Document Properties

Document Number:	D 5.3
Document Title:	**Use Case Implementation : 1st release**
Document Responsible:	Kévin Phemius (THALES)
Document Editor:	Kévin Phemius (THALES)
Authors:	Kévin Phemius (THALES) & Diego López (TID) Pedro A. Aranda (TID) & Bernhard Schräder (FTS)
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	1.0

Production Properties:

Reviewers:	Elisa Rojas (Telcaria) & Alexander Leckey (Intel) & Elio Salvadori (Create-Net)

Document History:

Version	Date	Issued by	
1.00	30-06-2015	Kévin Phemius	

Disclaimer:

This document has been produced in the context of the NetIDE Project. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2010–2013) under grant agreement n° 619543.

Abstract:

This deliverable presents the first iteration of NetIDE 's Use Case implementation and evaluation specified in the Work Package 5 (Use Cases and Evaluation). This work aims at presenting the first implementation of NetIDE 's three Use Cases with the current version of NetIDE 's tools and architecture. We will present how the three Use Cases were implemented and tested and how the different challenges that NetIDE has to solve, presented by the industrial partners as Use Case Requirements and metrics, were resolved or not. The deliverable exposes our approach to test how the current version of NetIDE , still a work in progress, holds to the commitments done at the beginning of the project and in which direction it's going. We also try to provide insight that could be helpful throughout the next iteration of the project.

Keywords:

clouds, networking, Information-centric, future internet, use cases, SDN, data centre, network applications

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

DC Datacentre

DMZ Demilitarised Zone

DNS Domain Name Service

ETSI European Telecommunications Standards Institute

ICMP Internet Control Message Protocol

IDE Integrated Development Environment

ISG Industry Specification Group

MAPE-K Monitor, Analysis, Plan, Execution, Knowledge w

NFV Network Functions Visualization

OB Operations & Business unit

ODL OpenDaylight

PoC Proof of concept

POP Point of presence

QA quality assurance

SDN Software Defined Networking

SVN Subversion

TCO Total cost of ownership

UC Use Case

UCR Use Case Requirement

vDC virtual Datacentre

VM Virtual Machine

Executive Summary

This deliverable presents the implementation and evaluation of the NetIDE project's Use Cases: *Vendor Agnostic Datacentre Evolution*, *Integrated IT System* and *Hybrid Environment Application Development*. The evaluation focuses on a preliminary version of NetIDE, so we expect some of the components to be either incomplete or absent. Nonetheless, we took into account these missing elements in our reports and provided some insight to guide the future work of the project. Similarly, some of the tests planned in Deliverable D 5.2 [1] were not completed but will be in a future deliverable. With these parameter in mind, we present in this deliverable the advances of the NetIDE project in terms of architecture, tools and respect of the Use Case Requirements. We aim to show how close is the project to its original objectives, leveraging on the specific needs each Use Case raises against NetIDE overall value proposition.

1 Introduction

This deliverable presents the Use Cases implementation and evaluation of the NetIDE project using the currently available release of NetIDE architecture: Engine, IDE and tools[1]. The industrial partners of the project evaluate different aspects of NetIDE against their expectations, which are reflected in the the following UCs:

- Vendor Agnostic Datacentre Evolution
- Integrated IT System
- Hybrid Environment Application Development

We present each of the UC implementation and evaluation by the UC owners following the stories defined in Deliverable D5.2 [1]. The implementations are defined with hardware and software parameters designed to allow the tests to be reproducible and provide a baseline to future tests. With the data collected, we aim to show how the first version of NetIDE fares in relation to the Metrics and Use Case Requirements defined in Deliverable D5.2 [1].
The Engine currently has three backends (Pyretic, Floodlight and Ryu) and three shim layers (Ryu, OpenDaylight, and POX). The API used is still the original unmodified Pyretic version. Although some of the shim layers and backends supports the NetIDE API 1.0, they are still not ready for our implementations (see Annex A). Because we are evaluating a preliminary version of NetIDE, we expect some of the components to be either incomplete or absent. For example the integrated platform is not a unique, all–in–one system, some of the tools are still in the early stages of definition, and there are still some shortcomings in the Application Programming Interface (API). Similarly, some of the tests planned in Deliverable D5.2 [1] were not completed but will be in a future deliverable. This deliverable exposes the implementation and evaluation of the Use Cases as well as a report on the different findings we encountered. Each UC owner used the available components of NetIDE to implement their UC: the Engine, the tools and the IDE. The deliverable is based on Deliverable D5.1 [2] for some of the concepts (e.g. the Use Case Requirements) as well as D5.2 for the definition of the Use Case. The Use Cases may be modified during the next part of the project for various reasons (compatibility issues, introduction of new concepts such as composition or server-level application, . . .); the justification for such changes will be reported in the milestone M5.2 (Use Case Redesign v1) due M24 and an updated version of this deliverable will be edited as D5.5 (Use Case Implementation - 2nd release) due M36.
The rest of the deliverable is structured the following way:

- Chapter 2 describes the evaluation of each Use Case in the following way:
  - The implementation parameters to allow the test to be reproducible
  - The evaluation itself which presents a step by step description
  - The results of the evaluation presented in this way:
    - The Metrics of the Use Case

[1] NetIDE Engine and tools release v0.2 of April 27th 2015

  - The scorecard telling which Use Case Requirement was fulfilled or not
  - A report concluding the evaluation and providing insight on the experience
- Chapter 3 concludes this deliverable.

2 Use Cases Evaluation

This section presents the results of the first implementation of NetIDE 's three UCs. Each UC will first present the implementation parameters that were used to get the data presented (in terms of hardware and software). It will then present the evaluation itself, with the success or setbacks that were encountered. The next section will be the results which consists of an analysis of the evaluation in regard to the UC's Metrics as well as the "scorecard" showing how the evaluation fared against the Use Case Requirements. Finally, a report of the Use Case owner on the evaluation will summarize the results.

2.1 Use Case 1: Vendor Agnostic Datacentre Evolution

The Telefónica Group is one of the drivers of the Network Functions Visualization (NFV) Industry Specification Group (ISG) at European Telecommunications Standards Institute (ETSI) [3] as a way of modernising the network infrastructure, enabling a more agile innovation process and reducing the Total cost of ownership (TCO). NFV will transform the essence of the current Point of presence, transforming them from equipment silos into specialised Datacentre (DC)-alike structures, in which mainly specialised servers and switching equipment will be used to implement network functions. Automating the switching process will require advanced Software Defined Networking (SDN) features. Additionally, the Telefónica Group also owns and operates large-scale Datacentres that offer virtual Datacentre (vDC) services. Currently, use of SDN technologies is increasing. However, one of the main obstacles to overcome right now in order to unleash the full potential of SDN is vendor dependency. This UC anticipates what will most probably be the migration from current DC infrastructures to SDN-driven DC infrastructures. Its objective is to provide a Proof of concept (PoC) of the processes involved in this migration and refine the requirements in terms of usability and applicability to the real world of the NetIDE system.

During the life-time of NetIDE , this UC[1] has been driving the development of the NetIDE architecture from the beginning of the project. We have realised full implementations of it in different environments to explore different implementation alternatives. All the code is available in the internal Subversion (SVN) repository [4]. Additionally, we have moved some of the implementations to our public GitHub repository [5]. We have used the Use Case in the following contexts:

1. to formulate a first attempt at the structure of the now discarded *Intermediate Representation Format* using the YANG modelling language [6]
2. to extract the interactions between the Pyretic implementation and the underlying controller framework to generate the Monitor, Analysis, Plan, Execution, Knowledge (MAPE-K) implementation of the Use Case [7]
3. to check the NetIDE APIv0.1 by implementing the Use Case on the Floodlight controller framework

[1] described in depth in Deliverable D 5.2 [1]

In the scope of this deliverable, and in order to evaluate the evolution of the project, we have selected the following three processes: 1. the validation of the design of the networking solution before it is implemented in a production environment; 2. the migration of the infrastructure supporting vDCs as a result of a technology update; and 3. the rollback process when the migration is deemed to have failed. The current time-scale for these processes is in the scale of days to weeks for the first case [8] and we are aiming at reducing it to hours. In the second and third cases, we are currently working with processes which combined cannot last more than the typical duration of an upgrade window (i.e. circa 4 hours) and we are aiming at reducing them so that the maximal duration is less than one hour. In addition and from a more general perspective, we evaluated the impact of the introduction of software development concepts on the management and operation of SDN-based networks. To perform the evaluation, we restricted the scenario to the use of a standalone Ryu implementation and a NetIDE engine implementation with OpenDaylight as server controller and Ryu as client controller.

2.1.1 Implementation parameters

As we will explain in Section 2.1.2, we have used a typical development workstation to evaluate the implementation environment of this Use Case.

- Hardware platform:
  - 3^{rd} generation $i5$ Intel processor,
  - 16 Gbytes of RAM
  - a hard disk with 500GByte storage capacity
- Software:
  - Mac OSX 10.9.5 (Mavericks)
  - Eclipse Luna
  - Virtualbox 4.3.28
  - Mininet virtual machine [9]
  - Python 2.7.4
  - Ryu 3.21
  - Java 7
  - OpenDaylight (ODL) (Helium release) with the current NetIDE Engine developments
- Other:
  - An Internet connection with reasonable bandwidth

2.1.2 Evaluation

Given the state of the development, we concentrated in this first phase on the development capabilities provided by the NetIDE system. Thus, our tests were conducted on a development computer. We recognise that all steps leading to the deployment on a *production–grade* environment still need to be designed and may involve processes that are beyond the scope of the NetIDE project. This include the integration with a Virtual Machine (VM) management system (like, for example, OpenStack [10]) which are currently used to deploy the VMs of end-users. With these limitations in mind,

we implemented the following components of the vDC Use Case on a Ryu controller framework[2]:

1. a simple firewall, allowing the Layer 2 traffic between the router and the elements of the Demilitarised Zone (DMZ) and the HTTP traffic (i.e. TCP on port 80) and the Domain Name Service (DNS) traffic between the elements of the DMZ and the *Internet*.
2. a simple router, advertising a prefix (assigned to the vDC) to the *Internet*
3. a simple Layer2 switch that controls the traffic between the elements in the DMZ and the firewall

Then we tried to combine all elements into a simple implementation of a DMZ, as shown in Figure 2.1.

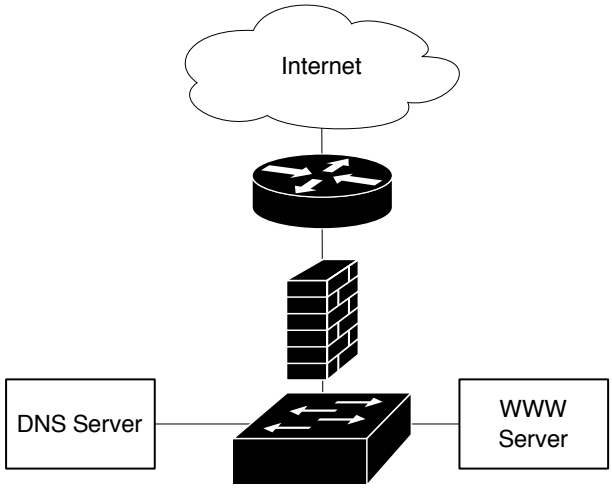

Figure 2.1: The initial Use Case 1 implementation

To test our development we used a Mininet network emulation environment. We compared the implementation process coding the elements with a Python IDE[3] and the NetIDE IDE. Figure 2.2 provides a snapshot of the topology editor of the NetIDE IDE during the development of the Use Case.

2.1.3 Results

This section presents the outcome of the evaluation in two subsections:

- The Metrics are defined in D5.2 [1]. Each metric, depending on its type, will have a qualitative feedback or a quantitative one which compares the measurements with the expected values.

- A "scorecard" table where we indicate for each Use Case Requirement (UCR) if, and to which degree, the requirement was met, with a short explanation of the result.

[2]We have selected these elements, because the Ryu controller framework bundles these functionalities and we will target to demonstrate re-usability with these well-proven components in the final implementation of the Use Case.
[3]We use PyDev in a clean Eclipse installation, i.e. without the NetIDE IDE components

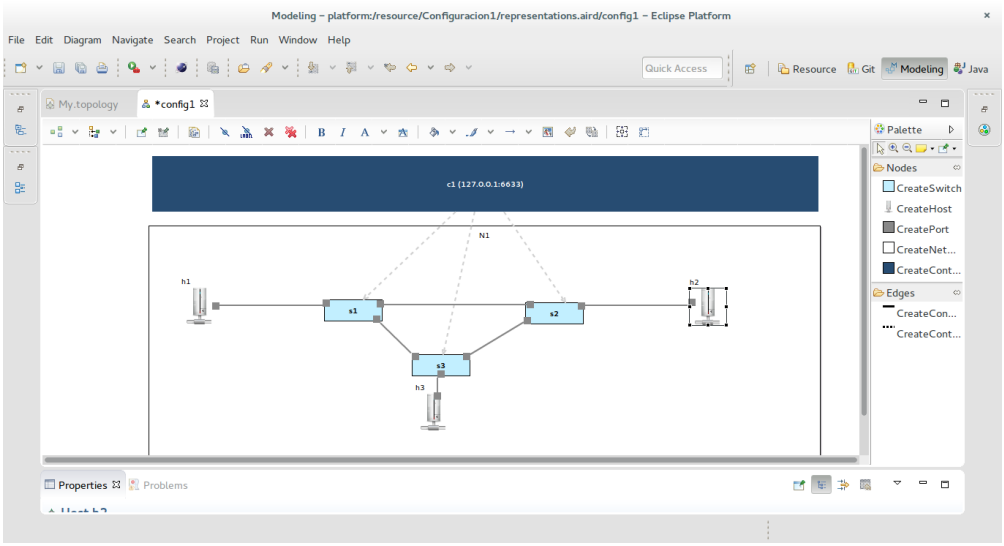

Figure 2.2: UC1 design in the IDE. A snapshot.

2.1.3.1 Metrics

Validation time

The time required to validate the design in the new technology environment by means of the simulation facilities.

1. We implemented and validated the vDC use case with current tools, i.e. Eclipse with PyDev, a virtual machine with Mininet and a controller framework. The developer was an advanced user with basic training in this OpenFlow environment.
2. We then implemented and validated the same vDC Use Case with the NetIDE IDE.
3. We compared the time needed for both processes.

Migration time

Time required to migrate to the new technology environment.

We started sending Internet Control Message Protocol (ICMP) packets between the different virtual machines of the virtual DC with 1 second intervals between ICMP packets with the *ping* program. Then we executed the following procedure:

1. We started from a virtual DC implemented on the Ryu controller framework.
2. We stopped the virtual DC
3. We restarted the virtual DC implemented on a NetIDE Engine, reusing the same code.

Once the ICMP packets reach all destination machines, we defined the down-time as the time the ICMP stream was interrupted.

Rollback time

The time required for rolling back the DC network infrastructure to the original state.

We reverted the steps followed to estimate the migration time and measure the down-time for this migration.

2.1.3.2 Use Case Requirements Scorecard

The following table summarises the tests planned for UC1. Given the overall state of the development, we have not been able to perform all of them. We therefore include two columns with tests we did perform in this round and tests we plan to implement in the next round of tests. For the tests we did perform, we have graded results as passed (✓), partially passed (∼) and failed (×). We also include the planning for the next round of tests[4]. For each UCR, we provide an additional comment to either clarify the outcome or reason why we did not perform the test.

Table 2.1: UC 1 requirement tests

Use Case Requirement	Tests on NetIDE system		Comments
	Current	Next	
UCR.1: The use case will use two different SDN control frameworks.	✓		We have used a Ryu based implementation of the use case. In the test, we compare the standalone Ryu environment vs. the NetIDE engine with Ryu as a client and OpenDaylight as a server controller.
UCR.2: The use case will make use of the abstraction capability to express the DC infrastructure network behaviour to be preserved across migrations.		✓	This abstraction capability will be provided by NetIDE applications. This test is left for the second round, waiting for the NetIDE application definition to be finalized by then.
UCR.3: The use case will require the capability of closely reproducing the production environment to accomplish the second phase described in the story section.	∼	✓	This Use Case Requirement is fundamental for NetIDE to be adopted in the production environment. We want to use it to demonstrate NetIDE to the Operations & Business units, however the NetIDE system is not there yet.
UCR.5: The use case will rely on NetIDE deployment capabilities for its basic goals.	✓		We have used the facilities provided by the NetIDE IDE in the implementation process.

[4]The second round of tests will be reported in Deliverable D 5.5.

Table 2.1: UC 1 requirement tests (continued)

Use Case Requirement	Current	Next	Comments
UCR.6: The use case will require the integration with external frameworks used to define and operate the whole DC infrastructure.	×	✓	The level of maturity of the NetIDE framework does not allow this test yet and we have decided to postpone. By external frameworks, we mean the VM lifecycle management and the integration with the networking infrastructure of the DC.
UCR.7: The use case will address multi-vendor coexistence and migration.	∼	✓	The NetIDE Engine supports different server and client controllers and is therefore, multi-vendor. However, we performed all tests on the Mininet simulator. We plan to address multi-vendor in the network infrastructure in future iterations of the tests.
UCR.8: The use case will explore the use of more than one South Bound Interface.	∼	✓	The ODL-based NetIDE Engine we used in the test is based on the NetIDE APIv0.1. It translates between OF1.0 and OF1.3. To completely validate this requirement, we are waiting for the next versions of the NetIDE API to be released.
UCR.9: The use case will require support for modular deployment and incremental testing.		✓	Waiting for the NetIDE application and composition mechanism to be defined at project level.
UCR.10: The use case will require large-scale simulation of the DC infrastructure network.	×	✓	The NetIDE IDE only supports small scale simulation on the Mininet environment. We plan large-scale simulation tests in future releases of NetIDE .
UCR.13: The use case will attempt to reuse the maximum amount of code possible in the different technology scenarios to be migrated.	∼	✓	We reuse the Ryu application in the ODL-based network engine
UCR.14: The use case will use debugging and tracing capabilities across the migration phases.	∼	✓	Currently, only logging is possible with the logger prototype. The output it provides is of limited value in our debugging process (see Figure 2.3).

Table 2.1: UC 1 requirement tests (continued)

Use Case Requirement	Current	Next	Comments
UCR.15: The use case will require different platform coexistence to support seamless migration in any direction.	∼	✓	We have verified that we can migrate from the Ryu standalone implementation to the NetIDE Engine 1.0 using OpenDaylight as a server controller and Ryu as a client controller. We are waiting for the NetIDE architecture to evolve in order to check the co-existence of different controller frameworks as client controllers in the NetIDE engine.
UCR.16: The use case will use Network application modules written for different client controllers.		✓	See comments to UCR.15. We are waiting for the NetIDE Engine to support the coexistence of several client controllers.
UCR.17: The use case will need to match policies against an abstracted network model.		✓	We are waiting for the NetIDE v2 architecture
UCR.18: The use case will require, as an essential aspect, to avoid conflicts with different choices of controllers.		✓	Conflict resolution is an essential aspect of the NetIDE v2 architecture. The current version of the NetIDE Engine does not implement it. Therefore we decided to postpone this test to the next round.
	On current:	✓	checked and passed
		×	checked and not passed
		∼	checked and incomplete
			not checked
	On next:	✓	planned
			not planned

2.1.3.3 Report

Taking into account that the project has suffered a major technological reorientation in the M6 technical review, we have lowered the expectations on this first systematic check. We conducted all tests using the IDE on the simulation environment it provides. The evaluation process covers the three main phases in our application cycle, namely **development**, **test** and **deployment**, which are treated separately further below. In terms of the metrics, as defined in Section 2.1.3.1, the results are shown in Table 2.2. These results should be interpreted as follows:

1. The validation time metric includes both the implementation and the validation of the NetIDE application. None of the *intrinsic* difficulties of the application are removed or reduced; however, the IDE provides mechanised solutions for some of the most error-prone tasks (like, for example, setting up the Mininet environment for a specific network topology, automated generation and setup for the different VMs).

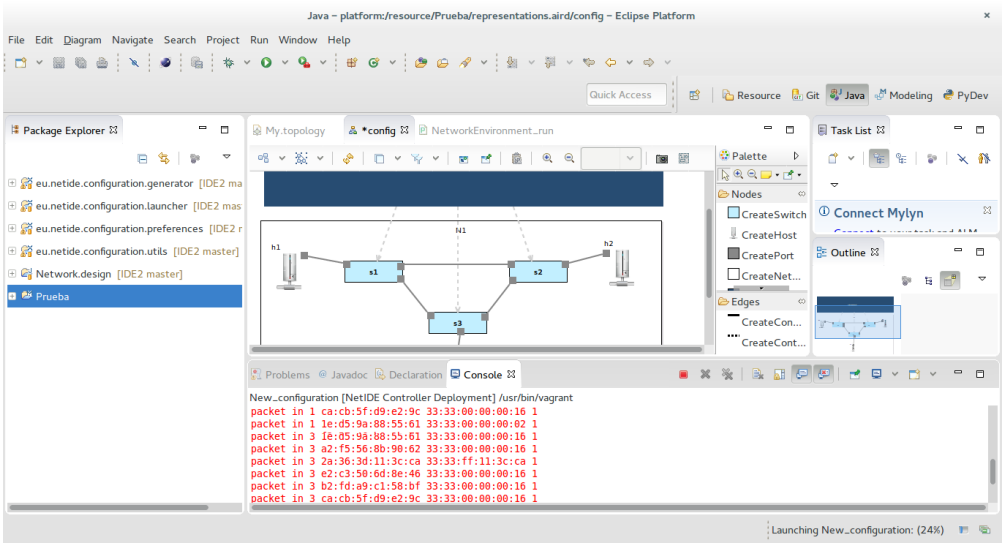

Figure 2.3: UC1: using the currently available debugging facilities

Metric	Value	
Validation time	6 days (before)	3 days (after)
Migration time	2 hours	
Rollback time	30 minutes	

Table 2.2: Use Case 1 evaluation metrics

2. The migration time comprises stopping the simulation environment, reconfiguring the affected VMs to integrate the Ryu back-end and the OpenDaylight server controller, and restarting the application. Unfortunately, since the integration with production-alike infrastructure is yet to be delivered by the project, we had to time this process in a small-scale emulation environment.
3. The rollback time suffers the same limitations as the migration time, since it could only be verified on a simulation environment.

Nonetheless, this first iteration of the UC implementation cycle allows us to assess the progress of the project towards implementing the full vDC use case and we can say that with the current tool-set, NetIDE lowers the entry barrier to SDN in general, and to OpenFlow-based networks in particular, for network application developers and for quality assurance (QA) teams.

The development process still has some way to go to meet the expectation of bringing the DevOps principles (as outlined in [11]) and facilities of a software development cycle to Software Defined Networking. We recognise that component reuse is not pervasively available outside NetIDE and that controller frameworks supporting it (e.g. Pyretic) still require a significant amount of user intervention to put in place an SDN application that is composed of available modules (i.e. modules provided by the framework itself or modules which were developed previously). We understand that the IDE does not provide means for component reuse and application composition basically because of two reasons: in first place because controller frameworks do not support it the way we expect

and then because we are still working on the NetIDE application format, as our project specific enabler for module reuse.

The testing process starts to show promising results. With the network editor provided by the IDE we were able to reduce the time to setup the test network on the mininet simulation environment significantly and eliminates errors that slow down the process. The fact that the IDE automates the generation and launching of the different Virtual Machines needed for specific tests is extremely helpful and smooths the learning curve. The logger is a good start for tooling the NetIDE system. We expect that once tools become available, the usefulness of the NetIDE tooling interface will become more clear.

The deployment process is still in its infancy. With the current version of the Network Engine we could, at least, show the principle of platform independence. While we are able to show that an application written for a specific controller platform can be used as-is on one of the controller platforms we designated as Server Controllers, we still need to work on the creation of NetIDE applications that use modules from the Client and the Server controller platforms simultaneously. Another aspect that needs thought is the integration of the IDE with production systems that would allow to automate the deployment of NetIDE applications to a production system.

2.2 Use Case 2 : Integrated IT System

Market demand for agile IT which rapidly supports the productive introduction of scalable new customer services reinforces Fujitsu's belief in the need for Integrated IT Systems using the design principles of our Use Case. Fujitsu's interest is to accelerate the time to market of our solutions while maintaining quality and performance standards. Fujitsu continues to see the principle benefit of NetIDE for Fujitsu in the improved tooling it provides during the development and test phases of network applications. Fujitsu is not currently focusing on the cross-controller capabilities of NetIDE. However, future scenarios are conceivable in which this capability may become significant. For instance, a third party security application, written for a different controller than our fabric application, could become a valuable addition to the Integrated IT System. The practical efforts to date to use and test the NetIDE software on Use Case 2 have focused on:

- Writing a reduced-capability mini-application which demonstrates the networking principles which have to be supported by NetIDE for Use Case 2
- Making this application run in an emulation environment (Mininet)
- Installing the current state of the NetIDE software in the Fujitsu labs
- Running the mini-application on the NetIDE software
- Using available NetIDE tools, currently this just means the network topology editor, to make an initial assessment of possible productivity gains in the development and test cycle

As more NetIDE software features become available, the testing effort will be expanded, in particular regarding the tooling.

2.2.1 Implementation parameters

We have used the following hardware configuration and software stack for the Integrated System Use Case evaluation.

- HW platform
  - Fujitsu Primergy BX620S5
  - 2 x Intel(R) Xeon(R) CPU E5504
  - 8 GB Memory
  - 2 x SAS 73 GB HDDs
  - 2 x 1Gbps Ethernet
- Software
  - Ubuntu 14.04.2 LTS
  - Eclipse Luna Vers.4.4.2
  - PyDev Vers.3.2.0
  - OpenJDK 7u79
  - NetIDE current version
  - Integrated System fabric application

2.2.2 Evaluation

For setting up the NetIDE framework we installed lots of different software packages. We did define Projects in the NetIDE framework, did set PATH's and did set up the Python and Eclipse environment. The NetIDE Engine itself didn't require significant effort for installation. We did customize views and figured out how to handle application development and how to import existing code. We integrated all the required tools into an VM image.

All currently supported capabilities of the NetIDE development environment were used for a first implementation of UC2. The current NetIDE version with the software stack described above and the Integrated System fabric application on top were installed and executed in Fujitsu's labs on a Fujitsu server system. At this stage of NetIDE's implementation, much manual work is necessary to install and run NetIDE. Further NetIDE implementation progress is also required in order to test a broader range of NetIDE's promising capabilities in the future.

The network topology editor included in NetIDE was used to create the network topology of UC2 for the Mininet simulation environment. After creating the integrated system network fabric, we combined all elements of the Use Case 2 integrated system application, as shown in Figure 2.4, for evaluation work.

The Mininet network emulation environment was used to evaluate the UC2 implementation within the NetIDE framework.

2.2.3 Results

This section presents the outcome of the evaluation in two subsections:

- The Metrics are defined in D5.2 [1]. Each metric, depending on its type, will have a qualitative assessment or a quantitative one which compares the measurements with the expected values.
- A "scorecard" table where for each UCR we check if, and to which degree, the requirement was met. If the requirement is not met we provide a relevant comment.

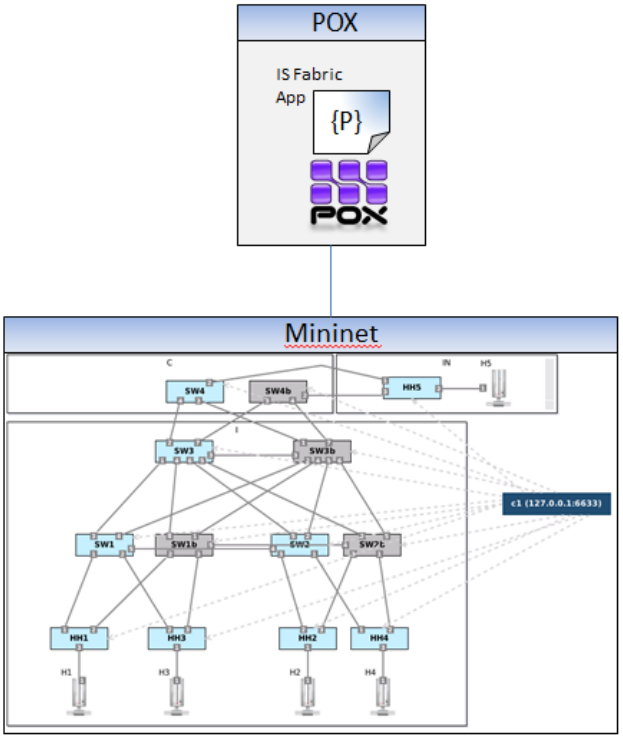

Figure 2.4: The initial Use Case 2 implementation

2.2.3.1 Metrics

Runtime Performance

An application written and deployed using the NetIDE environment shouldn't have worse runtime performance than the same application originally written and deployed for a specific controller.

1. This is tested by running the application on top of the NetIDE Engine and comparing the runtime performance characteristics with the same application run directly on the same target controller.
2. Once the shim layer in the NetIDE Framework gets usable, its impact on application runtime performance will be shown. The interception capability provided by the NetIDE shim layer, which will be usable e.g. for application event logging under NetIDE, shall not impact the application runtime performance under NetIDE. This has to be shown

Reduced Application Development Effort

Reduced time for the development and test cycle thanks to the availability of an integrated emulation environment which allows rapid testing in a virtual environment.

1. The time/effort required to setup NetIDE with all software parts and environment settings shall be tracked.(e.g. Eclipse and Python environment, Paths to be set, Project Setup)
2. It has to be shown that the NetIDE Engine helps installing the controller frameworks and that there is 0 minutes/hours to be spend to install frameworks manually, because they are installed automatically by the IDE.
3. The times required for development, compilation, deployment and simulation of the application using NetIDE vs not using NetIDE has to be measured.

Reduced Effort for different Environment Tests

The NetIDE network topology editor shall reduce effort and time to test network applications in different network environments.

1. For initial UC2 development and evaluation we use a normal editor to create the representation of the UC Ethernet network topology for simulation
2. Then we use the network topology editor to create the network topolgy for the simulation environment. Effort and time savings enabled by the usage of the network topology editor in comparison to manual network environment creation in Mininet will be shown. The figure above "The initial Use Case 2 Implementation" shows the Network topology as being created in the NetIDE network topolgy editor for UC2 implementation

Debugging Efficiency

Debugger capabilities have to be shown such as providing the status of the underlying network, of running applications and packets in flight and an inspection capability for the most important network elements.

1. First experience of the value of the debugger shall be gathered by starting the logger tool.
2. As soon as possible the Graphical User Interface of the debugger shall be evaluated.

2.2.3.2 Use Case Requirements

Table 2.3: UC 2 requirement tests

Use Case Requirement	Tests on NetIDE system		Comments
	Current	Next	
UCR.2: The coding effort required in order to formulate the behavior of the application when using NetIDE rather than alternative development environments *should not increase* and should preferably decrease. For UC2 the NetIDE environment with integrated tools like the network topology editor, the source code editor and built-in debug capabilities shall reduce the application development effort.	~	✓	Additional effort is necessary in order to use Eclipse with NetIDE. Projects need to be defined, Paths have to be set and the Python environment needs to be set up. The NetIDE Engine itself does not require significant additional effort. However, the handling when developing an application or importing existing code, the allocation of views etc. requires some customization. This should be simplified in future.
UCR.3: The use of the NetIDE development and runtime environment **must** not degrade the reliability of the application's behavior.	✓		Code is generated to describe the topology of the project for the Mininet simulation. It is comparable to the code a developer would write manually and thus does not degrade the project's reliablility more than the human written code does.
UCR.5: NetIDE should not make the installation and deployment of applications more complex. The NetIDE environment, including its repository for applications, should make the installation and deployment process of applications more convenient.	~	✓	At this stage we have little experience of deploying finished applications. Currently the initial setup of the development environment is quite complex. In particular the VM needs to be set up and all required tools must be integrated into the VM image. This work needs to be repeated in the case of updates and only pays off if the IDE is frequently used.
UCR.9: It **must** be possible to test multiple versions of the same modules.	~	✓	Versioning is not yet supported by NetIDE. It is in principal provided by tools like git, which can be used in an Eclipse environment.
UCR.11: NetIDE **must** support the development of Ethernet fabrics.	✓		The network topology editor was used to set-up an Ethernet Fabric topology for the Mininet simulator

Table 2.3: UC 2 requirement tests (continued)

Use Case Requirement	Current	Next	Comments
UCR.14: NetIDE *should enable* application debugging to a comparable level of alternative development environments. The debug information should contain the status of the underlying network and running applications, packets in flight and the composition and demultiplexing logic at runtime. Inspection of the information shall be supported by a graphical debugger in the IDE.	~	✓	This is not yet implemented in NetIDE. We expect to gather first experience of the value of the debugger interface when we start to use the logger tool.
UCR.14: The use of the NetIDE development and runtime environment **must** not compromise the security of the application (E.g. by making it easier to inject malicious code into a benevolent application).	✓		The IDE does not add code to the application. It simplifies the creation of the topology with generated code, but this does not affect the application. Thus NetIDE does not compromise the security of the application code itself. However, the runtime environment, in particular the NetIDE Engine, may create new security risks. The NetIDE Engine opens a new, non secured, TCP channel between backend and shim. The NetIDE Engine also provides a debugger interface which can be used to influence the behaviour of the application. Both of these mechanisms should be analysed to assess their potential for abuse and safeguards may need to be implemented.
UCR.17: NetIDE *should provide* a full network topology picture (i.e. map, links, load, events, etc.)	~	✓	NetIDE can be used to create a full network topology. However, the resulting graphs become confusing with increasing complexity. NetIDE should contain a graph visualization component which tries to present the topology with as few connection crossings as possible. A sophisticated visualization scheme will greatly contribute to the benefit of NetIDE vs manual coding.

Table 2.3: UC 2 requirement tests (continued)

Use Case Requirement	Current	Next	Comments
UCR.18: NetIDE tools *should uniformly support* code written for different SDN controllers.	∼	✓	The IDE does not interfere with coding itself. Within the Eclipse based GUI environment, additional editors can be provided in future to extend the number of requested application coding styles. POX was used for this use case initial implementation
On current:	✓	checked and passed	
	×	checked and not passed	
	∼	checked and incomplete	
		not checked	
On next:	✓	planned	
		not planned	

2.2.3.3 Report

Our initial experience with the NetIDE software is rudimentary but encouraging. Our assessment is that the topology editor has the potential to halve the time required to correctly describe a target typology, especially when the typology is non-trivial. The editor allows the easier creation of improved, graphic, typology documentation which eases the transfer of information between developers.

Although still not working perfectly the automatic VM creation and execution environment within NetIDE is very handy and significantly eases and by that significantly shortens the overall test and development cycle. For instance the direct switch from e.g. the Mininet console window to the network topology editor to change e.g. some host properties enables a rather interactive development method.

Further tools were not yet available for test. The tracer should give us a first indication of improved debugging support for network application developers. We see this as one of the key values of the NetIDE framework.

We are especially interested to see what kind of information will be available to the debugger user and what kind of manipulations will be possible (e.g. conditional break-points, analysis of packets in flight). A comfortable GUI is desirable and will be required for a commercialization of NetIDE. However, a useful first step for evaluation purposes would be access to the functionality through a simple CLI.

The initial set up of the NetIDE environment currently involves too much manual effort. We expect this to be addressed later in the project.

Our current experience is that the test program behaves the same way on the NetIDE Engine as when run directly on a controller. We have not been able to conduct any performance measurements yet. This will be a task for the next phase.

2.3 Use Case 3: Hybrid Environment Application Development

Thales has been involved with SDN since this novel concept started out in the academic world and set itself as one of the possible future direction of networking. During its investigation on the subject,

many controller frameworks were tested and network applications were developed with them. Seeing that eventually, the number of controller platforms would decrease, Thales saw an opportunity to investigate with NetIDE the possibility of porting code from one controller to another. The main rationale was to reduce the costs of manually porting time and NetIDE presented a interesting concept: 0-time portability. This UC is described in Deliverable D 5.2 [1] section 3.3. It shows how a company which has already invested resources in building multiple SDN networks and custom applications on specific platforms can reuse this work to seamlessly port these applications and thus reduce costs. The goal of this Use Case is to demonstrate that an application written on a controller platform can run on a NetIDE compatible server controller, reducing time to develop, complexity and gaining functionality.

Use Case 3 raised interesting suggestions during the discussion on the architecture of NetIDE , specifically pointing out the lack of support for essential messages in the Pyretic API and the expansion of the NetIDE API to include a more complete implementation of OpenFlow 1.0 and the support of LLDP messages.

To test the Use Case, we will compare the time to deploy a legacy application on a different controller platform from scratch versus using NetIDE . We will also check that switching to the NetIDE platform does not increase the complexity for the developer or the operator by requiring too much code modification or artefact definition. Finally, we will check the impact of deploying NetIDE on the hardware and the network.

2.3.1 Implementation parameters

As we will explain in Section 2.3.2, we have used multiple development workstations to evaluate the implementation environment of this Use Case.

To test this Use Case, we used the hardware defined in Deliverable D 5.2 [1]. To summarize, it consist of two identical servers to run the controller and NetIDE, an additional machine to run Mininet to simulate the network and basic management tools to test the behaviour of the network application.

- Hardware platform (Controllers) :
  - Intel Xeon 8 cores @ 2.93GHz
  - 10GB 1333MHz RDIMM RAM
  - 1Gbps network interfaces
  - a hard disk with 500GB storage capacity
- Hardware platform (network simulation) :
  - Intel Core i7 4 cores @ 1.90GHz
  - 10GB 1600MHz SODIMM RAM
  - 1Gbps network interface
  - a hard disk with 500GB storage capacity
- Software:
  - Linux Ubuntu 14.04.2 LTS x86_64
  - Virtualbox 4.3.28
  - Mininet virtual machine [9]
  - NetIDE v0.2

  - Floodlight 0.90
  - Monitoring application
  - Pox
  - iperf
  - tc
- Other:
  - An Internet connection with reasonable bandwidth

2.3.2 Evaluation

Due to the limitations of the current development of NetIDE, at this time we did not fully test the UC as much as we described in Deliverable D 5.2 [1]. For example, we did not simulate the development cycle on a cloud platform nor we had to use physical elements for the network. We did use our two physical servers for the controllers and only a simulation of the network on a third machine.

With these constraints, we implemented these components of the UC 3:

1. a Floodlight controller v0.90 [5] with the monitoring application
2. a NetIDE compatible server controller running shim layer
3. a Floodlight controller v0.90 [6] with the monitoring application and the backend layer for the Floodlight controller framework
4. a virtual machine running Mininet

The resulting deployment is shown in Figure 2.5.

To test the task of porting the code, we gave to two developers the code from the legacy application and ask them to either port it from scratch on OpenDaylight (see Figure 2.6) or to use NetIDE (see Figure 2.7). The developer which was in charge of manually porting the application (**Dev1**) had multiple steps to produce the code on unfamiliar platforms. First, he had to get the latest version of OpenDaylight [7] and go through the tutorial and the wiki to get accustomed with the platform. He then had to get the source code from the Floodlight application and go through it to understand how it interact with the controller platform [8]. Even if Floodlight and OpenDaylight are both written in Java, he couldn't copy/paste all the code to the new framework but had to adapt the code to ODL (meaning massive rewrites to the source code). ODL brings in a lot of technology to appropriate in order to be productive: OSGi, Maven, Yang, etc. You also need to understand how the ODL services (OSGi bundles), the Model-Driven Service Abstraction Layer (MD-SAL) and the MD-SAL Data Store fit together. In the end, even reproducing the *behaviour* of the legacy application wasn't the right idea because it meant an excessive loss in performance by the monitoring application, with CPU usage being on average at 43%. It thus needed a *complete rewrite* to get to the same level of performance as the legacy application (around 10% CPU usage). The second developer (**Dev2**) had to get the latest NetIDE source from the project's Github repository and read the tutorial available there. The next step was to integrate the Floodlight backend in the client controller and package it in a jar file (see about this addition in the paragraph about code and artifacts) and run it. He

[5] commit 0d4de9e93a3809b5baff4287e89f95e3a87ec539 from Oct. 22,2012
[6] commit 0d4de9e93a3809b5baff4287e89f95e3a87ec539 from Oct. 22,2012
[7] During all evaluations, the times to download or install a program are not taken into account.
[8] A guide was provided by the developer of the application.

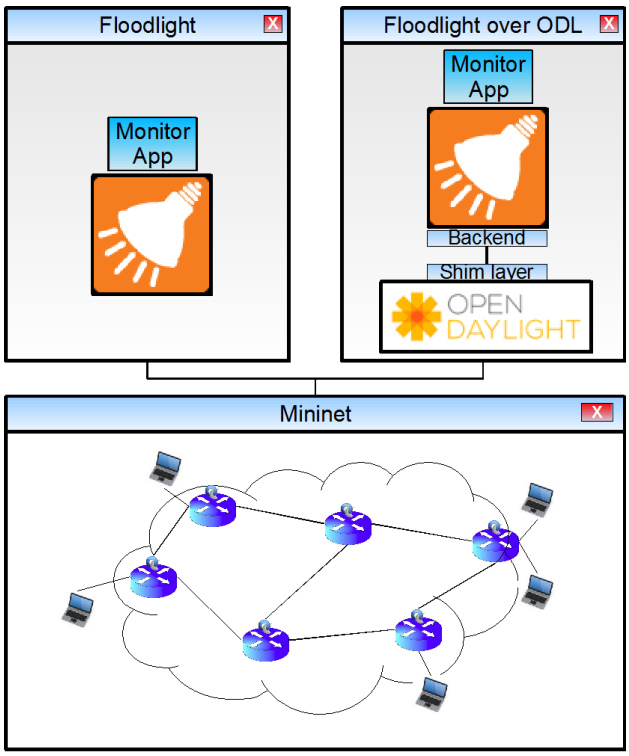

Figure 2.5: The initial Use Case 3 implementation

then had to use his server controller of choice already fitted with the shim and run it as well. This was all that was needed to run the legacy application on a different controller.

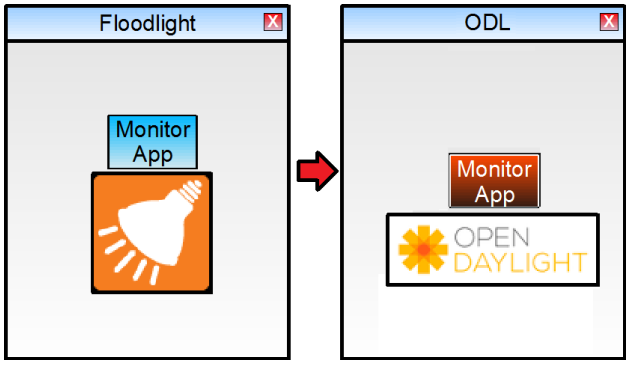

Figure 2.6: The goal of the Manual Port of the Network Application

To test the time for simulation we asked **Dev1** to use a Python file to represent the topology of UC 3 using Mininet. He had to get familiar with it and mostly got around the difficulty by copy/pasting existing example and then editing the configuration. **Dev2** used the Editor to graphically create the topology (see Figure 2.8) and generate the same Python file with the push of a button.

The code and artifact from NetIDE and the modification of legacy code were to be measured like described in Deliverable D 5.2 [1]. Because we did not use a "fully integrated version" of NetIDE (with application bundling or running everything from the IDE for example) we did not

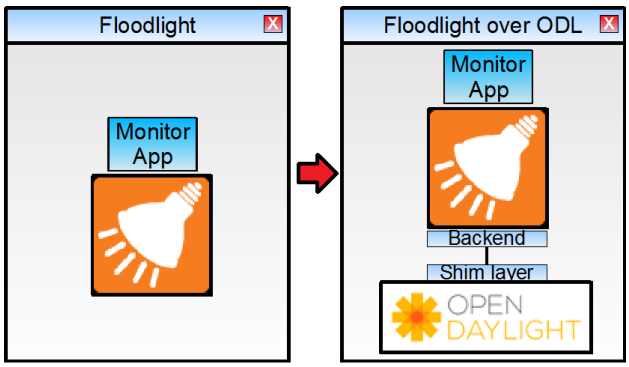

Figure 2.7: The goal of the NetIDE Port of the Network Application

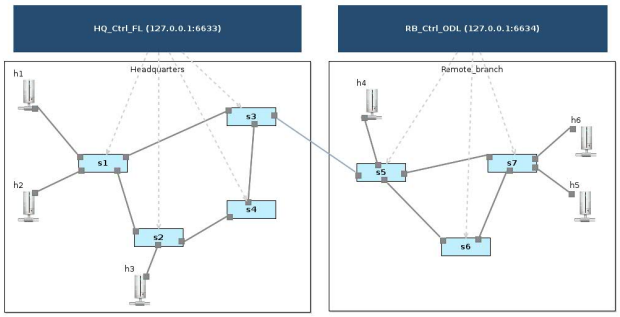

Figure 2.8: The NetIDE Editor with UC3's topology

have any configuration file associated with the test of the UC. Most components (Editor, Network Engine, ...) were tested separately. NetIDE also provide self-configured VMs with "shim-enabled" server controllers. Because we used off-the-shelf controller, we had to manually add the shims layers ourselves. We indicated that this wasn't going to be measured as a relevant metrics but in any case, the most work we had to do was to copy some packages and edit two lines of configuration, which is trivial.

Once the applications were running on the two systems, we verified their correct behaviour by sending traffic with *iperf* and see if the monitoring applications reported the same values, and changed some link latencies with *tc* to see if the change was visible in the applications. This data was obtained using the applications' REST APIs. This first measure was the one revealing the lack of performance of the first manual port of the application and provoked the rewrite mentioned earlier. The second test was more in line with what we expected and the two applications had comparable performances.

We measured the delay between switches and controllers using another REST command available

in the application [9]. To test the resource consumption we used common Linux tools for the CPU [10] and RAM [11] usage.

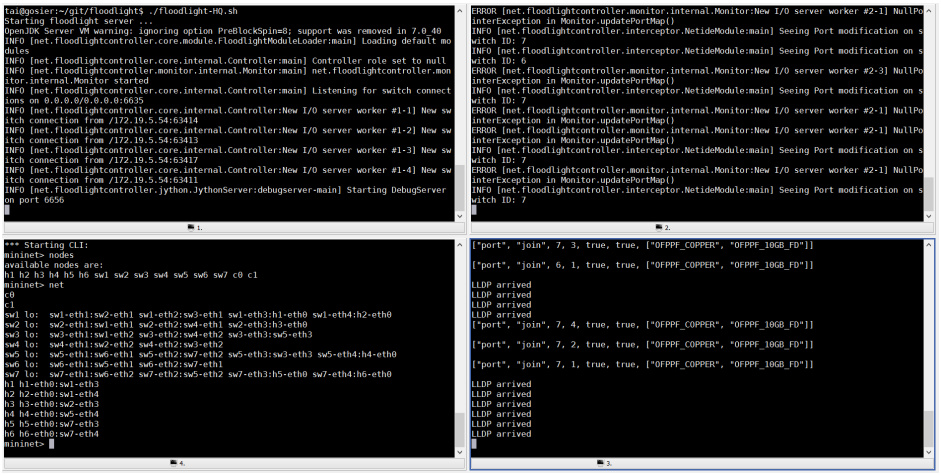

Figure 2.9: The view of the implementation, from the upper left, clockwise: HQ Controller (Floodlight), RB Client Controller (Floodlight + backend), RB Server Controller (OpenDaylight + shim layer), Mininet

2.3.3 Results

This section presents the outcome of the evaluation in two subsections:

- The Metrics defined in Deliverable D 5.2 [1]. Each metric, depending on its type, will have an qualitative feedback or a quantitative one which compares the measurements with the expected values.
- A "scorecard" table where for each UCR we check if, and on which degree, the requirement was met ; in case it's not validated, we provide a relevant comment.

2.3.3.1 Metrics

Time for development cycle

The time required to complete a development cycle.

[9] Get all the available data in the application : `http://[Controller_IP]:8080/wm/monitor/getAllData`
Get the latency between the switches and the controller : `http://[Controller_IP]:8080/wm/monitor/getSwitchLatencies`
Get the data from a switch : `http://[Controller_IP]:8080/wm/monitor/getSwitchData?id=[swId]`
Get the data from a switch's port : `http://[Controller_IP]:8080/wm/monitor/getPortData?id=[swId]&port=[swPort]`

[10] htop
[11] free

Manual port:

1. Each entity (developer, tester, ...) uses the target platform(s) for the first time, following the guides.
2. We port a legacy Floodlight application manually to the target platform(s) (the monitor application).
3. We write manually a *mininet* python script to create a representation of the network.
4. We launch the application with two settings: native Floodlight and with the target platform.
5. We test the behaviour of the network applications with fake traffic and REST calls.

NetIDE port:

1. Each entity (developer, tester, ...) uses NetIDE for the first time, following the guides.
2. We take the source code from a legacy Floodlight application (the monitor application).
3. We use the Editor to create a representation of the network.
4. We launch the application with two settings: native Floodlight and Floodlight with NetIDE .
5. We test the behaviour of the monitoring applications with REST queries and fake traffic.

We compare the time to complete each port.

Code and artefacts from NetIDE

How much the artifacts impact the deployment of the application. What will be written, how much code is needed, etc.

1. We check how much code and data is needed to bundle an application with NetIDE .
2. This value is be compared with the expected value and the leeway accorded.

Modification of the legacy code

How much of the original source code from the network application is needed to be modified in order to be compatible with NetIDE .

1. We take the original Floodlight application source code and make a *diff* with the final source code.
2. We check the amount of modification to the Network Application only (not the necessary changes to the client and server controllers).

Delay

How much delay is added to the processing of a packet by using NetIDE .

1. We run in parallel the two systems (with and without NetIDE) and check the controller-to-switch latency with the monitoring application.

Resource consumption

How much extra resource is needed to run an application with NetIDE .

1. We do a test on multiple pairs of server and client controllers.
2. We compare this value to the resource consumption of a controller running alone on the system.

2.3.3.2 Use Case Requirements Scorecard

The following table summarises the tests planned for UC3. Given the overall state of the development, we have not been able to perform all of them. We therefore include two columns with tests we did perform in this round and tests we plan to implement in the next round of tests. For the tests we did perform, we have graded results as passed (✓), partially passed (∼) and failed (×). We also include the planning for the next round of tests[12]. For each UCR, we provide an additional comment to either clarify the outcome or reason why we did not perform the test.

Table 2.4: UC 3 requirement tests

Use Case Requirement	Tests on NetIDE system		Comments
	Current	Next	
UCR.1: The Use Case 3 must operate on at least two platforms (Floodlight and OpenDaylight) to be validated. More platforms would be a welcomed addition to show that NetIDE is the best way to enable network application on multiple platforms.	×	✓	There are some issues with the Floodlight backend related to the NetIDE API. Some messages are not delivered correctly to the application. Nevertheless, the setup works on multiple platforms, especially Floodlight and OpenDaylight. The next iteration with the new NetIDE API should fix this issue.

[12]The second round of tests will be reported in Deliverable D 5.5.

Table 2.4: UC 3 requirement tests (continued)

Use Case Requirement	Current	Next	Comments
UCR.2: This component is necessary so that the statistics and network state can be accessible to the Monitor application. The list of parameters required by the application is only a subset of the total normally available list. Only the useful parameters are required.	✓		The necessary statistics are visible to the application. Work is in progress in the Floodlight backend, the NetIDE API and the shims layers to include more statistics.
UCR.3: This requirement is necessary for the Use Case when the developer needs to test the application for validation before deployment. It also should simulate realistic network conditions.	∼	✓	Mininet and fake traffic are used for validation. The Simulator tool will be tested in the next version.
UCR.4: The Developer will write the application with Floodlight. It's NetIDE's role to make the application run on Floodlight and OpenDaylight. There should be no intervention in the application source code to make it work on NetIDE, or only a minimal amount.	✓		It is completely transparent for the application running on the client controller. It doesn't know what server controller is in front of it or if it exist at all.
UCR.5: When the application is deployed on a physical network, NetIDE must ensure that there are no damaging interaction with the network platform already in place.	∼		Adding one client controller is seamless from the point of view of the shim. Multiple clients would have to be resolved in future releases with composition.
UCR.8: The homogenization is crucial for this Use Case. The Operator must be able to use NetIDE on other protocol than OpenFlow to have a broader reach on what technology can be used at the southbound interface.	×	✓	For now there is only partial OpenFlow support in the NetIDE API.
UCR.9: Testing multiple version would allow quicker development. For example by adding features in a development branch and being able to revert easily. If we can't do concurrent testing, we can still expect an easy way to swap application version within NetIDE.		✓	This feature was not tested as the testing platform is not ready yet.

Table 2.4: UC 3 requirement tests (continued)

Use Case Requirement	Current	Next	Comments
UCR.12: Depending on the controller platform, we could use multiple southbound protocols. For example, OpenDaylight can provide this functionality natively. In case OpenDaylight uses another southbound protocol than OpenFlow, the application must still have access to the same primitives (e.g. Port statistics, link state, . . .).		✓	Not tested as we only use OpenFlow network elements and the NetIDE API only supports OpenFlow.
UCR.13: This requirement is the key to the rationale of the Use Case. The application developed must be able to be deployed in the future on another branch of the company with another SDN platform seamlessly.	✓		The port of the legacy code on a new platform was successful.
UCR.14: The debugger is a must have aspect of application development during the testing phase to ensure an environment devoid of critical bugs. The other tools could also be used to improve the development of the application but are not strictly necessary for this Use Case.		✓	The Debugger tool is not available, it will be tested in the next version.
UCR.17: These levels of abstraction are necessary to allow the developer to focus on writing generic, portable code, instead of an application specific to a network topology or dataplane hardware.	∼	✓	The NetIDE API is the only point of contact for the developer so the abstraction is respected; but it's not complete. We will wait for the second version of the API.
UCR.18: This requirement is necessary to ensure code portability and prevent conflicts.		✓	This feature was not tested.

On current:	✓	checked and passed
	×	checked and not passed
	∼	checked and incomplete
		not checked
On next:	✓	planned
		not planned

2.3.3.3 Report

The NetIDE project had an extensive technical and architectural shift following the M6 review, we thus had to redefine how to test the UC implementation. Most tests were made on a hardware platform for the control plane with virtual machines for the network. We did not use or test some of the tools either because they were not readily available or that testing the full simulation environment would have been too time consuming. The metrics defined in Section 2.3.3.1 have their detailed result below. The main aspect tested by this UC are the temporal one (the time from development to run while porting an application), the amount of necessary work to do so and the behaviour of the system once the application is running.

Metric	Manual Port	NetIDE Port
Porting time	10 days	1 day
Simulation time	1 hour	0.5 hour
Total time	81 hours	8.5 hours

Table 2.5: Use Case 3 evaluation metrics

The temporal advantages of NetIDE were clearly seen during the testing of the UC as shown in Table 2.5. We can see that the NetIDE total porting time is a little over **10%** of the total manual port time, meaning tremendous gains in terms of time and resource. As we saw in Section 2.3.2, porting an application with NetIDE provide major advantages compares to a manual port from scratch. This is the main interest UC 3 had with NetIDE , the ability of seamlessly use legacy application code on different controller whilst saving time and resources. While we recognise that we didn't fully test the whole NetIDE system, its promises on zero-time portability holds true. For testing the application, the two duration are roughly similar. The scripting language of Mininet is mostly easy to understand so the developer did not have a hard time to create the file. It is however important to note that the UC 3 topology is not at all a complex one. The graphical topology Editor of the IDE is a good tool to have for complex topologies and to reduce errors when manually editing the file. Our limit for this metric as described in Deliverable D 5.2 [1] was to get at least a *50%* decrease in development and testing time. We can clearly see that NetIDE went beyond the expected requirements.

The code and artefact necessary to run the application was tested differently. First off, as we saw in Section 2.3.2, we did not bundle the application with the artifact and the parameters described in Deliverable D 3.2 [12] and went with a more "direct" approach when implementing the UC. Thus, there were no artefact per se. The amount of code written to port the application was thus **0KB**. This measure will most certainly change in future implementations and will be revised in another deliverable. Regarding the change of legacy code, we can see that NetIDE performed precisely as expected. While the manual port required an extensive amount of code (around 700 lines of code but some of it, in the order of 200 lines, was auto-generated) the NetIDE port required **no modification** of the source code. Even accounting for the controllers modifications (which we decided not to for this test), the total modification is two lines in a configuration file. We are confident that NetIDE will still allow in future version to port directly legacy application without the need of modifying their source code.

The behavior of the system was tested in two ways. First, the delay added in the data-plane by using NetIDE . Because there is an extra layer between the application and the network elements, we expected that the delay of communication between the client controller and the switches would increase. We saw in our tests that on average, the monitoring application measured that the delay between the network elements and the controller increased by a little over **10**s. Upon investigation, we observed that this value correspond to Floodlight's timeout when requesting statistics and the actual value was thrown in the mix. There seem to be an incoherence between Floodlight and the backend, due to the asynchronous nature of these messages. The next version on the Engine and API should fix this issue but we are still going to watch if the addition of the planned elements (NetIDE core, conflict resolution, ...) will have an influence on this value. Regarding the resource consumption, we saw that this metric was very dependent on the system configuration as well as the controller used. The CPU consumption test revealed that using NetIDE had a negligible impact on the system. The reason is that the different additions to the server and client controllers (the shim and backend) only do low computation tasks, which is the serialization/de-serialization of the Pyretic API. So launching a regular controller and a "NetIDE enhanced" one doesn't add a noticeable CPU overhead. The same goes with memory, so by knowing how much a controller use on a machine, adding an extra controller will only add its memory footprint, there is no additional memory use for NetIDE . The table 2.6 shows the average memory use for some of the most used controllers. We can clearly see that launching multiple instances of POX or Ryu won't have a lot of effect on the system whereas "bigger" controller such as Floodlight[13] or OpenDaylight would be more difficult.

Controller	Memory Use
POX	7MB
Ryu	30MB
Floodlight	400MB
OpenDaylight	450MB

Table 2.6: Use Case 3 evaluation metrics

This first implementation of the UC with NetIDE is encouraging. A lot of the key point of NetIDE were proven right, such as the zero-time portability and the use of legacy code as-is. We are still waiting to see how the tools will improve the development and deployment of applications. We did not test all the possible pairing between server and client controllers, but saw that not all of them were at the same point regarding the API and the messages that were compatible or not. Being developed in parallel by multiple developers, this is understandable. We will point out to the development teams the need to provide an up-to-date document describing this compatibility in detail. Nevertheless, we see the development of NetIDE as being in line with the work plan and look forward to test the features of the next version.

[13] Floodlight was lanched with a generous amount of memory. If the application is not very intensive in terms of resource, it can be launched with as little as 64MB of RAM.

3 Conclusion

This deliverable provided a synthetic view of the implementation and evaluation of the three Use Cases specified in the Work Package 5 (Use Cases and Evaluation) of the NetIDE project with the version of the tools available at this point in the life of the project. We acknowledge that the current version of NetIDE is preliminary and it would be too early to make a complete assessment of the quality of the project, but we can clearly see the the project is going in the right direction. The deliverable presented our report of the implementation of the Use Cases and provided concrete data to evaluate NetIDE 's progress through them. The Use Cases presented in the Description of Work and defined in Deliverable D 5.2 [1] have now a first implementation as well as their evaluation, the verification that their requirements has been followed and their metrics.

The main achievements reported in this deliverable include:

- Using NetIDE to show that the industrial partners' Use Cases are possible to implement even on a scaled down version
- Showing that NetIDE can be a solution to answer the issues reported by the industrial partner
- Providing some insight on the project direction with regards to the findings of the evaluation process

The next step will be to continue to work alongside the other Work Packages. We will focus on giving feedback on the architecture, the NetIDE API and the tools. We will also use the feedback from this work to adapt (if necessary) the Use Cases to a new direction by including new concepts such as composition and server-level application. The result of this work will be reported in milestone M5.2 and deliverable D5.4.

A Engine Compatibility

	Ryu	POX	ODL
Pyretic	✓	✓	×
Floodlight	×	×	✓
Ryu	×	×	×

Table A.1: Engine compatibility between shim layers (top) and backends (left)

Bibliography

[1] The NetIDE consortium. D5.2 - Use case requirements. Technical report, The European Commission, 2014.

[2] The NetIDE (NetIDE) Consortium. NetIDE deliverable 5.1 - Use Case Requirements, Jun 2014.

[3] ETSI Network Functions Virtualisation Industry Specification Group. `http://www.etsi.org/technologies-clusters/technologies/nfv`.

[4] NetIDE Subversion Repository. `http://gforge.create-net.org/svn/netide`, 2014.

[5] NetIDE GitHub Repository. `https://github.org/fp7-netide`, 2014.

[6] NetIDE Subversion Repository: IRF Model in YANG. `http://gforge.create-net.org/svn/netide/Projects/YANG`, 2014.

[7] NetIDE Subversion Repository: UC1 implementation for Pyretic. `http://gforge.create-net.org/svn/netide/Projects/Pyretic`, 2014.

[8] Jeronimo Bezerra, Julio Ibarra, and Heidi Morgan. Use of sdn in the amlight intercontinental research and education network. `https://tnc15.terena.org/getfile/1938`, 2015.

[9] MiniNet team. Download/Get Started With Mininet. `http://mininet.org/download/`, apr 2015.

[10] OpenStack Open Source Cloud Computing Software. `http://www.openstack.org`, jun 2015.

[11] Dhritiman Dasgupta. Launching Applications to the Cloud – It's all about Scalability and Speed. `http://devops.com/category/blogs/devops-in-the-cloud/`, jun 2015.

[12] The NetIDE consortium. D3.2 - Developer Toolkit v1. Technical report, 2014.