ACINO / D3.2

Report on the design of programmability elements for in-operation network control

Dissemination level	Public
Version	1
Due date	31/07/2016
Version date	23/08/2016

This project is funded
by the European Union

Document Information

Authors

Pontus Sköldström (Acreo)

Stéphane Junique (Acreo)

Antonio Marsico (CREATE-NET)

Editor

Pontus Sköldström, Stéphane Junique (Acreo)

Project funding

645127 — ACINO — H2020-ICT-2014

Revision and history chart

Version	Date	Authors	Comment
0.1	01/03/2016	Pontus Sköldström, Stéphane Junique (ACREO)	Milestone MS16
0.2	18/07/2016	Stéphane Junique (ACREO)	Updates Chapter 4
0.3	27/07/2016	Stéphane Junique (ACREO)	Updates to Chapter 1,2,3
0.4	29/07/2016	Pontus Sköldström, Stéphane Junique (ACREO), Antonio Marsico (CREATE-NET)	Updates Chapter 5
0.5	01/08/2016	Stéphane Junique (ACREO)	Major overall of Chapter 6
0.6	02/08/2016	Stéphane Junique (ACREO)	Fixed references and abbreviations; Fixed chapter 1, Summary written
0.7	03/08/2016	Pontus Sköldström, Stéphane Junique (ACREO)	Major update to chapter 7, Conclusion Review by the editors
0.8	05/08/2016	Marco Savi, Federico Pederzolli (CREATE-NET), Mohit Chamania (ADVA)	Internal reviews
0.9	19/08/2016	Stéphane Junique (ACREO)	New version according to internal reviews
1.0	23/08/2016	Domenico Siracusa (CREATE-NET)	Quality check, final version

Table of Contents

Table of figures

List of tables

Summary

This ACINO deliverable D3.2 presents the work performed in task T3.2 “Design of the programmability elements for in-operation network control” to design the northbound interface of the ACINO orchestrator.

The document begins with a review of the requirements of the northbound interface, derived from previous work done related to use cases and application requirements (see ACINO deliverable D2.1) and the expected properties of the ACINO framework (see ACINO deliverable D3.1).

The northbound interface of the orchestrator uses the intent paradigm: applications request *what* they want (in term of service properties), not *how* the requested services should be set up. This paradigm makes the interface potentially independent of the orchestrator, as it does not rely on the technical implementation of the control plane. Adopting an Intent-based interface as a standard for the orchestrator could make Software-Defined Networks much more popular, as applications communicating with them would 1) be easier to write (no knowledge of network technology required) and 2) not be tied to a specific orchestrator implementation. A review of the state of the art on the subject is presented in chapter 3.

The document defines two northbound interfaces: the Dynamic Intent-driven Service Management Interface or DISMI, which allows applications to request network services, and the Multi-Layer Topology and Planning Interface or MLTPI, which is an interface used for management purposes (e.g. interfacing with Network Management Systems (NMS)).

The network primitives exposed to applications through the DISMI are defined. Using the provided grammar, applications can combine them to build intents. The main types of primitives are:

- Actions, which describe the type of connection requested: point-to-point, point-to-multipoint or multipoint-to-multipoint, uni- or bi-directional;
- Nouns, which describe the network end points;
- Constraints, which specify the properties of the requested network service (bandwidth, delay, encryption, ...);
- Selectors, which allow discriminating traffic at the network end points and routing it over application-specific services.

Required properties for the DISMI, such as support for the standard CRUD operations (Create, Remove, Update, Delete), are discussed. Various ways to implement the intent grammar are presented. The DISMI is defined as a RESTful HTTP API, with JavaScript Object Notation (JSON) as data objects, and JSON schema to implement the grammar.

The possibility for an application to negotiate the network service properties after the initial intent request is discussed, and the steps to implement such functionality at the northbound interface are presented.

The architecture of the intent framework is presented. As defined, the DISMI is independent of ONOS, and the framework itself could be implemented as a component separate from ONOS. It is however integrated to the orchestrator to provide a full orchestrator implementation. The framework implements:

- The DISMI module, which performs the intent resolution: all the parameters and primitives of the intent are checked and validated;
- The various intent compilers (one per intent) that decompose intents and find a way to install them as sets of network requirements, then send them to ONOS for actual installation.

The network primitives exposed to applications through the MLTPI are defined. Network Management Systems can use them to pose questions, such as what-if questions (“what if this link fails?”) or re-optimization questions (“what is the benefit of a re-optimization with regard to energy usage?”). The result of such re-optimizations can be applied to the network or discarded.

The MLTPI is an extension of the DISMI: it adds a set of primitives and methods to the DISMI, that are only visible to privileged applications. Some of these primitives are to send questions; others provide the answers from the framework. Similarly to the DISMI, a RESTful HTTP API is defined for the MLTPI.

Finally, summarizing the design work presented in this report, four areas are identified in the network controller used as the base for the ACINO orchestrator, where programmability elements need to be modified or introduced to implement the functionalities described in this document.

1 Introduction

1.1 Goal of the northbound API

The ACINO orchestrator will have a northbound interface used by customer applications to request services. In addition, the orchestrator should provide a management interface for access to operations that go beyond the scope of a single service and require a network scope view. This may be achieved either using a northbound interface similar to that for applications that request services, or by developing an integrated network management system.

The application interface, which we call the *Dynamic Intent-driven Service Management Interface* or *DISMI*, provides a way for applications to easily request connectivity from the ACINO orchestrator without needing any knowledge about the network technology used inside the ACINO domain. This interface also lets an application specify its particular requirements for the requested connectivity, defining an individual, application-specific connectivity. The essence of the application-centric approach is to treat traffic flows differently even if they have the same ingress and egress points. In order to do so, data flows have to be classified and transported using the corresponding application-specific connectivity. The DISMI lets the application specify a "selector" that determines which flows of the traffic arriving at the ingress point should be use a particular application-specific connectivity.

The *Multi-Layer Topology and Planning Interface* (*MLTPI)* is an interface used primarily by a network planner or Network Management System (NMS) to access information and perform operations in the network “scope” such as the network topology, available resources, active services and instantiated paths. After obtaining the relevant information, the MLTPI can be used to pose questions to the Online Planning Tool, such as “what if this link fails?” or “what is the benefit of a re-optimization with regard to energy usage?” Depending on the answers to these questions, a re-optimization of service placement or the underlying network resource allocation may then be triggered.

1.2 Intent-driven interfaces

The *Dynamic Intent-driven Service Management Interface* uses *Intents* to specify connectivity services. *Intents* specify policies rather than the mechanism of how to realize them. This differs from earlier Software-Defined Network (SDN) interfaces where an application typically has to use an Application Programming Interface (API) to transform its connectivity requirements into specific Flowrules and generate OpenFlow commands to realize the paths, or set protocol- or hardware-specific parameters to configure the connection properties (bandwidth, protection, etc.).

Instead of forcing the application to calculate *How* to realize its connectivity, intents let the application specify *What* it requires and let the orchestrator determine how to best realize the request.

De-coupling the *How* and *What* of the service gives the controller/orchestrator more freedom in our design and implementation choices. Ideally, the intent framework can be portable, exposing a standard interface to the applications to define service requirements, while allowing the orchestrator the flexibility to choose

how to implement a request. Rather than letting the application decide, for example, which path is best, the orchestrator can evaluate the service request and handle it in the most appropriate fashion, applying e.g. additional information not know to the client, for instance taking into account other existing services. This separation also has the intrinsic capability to support multiple orchestration platforms.

1.3 Definition of Primitives, Intents and Service Requirements

In ACINO we consider three related concepts when discussing services and their realization: Network Primitives, Intents, and Service Requirements. These concepts, illustrated in Figure 1.1, are described below:

1. **A network primitive** is an object exposed to applications through the DISMI, which combines with other primitives to form intents. An application sends to the DISMI a service request, which is composed of a list of intents:
    a. A **Noun** is a primitive that abstracts in non-technical terms the termination point of a network connection point in non-technical terms. It can be altered by modifiers that specify:
        i. What traffic can pass through (and benefit from the network service);
        ii. What are the expected properties of the termination point;
    b. A **Selector** is a primitive that characterizes the traffic allowed to enter an ingress or leaving an egress;
    c. A **Constraint** is a primitive that adds extra requirements to an Action (see below);
    d. An **Action** expresses the requested connectivity between Nouns. Together with modifiers (selectors, constraints), it specifies *How* and *When* to provide the network service.
2. **An intent** is a syntactically and semantically correct combination of network primitives following a grammar, describing the service requested by the application;
3. **A service requirement** is a specific infrastructure-aware service with requirements that are provided to the Multi-layer optimization module.

To clarify these definitions, a short example, illustrated by Figure 1.1, goes through the communication flow between the application and the framework:

1. An application connects to the ACINO orchestrator to discover the API of the framework. It receives a list of p*rimitives and the grammar to use them;*
2. The application formulates and sends an intent request. For example, it uses the three simple primitives *Noun* "ConnectionPoint(String)", modifier "Constraint(Bandwidth(integer))", and *Action*: "Connect(ConnectionPoint, ConnectionPoint, Optional(Constraint)". These primitives can be

combined into the intent "Connect(Office, Internet, Bandwidth(10gb))", with the meaning of "Connect Office and Internet with a 10 Gb Bandwidth bidirectional connection";

3. The framework analyses the intent and *optionally* initiates a negotiation, allowing the application to choose between several possible solutions. For example, there may be two different solution satisfying the intent, providing different maximum jitter values, but at different costs;
4. The framework decomposes the intent into a list of service requirements. There may be several possible ways to decompose it, each forming a list of service requirements. The first list of service requirements is sent to the multi-layer orchestrator that tries to create the network service;
5. The orchestrator reports on the success (or failure) of the service creation. If the service creation fails, the framework sends the next list of service requirements corresponding to the intent (see point 4), until all possibilities have been exhausted.

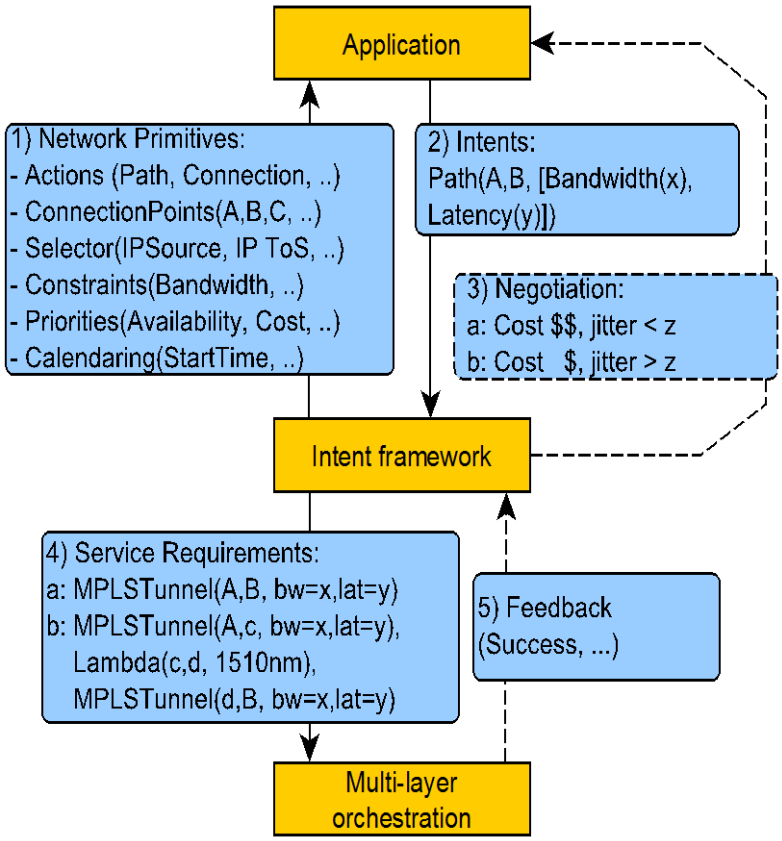

Figure 1.1: Illustration of the relation between Network Primitives, Intents, and Service Requirements.

1.4 Document structure

This report is laid out as follows:

Chapter 2 presents the requirements on the northbound interface. These requirements are derived from the work performed in two tasks: Task T3.1, presented in deliverable D3.1 “The framework for the

application-centric network orchestrator" [ACID31], in which a software environment has been chosen; and task T2.1, presented in deliverable D2.1 "Initial report on network architecture, use cases and application requirements" [ACID21], in which the requirements from specific use cases have been extracted.

Chapter 3 presents a review of the state of the art of intent-based interfaces for software-defined networks.

The network primitives defined for the DISMI are presented in chapter 4: the work that led to the choice of the set of primitives is discussed, and the grammar that allows combining primitives into intents is presented.

The northbound interface of the DISMI is discussed in chapter 5. The NBI defines how an application can communicate with the orchestrator in order to express its intentions. The requirements for such an interface are presented, and a detailed interface design is also provided.

Chapter 6 presents the design of the intent framework. The chapter exposes how a service request from an application, arriving in the form of an intent through the DISMI, is processed until it is installed as a network service by ONOS.

Chapter 7 presents the design of the MLTPI, which is the counterpart of the DISMI for Network Management Systems, for posing planning questions. The required functionalities are detailed, as well as an NBI.

Chapter 8 discusses which programmability element modifications and additional components are needed to support the DISMI and MLTPI.

Finally, chapter 9 concludes the report.

2 Requirements and environment

This section summarizes the requirements and choices made in two tasks:

- Task T3.1 “Definition of the framework for the application-centric network orchestrator” and provided through deliverable D3.1 “The framework for the application-centric network orchestrator” [ACID31];
- Task T2.1 “Network definition and specifications” and provided through deliverable T2.1 “Initial report on network architecture, use cases and application requirements” [ACID21].

2.1 Orchestrator and its internal architecture

ACINO has chosen ONOS, the Open Network Operating System [ONOS] as the base for its orchestrator. ONOS is a modular network controller written in Java. It is built as a multi-module project whose modules are managed as OSGi bundles [OSGI] in Apache Karaf [Apache, KARAF]. The overall functional architecture of the ONOS-based ACINO Orchestrator architecture is described in deliverable D3.1 [ACID31] and is shown in Figure 2.1.

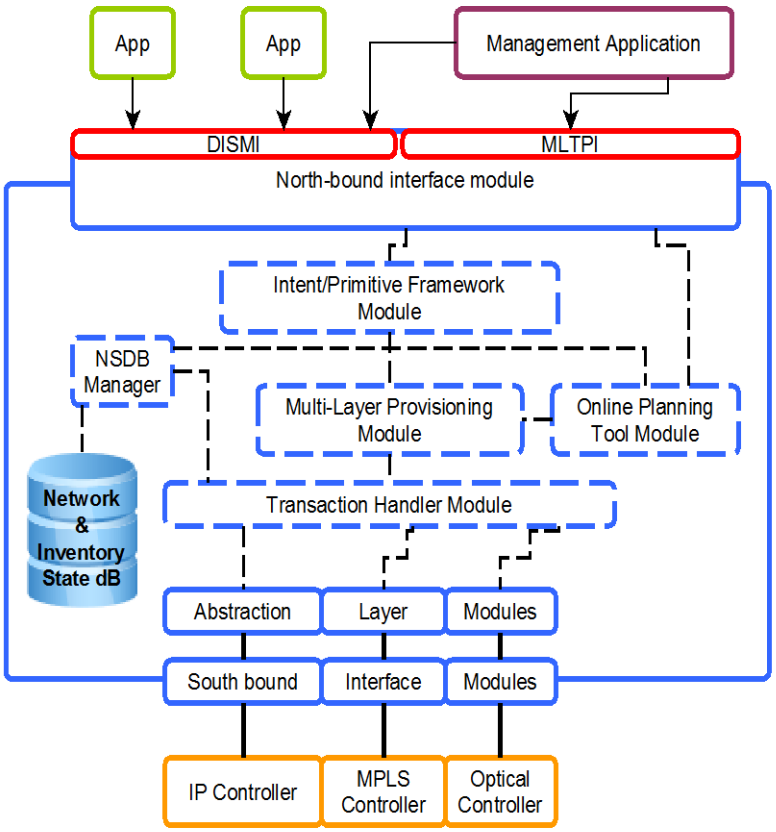

Figure 2.1 : ACINO Orchestrator architecture, as described in D3.1. The two northbound APIs to applications (DISMI) and management (MLTPI) are highlighted.

2.2 Requirements on the Northbound interface

Deliverable D3.1 has also defined the requirements for the northbound interface in its section 3.1. There are two sets of interfaces:

- The high-level **Dynamic Intent-driven Service Management Interface**, or **DISMI**, which allows applications to provision and manage connectivity services using high-level service descriptors;
- **The Multi-Layer Topology Planning Interface**, which exposes topological details (such as information from the optical layer). It will either take the form of an integrated online planning tool, or export enough information to feed an external planning tool.

2.2.1 Dynamic Intent-driven Service Management Interface (DISMI)

The requirements from deliverable D3.1 are shown in Table 1:

Table 1: List of requirements for the high-level dynamic intent-driven service management interface.

Requirement	**Description**
[NB-INT-1]	The ACINO orchestrator shall allow an application to request connectivity for a particular host from a specified network ingress point.
[NB-INT-2]	The ACINO orchestrator shall allow applications to specify any combination of the following constraints: a) Bandwidth (capacity), potentially variable; b) Maximum latency, or latency class; c) Cost (monetary); d) Security (some services may be encrypted); e) Reliability, expressed as survivable downtime: none, low, high, don't care (possibly a numeric threshold); f) Calendaring, expressed as time of activation, deactivation and potentially recurrence. Optionally, in conjunction with a Cost limit, this option could be interpreted as "provide a service when the cost is below a certain threshold".
[NB-INT-3]	An application shall be able to **learn or be told the network ingress (and possibly egress) points of the network(s) managed by the orchestrator.**
[NB-INT-4]	An application shall be able to **poll the status** (e.g. active, scheduled, terminated, refused) of a service it has requested.
[NB-INT-5]	An application shall be able to **change some parameters of active/scheduled services**,

	such as bandwidth. The orchestrator may reject the changes if resources are insufficient to enact them.
[NB-INT-6]	An application shall be able to **terminate or unschedule services** it has requested.

2.2.2 Multi-layer Topology and Planning Interface (MLTPI)

The requirements from D3.1 are that the orchestrator shall allow to (requirements [NB-PLAN-1]):

a. Simulate the effect of failures, i.e., observe the effect of a node or link failure on existing traffic and check which application requirements would no longer be fulfilled;

b. Simulate the effect of changes to the physical (optical) or virtual (IP/MPLS) topology, i.e. check how traffic would be rerouted by installing new fibres or light paths; for the latter, the tool shall also be able to install the new configuration into the network;

c. Compute a re-optimized configuration of the network, and later install it.

Chapter 7 describes the MLTPI as well as where the required functionalities are implemented.

2.2.3 Use-case requirements

The use–case requirements were extracted from deliverable D2.1 (task T2.1). They are intended to show the requirements in terms of connectivity specifications and constraints given by the use-cases.

2.2.3.1 Applications-specific protection strategies

This use-case describes the need for protection against network connection loss. The traditional way to perform restoration is in the IP layer, but it can be much more efficient to perform an optical or – even better – a multi-layer restoration. Indeed, IP restoration is faster but spare capacity is much more cost effective in the optical layer.

With an optical restoration, the selection of the restoration path is insensitive to the needs of the IP layer. A multi-layer restoration takes the needs of the IP layer into account for the aggregate traffic that traverses the failed IP links.

Several application-specific constraints are relevant for restoration. We consider here two who have different implications for the design of the restoration scheme:

- Maximum allowable outage time
- Maximum allowable latency.

Table 2 summarizes how these constraints impact the use of optically-restored capacity.

Table 2 : Constraints related to application-specific protection strategies.

Constraint	**Wait for optical restoration?**	**Re-optimize to use optically restored path?**
Max outage time	No – restore in the IP layer	Yes
Max latency	OK to wait for the optical layer	Do not re-optimize for traffic that meets the max latency requirement.

2.2.3.2 Secure transmission as a service

The frequency of cyber-attacks on critical network infrastructure and public/private entities connected to the Internet has been growing significantly and has translated into significant financial costs for end-customers or businesses.

Server-to-server communication, especially data centre interconnects (DCI), are increasingly becoming a lucrative service offering for network operators. In DCI applications, the end points (data-centres) are trusted entities, and as a result, the responsibility of securing the communication can be moved away from the actual end points to the network infrastructure. This reduces the processing complexity at the servers and allows network operators to propose *secure transmissions as a service*.

This use case has the following characteristics:

- Type of connectivity: point-to-point connection to the Internet;
- Risk: DDOS and other cyber attacks;
- Possible constraints are: Encryption mechanism (IPSec, IEEE MacSec, and custom all-optical encryption), Availability, Latency, and Throughput.

Table 3 shows a detailed list of requirements and capabilities of encryption solutions.

Table 3: Detailed list of constraints and capabilities of solutions for secure transmission as a service.

Requirement	**IPSec**	**MACSec (L2)**	**Layer 1 (ADVA ConnectGuard)**
Latency	high	medium	low
Data Throughput	low	medium	line speed (no overhead)
Protocol Transparency	low	medium	high
Flexible Encrypted Payload Size	restricted	restricted (standard MAC size)	1G – 100G
End–to–End	IP only	layer 2 only	Fiber/OTN, SONET/SDH

Compatibility			
Flexibility (Meshed)	high	low	medium

2.2.3.3 Cloud applications

Cloud applications cover several use cases:

- **Inter Data-Centre (DC) traffic**: This type of traffic can have bandwidth or latency requirements, depending on the type of traffic. One growing requirement is encryption, for protection against data snooping.
- **VM migration**: This is a specific type of inter-DC traffic that typically requires High bandwidth and Low latency, in particular to finish the synchronization during live migration
- **Database (DB) synchronization**: There are typically three approaches to DB synchronization, either synchronously or asynchronously. An asynchronous synchronization requires mostly high bandwidth, whereas a synchronous synchronization requires mostly low latency. The three approaches are:
    - **Approach 1 – Active stand-by**: a single primary DC is used to serve users, but is synchronized to one or more standby DCs. This is done as a background task, either asynchronously, if inter-DC latency is high, or even synchronously, if latency is low enough.
    - **Approach 2 – Single DB instance over multiple DCs**: used when a single DC has insufficient infrastructures, and multiple DC have low latency connections, then they can be used as a single logical DC, provided there is also a high-bandwidth, possibly dedicated, WAN link between them. Data is stored in just one DC at a time (no redundancy), so no particular synchronization is needed.
    - **Approach 3 – Multiple DB instances over multiple DCs**: multiple DBs are independently run in multiple, potentially far DCs. Avoiding data collisions requires support from the application layer. Full bi-directional synchronization of DBs is needed.

The requirements for data centre applications are summarized in Table 4.

Table 4: Summary of the requirements for various data centre applications.

Requirements/ Application	**VM Migration**	**DB Synchronization and VM replication**
Capacity	Tens of GB to TB, PB (e.g. 128 GB for the memory, 1 TB for the	Unknown (Values in the order of GBs could

	disk)	be assumed)
Duration	Minutes to hours (depends on capacity, bandwidth, dirtying rate) Reasonable values: - Cloudburst (minutes) - Enterprise IT consolidation (hours to days) - Follow the sun (minutes to hours) - Disaster Recovery/Preparedness (hours)	Typically continuous, or scheduled (e.g. during the night)
Bandwidth	Multiple 1/10 Gbps VM migrations are reported as examples for the hybrid cloud scenario. 100 Gbps VM migrations are foreseeable	1 Gbps can be assumed
Distance	Regional to inter-continental	Regional to national network (synchronous copy, limited by propagation delay, i.e., latency) to continental (asynchronous copy, eventual consistency)
Latency	Current solutions work around it Latency is critical since it affects convergence. 10ms is specified as max for vMotion (intra-dc); could be higher for WAN migration	Low latency required for synchronous configurations Stricter than VM migration one. Reasonable value: in the order of ms, latency affect DB performance with synchronous replication
Protection / Restoration	Protection or IP restoration desirable	Synchronous configuration benefit from Protection or IP restoration, asynchronous ones could live with optical restoration and slow IP reroutes
Encryption	Desirable	Desirable

2.2.3.4 Applications using a 5G backhaul network

5G networks are a very active research area. It is not clear what the requirements of such networks will be, but they are expected to be deployed in a few years, and there are many different visions on what they might be and what they should be able to provide.

5G networks are expected to provide massive network capacity with a high focus on data rates and the number of connected user terminals. Mobile traffic demand is predicted to increase dramatically, and very high bandwidth, very low latency and high reliability are prime goals. Deliverable D2.1 [ACID21] investigated these high-level goals and concluded that an application centric IP/optical network architecture could aid the future 5G Radio Access Network to provide services that previously have been difficult to realize. Some of the requirements are presented in Table 5 (combining information from D2.1 and from [METISD11].

Table 5: Summary of requirements for various use cases and applications using a 5G backhaul network.

Requirement / Use case	**Stadium**	**Festival**	**Shopping mall**	**Office**
Dominating application	Video/photo sharing (upload) + video streaming (download)	Similar to stadium + “Internet access”	Augmented reality	Cloud storage
Capacity	DL: 750 Gbps UL: 1500 Gbps	DL+UL: 900 Gbps	DL: 34 Gbps UL: 13 Gbps (assuming 0.2 km^2)	DL: 15 Gbps UL: 2 Gbps
Time scales	2-3 events per week 2-3 hours per event	1-2 events per year lasting for 1-4 days	Opening hours, e.g. 10-18 every day	Office hours
Distance	Regional to international	Regional to international	Regional to international	Regional to international
Latency	10ms e2e	10ms e2e	10ms e2e	10ms e2e
Protection/ Restoration	Desirable	Desirable	Desirable	Maybe required?
Encryption	Maybe	Maybe	Maybe	Maybe

3 State of the Art

The intent paradigm is an attractive choice for the design of a NBI as it makes the interface portable, easy to use, and agnostic to the underlying network and orchestration technology. A review of the current frameworks is presented in this section. Our conclusions from the state of the art is that while there is no formal definition of exactly what constitutes an intent interface, there are common features of them all, namely to decouple *what* the users want from *how* to achieve it.

3.1 Publications

3.1.1 DOVE: Distributed Overlay Virtual Ethernet

DOVE is a Distributed Overlay Virtual Ethernet network developed by IBM [DOVE1, DOVE2], providing a high-level network abstraction. The abstraction uses an intent-based model. The publication cited above discusses what a good network abstraction is:

- A possible abstraction is to describe the network functionality in terms of connectivity between endpoints and the policies associated with the connectivity. This approach has a major drawback: the complexity explosion required to manage a pair-wise mesh of endpoints. In addition, it fails to decouple the network definition from the endpoints' instantiation. Network abstractions with this property cannot separate the relatively static application lifecycle from dynamic aspects such as the addition, removal and migration of components.
- A better abstraction is to keep the static and dynamic aspects of the network separate, to allow networks to grow and shrink. This can be done by defining policy domains: entities that aggregate endpoints with common policy criteria. This model, with virtual networks, closely reminds of Software-Defined infrastructures such as OpenStack.

The source code for DOVE (now OpenDOVE) has been contributed to the SDN controller OpenDaylight [DOVEAPI]. The API and data structures proposed by DOVE are interesting and should be compared to that of other projects that we use as references. DOVE provides an intent-based API based on the Representational State Transfer (REST) architecture to provision and release resources [DOVEREST].

3.1.2 InSeRt: An intent-based Service Request API for Service Exposure in Next Generation Networks

This publication [INSERT] describes a policy-based service broker for a network-based telecommunications service. The focus of the work is clearly voice telecommunications: GSM, and PSTN. The goal of using intent-based requests to set up services is to allow a description of services in business terms: intentions and strategies to achieve them.

The paper claims that the proposed intent-based service broker has been implemented in openSOAplayground [OPENSOA]. However, the project does not seem to be available anymore.

3.1.3 ONF Webinar: What's Going On With Intent-Driven Networking & Northbound Interfaces?

The webinar [ONFINT] from the Open Networking Foundation (ONF) presents the concept of intents and the work done in various Open Source Software (OSS) projects. Although it tries to cover all projects in the field (and several of them are mentioned), it very much focuses on open source project Aspen, in which one of the presenters is involved.

The main message from the webinar is that intents are a model of communication requirements, with the following characteristics:

- Intents are about What, not How;
- Intents want to be a universal language;
- Intents are dynamic but invariant in expression: they don't depend on the particular infrastructure;
- Intents are portable: they are independent of protocols, vendors, infrastructure providers, etc.;
- Intents are compose-able: disparate services, developed independently, can express their resource requirements in a common language;
- Intents provide context;
- Intents have a small attack surface;
- And finally: "If your boss cannot understand it, it's not intent".

A list of projects currently working with intents (defining models and building reference implementations) is provided, some of which are of particular interest:

- ONF's open source SDN projects Boulder [BOULDER] and Aspen [ASPEN]: Boulder investigates the server-side (the SDN or orchestrator) while the goal of Aspen is to build client applications using intents based on Boulder;
- OpenDayLight Network Intent Composition (NIC) project;
- ONOS intents.

3.2 Boulder cross-controller Intent-NBI

Boulder [BOULDER] is an OSS project that aims at developing an intent-based northbound interface for SDN controllers. The project focuses mainly on defining what an intent-based interface should be: semantics and information models necessary to tell a network what to do. It discusses what the key properties of such an interface should be, to make it valuable.

In parallel with the work on what an intent-interface should be, the Boulder project has the objective to propose an implementation. The current status of the source code is lacking, though: the last update in the git repository [BOULDERGIT] is from September 2015 and the code cannot be compiled following the documentation.

Much of the work going on in Boulder is in what an intent-based interface for SDN controllers should be. This work is formalised by the release of documents describing the intent NBI paradigm [BOULDERPRES, BOULDERDASH]. The key properties of an intent-based interface are defined as:

- **Non prescriptive**: It describes the request purely as needs and constraints, and contains no mention of how to meet them. In other terms, they are about *What*, not *How*;
- **Portable**: The interface is purely intent-based, which makes it portable across SDN controllers: they are independent of protocols, vendors, infrastructure providers, etc.;
- **Composable**: Separate services/constraints developed independently can be mixed in an intent interface to let client applications create an intent;
- **Context**: The intent interface provides the context necessary for the proposed services to be understood by the client application. In other terms, an intent interface allows the auto-discovery of the available services;
- **Universal**: It is universal enough that any application request can be expressed in terms of intents.

3.3 The Aspen open source project

The OSS project Aspen [ASPEN] is an implementation of an intent-based interface for SDN controllers targeting a specific goal: allowing client applications to inform the controller about their Quality of Service (QoS) requirements for real-time media traffic. The project implements a use-case of the ONF real-time media NBI REST specification [ONFRTM].

The main focus of Aspen is in the implementation (since it uses the specification and use case from another project as its base). However, like with Boulder, the last update in the git repository [ASPENGIT] is from September 2015 and the code cannot be compiled following the documentation.

3.4 OpenDayLight Network Intent Composition (NIC) project

The OpenDayLight (ODL) [ODL] Network Intent Composition (NIC) [ODLNIC] project is a working group to add an intent interface to the OSS in the ODL project. It is actively working with the design, and some experimental code has been released.

3.5 ONOS Intent-based interface

The Open Network Operating System (ONOS) [ONOS] is a commercially used, open source SDN controller. It is implemented in Java as a multi-module project. The modules are managed as OSGi [OSGi] bundles inside an Apache Karaf [KARAF] framework. This architecture allows an automatic dependency control and an isolation of the services implemented in each module.

One of ONOS subsystems is an intent framework [ONOSINT], which allows applications to request network services in terms of policies (the *What* described in section 3.2) instead of mechanisms (the *How*). These intents, sent by client applications, are transformed into “installable intents”, which are operations that can be performed on the network environment.

The ONOS intent framework provides an intent interface that complies with the criteria proposed by the Boulder project. In particular, the interface is independent from the underlying network implementation, and it is extensible.

The ACINO orchestrator is developed on top of the ONOS framework, therefore, it inherits its features. Chapters 6 and 8 will explain what are the main limitations of the intent framework developed within ONOS with respect to the requirements of the ACINO project and how we plan to evolve the programmability of the former to realize our objectives.

4 Network primitives for DISMI

As stated earlier, network primitives are individual parts of the intent language that combine to form a complete sentence. The available network primitives and their syntax are exposed to the Application when it connects and authenticates with the orchestrator, with the possibility of exposing a different set of primitives depending on the (business) relationship between the application and the orchestrator. As with words in natural languages the primitives can be assigned into different classes and expressed in different groups.

4.1 Nouns – Connection points and End points

A *Noun* is a primitive characterizing the network ingress and egress points. A *Noun* contains a *ConnectionPoint* primitive, as well as some optional modifiers.

A *ConnectionPoint* makes the junction between the *Intent* representation of a network edge point, and its network representation: It contains a string describing the edge point to the application (e.g. “Office-Berlin”, “Customer-A”).

The orchestrator associates each *ConnectionPoint* with a list of network *EndPoint* primitives. An *EndPoint* is a network point relevant to the ACINO orchestrator, described in technical terms: an IP subnet on a router port, a port on a switch, or a lambda or fibre on an optical device.

The mapping between a *ConnectionPoint* and its *EndPoint* list has to be done *a priori*, for example as part of a contract negotiation between the ACINO provider and the Application owner. The primitive *EndPoint* is not exposed to applications, except through a specific DISMI interface to allow updating the mapping between the name of a *ConnectionPoint* and its list of *EndPoint* during runtime, as well as to create or delete *ConnectionPoint* objects. In addition to a *ConnectionPoint*, a *Noun* can optionally contain modifiers: a list of *Selector* and a list of *Constraint primitives.* Table 6 and Table 7 show the details of the *Noun* and *ConnectionPoint* primitives, respectively. When a line names a primitive, the column “required” states whether it is mandatory for the DISMI to implement it. When a line names a primitive parameter, the column “required” states whether it is mandatory to provide the parameter when using/creating the primitive.

Table 6 : Noun primitive.

Primitive	**Parameters**	**Required**
Noun		Yes
	ConnectionPoint	Yes
	List(Selector)	No
	List(Constraint)	No

Table 7: ConnectionPoint primitive.

Primitive	**Parameters**	**Required**
ConnectionPoint		Yes
	Unique name	Yes

As stated, an *Endpoint* refers to a specific point in the ACINO controlled network that can ingress and egress customer traffic. For ACINO use-cases, the main type of *EndPoint* is a port on an IP router, but other types are possible too, such as ports on layer 2 switches, OTN timeslots, lambdas or fibre connections (although not clearly required by any use-cases). The traffic can be filtered in a very generic way (i.e. independently of the specifics of a particular equipment manufacturer) based of the properties of each type of EndPoint, as shown in Table 8. End points other than the *IPEndPoint* are marked as not required (in the DISMI) as they are not needed for the use cases that we have identified.

Table 8: List of available EndPoint primitives.

Primitive	**Parameters**	**Required**
IPEndPoint		Yes
	Router-id	Yes
	IPAddress	No
	Port-id	No
	Subnets	No
EthEndPoint		No
	Switch-id	Yes
	MAC address	No
	Port-id	No
FiberEndPoint		No
	Switch-id	Yes
	Port-id	No

LambdaEndpoint		No
	Switch-id	Yes
	Port-id	No
	Lambda Frequency	No
	Width	No

4.2 Actions – connectivity primitives

Actions express the expected connectivity between *Noun* primitives. An *Action* can be modified to add a list of *Constraint* and *Calendaring* when wanted. The DISMI defines the actions shown in Table 9, together with the corresponding types of connectivity. Details about each primitive are given in Table 10. When a line names a primitive, the column "required" states whether it is mandatory for the DISMI to implement it. When a line names a primitive parameter, the column "required" states whether it is mandatory to provide the parameter when using/creating the primitive.

Table 9: Type of connectivity and the corresponding Action primitives.

Connectivity	**Uni-directional**	**Bi-directional**
1:1	Path(src, dst)	Connection(src, dst)
1:N	Multicast(src, [dst])	Tree(src, [dst])
N:1	Aggregate([src], dst)	Tree(src, [dst])
N:N	–	Mesh([dst])

Table 10: Action primitives – The Path primitive can be used to build all the other ones.

Primitive	**Parameters**	**Required**
Path		Yes
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Connection		No
	ConnectionPoints	Yes

	Selectors, Constraints, Priorities	No
Multicast		No
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Aggregate		No
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Tree		No
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Mesh		No
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No

When installing a DISMI intent (see section 6.2), these actions can be implemented directly if the underlying system supports it, and/or be translated into each other. For example, a complex *Action*, such as *Connection*, can be implemented directly if the underlying systems (the Multi-layer optimization module and Controllers) support a bidirectional P2P connection. If not supported, the *Connection* can be implemented as two *Path* primitives, one for each direction. Choosing which compilation is the most appropriate depends on the costs of the underlying primitives in lower-levels, if e.g. there is native support for a Mesh, a single native Mesh is likely a preferable solution (in some sense) than multiple Path connections between all nodes in the mesh.

The connectivity primitives *Multicast*, *Aggregate*, *Tree*, and *Mesh* are ambiguous as they are implicitly defined as sets of *EndPoint*-to-*EndPoint* paths. But this does not need to be the case. For example, Mesh can be interpreted:

- As a logical connectivity, i.e. all nodes can communicate pair-wise with each other. In such a case, the requested constraints (such as bandwidth, delay, …) between two nodes can only be guaranteed as long as no traffic flows through other nodes;
- As an actual mesh of links between the nodes, i.e. all pairs of nodes are connected by a link. The requested constraints between two nodes are then guaranteed independently of the traffic flowing through other nodes.

In a Mesh with four nodes, the logical connectivity interpretation leads to the need of allocating one link per node, whereas in the physical interpretation three links per node are needed. This problem is illustrated in Figure 4.1, and is highlighted clearly when adding a Bandwidth constraint to the *Intent*: does *Mesh(A,B,C,D, 10 Gbps)*, mean that each node has a total of 10 Gbps of capacity or that it has 10 Gbps of capacity to each of the nodes in the *Mesh*? This ambiguity can be solved by either extending the affected primitives with a flag to indicate which interpretation is intended, or by splitting the primitive into one for each interpretation. A final possibility would be to choose just one interpretation and clearly indicate which one is meant.

In the remainder of this work, we define that the more advanced *Action* primitives should always be interpreted as physical connectivity as shown in Figure 4.1 (right): an *Action* is decomposable into a set of *Path* between pairs of *EndPoint*. Other primitives implementing logical connectivity can be added later, should the need arise.

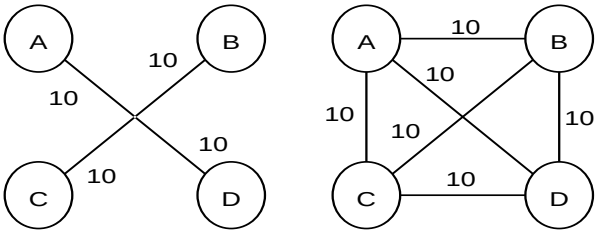

Figure 4.1 : Illustration of the logical (left) and physical (right) connectivity interpretations.

4.3 Modifiers – Selector, Constraint, and Calendaring

ConnectionPoint and *Action* primitives are enough to express best-effort connections. But to enable application-centric networking, additional information is necessary in order to define the ingress traffic profile (selector), the connectivity requirements (constraints), and when the connectivity should be available (calendaring).

4.4 Selector – Classifying applications at ConnectionPoints

A *Selector* specifies which traffic belongs to an application class at an ingress connection point, by specifying which flow(s) of the traffic the *Action* is referring to. Classifying traffic can be done in many ways with potentially a very high granularity depending on the type of *EndPoint* and the equipment that implements it. As ACINO is focusing on IP/Optical networks, the DISMI provides support for selecting traffic at Layer 3 and 4, which can be handled by both layer 2 and 3 devices (switches and routers). However, a goal is to keep the *Selector* definitions flexible so that the DISMI can be extended in the future to support other use-cases. With that in mind, deciding how to define the *Selector* depends mainly on two things: 1) The features supported by the equipment; and 2) The requirements of the ACINO use-cases.

Relevant equipment for ACINO are OpenFlow-based solutions (OpenFlow version 1.0 [OFLOW10] and 1.3 [OFLOW13] will be used), Linux-based solutions (in particular for the emulation framework), and routers

from Cisco and Juniper. This section goes through them one by one to see what they support in terms of traffic selection.

4.4.1 Matching in OpenFlow 1.0

If the end-point is for example a port on an OpenFlow 1.0 switch, a 12-tuple can be used to specify a flow. Some of these fields are overloaded, such as UDP/TCP source and destination ports, so more than 12 logical fields are available for matching. Fields can also be wild-carded, meaning that the field should be ignored during matching. See the OpenFlow 1.0 specifications [OFLOW10], table 3.

All these fields can be used (although since some of them are overloaded, one cannot match both TCP and UDP ports in a single packet) to classify a flow and make a routing decision:

- Ingress Port
- Ethernet source address (Source MAC address)
- Ethernet destination address (Destination MAC address)
- Ethernet protocol type (Ethertype, e.g. IP, ARP, VLAN)
- Virtual LAN identifier (VLAN id)
- Virtual LAN priority (VLAN pcp)
- IP source address + mask (IP source subnet)
- IP destination address + mask (IP destination subnet)
- IP protocol type (e.g. TCP, UDP, ICMP)
- IP Type-of-Service bits
- Transport source port (TCP/UDP source port, ICMP Type)
- Transport destination port (TCP/UDP destination port, ICMP Code).

4.4.2 Matching in OpenFlow 1.3

The 12-tuple in OpenFlow 1.0 [OFLOW10] was extended in subsequent definitions of the protocol to support "Extensible Match", a flexible TLV structure of fields that should be matched rather than the initial fixed tuple provided in version 1.0. In this version, at least 13 different logical fields are required, with 26 additional fields defined but not required, such as support for PBB tags [PBB], IPv6, and SCTP [RFC4960]. As in the case of OpenFlow 1.0, all the fields which are not mutually exclusive can be used to classify a flow and decide where to route it. See the OpenFlow 1.3 specifications [OFLOW13], tables 10 and 11.

4.4.3 Policy-based Routing in Linux

Policy-based Routing (PBR) is a mechanism for IP routers to route traffic based on more than just the IP destination address and the normal routing table. In the case of a Linux-based router, Policy-based Routing is performed by configuring the routing policy database (RPDB) [RPDB] that selects which route to apply

using a set of matching rules. An incoming packet iterates through the tables defined in the RPDB and if nothing matches it, normal destination-based routing typically ensues.

The fields that can be used to match a packet in the RPDB are:

- IP source prefix
- IP destination prefix
- Incoming port
- IP Type-of-Service / DSCP code (Quality of Service marking)
- Firewall tagging.

While seemingly providing less matching possibilities than the OpenFlow protocol, the ability to match on firewall marks greatly extends these capabilities. Any incoming packet passes through the Linux firewall system, Netfilter/IPTables [NETFILTER], before the first routing decision is made. Netfilter supports marking packets [NETMARK] based on many stateless criteria such as protocol headers (e.g. DCCP, SCTP, IPSec, ...), parts of the payload such as arbitrary strings or byte values, incoming device, packet sizes, hash values, or which CPU is processing the packet. Additionally, many stateful criteria such as the current state of a TCP connection can be used through connection tracking, the traffic rate, or even the time of day when the packet arrived. The recent IPtables version 1.4.21 supports over 50 different modules, each with multiple matching parameters.

4.4.4 Policy-based Routing on Cisco routers

Cisco routers support Policy Based Routing through the use of "route maps", which can be used for many things. Route maps contain a series of "if then" statements that allows matching upon some parameters of incoming traffic and perform some actions. Similarly to the Linux approach, the system attempts to find a match in the route map by iterating through it in priority order. If no match is found, normal routing is applied. Route maps in Cisco IOS version 12.2 supports creating policies matching on many criteria. The relevant ones for ACINO are IPv4/6 source prefixes, IP protocols (such as ICMP, UDP, and TCP), application (such as FTP, HTTP, Telnet, and SMTP), and packet size range [CISCIOS122]. Upon matching packets, relevant actions that can be taken are applying route tags (e.g. to route matched traffic into a Multiprotocol Label Switching (MPLS) tunnel), setting the outgoing interface, setting the next-hop router, and setting Type-of-Service and precedence bits.

Some examples of how Access Control Lists (ACLs) can be used together with route maps to perform PBR are shown in Appendix (section 10.3).

4.4.5 Policy-based Routing on Juniper routers

Juniper routers support PBR through their *policy framework*. The feature "filter-based forwarding (FBF)" is particularly relevant to ACINO: FBF supports functionalities comparable to those of the Cisco route-maps approach, but through a different design [JUNOS].

4.4.6 Selector primitives

Based on the requirements from use-cases and what different technologies are capable of, the *Selector* primitives are defined in Table 11. Primitives at the IP and Ethernet level are marked as required because they are needed to support ACINO's use-cases. Primitives for Lambda and GPRS tunnelling protocols are not necessary for ACINO's demonstrations, but they allow the DISMI to provide a more complete and generic API. When a line names a primitive, the column "required" states whether it is mandatory for the DISMI to implement it. When a line names a primitive parameter, the column "required" states whether it is mandatory to provide the parameter when using/creating the primitive.

Table 11: Selector class primitives.

Primitive	**Parameters**	**Required**
Internet protocol		
IPSource		Yes
	IPv4/6 address, mask	Yes
IPDestination		Yes
	IPv4/6 address, mask	Yes
IPProtocol		Yes
	protocol number/enumerator	Yes
IPToS		Yes
	Type-of-Service	Yes
IPDSCP		Yes
	Differentiated Services Code point	Yes
Ethernet		
EthSource		Yes
	MAC address	Yes
EthDestination		Yes
	MAC address	Yes
EthType		Yes
	Ethernet protocol type	Yes

VLAN		Yes
	VLAN identifier	Yes
VLANPCP		Yes
	VLAN priority number	Yes
Lambda		
LambdaFrequency		No
	Channel centre frequency	Yes
LambdaWidth		No
	Width of channel	Yes
GPRS Tunnelling protocol		
GTPTEID		No
	Tunnel identifier	Yes

4.5 Constraints – restrict the potential paths

While a *Selector* defines which traffic belongs to a particular application, a *Constraint* expresses the minimal requirements of an application on the requested connectivity. As with the *Selector*, an almost infinite amount of possibilities can be imagined. The DISMI therefore focuses on the constraints required by the ACINO use-cases and what can be realistically realized. All use-cases require basic traffic profile constraints, i.e. constraints on minimum bandwidth, maximum latency, and maximum jitter.

4.5.1 Bandwidth

Reservation of bandwidth has at least two important aspects to consider: the *description* of the reservation and the *mechanisms* for enforcing it. These two aspects are, in many protocols, closely coupled to each other: one typically has to supply the parameters that configure the traffic shapers, policers, queuing disciplines, and scheduling tools that implement the Quality of Service policies [RFC2210].

How to decouple the bandwidth description from the implementation in a generic and flexible way is not obvious. The two sections below present the mechanisms used by the Metro Ethernet Bandwidth Profile and the Resource Reservation Protocol, respectively, to control bandwidth usage.

4.5.1.1 Metro Ethernet Bandwidth Profile

One example on how bandwidth profiles can be formulated is the Metro Ethernet Forum UNI (See MEF Bandwidth Profiles for Ethernet Services [MEFBW]): a profile is defined based on 5 parameters that define two token buckets, Committed and Excess, and how to treat priority tags.

Committed Token Bucket:

- CIR, Committed Information Rate (bits per second)
- CBS, Committed Burst Size (bytes);

Excess Token Bucket:

- EIR, Excess Information Rate (bits per second)
- EBS, Excess Burst Size (bytes);

Priority handling:

- CM, Color Mode (yes/no), whether to take customer priority tags into account or not.

Packet treatment:

Incoming packets first pass the Committed Token Bucket. If that token bucket is not empty, the packets are forwarded with full guarantees. If the Committed Token Bucket is empty, packets are passed on to the Excess Token bucket. If this bucket is not empty packets are forwarded on a best effort basis. If both the Committed Token Bucket and the Excess Token Buckets are empty, the incoming packet is dropped.

4.5.1.2 Bandwidth Profiles in RSVP

Another example is provided in the Resource Reservation Protocol (RSVP) where two objects are used to define the QoS requirements of a reservation, the SENDER_TSPEC (which indicates the requested bandwidth) and the FLOWSPEC (which indicates actual reserved resources) objects. RSVP defines several different versions of these objects, aimed at different types of services (e.g. Controlled-Load and Guaranteed QoS) and underlying technologies (e.g. packet forwarding networks and time-division networks) [RFC2210, RFC2215, RFC6003].

The RSVP SENDER_TSPEC [RFC2210] characterizes the traffic generated at the source, it is defined as:

- Token Bucket Rate (r) - Senders estimation of its generated traffic
- Token Bucket Size (b) - Senders estimation of its generated traffic
- Peak Data Rate (p) - Senders maximum generated traffic rate (e.g. physical interface speed)
- Minimum Policed Unit (m) - Minimum packet size generated, including the payload and headers above IP layer header
- Maximum Packet Size (M) - Maximum packet size generated.

Exactly how these parameters are used then depends on the different types of reservations that are supported by RSVP. However, as one can see this approach uses a single token bucket compared to the one taken by MEF.

4.5.1.3 Bandwidth Profiles in the DISMI

The objective of the DISMI is to provide an interface that lets applications request features in a descriptive way, not by configuring mechanisms. Therefore, the *Bandwidth* primitive allows configuring only the minimum guaranteed bandwidth. This simple definition than can be further extended in case it is needed.

The *Bandwidth* constraint simply specifies the minimum guaranteed rate, as shown in Table 12, allowing underlying layers to set typical default values, or estimate them based on the rate (e.g. using a heuristic as bucket size should correspond to number of packets send during time *t* at maximum rate). *Bandwidth* specifies only the minimum guaranteed bandwidth. There may still be additional bandwidth available but ACINO does not provide any guarantees on that bandwidth, nor does it imply that any bandwidth throttling will take place. The actual configuration of the network policing/shaping mechanisms and accounting of resources could also be affected by other parameters such as latency, as larger token buffers cause higher latencies in the network due to packets waiting in queues. It is the task of the orchestrator to account for such parameters in order for the bandwidth requirement to be met, with some margin if necessary (or if the algorithm predicting the available bandwidth is not very accurate).

Table 12: Bandwidth class primitive.

Primitive	**Parameters**	**Required**
Bandwidth		Yes
	Bitrate	Yes

4.5.2 Delay - Latency and jitter

Latency limits are also important for many use-cases and can be divided into one-way and two-way delays (round-trip time). Latency is the sum of two components: the fixed delay (consisting of transmission and propagation delay), which depends on the particular path chosen, and a variable delay composed of processing and queuing delays. All of these components can be estimated, and the variable delay controlled through e.g. QoS settings. Jitter, or packet delay variation, is also an important constraint for some use-cases (in particular related to the 5G infrastructure) and can be controlled in similar ways as latency.

Maximum latency and maximum jitter (see Table 13) are simple primitives that are interpreted as one-way latency/jitter or two-way latency/jitter, depending on the *Action* they are associated with (i.e. one-way for Path, two-way for Connection).

Table 13: Delay class primitives.

Primitive	Parameters	Required
Latency		Yes
	Time	Yes
Jitter		Yes
	Time	Yes

4.5.3 Availability

Availability is an important constraint for many use-cases. Availability in networking typically entails having redundant paths, both at the link and node level, and having mechanisms for quickly detecting a failure and re-establishing connectivity using alternative links and/or nodes. Alternative links or paths can be proactively allocated during provisioning of the primary link (typically called *protection*), or calculated in a reactive fashion once a failure has been detected (typically called *restoration*).

Protection is usually able to restore connectivity within tens of milliseconds, while restoration can take minutes depending on e.g. the topology. In the protection case, one can also distinguish between 1+1, 1:1, and 1:N protection schemes, where 1+1 means that the primary and backup path are active at the same time, 1:1 that the backup path is reserved but not active, and finally 1:N where multiple primary paths share a single backup. These alternatives have different cost since different amounts of resources have to be allocated, and provide different level of reliability: if multiple paths fail simultaneously in the 1:N case, only the first one will be protected whereas in the 1:1 case all are protected.

Availability can be specified in percentage or as its equivalent downtime per a certain time period. For example, an availability of 99% translates into a maximum of 3.65 days of downtime over a year, 99.999% to 5.26 minutes over a year, and so on. This formulation is understandable when describing time-bound services, e.g. a point-to-point connection service allocated for a year. It is more difficult to translate uptime and downtime measurements into an availability when the service life time is unknown. In addition, a 99% availability over a year may translate into a single event of 3.65 days of consecutive unavailability or 313650 events per year each with one second of unavailability. While both of these situations represent 99% availability, the consequences for the application are likely quite different.

Availability of a certain component can be calculated from the Mean Time Between Failures (MTBF) and Mean Time To Recovery (MTTR) as Availability = MTBF/(MTBF+MTTR). If the availability of all the components and links is known, the availability for the whole path can be calculated [BISMUK].

The MTBF is the predicted mean time between when a failure is repaired and when a failure appears again. The MTBF can be predicted by analysing the different components of the network, or estimated from previous data from similar environments.

The MTTR is the average time it takes to recover from a failure once it has happened. This parameter can be controlled during service deployment by employing various fault detection and recovery mechanisms. The MTTR for different networking technologies can be seen in Table 14.

The DISMI takes advantage of the coupling between the availability and MTTR and MTBF. It provides MTTR and MTBF as primitives to characterize the availability constraints. These parameters are more easily used by applications, where in particular MTTR can be mapped to e.g. a timeout in a higher layer protocol. It simplifies the orchestrator's choice of implementation method. MTTR can be mapped to the recovery mechanism e.g. packet-based restoration or optical protection, and MTBF to the quality of links and nodes used. ACINO provides the availability primitives shown in Table 15. Like the previous tables about primitives, when a line names a primitive, the column “required” states whether it is mandatory for the DISMI to implement it; when a line names a primitive parameter, the column “required” states whether it is mandatory to provide the parameter when using/creating the primitive.

Table 14: Mean Time To Recovery (MTTR) for different technologies [WPPROT].

Technology	**MTTR**
SONET Automatic Protection Switching	< 50 ms
Ethernet Resilient Packet Ring (IEEE 802.17b)	< 50 ms
MPLS-TP with BFD	< 50 ms (depending on configuration)
Optical protection (1+1, 1:1, 1:N)	< 20 ms
MPLS Fast Reroute	< 50-100 ms (depending on configuration)
Ethernet Rapid STP (RSTP, IEEE 802.1w)	1-3s, depending on topology
Ethernet Spanning Tree (STP,IEEE 802.1D)	30-60s depending on topology
OSPF Convergence	> 5-60s (depending on topology/configuration)
Optical restoration	Second to Minutes

Table 15: Availability class primitives.

Primitive	**Parameters**	**Required**
MTTR		Yes
	Time	Yes
MTBF		Yes

	Time	Yes
Availability		No
	Percentage	Yes
SurvivableDowntime		No
	String: “none”, “low”, “high”	Yes

4.5.4 Information security

Confidentiality and integrity are at the core of information security, and both of them can be provided at different degrees during service deployment. Confidentiality, making sure that information is not made available to unauthorized individuals, can be provided through encryption mechanisms at various layers of the network: for example, at the IP layer through IPSec Encapsulating Security Payloads [RFC4303] or at the optical layer through the encryption of the optical channel (See FSP 3000 [ADVAFSP]). Another method of providing a measure of confidentiality is to avoid traversing certain regions of the network, for example avoiding sending the traffic through links passing through another country’s territory.

In the ACINO case, confidentiality through encryption can only start at the ingress node, meaning that the customer would have to trust the network until and including that point. While typically encryption requires the user (or in this case, the application) to provide either a public or secret key, this is not useful in the ACINO case as the encryption is expected to be transparent to the user: packets arrive and leave the ACINO network in plain text, but traverse it encrypted. For this reason, ACINO cannot provide end-to-end encryption, and if it is needed, the application has to implement it by itself. However, this simplifies the interface as no parameters are necessarily required, either encryption is turned on or it is not. This could potentially be extended with a "strength" parameter to indicate e.g. which key-length or encryption algorithm to use, usually at the cost of lower throughput, or through a monetary cost for the Application.

Confidentiality through region exclusion is a more complicated scenario, with a political and business level in addition to the technical one, as it is not clear exactly what it means for every use-case. One option would be to e.g. tag all components in the topology with their geographical location, manufacturer of the equipment, provider of the links in case they are provided by a third party, etc. This model however requires that the application is able to retrieve and understand these labels (or that they are grouped into meaningful categories that the application understands), and that such information can be found and inserted into the topology, which is a daunting task.

Another, more realistic option is to resolve this case on the political/business level when negotiating with the ACINO network provider, to clarify on a per application basis what paths and nodes should be excluded from the requested connectivity. The Sensitive primitive can be included in the intent to show that any predefined exclusion zones should be applied to this request.

Integrity, making sure that information cannot be modified in an unauthorized manner, can also be provided at various degrees at deployment time, in the strictest sense by cryptographic means for example through IPSec Authentication Headers which digitally signs packets. Integrity in the sense of random errors during transmission over the network can also be improved by configuring more stringent, and costly, error detection and correction methods.

The primitives related to information security defined by ACINO are shown in Table 16.

Table 16 : Security class primitives.

Primitive	**Parameters**	**Required**
Encryption	--	Yes
Encryption	Strength	No
Sensitive	--	No
Integrity	--	No
Integrity	Strength	No

4.6 Priorities – Defining Application priorities

Another set of primitives are priorities: They allow the application to specify soft constraints, or goals, of the requested service. If several network configurations meet the requirements of the requested network service, priorities are used to choose the most appropriate one. If multiple priorities are included, they are taken into account by order. The primitives available for the *Priority* primitive are shown in Table 17.

Table 17: Priority class primitives.

Primitive	**Parameters**	**Required**
Availability	--	No
Cost	--	No
Latency	--	No
Jitter	--	No
Bandwidth	--	No

- **Availability**: Choose the configuration that is least likely to fail (if all the components have the same availability, this will typically be the shortest path, as the lowest number of components is involved);
- **Cost**: Choose the path with the least monetary cost for the application (note that this is not the same as "least-profit to the operator");
- **Latency**: Choose the path with the least end-to-end latency;
- **Jitter**: Choose the path with the least end-to-end jitter;
- **Bandwidth**: Choose the path with most available extra bandwidth or bandwidth potential for scaling the service later.

4.7 Calendaring - when to start and stop the service?

Service calendaring is the ability to request a service from and to a certain time, or for a certain cost. The associated primitives are shown in Table 18. This constraint is of interest for cloud applications such as dB synchronisation and VM replication (see section 2.2.3.3), which may be scheduled during low-load hours or when the connectivity is inexpensive. The primitives allow the application to define:

- An open-ended service starting at a certain time but without a specified ending;
- A service with both start and end;
- A service terminating at a certain point;
- A service that is not started unless the hourly cost is lower than specified.

Table 18: Calendar class primitives.

Primitive	Parameters	Required
StartTime	Datetime	Yes
StopTime	Datetime	Yes
CostLimit	Cost/h	No

4.8 Information model – The intent grammar

Based on the defined primitives, we can create an information model that specifies what information is needed and how it can be organized to construct an *Intent*. The information model for the DISMI, keeping most optional primitives out, is shown in Figure 4.2. In the figure, blue boxes indicate containers that define a group but do not hold any information in themselves (like abstract classes or interfaces), brackets ([]) indicate a list and braces ({}) an optional list.

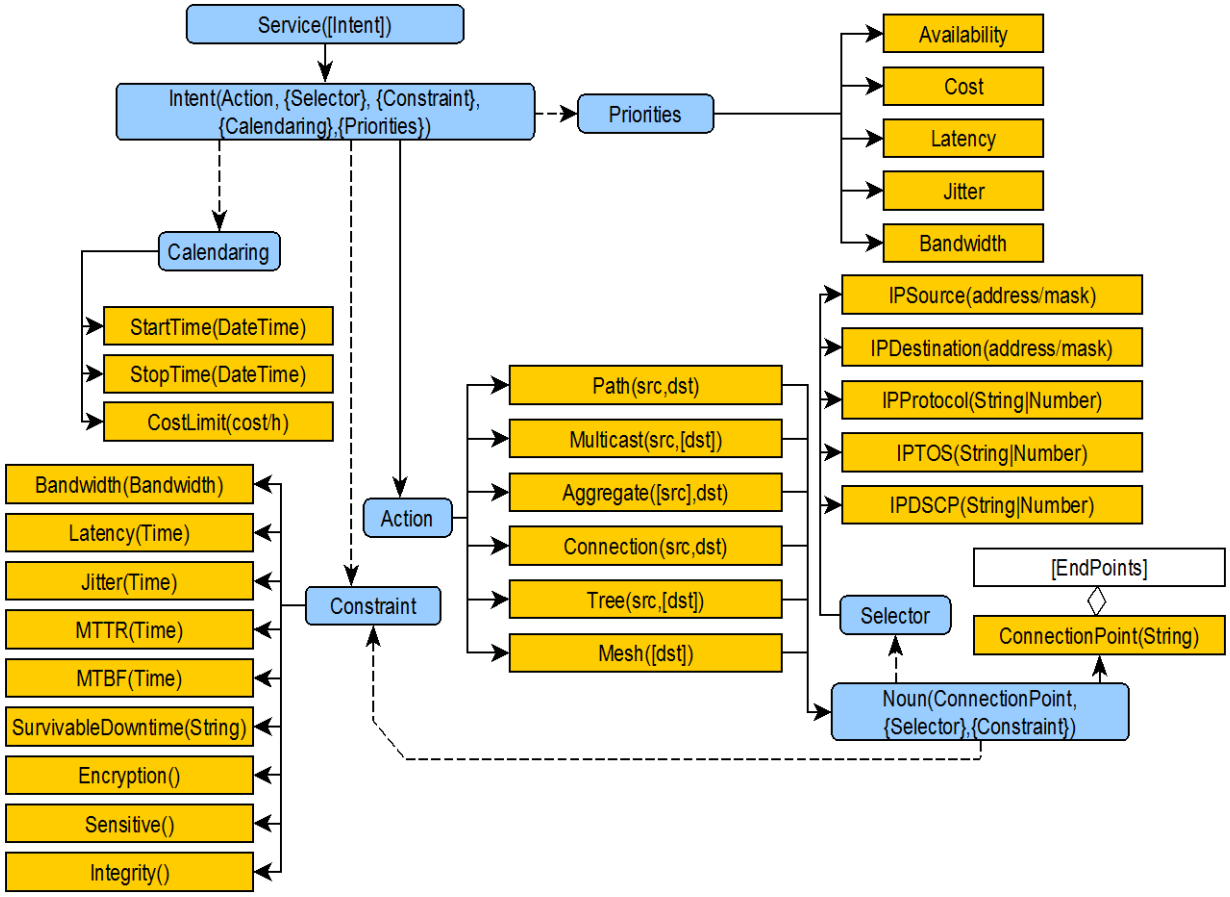

Figure 4.2 : Information model to construct an Intent.

From the figure we can read that:

- A *Service* consists of a list of *Intent*, containing at least one;
- An *Intent* consists of an *Action*, an optional list of *Selector*, an optional list of *Constraint*, an optional list of *Calendaring*, and an optional list of *Priorities*;
- *Action* classes contain one or two (lists of) *Noun*; There are six *Action* classes;
- A *Noun* consists of a *ConnectionPoint*, an optional list of *Selector*, and an optional list of *Constraint*;
- A *ConnectionPoint* has a single string parameter and is mapped to a list of *EndPoints* inside the intent framework. The mapping is not exposed to the application, except through the API used to handle the mapping;
- A *Constraint* consists of six classes with either a bandwidth, time or no parameter;
- *Calendaring* consists of two classes each with a date-time parameter;

- *Priority* consists of five classes without parameter;
- *Selector* consists of five classes with different parameters (IP prefixes, names/numbers, etc);
- Note that the *Constraint* and *Selector* primitives can appear both at the *Intent* and *Noun* levels. This is to allow applications to either set a default value for all links in the *Intent* by placing constraint(s) at the *Intent* level, and/or to specify *ConnectionPoint*-specific constraints at the *Noun* level.

4.9 Constructing Intents from network primitives

Using the various primitives described in the previous section we can start to construct Service intents. Three examples are given below, showing how simple and complex intents can be constructed from the basic network primitives.

4.9.1 Example Intents from primitives

Initially a mapping between ConnectionPoints and EndPoints has to be established. This is done prior to the Application connecting to ACINO, for example as part of contract negotiations between the ACINO and Application owners (see Table 19).

Table 19: Mapping example between ConnectionPoint and Endpoint.

ConnectionPoint	**EndPoints**
"Office-Madrid"	[IPEndpoint(Router1,port2)]
"Office-TelAviv"	[IPEndpoint(Router3,port5)]
"Office-Trento"	[IPEndpoint(Router6,port1)]

4.9.1.1 Unidirectional path

Establish a unidirectional, best-effort, connection from Madrid to TelAviv:

```
Path("Office-Madrid", "Office-TelAviv")
```

4.9.1.2 Bidirectional connection with constraints, calendaring, and priorities

Establish a bidirectional connection between Madrid and TelAviv, with a minimum bandwidth of 10Gbit/s and maximum latency of 20ms, and a MTBF of 50ms. Prioritize reliable, low-latency, connections and start the service on New Year's Eve:

```
Connection(["Office-Madrid", "Office-TelAviv"],
constraints=[Bandwidth("10Gbit/s"),Latency("20ms"), MTBF("50ms")],
calendar=[StartTime("2015-12-31,10:00:00")], priority=[Availability,
Latency])
```

4.9.1.3 Encrypted mesh between all offices, with constraints, calendaring, and priorities

Establish an encrypted mesh between all offices, with minimum 20 Gbit/s bandwidth and maximum 20ms latency. Priorities are Availability and Latency, the service should be started at New Year's Eve.

```
Mesh(["Office-Madrid", "Office-TelAviv", "Office-Trento"],
constraints=[Bandwidth("20Gbit/s"),Latency("20ms"), Encryption],
calendar=[StartTime("2015-12-31,10:00:00")], priority=[Availability,
Latency])
```

4.10 Templates – further simplifying service requests

With as many as 8 different classes of primitives, for an initial total of about 30 required primitives, finding and composing the appropriate ones for a particular application may be a difficult task. For this reason, it would be good to construct high level templates for commonly requested service types, such as Voice traffic, Bulk data transfers, Video broadcast, etc. Templates for such functions can be created by, for example, providing a list of typical services containing a list of appropriate primitives with default values. This list could then be exposed to applications which could then use them to create their intents.

However, with the intent framework in place, extending it with additional actions that use the existing ones to build a high level service is a more natural solution. For example for bulk data transfer, a new intent could look like:

```
BulkTransfer("ConnectionPoint A", "ConnectionPoint B", "amount=20 TB")
```

This intent would be built using two uni-directional paths with different properties:

- One high-bandwidth path from the source to the destination to transfer data. The optimal bandwidth of the path might be calculated by the orchestrator, for example by taking into account the maximum bandwidth of both end points.
- One best effort, low-latency, path from the destination to the source for acknowledging chunks of transferred data and another from the source to destination with a high bandwidth for transfer of data chunks; "`Path(A, B, constraints=[Bandwidth(maxBandwidth(A, B))]), Path(B, A, priority=[Latency])`".

Intents for different types of ACINO services can be defined in the same way. However, the definition of such high level actions is left for future studies.

5 Northbound application interface

This chapter presents the protocol requirements on the northbound interface (NBI) that is to be used by applications to communicate with the Orchestrator. The NBI defines how an application can communicate with the ACINO orchestrator in order to deliver its intentions as well as modify, delete, and read the status of ongoing services and service requests.

5.1 Requirements on the interface

The Northbound interface should support automatic discovery of available primitives, configured ConnectionPoints, the intent grammar for composing primitives, and available NBI commands. In addition to this, asynchronous notifications should be supported in order to inform an application where in the life-cycle an intent is (e.g. if it has Failed). For the different types of services and resources (Services, Intents, Primitives) the typical CRUD operations (Create, Read, Update, Delete) should be supported.

The interface should also support mutual authentication between the Application and the ACINO orchestrator, with the ability of restricting the available primitives and NBI commands depending on the role of the Application (e.g. normal user, super user, etc.). Finally, it should be easy and intuitive to use, easy to access, and easily extendable. The requirements on the NBI can then be summarized as follows:

1. Discovery of primitives and ConnectionPoints;
2. Discovery of intent grammar;
3. Discovery of NBI commands;
4. CRUD for Services, Intents, ConnectionPoints (requirements NB-INT-1, NB-INT-3, NB-INT-5 and NB-INT-6 in Table 1);
5. Asynchronous notifications (requirement NB-INT-4 in Table 1);
6. Mutual authentication between Applications and Orchestrator;
7. Easy to use, generic, extensible.

5.2 Defining the Intent Grammar

The intent grammar can be defined in several different ways, for example as a context-free grammar or as data models using for example YANG or JSON-Schema. It is important to note that intents need a grammar in two locations, at the application-facing northbound interface, and internally in the intent framework. This section focuses on the former.

5.2.1 Formal grammar

One of the most flexible ways of defining a grammar is through the use of a formal grammar using a series of production rules, for example in Augmented Backus-Naur Form [RFC5234]. This method is a very generic

method that allows us to define unambiguous grammars, for example for whole human languages. One example is Lojban [LOJBAN], which is an artificial spoken language with a non-ambiguous grammar.

Using this approach we can define the grammar for a language able to parse the string "Path from office to internet with bandwidth 10 gbps and 10 ms latency" and extract the important information. In the syntax of the pyParsing recursive descent parser [PYPARSING], the grammar for this intent can be expressed as shown in the appendix (section 10.4).

Running the code from the appendix on the string "Path from office to internet with bandwidth 10 gbps and 10 ms latency" or "Path to internet from office with 10 gpbs bandwidth and latency 10 ms" and then printing the parse-tree as XML results in the output shown in the same section of the appendix.

As it can be seen, the different relevant parts of the sentence have been extracted and grouped in a meaningful way, together with particular words that are the variables for the different parts of the intent. However, this method is likely too flexible and writing an unambiguous grammar that is able to express an intent in something similar to e.g. English is quite a challenge. Finally, it is not very useful as a machine-to-machine interface between an orchestrator and applications (although potentially useful for human interaction).

5.3 Realizing the intent interface: protocols and APIs

Many protocols are potential candidates to implement the NBI of the orchestrator:

- Binary protocols on top of TCP/UDP, using for example Avro/GoogleProtobufs/Thrift;
- Messaging systems such as RabbitMQ, ActiveMQ, ZeroMQ;
- NETCONF and other network controller protocols;
- Web protocols such as REST over HTTP/JSON, HTTP/XML, ...

APIs in the Web environment typically try to apply the RESTful architectural style using HTTP and JSON [RESTFUL] to define resources/function calls (HTTP Method + URI) and data types (JSON). In our case, we want the application to be able to fulfil the requirements defined in section 2.2.1 and 5.1, these can all be fulfilled by a RESTful HTTP/JSON interface.

In addition, the existing intent interface in ONOS uses a RESTful API over HTTP with JSON objects. Implementing the DISMI in a similar fashion was hence an easy choice. The reminder of this section presents the protocols and APIs designed for the DISMI.

5.3.1 External interface - Data model for RESTFul interface over HTTP/JSON

Connection setup and Application authentication can be handled by HTTP [RFC7235], with Orchestrator authentication (and connection security) handled by HTTPS with SSL or TLS. By using the various HTTP Methods (GET to read, POST to create, PUT to update, and DELETE to delete) with different URIs we can define an API for interacting with the ACINO orchestrator. Additional parameters to the requests are

provided as JSON data in the requests. The success of the request is indicated by a response code (e.g. 200 OK, 404 Not Found), and data is encoded as JSON. The API is presented in Table 20:

Table 20: RESTful HTTP API with methods and routes.

Method	**Route**	**Description**
GET	/service	Get a list of the current services
GET	/service/{id}	Gets overview of a specific service
GET	/service/{id}/{id}	Gets overview of a specific intent in a service
GET	/service/all	Gets overview of all current services
POST	/service	Create a new service
PUT	/service/{id}	Update a service
PUT	/service/{id}/{id}	Update an intent in a service
DELETE	/service/{id}	Delete a service
DELETE	/service/{id}/{id}	Delete an intent in a service
GET	/connectionpoint	Gets a list of the defined ConnectionPoints
GET	/connectionpoint/{name}	Gets overview of a specific ConnectionPoint
GET	/connectionpoint/all	Gets overview of all ConnectionPoints
PUT	/connectionpoint/{name}	Update a ConnectionPoint (e.g. change name)
POST	/connectionpoint/{name}	Create a new ConnectionPoint to EndPoint mapping
DELETE	/connectionpoint/{name}	Delete a specific ConnectionPoint
GET	/primitives	Gets a list of primitives and associated grammar
GET	/primitives/{name}	Gets the grammar for a particular primitive

The request and response parameters are missing from Table 20, to keep it easily readable. These can be defined using data modelling mechanisms such as YANG [RFC6020] or JSON Schema [JSON].

Using JSON Schema we can define a data model for the information model shown in Figure 4.2 above. For example, a GET request to /connectionpoint returns a list of *ConnectionPoint*, this response can be modelled in JSON Schema as:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "ConnectionPoints",
  "description": "List of ConnectionPoints",
  "type": "object",
  "properties": {
    "connectionpoints": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "uniqueItems": true
    }
  },
  "required": [
    "connectionpoints"
  ]
}
```

The Application can use this schema to determine how the data it receives is structured, as well as validating it. A valid response, according to the schema, is: {"connectionpoints": ["Office-Berlin", "Office-Stockholm", "Office-Madrid", "Office-Trento", "Internet"]}.

Using JSON Schema, we can define the structure of the requests and responses of the API components (i.e. the *Action*, *Constraint*, *Priority*, and *Selector* primitives).

We can go even further and define the whole interface using JSON Hyper-schema, which includes the HTTP Methods, the URIs, and the expected response and request data structures. For example, for the "GET /connectionpoint" request, the schema can be defined like this, with "targetSchema" referring to the already defined "connectionpoints" as expected response data.

```
{
  "description": "Gets a list of the defined ConnectionPoints",
  "href": "/connectionpoint",
  "method": "GET",
  "rel": "instances",
  "title": "Get list",
  "targetSchema": {
   "$ref" = "#/connectionpoints"
  }
}
```

5.3.2 Asynchronous operations

The RESTful API defined in Table 20 provides methods for the application to send requests to the orchestrator, but it does not define how replies are sent back to the application. For calls that are fast enough, it may make sense to keep the TCP connection open until the orchestrator has replied to the application. However, for more time-consuming tasks, keeping TCP connections open has some drawbacks:

- The server needs to be able to handle many open connections simultaneously, which is resource-hungry;
- Connections may time-out, leaving the resources created by the orchestrator unreachable (as the application won't know whether they have been created).

Asynchronous operations are a lot more flexible: they provide a way to check the status of ongoing operations, but also of established ones. There are at least two ways to handle asynchronous access to resources [RESTASYNC1]:

- Resource-based: The API call returns a resource reference [RESTASYNC2] that can be later used to check the status of the request (by polling);
- Callback-based:
    - Direct: The application provides a callback URI that the orchestrator later uses to inform the application of the status (completion/failure) of the request. This solution presents a practical problem, as it assumes that the application is reachable from the orchestrator, which may not be the case (NAT, firewall rules, …);
    - Using a callback queue: for example, the orchestrator maintains a callback queue using a messaging system such as RabbitMQ. The orchestrator sends updates to the queue about the success/failure of requested operations, and connected applications receive messages as they arrive. A tutorial using RabbitMQ is provided in [RMQCQ]. Being located behind a NAT or firewall is not a problem either, as it is the application that connects to the queue. Finally, if an application is offline when a message arrives, the message stays in the queue and can be retrieved by the application when it gets back online;
    - Websockets: The API call initiates a Websocket [RFC6455, WEBSOCKET], which is a full-duplex TCP connection using port 80. NAT and standard firewall rules do not block websockets, and this method allows bidirectional communication between the server and the clients.

The direct callback solution has practical limitations. A callback queue is a more attractive solution, but implies the coexistence of two different technologies to implement the DISMI: a Web protocol (REST over HTTP/JSON) and a messaging system over e.g. RabbitMQ. This does not represent a very clean design.

Using Websockets is a popular solution, but keeping TCP connections open is not ideal. Connection time-outs may not be a problem, as a resource number may be provided to clients for every intent request, allowing them to reconnect. Nevertheless, websockets force the server to keep a TCP connection open with each client waiting for a reply, which is resource-hungry.

Hence, a resource-based implementation of asynchronous operations is chosen. One of the main advantages of this solution is that no update of the API proposed in Table 20 is necessary, as GET methods already allow the asynchronous polling of intent and service status after their creation.

Further, we can specify which values the API should/is allowed to return, depending on whether the call is asynchronous or not. According to the IETF RFC 7231 [RFC7231]:

1. **HTTP 201 "Created"** should be used when the request has been fulfilled and the expected resources created;

2. **HTTP 202 "Accepted"** should be used when the request has been accepted for processing, but the processing is not completed. A link to the resource reference can be provided in the Location field of the header:

```
http POST https://api.acino.eu/service Service={xxxxx}
HTTP/1.1 202 Accepted
Location: /service/12345
```

5.4 Negotiation

The idea behind the negotiation of network services is to offer to the application not only the possibility to request a connectivity service, but also to evaluate which solution it prefers, in terms of price and performance provided. A service negotiation may happen in two cases:

1. There are several solutions that fit the application request;

2. The network resources are not enough to satisfy the original intent.

In the first case, the application requests an intent and the DISMI answers with several possible solutions. The application evaluates them and then chooses the solution it prefers. In simple cases, one may include *Priority* primitives in order to avoid a negotiation, as illustrated in Figure 5.1. That approach does however not allow more than a basic ordering of the solutions.

In the second case, the status of the network does not permit the allocation of the intent. This may happen e.g. because some network resources are overloaded during the requested time span, or because some requested constraints (encryption, maximum delay, ...) are impossible to meet between the specified connection points. In order to permit the installation of a service anyway, the DISMI offers the application the possibility to negotiate a different service from the network, by accepting a modification of the intent constraints. Using negotiation, a valid intent request containing no priority primitives would be processed as illustrated in Figure 5.2.

It should be noted that evaluating which constraint(s) need to be relaxed for the intent to be allocated may be a daunting task for all but the simplest situations. We assume here that the multi-layer optimisation routine that will be used in the ACINO orchestrator has a way to do so, possibly using a crude method such as removing a constraint and trying again to install the (now relaxed) intent.

To support negotiations, the DISMI needs to keep track of the states of all intents: not only those that are installed, but also those that are under installation and the relaxed intents proposed to the application.

When an application sends an intent-based service request to the orchestrator, the request is processed by the intent framework. There can be several results to this processing, and the framework reacts differently depending on whether the intent contains priorities and whether negotiation is used. An overview of the request processing is provided below:

1. The request is invalid: for example, the user is not allowed to request such a connection. The intent framework returns an error message;
2. The request is valid, but no connection that fulfils all the requested criteria is found. The intent framework may opt for:
    a. **Priorities** or **no negotiation**: Send an error message back to the NBI and terminate the request;
    b. **Negotiation**: If the framework has a list of connections that partly fulfil the request, a negotiation with the application is started to let the application choose whether one of these connections is acceptable;
3. The request is valid, and there is only one solution found by the intent framework:
    a. **Priorities** or **no negotiation**: This connection is set up and the intent framework returns a success message;
    b. **Negotiation**: This connection is proposed to the application, which either accepts or refuses it;
4. The request is valid, and there are several solutions found by the intent framework:
    a. **Priorities:** The priority list is used to order the solutions to the request, from the preferred to the less interesting;
    b. **No negotiation**: The framework orders the solutions using its own priority function (which is set up by the network operator and reflects policies such as lowest operational costs, concentration of the traffic on the lowest number of links, spreading out the traffic evenly over the links to minimize the effect of a hardware failure, …);
    c. **Negotiation**: The list of solutions is proposed to the application, which chooses the connection it prefers.

So, depending on how the intent framework handles requests, an application may be given the opportunity to negotiate for the properties of the connection. From the application's perspective, not supporting negotiations makes for a simpler interface with the orchestrator, but finding out why a connection request has been denied may be more difficult. Supporting negotiations gives more flexibility to the application, at the expense of complexity for both the application and the intent framework.

From the application's perspective, one can distinguish three cases with respect to negotiation:

1. **No negotiation**: the intent framework makes the decisions using the request as well as its internal rules, and either provides a connection or returns an error message;
2. **Short negotiation**: When several possible connections exist, the intent framework chooses the most appropriate one; when none exists, the application is given the possibility to revise its request. This provides an added value compared to the no negotiation case, while limiting the complexity of the intent framework;
3. **Active negotiation**: In this case, every time there is a choice to be made, the application is contacted to make the most appropriate one.

The *no negotiation* solution is the simplest to implement, but it significantly limits the available functionality. In the context of ACINO, the *short negotiation* solution is the favoured solution as it allows negotiation with limited complexity. The *active negotiation* solution is quite attractive from a research perspective, and it may be investigated but is outside the implementation scope of the ACINO orchestrator.

Providing support for negotiation does not require any additional API: getting an overview of a specific *Service* or *Intent* is done by the following calls, respectively:

```
GET /service/{service_id}
GET /service/{service_id}/{intent_id}
```

If the service status indicates that a negotiation is necessary, then the application can either cancel the service request or go through the list of intents pertaining to the service to discover which one(s) require negotiation. Implementation-wise, it requires that the overview of a specific intent contains an optional list of connections.

Choosing a proposed connection for an intent from a list provided by the intent framework is performed by a call to update the intent: `PUT /service/{service_id}/{intent_id}`. Implementation-wise, this function call needs to support an optional parameter specifying which connection has been chosen.

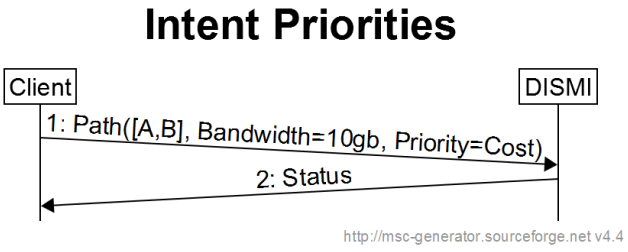

Figure 5.1: Intent process with priorities in the Intent.

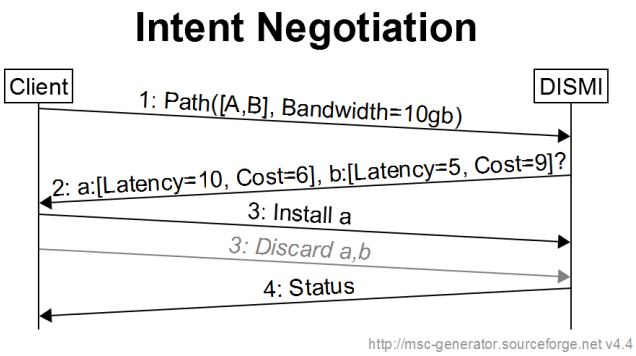

Figure 5.2: Intent negotiation process.

It is expected that the intent framework will first be implemented without support for negotiations, and that short negotiations will be added once the framework is working in a satisfactory manner.

6 Design of the intent framework

The intent framework is a module of the ACINO orchestrator implementing the DISMI, a northbound interface for applications. Chapter 4 has presented the network primitives of the DISMI, which applications can use to build intent requests. Chapter 5 has presented the NBI itself, a RESTful HTTP API. This chapter describes the design of the intent framework, how intent requests are processed and how the intent framework connects to ONOS, the network controller that the ACINO orchestrator is based upon.

Section 6.1 presents the various types of intent primitives that the framework has to deal with. Section 6.2 presents the intent framework architecture. Section 6.3 discusses the possible states that a DISMI intent can take during its processing, once it is installed and when the application decides to remove it. This is of interest because this state is exposed to the application, which can query it at any time.

6.1 ACINO intents and ONOS intents

The term *intent* is used heavily within ACINO as well as in ONOS. Figure 6.1, which presents the architecture of the DISMI and its connection the application-centric compiler (that itself connects to ONOS), shows where the intents defined within the ACINO orchestrator are used. The various types of intents that we have to deal with within ONOS and the ACINO orchestrator are:

- **DISMI intent:** is the intent primitive defined as part of the DISMI. The DISMI is independent of ONOS, and the DISMI can be ported to other network controllers. With respect to the ONOS intents, DISMI intents have broader scope and higher abstraction. For instance, the connection point provided by the application to the DISMI interface may not be actually represented within the ONOS topology (e.g. a Data Center with multiple ingress/egress points to/from the network), while the requested connectivity graph may be richer than the one available by means of ONOS intents.
- **ACI intent**: The intent framework needs to compile DISMI intents into ONOS intents so they can be installed on the network. However, ONOS intents are missing some features necessary to implement ACINO. Indeed, ONOS intents are not developed to interact with an external computing entity (see Section 6.2.2), only provide a small set of primitives with respect to the ones presented in this document, and are not fully compliant with the ACINO approach for managing multi-layer IP/Optical networks, as reported in D4.1 [ACID41]. To remedy this, the Application-Centric Intents or ACI intents have been created [ACID41]. ACI intents derive from ONOS intents but have added functionalities. Like ONOS intents, they need their own compilers. Because they are tailored for ACINO, DISMI intents can easily be transformed into ACI intents;
- **ONOS intent**: ONOS provides its own intent-based interface, with several intent primitives. We refer to them as ONOS intents;
- **Installable intent**: ONOS compiles its ONOS intents into one or more installable intents. Each ONOS intent has its own compiler that generates one or more installable intents. Installable intents are

used by ONOS to keep track of which ONOS intent created which connections, provide event-based information about network disruption, etc..

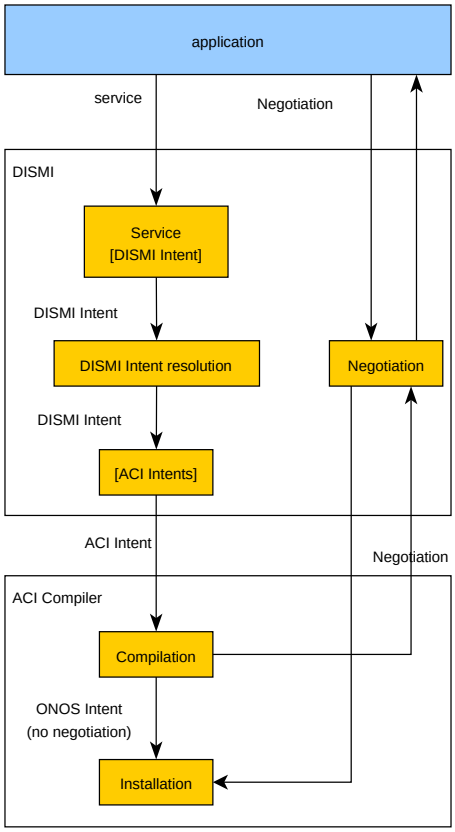

Figure 6.1: Architecture of the DISMI and connection to the ACI compiler.

6.2 Installing a DISMI intent: the intent framework architecture

Section 5.4 presented the negotiation process between the DISMI and the application, and how an intent request is processed. This section builds upon that discussion to present the architecture of the intent framework: section 6.2.1 discusses the DISMI module and section 6.2.2 the ACI compilers.

6.2.1 The DISMI module

The DISMI module is shown in Figure 6.1. When a service request arrives, each DISMI intent is checked for validity, then compiled into a (set of) ACI intents. ACI intents are then passed on to the ACI compiler, where possible solutions are looked for. If negotiation is supported and expected (i.e. no priority constraints were provided), then a negotiation is initiated. The chosen intent is sent back to the ACI compiler for installation.

The DISMI module resolves and validates the incoming intents as illustrated in the flowchart in Figure 6.2:

1. **Resolve the Actions**:
   a. Find the module(s) that support Action X, transform the action into a set of simpler actions if necessary;
   b. Check that the required parameters are passed (e.g. no empty Noun lists);
2. **Resolve the Endpoints from Nouns**: Check that Endpoint(s) mapped by ConnectionPoints match (e.g. cannot connect a LambdaEndpoint to an IPEndpoint), Select the union of the Endpoints to find one or more that are in common for the Endpoints;
3. **Resolve the Selector primitives**: Check that Selectors are supported by the EndPoints (e.g. cannot apply a LambdaSelector to an IPEndpoint);
4. **Resolve the Constraint primitives**: Validate the Constraints, e.g. impossible bandwidths or latencies;
5. **Resolve the Calendaring primitives**: Validate the Calendaring, e.g Start/StopTimes in the past, wrong order of StopTime/StartTime;
6. **Resolve the topology**: Retrieve the expected topology during the time span set by the calendar.

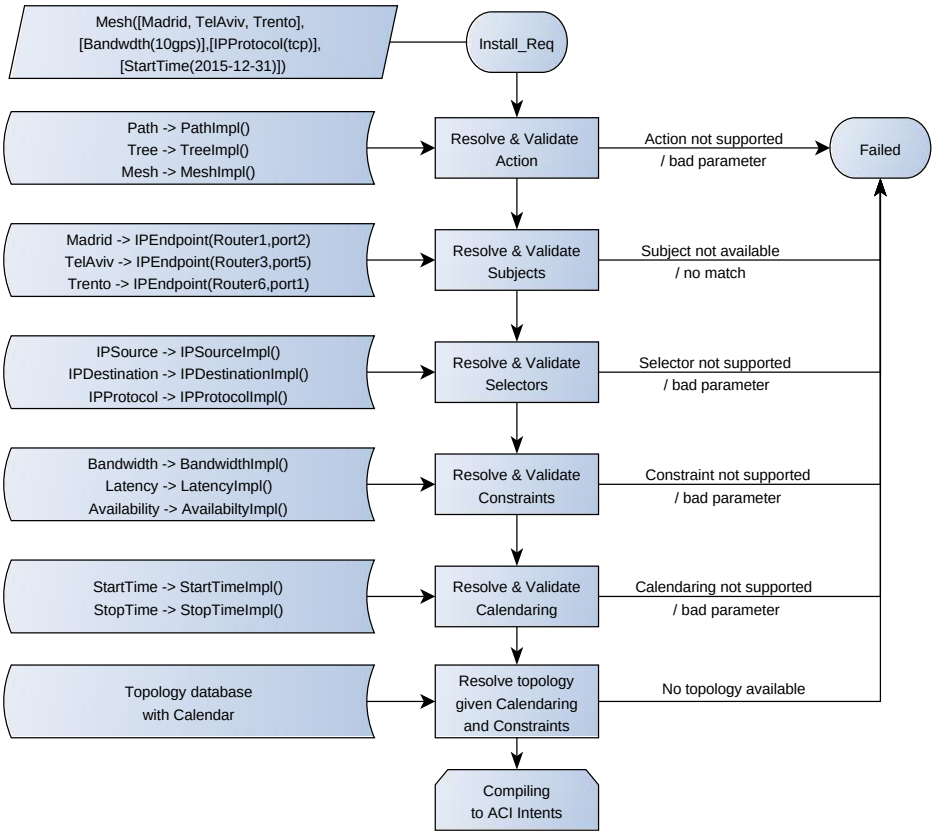

Figure 6.2: Flowchart illustrating the DISMI intent resolution process.

6.2.2 Architecture of an ACI compiler

Each ACI intent needs its own compiler, but compilers can be built by extending other basic intent compilers to reduce duplication. For example, for all intents that can be decomposed into a set of unidirectional paths, the compiler can be built by extending a unidirectional path intent compiler. In fact, the only necessary compiler is the one compiling a uni-directional path, as all other intents can be built from it. In practice, it is expected that ACI compilers will share most of their code.

The generic architecture of an ACI compiler is shown in Figure 6.3. The compiler will use Net2Plan [NET2PLAN] as its Multi-Layer Optimizer (MLO). With respect to the functional architecture presented in Figure 2.1, the MLO can perform computations in case of dynamic provisioning and/or re-optimization, thus operating as the Multi-Layer Provisioning Module and/or the Online Planning Tool Module. The ACI compiler is divided into two main modules: the compilation and the installation modules. The processing of a DISMI intent is as follows:

Compilation module: The role of the compilation module is to encode all the data needed by the MLO into a format that it can decode and decode. The MLO needs the ACI intent that should be installed, the current network topology and the list of all the already installed intents. Two possible solutions are proposed for the encoding format:

- **The ONOS format**: Using the ONOS format has the advantage of reducing the amount of development work: the encoder from ONOS to the MLO is trivial (no encoding, just pack the data together), as is the decoder from the MLO to ONOS. The main drawback of this format is that it is an *ad hoc* solution;
- **The T-API format**: The Transport API format is a standard format, and the implementation of encoders and decoders in ONOS and Net2Plan would bring value to both platforms. However, it requires twice as much development work, and a good understanding of the T-API format. In addition, the format does not cover all our needs. This is both a drawback and an opportunity if we can propose appropriate extensions of the standard.

The choice for the right encoding format is left to Work Package 4 (development), where the practicality of each solution will be further evaluated.

Multi-Layer Optimizer: The MLO decodes the request received and runs an optimization to install the additional ACI intent. The result is sent back to the compilation module as sets of operations to perform in the network (as a sequence of add/remove operations on ONOS intents), each set corresponding to a possible solution. The characteristics of the actual intents that have been found as a solution are also provided with each set.

Intents evaluation and ordering module: The solutions are passed on to the *intents evaluation and ordering* module. If a negotiation is expected, the solution sets are sent to the negotiation module of the DISMI. The characteristics of the solutions allow the negotiation module to build DISMI intents to be presented to the application. The negotiation module of the DISMI sends back the chosen solution. If no

negotiation is to take place, the *intents evaluation and ordering* module uses its internal cost function to order the solutions.

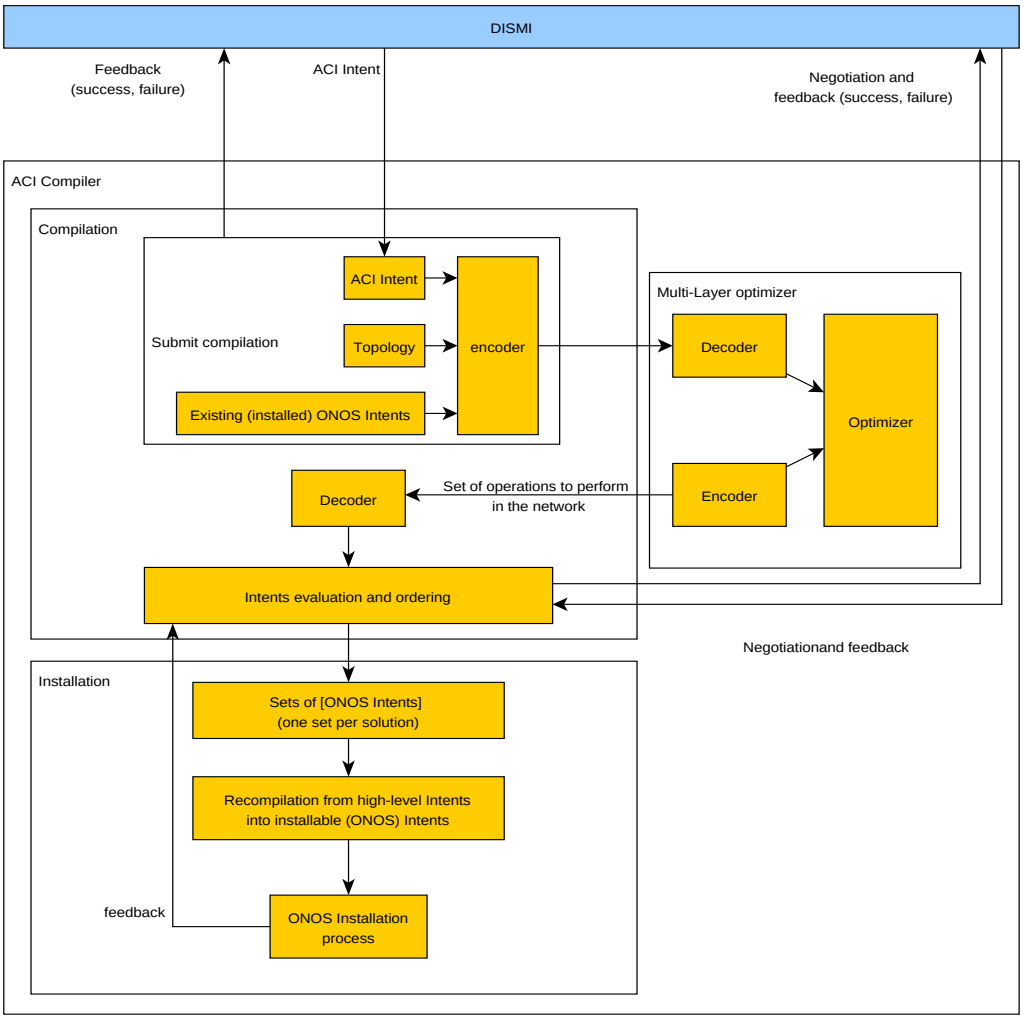

Figure 6.3: Architecture of a generic ACI compiler.

Installation module: The preferred solution is passed on to the installation module as a set of ONOS intents. These intents are recompiled into *installable intents* using ONOS, then sent to ONOS for installation. The success or failure of the installation is reported back to the *intent evaluation and ordering module*. If more solutions are available, then the next one is sent for installation. If the list of solutions has been exhausted (or there was only one), then:

- If the network has changed (other intents have been removed or the topology has changed), then a new compilation of the ACI intent is attempted;
- If the network has not changed, the failure is reported back to the DISMI.

ONOS Intents: Within ONOS, each ONOS intent is associated with a finite state machine that manages its life-cycle. The description of such a process is outside the scope of this document, as the compiler will interact with ONOS intents using the ONOS API only and not by modifying the finite state machine.

6.3 Requesting the status of a DISMI intent

A DISMI intent may be under several states. To keep track of the status of each of them, the orchestrator makes use of the database service of ONOS to save the status and content of each intent, and the relationship between the various objects created during the processing. The state of the DISMI intent can be requested at any time by the application using the DISMI interface (URI /service/{id}/{id}, see Table 20).

6.3.1 Installation of a DISMI intent

When a DISMI intent is processed for installation, it can be in one of the following states:

1. **In the DISMI module** (see Figure 6.1):
    a. **When it enters the DISMI module**, it takes the state DISMI_INTENT_PROCESSING_DISMI;
    b. **At the end of the processing**, a list of ACI intents corresponding to the DISMI intent has been generated. The DISMI intent is in state DISMI_INTENT_PROCESSING_DISMI_DONE;
2. **In the ACI compiler: compilation** (see Figure 6.3):
    a. **When entering the ACI compiler**, The DISMI intent takes the state DISMI_INTENT_COMPILING. The list of ACI intents is sent to the multi-layer optimizer;
    b. **No negotiation:** If the multi-layer optimizer is configured for no negotiation (see discussion in section 5.4), it can return at most one solution:
        i. **Success**: If the compilation succeeded, each ACI intent is returned with an associated sequence of operations on ONOS intents. The DISMI intent takes state DISMI_INTENT_INSTALLABLE;
        ii. **Failure**: If the compilation fails, at least one ACI intent does not have a corresponding sequence of operations on ONOS intents. The DISMI intent goes into state DISMI_INTENT_COMPILATION_FAILED;
    c. **Negotiation:** If the multi-layer optimizer is configured for negotiation (see discussion in section 5.4), it can return zero, one or more solutions;
        i. **Success**: If the compilation succeeded, it returns one or more lists of ACI intents, with their corresponding sequences of operations on ONOS intents. Depending on the configuration of the multi-layer optimizer, each such list of ACI intents may fulfil the criteria of the original DISMI intent or have relaxed constraints. The DISMI intent goes into state DISMI_INTENT_NEGOTIATION;

ii. **Failure**: If one or more of the ACI intents has no solution, the DISMI intent goes into state DISMI_INTENT_COMPILATION_FAILED;

3. **Negotiation and feedback** (see Figure 6.3):

   a. **Failure**: If the compilation failed, the *Intents evaluation and ordering* module informs the *negotiation* module of the DISMI (Figure 6.1). The application will be informed by polling the DISMI (see section 5.3.2). It can then choose to discard the DISMI intent or modify it and resubmit it. All ACI intents are deleted and the DISMI intent re-enters the DISMI with state DISMI_INTENT_PROCESSING_DISMI;

   b. **Negotiation**: In case of negotiation, the *Intents evaluation and ordering* module sends to the *negotiation* module of the DISMI (Figure 6.1) the original DISMI intent with its lists of ACI intents. Each list is converted to a proposed DISMI intent with state DISMI_INTENT_PROPOSAL;

      i. **Cancel**: The application may choose to cancel the service request, and the DISMI intent is deleted;

      ii. **Installation request**: If the application chooses one of the proposed solutions, the original DISMI intent goes to state DISMI_INTENT_INSTALLABLE and the chosen DISMI intent goes to state DISMI_INTENT_NEGOTIATED. The other candidates of the negotiation are deleted. The original DISMI intent is sent back from the *negotiation* module to the *Intents evaluation and ordering* module;

4. **Installation** (see Figure 6.3): The DISMI intent with state DISMI_INTENT_INSTALLABLE is sent by the compilation block of the ACI compiler to the installation block for installation:

   a. **Success**: if the installation succeeds, the original DISMI intent takes state DISMI_INTENT_INSTALLED. Feedback is sent to the DISMI through the *Intents evaluation and ordering* module;

   b. **Failure – recompilation needed**: If the installation fails, the original DISMI intent takes state DISMI_INTENT_RECOMPILING_NEGOTIATED, and the negotiated DISMI intent takes state DISMI_INTENT_NEGOTIATED_RECOMPILING. The ACI compiler tries to recompile the negotiated DISMI intent without negotiation and install it;

      i. **Success**: Upon success of the recompilation, the original DISMI intent takes state DISMI_INTENT_INSTALLABLE and the chosen DISMI intent goes to state DISMI_INTENT_NEGOTIATED. The installation step is run again;

      ii. **Failure**: The negotiated DISMI intent is discarded. If the network state (topology, installed ONOS intents, etc.) has changed, a new compilation is attempted and the original DISMI intent takes state DISMI_INTENT_RECOMPILING. Otherwise, the DISMI intent takes state DISMI_INTENT_FAILED.

6.3.2 Removal of a DISMI intent

When an application uses the DISMI to remove a DISMI intent, it takes state DISMI_INTENT_RETIRING. It is then processed by the DISMI in order to remove all the ONOS intents from the network. Once this is done, the DISMI intent takes state DISMI_INTENT_RETIRED. Keeping the intent in the database allows maintaining some bookkeeping, for example for statistics or billing purposes (even though no billing feature is considered within the project).

6.3.3 Network events

For each ONOS intent that is installed in the network, it is possible to register a *listener*, a worker thread that sends information about events affecting that ONOS intent. The DISMI can therefore be informed by these callbacks of the network failures that affect DISMI intents and use the data stored in the database to find which ONOS intent corresponds to which DISMI intent.

When a network event affects a DISMI intent, the intent takes state DISMI_INTENT_FAILURE. Depending on the type of failure and the recovery strategy, the orchestrator may let the network fix the failure in a way that is transparent to the DISMI intents (for example by setting up a new optical path), or recompile the intent:

1. If the network fixes itself or if the compilation succeeds, the DISMI intent goes back to state DISMI_INTENT_INSTALLED;
2. If one of these steps fails, the DISMI intent takes state DISMI_INTENT_FAILED.

7 Multi-layer Topology and Planning Interface (MLTPI)

The management interface, the Multi-Layer Topology and Planning Interface or MLTPI, is an interface used by a client such as a network planner or a network management system (NMS) to access information about the network. This information includes the network topology, available resources in the topology, active services and reserved resources. After obtaining the relevant information, the client can use the MLTPI to pose questions to the Online Planning Tool, such as what-if questions ("what if this link fails?") or re-optimization questions ("what is the benefit of a re-optimization with regard to energy usage?"). Depending on how the Online Planning Tool answers the question, the client may decide to trigger the implementation of the optimization in the actual network. We see the MLTPI as an extension of the DISMI, i.e. the MLTPI adds a set of additional primitives and methods to the DISMI. However, these are only visible and available to privileged clients of the NBI, depending on their authentication credentials.

When planning the project, an interface like the MLTPI interface seemed the appropriate way of exposing this functionality. However, as the project has progressed, the adoption of the "Net2Plan" network planning tool [NET2PLAN] has highlighted another option. This option would be to expose the same functionality through the existing Net2Plan graphical user interface, providing similar functionality at a lower implementation cost. While this option is likely to be the one chosen in WP4, we include here the design of the MLTPI interface for reference and to highlight the required functionality.

7.1 Re-optimization

One of the main functionalities provided by the On-line planning tool is to re-optimize the allocation of resources in the underlying network. Here we envision two different types of optimizations: optimization on a set of intents, and optimizations on the whole network resource allocation.

Intents optimization selects one or more installed intents and re-optimizes their network resource allocations based on a number of different goals, such as SLA compliance, energy use, monetary cost, and spreading or packing of network load. Information about intents in the network can be retrieved using the DISMI methods.

Network optimization, on the other hand, looks at all the allocated network resources and re-optimizes them with regards to a network global goal, such as the average link utilization percentage.

The process of requesting an optimization is very similar to the negotiation of an intent, see Figure 7.1. In step 1) a re-optimization is requested by a client of the MLTPI, which tells the Online Planning Tool to perform the required calculations. In step 2) the results of the optimization, likely a set of statistics of expected gains / losses, are presented to the requester. The calculation may be a time-consuming process, which therefore requires asynchronous notification to the client when the results are ready. In step 3), once the results are ready, the client may decide whether to actually implement the optimization in the network or to discard the results and not realize the optimization. Finally, in step 4), the MLTPI replies with the status of the execution of the optimization, or whether it was successfully discarded. To support re-

optimization, we add re-optimization actions to the list of connectivity actions described in section 4.2 above. Additional optimization primitives are listed in Table 21, Table 22 and Table 23.

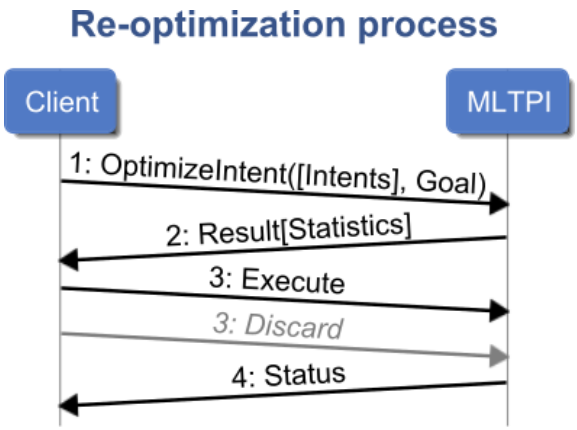

Figure 7.1: Re-optimization process, note the similarity with Figure 5.2 that describes the negotiation process.

The goals of an intent optimization can be summarized as:

- **SLA compliance**: Check whether the included intents are currently meeting their constraints and if not, try to find a new allocation where the constraints are met;
- **Energy use:** Try to allocate network resources for the included intents in such a manner that the energy requirements are reduced, e.g. by preferring optical connectivity over packet routers;
- **Monetary cost**: Try to allocate network resources for the included intents in order to reduce the monetary cost of the connection, e.g. by avoiding leased connections;
- **Load spreading:** Try to allocate network resources for the included intents in order to spread the network load over the available links and nodes;
- **Load packing**: Try to allocate network resources for the included intents in order to concentrate the network load on a subset of available links and nodes, e.g. in order to create larger pools of unused resources.

Similarly, the goals for network wide optimizations are:

- **Link utilization**: Try to allocate network resources in a manner that equalizes link utilization in the network. This is the network wide version of the **Load spreading** goal for intents;
- **Link release**: Try to allocate network resources in a manner that maximizes utilization of a subset of links in the network. This is the network wide version of the **Load packing** goal for intents;

- **Monetary cost**: Try to allocate network resources in a manner that minimizes cost for the operator. This is the network wide version of the **Monetary cost** goal for intents;
- **Energy use:** Try to allocate network resources in a manner that minimizes the energy requirements for the network, this is equivalent to the **Energy use** goal for intents;
- **SLA compliance**: Try to allocate resources to better meet SLA compliance for the active intents in the network.

The results of these optimizations are network statistics, showing what would change if the optimization was actually applied to the allocated network resources e.g.:

- Network utilization;
- Availability, failure probability, average and/or per link;
- Blocking probability, average and/or per link;
- Packet drop probability, average and/or per link.

As these statistics depend on the detailed implementation of the optimization algorithms, we do not attempt to define the response object in detail but rather leave a placeholder that can be used to return relevant information, as determined by the algorithm (Table 24).

Table 21: Intent re-optimization primitives.

Primitive	**Parameters**	**Required**
OptimizeIntent		Yes
	List of intents	Yes
	Intent optimization goal	Yes
OptimizeNetwork		Yes
	Network optimization goal	Yes

Table 22: OptimizeIntent goal primitives.

Primitive	**Parameters**	**Required**
SLA compliance	--	Yes
Energy	--	Yes
Cost	--	Yes

Spread	--	Yes
Pack	--	Yes

Table 23: OptimizeNetwork goal primitives.

Primitive	**Parameters**	**Required**
SLA compliance	--	Yes
Energy	--	Yes
Cost	--	Yes
Link utilization	--	Yes
Link release	--	Yes

Table 24: Response statistics primitives.

Primitive	**Parameters**	**Required**
Statistics		Yes
	Collection of statistics objects	Yes

7.2 What-if questions

What-if questions allow the client to ask the planner hypothetical questions relating to different network resources, and get a response as to what would be the consequence of the hypothetical situation. The hypothetical situations can be the addition of links or nodes to the network, unavailability of links or nodes (due to failures or administrative actions), and triggering of resiliency mechanism.

7.2.1 What if resources fail?

These questions, sent to the MLTPI using the primitive shown in Table 25, allow the client to simulate failures or administrative actions on existing links and nodes in the network. The answer to these questions is a collection of consequences for active intents, caused by the simulated failure. An answer may also include statistics of overall network performance, showing the wider impact of a failure on the system. The list of response primitives is shown in Table 26.

A list of consequences could be, for example, "Failed: [Intent A, B, C], Recovered: [Optically: [Intent D, E, F], MPLS-FRR: [Intent G, H]]". In this example intents A, B, and C would fail as a consequence of the network failure, while intents D to H would recover, either through optical protection (D, E, F) or through MPLS Fast-

Reroute (G, H). As the calculation of these consequences may be time consuming, like the re-optimization, it requires asynchronous notifications.

There is usually no one guaranteed state for a failure. In particular, the list of consequences depends on the algorithm used to re-optimize the intents and in which order they are treated, and possibly on statistics gathered by monitoring the network. As the same multi-layer optimizer is used to run simulations and to perform actual recovery actions, monitoring statistics (on available bandwidth, jitter, etc.) are the main probable source of discrepancy between the result of a *What if* simulation and a real recovery.

Table 25: Primitive for resource failure simulations.

Primitive	**Parameters**	**Required**
ResourceFailure	List of Links and Nodes	Yes

Table 26: Response primitives used to answer the resource failure question.

Primitive	**Parameters**	**Required**
Consequences	List of Failed-/RecoveredIntent	Yes
FailedIntent	Intent identifier	Yes
RecoveredIntent	Intent identifier, mechanism	Yes
Statistics	Undefined	No

7.2.2 What if additional resources are available?

This question allows the client to simulate the consequences of introducing new links and/or nodes to the network, for the network performance as a whole and for inactive intents (that the orchestrator has so far not been able to accommodate). The response to this question is the same as for the resource failure: a list of primitives describing consequences and possibly network statistics, as shown in Table 26.

To perform such a request, a new primitive is necessary that allows augmenting the existing topology with additional links and/or nodes, with relevant parameters such as bandwidth, distance, node types, etc., and triggering a calculation. This can be done with the *AdditionalResources* primitive, containing a list of new links and nodes, with relevant parameters, as shown in Table 27. As these parameters depend on the exact model of the topology, they will not be elaborated here.

Table 27: Primitive that allows adding resources (links, nodes) to an existing topology.

Primitive	Parameters	Required
AdditionalResources	List of additional Links, Nodes and Interfaces	Yes

7.3 RESTful HTTP API for MLTPI

With the primitives defined in sections 7.1 and 7.2, a RESTful API can be constructed for asking the questions, obtaining results, and discarding results / cancelling calculations. The MLTPI API is shown in Table 28. The various primitives for optimizations and questions are sent as JSON objects in the body of the corresponding HTTP methods.

Table 28: MLTPI RESTful HTTP API.

Method + URI	Functionality
POST /optimization/{optId}	Start a new optimization with identifier optId
GET /optimization/	Get a list of performed / running optimizations and their status
GET /optimization/{optId}	Get status and results of an optimization with identifier optId
DELETE /optimization/{optId}	Delete the results of an optimization or cancel the calculation
POST /optimization/install/{optId}	Trigger the installation of an optimization into the underlying network
GET /optimization/install/	Get a list of optimization(s) being installed, and their status
GET/optimization/install/{optId}	Get the details about an optimization being installed or done installing
DELETE /optimization/install/{optId}	Delete the status of an optimization, or cancel the installation if ongoing
POST /question/{questId}	Start a new what-if question with identifier questId
GET /question/	Get a list of questions and their status
GET /question/{questId}	Get status and results and status of a question

DELETE /question/{questId}	Delete the results or cancel a question

7.4 Assisting functionality

Several of the previously mentioned primitives require knowledge of the network topology as well as the committed *Intents*, *ConnectionPoint*/*EndPoint* mappings. This information can be obtained through either the native ONOS interfaces (topology) or through the DISMI (intents, ConnectionPoints/EndPoints, see section 5.3.1).

The ONOS interfaces are documented in [ONOSRESTAPI], and the relevant ONOS API commands are shown in Table 29.

Table 29: Parts of the ONOS REST API relevant to MLTPI, from [ONOSRESTAPI].

Method + URI	**Functionality**
GET /devices/	Lists all infrastructure devices.
GET /devices/{deviceID}	Lists details of a specific infrastructure device
GET /devices/{deviceID}/ports	Lists ports of a specific infrastructure device
GET /links/	Lists all infrastructure links
GET /links?{device=deviceId}{port=portNumber} {direction=ALL,INGRESS,EGRESS}	Lists details of a link
GET /topology	Gets overview of the current topology
GET /topology/clusters	Gets list of topology clusters overview
GET /topology/clusters/{clusterId}	Gets overview of a specific topology cluster
GET /topology/clusters/{clusterId}/devices	Get infrastructure devices that belong to the specified topology cluster
GET /topology/clusters/{clusterId}/links	Gets infrastructure links that belong to the specified topology cluster

8 Programmability elements

The ACINO orchestrator is developed on top of the ONOS controller. To support the DISMI and MLTPI, some of the existing ONOS components need modifications while some new ones need to be introduced. Figure 8.1 shows these components on an overview of the ACINO orchestrator. Figure 8.2 shows them on a more detailed view of the ONOS controller.

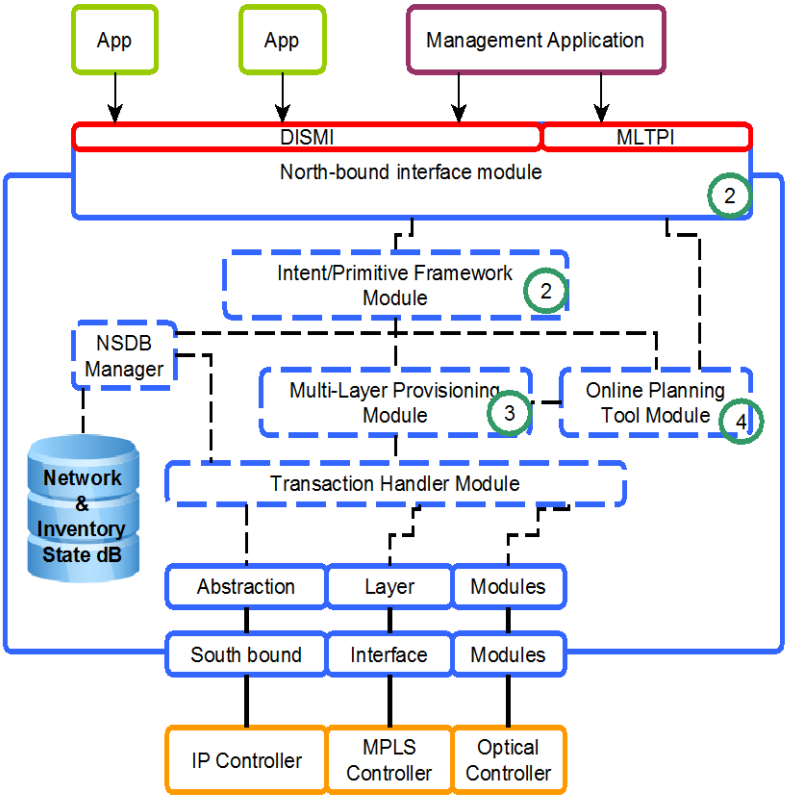

Figure 8.1: The ACINO orchestrator with programmability elements that need modifications numbered from 1 to 4.

These elements are presented below:

1. The DISMI and MLTPI interfaces have to be implemented alongside the existing northbound interfaces. As there is currently an existing REST interface, re-using the existing infrastructure should simplify this step;

2. The intent framework needs to be modified, in particular the path calculation. Currently the ONOS intent framework calculates paths in parallel with the intent verification. This will be modified to first verify the intent, then generate potential solutions that are delivered to the Multi-Layer Provisioning Module;

3. The Multi-Layer Provisioning Module needs access to the Network State and Inventory State Database, together with the ability for triggering the implementation of calculated solutions;

4. In order to integrate Net2Plan as the Online Planning Tool, modules to import and export data from the Network State and Inventory State Database are needed. Additional modules to relay MLTPI questions and answers to and from the Online Planning Tool are also needed, together with mechanisms for communicating the results.

The specific implementation details will be discussed and analysed in WP4.

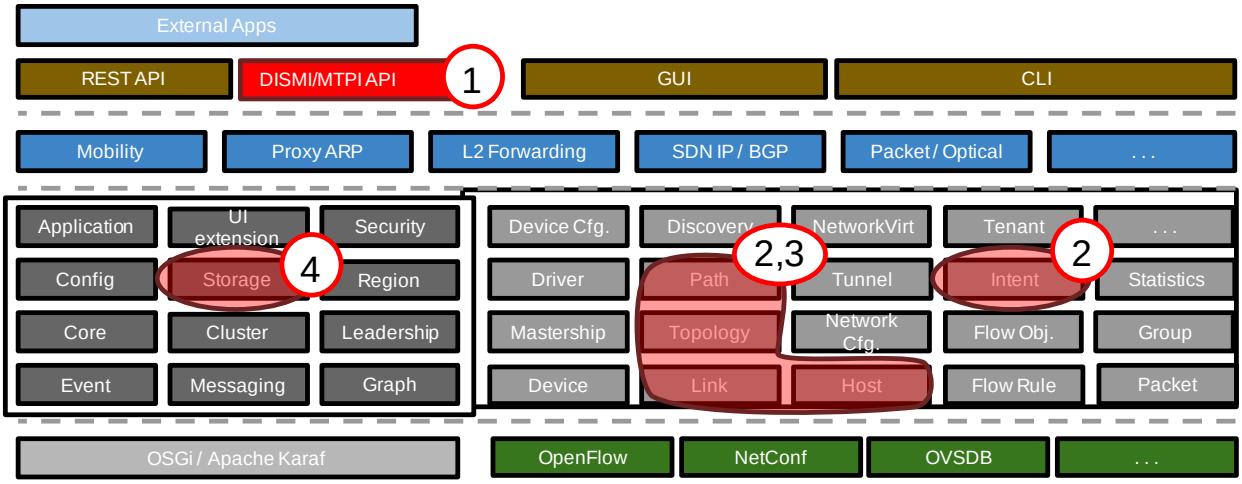

Figure 8.2: ONOS architecture with areas needing modification to support DISMI/MLTPI highlighted. From ONOS Developer Workshop 2015, slide 7.

https://docs.google.com/presentation/d/1iWrZ5JxQnQi7Q5OdMu4DxaXnmLxSm7fYIfHx8UpenQE/present?ueb=true#slide=id.gb34258f82_0_22

9 Conclusions

This deliverable has presented the work performed to design the northbound interface of the ACINO orchestrator. The work started with the requirements placed on the northbound interface by previous deliverables (D2.1 and D.3.1).

We decided to apply the intent paradigm for the design of this interface, in order to make it portable, easy to use, and technology agnostic. For these reasons, we evaluated the state of the art regarding intent-based interfaces. Our conclusions from the state of the art is that while there is no formal definition of exactly what constitutes an intent interface, there are common features of them all, namely to decouple *what* the users want from *how* to achieve it.

We have defined two northbound interfaces: the Dynamic Intent-driven Service Management Interface (DISMI) and the Multi-Layer Topology and Planning Interface (MLTPI).

DISMI exposes a number of primitives and a grammar showing how primitives can be combined in order to formulate intent(s), which then can be used to request services. The definition of these primitives has been based on both the requirements on the interface as well as what can be provided by common networking equipment. This has led to a generic interface, applicable on a wide array of underlying network technologies. To use the data- and information-model designed for DISMI we have defined a RESTful HTTP API, and described the means needed for making it auto-discoverable, using common web technologies such as JSON schemas.

We have discussed the negotiation phase, and its different variants, triggered when an application wants more control over the resource allocation, or when the requested service cannot be provided. We then have presented the architecture of the intent framework, which implements intent resolution and validity checking of intents, and the intent compilers, which decompose intents into network requirements.

We have also defined the MLTPI, an extension of the DISMI for planning operations such as *what-if* questions (“what if this link fails?”) or re-optimization questions (“what is the benefit of a re-optimization with regard to energy usage?”).

Finally, summarizing the presented design, four areas are identified in the ONOS network controller used as the base for the ACINO orchestrator, where programmability elements need to be modified or introduced to implement the described functionalities.

List of abbreviations and acronyms

Abbreviation	**Meaning**
ACL	Access Control List
API	Application Programming Interface
ARP	Address Resolution Protocol
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DCCP	The Datagram Congestion Control Protocol
DDOS	Distributed Denial of Service
DISMI	Dynamic Intent-Driven Service Management Interface
DSCP	Differentiated Services Code Point
eNodeB	Evolved Node B
FBF	Filter-Based Forwarding
FTP	File Transfer Protocol
GPRS	General Packet Radio Service
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	The JavaScript Object Notation format
MLTPI	Multi-Layer Topology and Planning Interface
MPLS	Multiprotocol Label Switching
NETCONF	The Network Configuration Protocol

NMS	Network Management System
ONF	Open Networking Foundation
ONOS	The Open Network Operating System
OSGi	The Open Service Gateway Initiative
OSPF	Open Shortest Path First
OSS	Open Source Software
PBR	Policy-Based Routing
QoS	Quality of service
RFC	Request For Comments
RPDB	The Linux Routing Policy DataBase
RSVP	Resource Reservation Protocol
SCTP	The Stream Control Transmission Protocol
SDN	Software-Defined Network
SMTP	Simple Mail Transfer Protocol
T-API	Transport-API
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
YANG	The “Yet Another Next Generation” data modelling language

References

[ACID21] ACINO deliverable D2.1 “Initial report on network architecture, use cases and application requirements”, Feb. 2016.

[ACID31] ACINO deliverable D3.1 “The framework for the application-centric network orchestrator”, Feb. 2016.

[ACID41] ACINO deliverable D4.1 “First implementation activities of the orchestrator”, July 2016.

[ADVAFSP] The ADVA FSP 3000 optical network encryption. Retrieved on July 28, 2016.

http://www.advaoptical.com/~/media/Resources/Data%20Sheets/FSP%203000%20Optical%20Network%20Encryption.ashx

[APACHE] Website of the Apache software foundation. Retrieved on August 2, 2016.

https://www.apache.org

[ASPEN] Website of the Aspen project. Retrieved on July 28, 2016.

http://opensourcesdn.org/projects/project-aspen-real-time-media-interface-specification/

[ASPENGIT] GIT repository of the Aspen project. Retrieved on July 28, 2016.

https://github.com/OpenNetworkingFoundation/ASPEN-Real-Time-Media-Interface

[BISMUK] Biswanath Mukherjee, ”Optical WDM networks”, Springer Science & Business Media (2006).

[BOULDER] Website of the Boulder project. Retrieved on July 28, 2016.

http://opensourcesdn.org/projects/project-boulder-intent-northbound-interface-nbi/

[BOULDERDASH] The Boulder project dash board. Retrieved on July 28, 2016.

https://community.opensourcesdn.org/wg/IntentNBI/dashboard

[BOULDERGIT] GIT repository of the Boulder project. Retrieved on July 28, 2016.

https://github.com/OpenNetworkingFoundation/BOULDER-Intent-NBI

[BOULDERPRES] Presentation of the Boulder project. Retrieved on July 28, 2016.

https://www.opennetworking.org/images/stories/sdn-solution-showcase/germany2015/Boulder%20-%20Intent%20Based%20NBI.pdf

[CISCIOS122] Solie, Karl, Lynch and Leah, “CIEE practical studies”, Cisco Press, vol. 2 (2003).

[DOVE1] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain, “An intent-based approach for network virtualization,” in 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pp. 42-50 (2013).

[DOVE2] R. Recio, “Distributed Overlay Virtual Ethernet (DOVE) Networks,” PowerPoint Presentation, pp. 1–27 (2012). Retrieved July 28, 2016.

www.ethernetsummit.com/English/Collaterals/Proceedings/2012/20120222_2-103_Recio.pdf

[DOVEAPI] API of the OpenDayLight DOVE module. Retrieved on July 28, 2016.

https://wiki.opendaylight.org/view/Open_DOVE:API

[DOVEREST] REST API of the OpenDayLight DOVE module. Retrieved on July 28, 2016.

https://jenkins.opendaylight.org/opendove/job/opendove-merge/lastSuccessfulBuild/artifact/odmc/rest/northbound/target/enunciate/build/docs/rest/index.html

[INSERT] N. Blum, S. Dutkowski, T. Magedanz, "InSeRt: An Intent-Based Service Request API for Service Exposure in Next Generation Networks." 32nd Annual IEEE Software Engineering Workshop (SEW'08), (2008).

[JSON] Website dedicated to JSON schema. Retrieved on July 28, 2016.

Jsonschema.org

[JUNOS] Configuration example of the Juniper operating system JUNOS. Retrieved on July 28, 2016.

http://www.juniper.net/techpubs/en_US/junos14.2/topics/example/firewall-filter-stateless-example-trusted-source-block-telnet-and-ssh-access.html

[KARAF] Website of the Apache Karaf OSGi container system. Retrieved on August 2, 2016.

http://karaf.apache.org/

[LOJBAN] The Logical Language Group Inc, “Lojban Machine Grammar” (1997). Retrieved on July 28, 2016.

http://www.lojban.org/publications/formal-grammars/bnf.300.txt

[MEFBW] The Metro Ethernet Forum (MEF) bandwidth profile definitions. Retrieved on July 28, 2016.

https://www.mef.net/Assets/White_Papers/Bandwidth-Profiles-for-Ethernet-Services.pdf

[METISD11] METIS project deliverable D1.1, “Scenarios, requirements and KPIs for 5G mobile and wireless system”.

[NET2PLAN] Website of the open source network planner Net2Plan. Retrieved on August 2, 2016

http://net2plan.com/

[NETFILTER] The Linux Documentation Project, “Netfilter and IP Tables”, The Linux Network Administrators guide, section 9.8. Retrieved on July 28, 2016.

http://www.tldp.org/LDP/nag2/x-087-2-firewall.future.html

[NETMARK] The Linux Documentation Project, “Netfilter & iproute - marking packets”, Linux Advanced Routing & Traffic Control HOWTO, Chapter 11. Retrieved on July 28, 2016.

http://www.tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.netfilter.html

[ODL] Website of the OpenDayLight project. Retrieved on July 28, 2016.

https://www.opendaylight.org/

[ODLNIC] Website of the OpenDayLight Network Intent Composition project. Retrieved on July 28, 2016.

https://wiki.opendaylight.org/view/Network_Intent_Composition:Main

[OFLOW10] The OpenFlow 1.0.0 specification. Retrieved on July 28, 2016.

https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf

[OFLOW13] The OpenFlow 1.3.0 specification. Retrieved on July 28, 2016.

https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[ONFINT] Open Networking Foundation webinar about Intent, Feb. 2015. Retrieved on July 28, 2016.

https://www.sdxcentral.com/resources/nfv-sdn-training-sdnuniversity-archives/intent-driven-networking-onf-webinar/

[ONFRTM] ONF real-time media northbound interface specification. Retrieved on July 28, 2016.

https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/Real_Time_Media_NBI_REST_Specification.pdf

[ONOS] Website of the ONOS project. Retrieved on July 28, 2016.

http://onosproject.org/

[ONOSINT] Documentation about the intent framework in ONOS. Retrieved on July 28, 2016.

https://wiki.onosproject.org/display/ONOS/Intent+Framework

[ONOSRESTAPI] Wiki-page documenting the ONOS REST API, "Appendix B: REST API (Draft)-ONOS - Wiki". Retrieved on August 1, 2016.

https://wiki.onosproject.org/pages/viewpage.action?pageId=1048699.

[OPENSOA] FOKUS Open SOA Telco Playground; Link from [INSERT]. Unavailable on August 8, 2016

http://www.opensoaplayground.org

[OSGI] Website of the Open Service Gateway Initiative (OSGi). Retrieved on August 2, 2016.

https://www.osgi.org/

[PBB] “Understanding PBB”. Retrieved on July 28, 2016.

https://sites.google.com/site/amitsciscozone/home/pbb/understanding-pbb

[PYPARSING] Home page of the Python Pyparsing module. Retrieved on July 28, 2016.

http://pyparsing.wikispaces.com/

[RESTASYNC] Discussion on website Stackoverflow about how to handle asynchronous operations in REST. Retrieved on July 28, 2016.

http://stackoverflow.com/questions/16214244/how-to%20-handle-asynchronous-operations-in-rest

[RESTASYNC2] Discussion about how to handle long running job operations in REST. Retrieved on July 28, 2016.

http://farazdagi.com/blog/2014/rest-long-running-jobs/

[RESTFUL] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000

[RFC2210] J. Wroclawski, “The Use of RSVP with IETF Integrated Services”, Internet Requests for Comments, RFC Editor, RFC 2210, September 1997. Retrieved on July 28, 2016.

http://www.rfc-editor.org/rfc/rfc2210.txt.

[RFC2215] S. Shenker and J. Wroclawski, “General Characterization Parameters for Integrated Service Network Elements”, Internet Requests for Comments, RFC Editor, RFC 2215

(1997). Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc2215

[RFC4303] S. Kent, “IP Encapsulating Security Payload (ESP)”, Internet Requests for Comments, RFC Editor, RFC 4303, December 2005. Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc4303

[RFC4960] R. Stewart, “Stream Control Transmission Protocol”, Internet Requests for Comments, RFC Editor, IETF RFC 4960 (2007). Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc4960

[RFC5234] D. Crocker, P. Overell, “Augmented BNF for Syntax Specifications: ABNF”, IETF RFC 5234, January 2008. Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc5234

[RFC6003] D. Papadimitriou, “Ethernet Traffic Parameters”, Internet Requests for Comments, RFC Editor, RFC 6003, October 2010. Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc6003

[RFC6020] M. Bjorklund, “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)”, IETF RFC 6020, October 2010. Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc6020

[RFC7231] R. Fielding, J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content”, section 6.3.2, Internet Requests for Comments, RFC Editor, RFC 7231, June 2014. Retrieved on July 28, 2016.

https://tools.ietf.org/html/rfc7231#section-6.3.2

[RFC6455] I. Fette, A. Melnikov, “The WebSocket Protocol”, Internet Requests for Comments, RFC Editors, RFC6455, December 2011. Retrieved on August 19th, 2016.

https://tools.ietf.org/html/rfc6455

[RMQCQ] Website of the RabbitMQ messaging system, Tutorial to implement Remote Procedure Calls (RPC) to implement a callback queue. Retrieved on July 28, 2016.

https://www.rabbitmq.com/tutorials/tutorial-six-python.html

[RPDB] The Linux Routing Policy DataBase. Retrieved July 28, 2016.

http://www.tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.rpdb.html

[WEBSOCKET] Website if the Websocket communication system, Retrieved on August 19th, 2016.

https://www.websocket.org

[WPPROT] White paper, “Carrier-Class Protection and Restoration for Ethernet MANs”, Metrobility Optical Systems. Retrieved on July 28, 2016.

http://www.techdata.com/content/tdenterprise/whitepapers/03Aug_MetrobilityOptical_CarrierClass.pdf

10 Appendix

10.1 List of the DISMI primitives

Table 30 below lists all the required and optional primitives discussed in this document. Text in bold indicates a category of primitives. For primitives, the column “required” indicates whether implementing the primitive in the DISMI is mandatory. For primitives taking parameters, the “primitive and “parameter” cells are merged and the parameters are listed in separate lines. For each parameter, the column “required” states whether the parameter is mandatory when using/creating the primitive. For primitive without parameter, the parameter field is marked as “—“.

Table 30: List of the DISMI primitives.

Primitive	**Parameter**	**Required**
Service		Yes
	List(Intent)	Yes
Intent		Yes
	Action	Yes
	List(Selector)	No
	List(Constraint)	No
	List(Calendaring)	No
	List(Priority)	No
Noun		Yes
	ConnectionPoint	Yes
	List(Selector)	No
	List(Constraint)	No
ConnectionPoint		Yes
	Unique name	Yes
EndPoint		
IPEndPoint		Yes

Primitive	Parameter	Required
	Router-id	Yes
	IPAddress	No
	Port-id	No
	Subnets	No
EthEndPoint		No
	Switch-id	Yes
	MAC address	No
	Port-id	No
FiberEndPoint		No
	Switch-id	Yes
	Port-id	No
LambdaEndpoint		No
	Switch-id,	Yes
	Port-id	No
	Lambda Frequency	No
	Width	No
Actions		
Path		Yes
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Connection		No
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Multicast		No
	ConnectionPoints	Yes

Primitive	Parameter	Required
	Selectors, Constraints, Priorities	No
Aggregate		No
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Tree		
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Mesh		
	ConnectionPoints	Yes
	Selectors, Constraints, Priorities	No
Selectors – Internet protocol		
IPSource		Yes
	IPv4/6 address, mask	Yes
IPDestination		Yes
	IPv4/6 address, mask	Yes
IPProtocol		Yes
	protocol number/enumerator	Yes
IPToS		Yes
	Type-of-Service	Yes
IPDSCP		Yes
	Differentiated Services Code point	Yes
Selectors – Ethernet		
EthSource		Yes
	MAC address	Yes
EthDestination		Yes

Primitive	Parameter	Required
	MAC address	Yes
EthType		Yes
	Ethernet protocol type	Yes
VLAN		Yes
	VLAN identifier	Yes
VLANPCP		Yes
	VLAN priority number	Yes
Selectors – Lambda		
LambdaFrequency		No
	Channel centre frequency	Yes
LambdaWidth		No
	Width of channel	Yes
Selectors – GPRS Tunnelling protocol		
GTPTEID		No
	Tunnel identifier	Yes
Constraints		
Bandwidth		Yes
	Bitrate	Yes
Latency		Yes
	Time	Yes
Jitter		Yes
	Time	Yes
MTTR		Yes
	Time	Yes
MTBF		Yes

Primitive	Parameter	Required
	Time	Yes
Availability		No
	Percentage	Yes
SurvivableDowntime		No
	String: “none”, “low”, ”high”	Yes
Encryption		Yes
	Strength	No
Sensitive	--	No
Integrity		No
	Strength	No
Priorities		
Availability	--	No
Cost	--	No
Latency	--	No
Jitter	--	No
Bandwidth	--	No
Calendaring		
StartTime		Yes
	Datetime	Yes
StopTime		Yes
	Datetime	Yes
CostLimit		No
	Cost/h	Yes

10.2 List of the MLTPI primitives

Table 31 below lists all the required and optional primitives discussed in this document. Text in bold indicates a category of primitives. For primitives, the column “required” indicates whether implementing the primitive in the DISMI is mandatory. For primitives taking parameters, the “primitive and “parameter” cells are merged and the parameters are listed in separate lines. For each parameter, the column “required” states whether the parameter is mandatory when using/creating the primitive. For primitive without parameter, the parameter field is marked as “--“.

Table 31: List of the MLTPI primitives

Primitive	**Parameter**	**Required**
Intent re-optimization		
OptimizeIntent		Yes
	List of intents	Yes
	Intent optimization goal	Yes
OptimizeNetwork		Yes
	Network optimization goal	Yes
OptimizeIntent goals		
SLA compliance	--	Yes
Energy	--	Yes
Cost	--	Yes
Spread	--	Yes
Pack	--	Yes
OptimizeNetwork goals		
SLA compliance	--	Yes
Energy	--	Yes
Cost	--	Yes
Link utilization	--	Yes
Link release	--	Yes

Response statistics		
Collection of statistics objects		Yes
	Collection of statistics objects	Yes

10.3 Access Control List configuration for Cisco

```
!Create ACL for source subnet 172.16.100.0/24
access-list 5 permit 172.16.100 0.0.0.255
!Create ACL for source subnet 172.16.100.0/24, application ftp
access-list 10 permit 172.16.100 0.0.0.255 eq ftp any
!Create ACL for source subnet 172.16.100.0/24, application ftp-data
access-list 15 permit tcp 172.16.100 0.0.0.255 eq ftp-data
!Create ACL for any TCP traffic to 192.168.1.100 with port greater than 1024
access-list 20 permit tcp any host 192.168.1.100 gt 1024

!Create a route map "set_tag10", permit traffic, with priority 100
!Tags matching traffic with route tag 10
route-map set_tag10 permit 100
 !Match IP addresses using ACL 5
 match ip address 5
 set tag 10
! Forward FTP traffic to 172.16.4.3
route-map forward_ftp permit 101
 !Match IP addresses using ACL 10
 match ip address 10
 !Forward to 172.16.4.3
 set ip next-hop 172.16.4.3
! Forward large packets to 172.16.4.6
route-map bulk permit 102
 !Match large packets
 match length 1000 1600
 !Forward to 172.16.4.6
 set ip next-hop 172.16.4.6
! Forward small packets to 172.16.4.6 after setting their ToS and precedence
bits
route-map live permit 102
 !Match small packets
 match length 1 1000
 !Forward to 172.16.4.6
 set ip next-hop 172.16.4.6
 !Set IP TOS
 set ip tos 10
 !Set IP Precedence
 set ip precedence priority
```

10.4 PyParsing grammar and output

10.4.1 Code

```
lfrom = Literal('from')('ingress')
to = Literal('to')('egress')
lwith = Literal('with').suppress()
number = Word(nums+'.')
connectionPoint = Group((to|lfrom)+Word(alphanums)('CP'))('connectionPoint')
bwunit=(Literal('tbps')|Literal('gbps')|Literal('mbps')('unit')
microsec = Literal('µs')|Literal('microsec')|Literal('microseconds')
millisec = Literal('ms')|Literal('millisec')|Literal('milliseconds')
timeunit = (microsec|millisec)('unit')
latency = Group((number('value') + timeunit('unit')
          + Literal('latency').suppress()) | Literal('latency').suppress()
          + number('value') + timeunit('unit')))('latency')
bandwidth = Group((Literal('bandwidth').suppress()
            + Group(number('value') + bwunit))| (number('value') + bwunit
            + Literal('bandwidth').suppress()))('bandwidth')
action = (Literal('Path')|Literal('Connection'))('action')
constraints = Group((lwith + OneOrMore((bandwidth|latency)
              + Optional(Literal('and').suppress()))))('constraints')
params = Group(connectionPoint + connectionPoint)('connectionPoints')
intent = (action + pathParams + constraints)
self.intent = intent('intent')
```

10.4.2 Output

```
<intent>
  <action>Path</action>
  <connectionPoints>
    <connectionPoint>
      <ingress>from</ingress>
      <CP>office</CP>
    </connectionPoint>
    <connectionPoint>
      <egress>to</egress>
      <CP>internet</CP>
    </connectionPoint>
  </connectionPoints>
  <constraints>
    <bandwidth>
      <value>
        <value>10</value>
        <unit>gbps</unit>
      </value>
    </bandwidth>
    <latency>
      <value>10</value>
      <unit>ms</unit>
    </latency>
```

```
    </constraints>
  </intent>
```