important aspect is timeliness. The MA may record raw data for later uploading, so that situations can be analyzed after the fact. This data will not only be used for monitoring the performance of the network but for diagnosing problems as well.

6.1.2. Policies involved[81]

Apart from the DIF Management System (Manager Agent and Manager), the RINA components and policies that are involved in the monitoring processes are the following:

Policies that are intended to provide monitoring information

- Relaying an Multi-plexing Task
  - Max. queue policy. This policy will indicate when the queues reach the maximum buffering capacity.
  - Monitoring policy. This policy will monitor scheduling aspects of the Relaying an Multi-plexing Task.

Policies that are intended to use monitoring information

- Resource Allocator
  - Forwarding table generator policy. This policy will use the monitored load of the N-1 DIFs to create the PDU[82] forwarding table in the case of the Dynamic QoS-aware multi-path routing.

6.2. Management Agent and RINA stack[83]

Some improvements on the Management Agent and RINA stack side are necessary. These have been previously discussed so a short list is given here.

1. Completion of the ongoing work on the delegation of the IPC processes sub-RIB.
2. Improvements to librina. For example, improving the scalability of the API for RINA applications (and by extension the DMS Manager)
3. Implementation of the notification system (as specified in previous section), which will allow the DMS Manager to configure notifications.

[81] file:///work/pristine/pristine-rawwiki/wp5/d53/8-conclusion.asciidoc:41
[82] 91.glossary.xml#pdu
[83] file:///work/pristine/pristine-rawwiki/wp5/d53/8-conclusion.asciidoc:56

6.3. DMS Manager[84]

The described functionality of the DMS Manager (event DSLs, RIB Model, RIB Model Code Generator, Strategy Framework, DMS Manager) is stable and tested. Future plans focus first on the extension of the described software system to ease the creation and deployment of strategies as well as to allow for multiple RIB models to be defined in the same code base.

1. Add an abstract schema for Strategy assembly and provide an XML as well as a JSON concrete schema. The content of the abstract schema will be identical to the current manual assembly.
2. Add an automated mechanism for the assembly of Strategies to the existing manual (implementation) using the concrete XML and JSON schemas
3. Add an abstract schema for Strategy deployment and provide an XML as well as a JOSN concrete schema. The content of the abstract schema will be identical to the CLI commands currently being used.
4. Add an automated deployment using the concrete XML and JSON schemas in addition to the current CLI deployment method.
5. Add an OODA implementation that fully uses the DMS-ES generic state machine.
6. If required, add an ECA implementation for strategies to demonstrate a second management policy model.
7. Provide a DIF-Viz component that can visualise DIFs based on actual RIB runtime information.
8. Validate the monitoring solution to allow dynamic notification filtering.
9. Continue work on the CDAP connector and align with librina API changes.

[84] file:///work/pristine/pristine-rawwiki/wp5/d53/8-conclusion.asciidoc:65

References

[asciidoc] Stuart Rackham. Text based document generation. Available online[85].

[adoctor15] Asciidoctor, The Asciidoctor Maven plugin, Available online[86].

[boyd96] Boyd, J.R., The Essence of Winning and Losing. Lecture notes, June, 1996. Available online[87].

[D51] PRISTINE Consortium. Deliverable D5.1. Draft specification of common elements of the management framework. June 2014. Available online[88].

[D52] PRISTINE Consortium. Deliverable D5.2. Specification of common elements of the management framework. December 2014. Available online[89].

[D61] PRISTINE Consortium. Deliverable D6.1. First iteration trials plan for System-level integration and validation. April 2015. Available online[90].

[fowler05] Martin Fowler. Event Sourcing. Web blog, December 2005. Available online[91].

[json-s] Internet Engineering Task Force (IETF). JSON Schema: core. January 2013. Available online[92].

[M3010] ITU-T. M.3010 : Principles for a telecommunications management network. February 2000. Available online[93].] Internet Engineering Task Force (IETF). Introduction and Applicability

85 http://www.asciidoc.org
86 http://asciidoctor.org/docs/asciidoctor-maven-plugin/
87 https://fasttransients.files.wordpress.com/2010/03/essence_of_winning_losing.pdf
88 http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d51-common-management-elements_draft.pdf
89 http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d52-draft.pdf
90 http://ict-pristine.eu/
91 http://martinfowler.com/eaaDev/EventSourcing.html
92 http://tools.ietf.org/html/draft-zyp-json-schema-04
93 https://www.itu.int/rec/T-REC-M.3010/en

Statements for Internet Standard Management Framework. IETF RFC 3410, December 2002.

[rfc3549] Internet Engineering Task Force (IETF). Linux Netlink as an IP Services Protocol. IETF RFC 3549, July 2003. Available online[94].

[rfc6455] Internet Engineering Task Force (IETF). The Websocket protocol. IETF RFC 6455, December 2011. Available online[95].

[S101] Structure 101. Available online[96].

[X700] ITU-T. X.700 : Management framework for Open Systems Interconnection (OSI) for CCITT applications. September 1992. Available online[97].

[94] http://tools.ietf.org/html/rfc3549
[95] https://tools.ietf.org/html/rfc6455
[96] http://www.structure101.com
[97] https://www.itu.int/rec/T-REC-X.700/en