## Abstract

For cloud computing to be successful in the future, PaaS frameworks for deploying cloud services are essential since service developers need a platform on which they can deploy their services.
Existing PaaS cloud frameworks do not support automatic handling of non-functional aspects, such as authentication. In most cases they are limited to a single platform and only allow the deployment of software written in a fine number of programming languages.
By building on the existing JSOA framework, it is possible to create a PaaS framework with automatic authentication handling, which runs on all major platforms and supports deployment of services written in all or most popular programming languages. In addition to allowing developers to extend the framework to support additional programming languages and platforms.
This thesis presents the extended JSOA framework. By taking advantage of its heterogeneity, scaling the framework to support multiple programming languages, while still allowing for further scaling, is achieved. The JSOA framework runs natively on the Windows and Linux platforms and automatically handles authentication independently of the deployed services.

# Preface

This report is written by Bergar Simonsen, at the IT-University of Copenhagen, as part of the bachelor thesis in Software Development. It contains work from February to May of 2014.

The main supervisor of the project has been Fabrizio Montesi.
A special thanks goes to Claudio Guidi, for providing assistance and expertise on Jolie and the JSOA framework.

The code for this project can be found on sourceforce:

Git repository containing the code for the JSOA framework:
`http://sourceforge.net/p/jsoa/code/ci/jsoa_windows/tree/`

SVN repository containing the code for the Jolie implementation of Windows, along with running examples of C# services and benchmarking results:
`http://sourceforge.net/p/jolie/code/HEAD/tree/experimental/jolie-windows/`

# Contents

# 1 Introduction

In recent years, cloud computing has become a hot topic in modern computer science. The ability to host a service in a single cloud somewhere on the internet is very appealing, and is a subject that is under constant research and development.
Cloud computing consists of multiple servers that are connected to each other via the internet. Combining all these servers, allows developers to deploy software which then can be accessed from anywhere in the world. These servers supply the processing power needed to perform certain tasks, freeing the clients from doing the heavy computing work which might needed for a service.

Cloud computing can be divided into three layers. Infrastructure as a service (IaaS), Platform as a service (PaaS) and Software as a service (SaaS)[1].
IaaS deals with the hardware infrastructure, such as the underlying server or virtual machine, network infrastructure and so on. PaaS is the computing layer. Operating system, programming environments, databases are all running on the PaaS layer. The final layer, SaaS, is the actual service running on the cloud framework. This could be anything from email server applications, games, and other software.

## 1.1 Structure of the thesis

This thesis is structured as follows:

- Section 1 gives an introduction to the thesis, followed by a problem description, aim of the thesis, thesis statement and finally the contributions of the project
- Section 2 provides some background information on cloud computing and the Jolie programming language
- Section 3 is the methodology used in the thesis. Technical analysis and implementation, followed by a running example and evaluation of the JSOA framework are analysed in this section
- Section 4 reports the conclusions and future work for the project

[1]see `section 2.1` for more details on cloud computing

## 1.2 Problem description

For cloud computing to be successful in the future, PaaS frameworks are essential since developers need frameworks on which they can deploy their software.
Currently there are a number of PaaS frameworks on the market. Microsoft Azure[1], Amazon EC2[2] and Google App Engine[3] are some notable examples. Although these examples are widely used, they do have some limitations.
The first limitation is that developers are tied to having their entire cloud at a single provider, e.g., these frameworks do not allow having the IaaS at one place, and have a PaaS framework from another provider running on top. Another limitation is support for multiple programming languages and platforms. Although these examples mentioned here do support multiple platforms and a range of programming language, they do not allow developers to extend the framework, e.g., developers can not add additional programming languages or additional features to the framework.
In addition to the limitations mentioned above, most of the time these frameworks give the developer a platform for deploying software and nothing else. Everything, including non-functional tasks such as authentication, need to be implemented for each service. Some PaaS frameworks (e.g., Google App Engine) do offer APIs for authentication which aid developers with the implementation, but they still need to be handled by each service.

To the best of my knowledge, there is no PaaS framework which runs on all major platforms (e.g., Windows, Linux and OSX) and supports all major programming languages[2], or at least have the ability to extend the framework to support a particular programming language. Neither is there a framework that supports those features mentioned here while also allowing the developer to abstract from non-functional tasks such as authentication.

**Problem statement**
Existing PaaS cloud frameworks do not support automatic handling of non-functional aspects, such as authentication. Neither do they support deployment and monitoring of services written in all major programming languages, or at least have the ability to be extended, by the developer, to support additional programming languages.

[2]This thesis defines popular programming languages as the major programming languages used today, e.g., Java, C#, Objective-C, C etc.

## 1.3 Aim

The goal of this thesis is to contribute to a PaaS framework which runs on all major platforms and supports the deployment and management of software written in different programming languages. Particularly the more popular programming languages are important in order to give the framework a commercial appeal.
The framework needs to be heterogeneous in the sense that the framework can be distributed across multiple servers, and thus allowing scaling of the framework to support additional programming languages.
In addition to supporting multiple programming languages and platforms, the framework should handle most of the non-functional and administrative tasks so that developers of software services can abstract away from these features. These tasks include monitoring services as well as authentication.

**Aim**
To contribute to a general purpose PaaS framework which supports automatic handling of non-functional aspects such as authentication and integrate it with commercial programming languages to give a commercial appeal to the framework, while still allowing further extension to support additional languages and platforms.

The methodology of this thesis will be to extend the existing JSOA[4] framework in order to create a PaaS framework which reaches the aim of the thesis.

There are a number of reasons why JSOA is a good starting point for developing such cloud frameworks:

- **Solid foundation.** The JSOA framework is built for running on the Linux platform. Since JSOA is written in Jolie, which is built on Java, the foundation is already in place to make it platform independent.
- **Working prototype.** JSOA is a working prototype which currently supports deployment of services written in Jolie, Java and Javascript. JSOA also currently has a working administration module so there is no need to implement this from scratch.
- **Heterogeneous.** JSOA supports heterogeneous computing in the way that a single JSOA central control panel supports multiple "domains", each running on the same server or even distributed across multiple

servers. This makes JSOA a good starting point for extending its functionality, while at the same time distributing it among multiple servers (see `section 3.1.1`).

- **Separate deployment from behaviour.** As stated, JSOA is built on Jolie. The very nature of Jolie makes it very easy to separate deployment from behaviour (see `section 2.2`). Because of this, Jolie does not differentiate between local communication, and communication with services running on remote servers. This makes Jolie, and by extension JSOA, ideal for developing distributed systems.

As stated above, the foundation of JSOA is already in place and therefore makes it an ideal starting point for this thesis.

## 1.4 Thesis statement

This thesis attempts to find a solution to the issue regarding cross-platform, multilingual PaaS cloud frameworks and to help service developers by automatically handling some non-functional aspects.

**Thesis statement**
By extending the JSOA framework and make it platform independent, and by taking advantage of it's heterogeneity, one can develop a PaaS framework which runs on all major platforms, supports the running of services written in all major programming languages and allow service developers to abstract away from non-functional aspects such as authentication.

More specifically, this thesis attempts to port the JSOA framework to the Windows platform, making it runnable on both Linux and Windows, and add the C# language to its list of supported programming languages.
In addition, an attempt to integrate authentication into the framework, freeing service developers from having to handle authentication in every service, is made.

## 1.5 Contributions

This thesis provides three main contributions.

**Windows and C# integration**
JSOA currently supports deployment of services written in Jolie, Java and JavaScript and runs on the Linux platform.
The first contribution is to integrate JSOA to the Windows platform. Although JSOA is built on Jolie which is platform independent, it is designed for running on Linux, and therefore there has not been any effort put into making it platform independent, so some changes need to be made to JSOA in order for it to run natively on Windows.
The first contribution is to integrate JSOA to run on the Windows platform, and support the deployment of services written in the .NET (C#) programming language.

**Authentication**
Currently, JSOA does not offer any authentication. All authentication for deployed services need to be implemented ad-hoc.
The second contribution will be to add authentication to the JSOA framework, allowing service developers to abstract away from handling any authentication on each service.

**Evaluation**
The third contribution is to develop a proof-of-concept service, written in C#. This proof-of-concept service will be used to evaluate the JSOA framework and demonstrate that the above contributions are met.

# 2 Background

## 2.1 Cloud computing

As stated in the introduction, a cloud system basically is collection of servers or virtual machines, connected by the internet, that form a "cloud" which services can be deployed on. This definition is very rough, so this section will give a more thorough definition of what cloud computing actually is.

The National Institute of Standards and Technology (NIST) defines cloud computing as:

> *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.*[5]

As stated by the NIST, cloud computing consists of *"five essential characteristics, three service models, and four deployment models."*[5] The following section will describe these characteristics, service models and deployment models.

**Characteristics**
In order to be qualified as a cloud system, there are some characteristics which need to be fulfilled in the system.

- ***Always on***
  The first of these characteristics is that the service is always running. What this means in practice is that a user can, at any time, connect to the service in order to use storage, processing power, or other resource without requiring any human interaction.

- ***Broad network access***
  The capabilities (or resources) of the service are available through the internet at any time. In addition, they are available through any standard device that has access to the internet (e.g PCs, mobile phones and PDAs)

- ***Resource sharing***
  Another core characteristic of cloud computing is resource sharing.

The cloud is organized in such a way that multiple clouds or services running in the cloud share a large resource pool. This resource pool is then assigned to which ever service requires it. For example if a service is attracting more users at a certain time, additional resources from the resource pool are assigned to the service. When the demand falls back to normal, the resources are put back in the pool.

- ***Elasticity***
  Elasticity means that a service can expand or shrink at any time.
  When a service is deployed in the cloud, the deployer does not have to allocate a specific amount of storage or bandwidth to the service but these can be expanded at any time.

- ***Monitoring***
  Cloud systems monitor their running service in order to determine how many resources are required. Monitoring is essential for *Resource sharing* and *Elasticity* to work properly without human interaction.

**Service models**
As also stated in the introduction, cloud computing can be divided into three layers, or service models:

- ***Infrastructure as a service (IaaS)***
  IaaS' supply consumers with computing power. This includes processing power, storage, network and other computing resources. The consumer can then deploy and run software on this service. An example of software running on IaaS is the operating system.

- ***Platform as a Service (PaaS)***
  The primary task of the PaaS is to provide the deployment environment to the consumer. Deployment environment include a framework for running software, programming languages and compilers, databases, and other related software.
  The PaaS abstract away from the underlying infrastructure, but often has some control of the service deployed on top of the PaaS. Monitoring of running services is an example.

- ***Software as a Service (SaaS)***
  SaaS is the actual cloud service deployed on the cloud framework. This service could be anything from an email application to social networks. These services are available for users to interact with, through their clients (PC and mobile phone), via the internet.

**Deployment models**
Deploy models define how a cloud is deployed and to what extent users have access to the cloud system.

- ***Public cloud***
  This is the most common deployment model. Public clouds are available for all users who have access to the internet.

- ***Community cloud***
  The community cloud is a cloud that is owned by several organizations or companies. An example of this could be a large company, that consists of several smaller companies, where all share a cloud for internal storage and communication.

- ***Private cloud***
  The private cloud belongs to a single organization or company. The private cloud is inaccessible for anyone except the owner. This deployment model is often used when the owner deals with sensitive data, but needs to share the data internally.

- ***Hybrid cloud***
  Hybrid clouds are a combination of the other three deployment models. For example, if a large company which owns a lot of smaller companies, needs allow the smaller companies to see data, but not be able to edit it, a hybrid cloud is ideal. The cloud would be a community cloud in the sense that multiple companies share the same cloud, while being a private cloud at the same time since only the head company can edit the data.

## 2.2 Jolie

Jolie is a service-oriented programming language built on Java.
What makes Jolie an ideal choice for programming distributed systems, is its ability to separate deployment from behaviour.
This section will describe some of the important Jolie features which will be used in this thesis. For more information about Jolie, see [6].

A standard Jolie service is divided into two parts, the deployment, which deals with how the service is deployed and how it communicates with other services. The second part, the behaviour, deals with what the service does, e.g., the actual work performed by the service.

Listing 1: Separation of deployment and behahaviour in Jolie

```
1 inputPort inputPort_1 {
2        Location: "socket://localhost:9999"
3        Protocol: sodep
4        Interfaces: interface_1 }
5 main { /* some code */ }
```

`Listing 1` shows an example Jolie service. Lines 1 - 4 show an input port, which is the deployment part, while line 5 shows the main operation, which is the behaviour part.
In the example above, the input port can be considered to be like a `socket listener` [7] in Java. This analogy is not 100% correct since Jolie input ports support multiple technologies (e.g., bluetooth, local memory etc.). What they all do have in common is that the input port listens for incoming connections at the specified location.
This particular input port listens for request on localhost, port 9999 and uses the Sodep[8] protocol (lines 2-3), and implements the interface interface_1 (line 4). A Jolie output port has the same syntax as the input port, only the *inputPort* keyword is changed to *outputPort*.
The second part, line 5, is the behaviour of the service. This contains the actual work the service performs.
`Listing 1` demonstrates one of the major strengths of Jolie. Not only does it elegantly separate the deployment from the behaviour, but it also completely abstracts from which technology is used. Whether it communicates with a file in the same folder, a remote server or a bluetooth device, it makes no difference to the input port, or the rest of the service.

### Data types

Data types in Jolie are designed as an XML like tree structure.

Listing 2: Jolie data types

```
 1  type DataType: void {
 2          .id: int
 3          .value: int
 4          .domain: void
 5                  .address: string
 6                  .port: int }
 7
 8  dataType.id = 1;
 9  dataType.value = "some value";
10  dataType.domain.address = "localhost";
11  dataType.domain.port = "8000"
12
13  <DataType>
14          <id> 1 </id>
15          <value> some value </value>
16          <domain>
17                  <address> localhost </address>
18                  <port> 8000 </port>
19          </domain>
20  </DataType>
```

`Listing 2` shows a Jolie data type and the same data type implemented in XML. This structure allows Jolie to easily implement complex, tree like data structures.
Lines 1 - 6 show how the data type is structure. Lines 8 - 11 show how the data type is used in practice. Finally, lines 13 - 20 show how the same data type would be implemented in XML.

In Jolie, every data type is a dynamic array. In practice, this means that there is no syntactical difference between a ”regular” data type, and a list. Note that line 8 in `Listing 2` implicitly is referring to the first element in the dynamic array, e.g., datatype.id = 1 is the same as datatype.id[0] = 1. The only difference between a list and a single object in Jolie, is in the type definition.

Listing 3: Jolie dynamic array

```
type Person: void { .name: string, .phoneNumber*: string }
```

`Listing 3` shows the syntactical difference between a single value and a dynamic array. The Person type has a name (string) and any number of phoneNumbers, specified by the * notation.

### Embedding

Jolie supports embedding of other services. This means that when developing a Jolie service, the developer can embed another service inside that service, which allows the running of a service as a sub-service, creating a hierarchy of services. Embedding currently supports services written in Jolie, Java and JavaScript.
`Listing 4` shows how embedding in Jolie is used. Line 1 shows the syntax, while line 2 shows a specific example of how embedding is used.

Listing 4: Jolie embedding

```
1 embedded { Language : path [ in OutputPort ] }
2 embedded { Jolie: "service_1.ol" in outputPort_1 }
```

### Aggregation

Aggregation in Jolie allows the creation of proxy services that can forward invocations to other services. Aggregation is purely related to deployment and has nothing to do with the behaviour of the service. Aggregation allows an input port to aggregate an output port, creating a bridge between them.

Listing 5: Jolie aggregation

```
outputPort outputPort_1 {
       Location: "socket://localhost:8991"
       Protocol: sodep
       Interfaces: Interface_1 }

inputPort inputPort_1 {
       Location: "socket://localhost:8009"
       Protocol: http
       Aggregates: outputPort_1 }
```

`Listing 5` shows an example of aggregation. When a request is sent to `inputPort_1` via the http protocol, the request is forwarded to the aggregated `outputPort_1` via the sodep protocol.
Using aggregation makes it easy to develop large, distributed systems. For example, `inputPort_1` from the example above could aggregate multiple output ports, allowing the input port to forward requests to multiple different location with just a single line of code.

The following example (`Listing 6`) demonstrates how embedding and aggregation works in Jolie.

Listing 6: Aggregation and embedding example

```
// Service A
outputPort B {
        Location: "serviceB.location"
        Protocol: http Interfaces: Interface_B }
outputPort C {
        Location: "serviceC.location"
        Protocol: soap Interfaces: Interface_C }
outputPort D { Interfaces: Interface_D }
embedded { Jolie: "serviceD.ol" in D }
inputPort A {
        Location: "socket://sample.location:99"
        Protocol: sodep Aggregates: B, C, D }
```

Assume that we have four services A, B, C and D. Services B, C, D are independent services and service A is described above (`Listing 6`)
As seen in `Listing 6`, service A has three output ports, one for each service, and a single input port. Output port C and B are ”regular” output ports, while output port D *embeds* a service (service D). Input port A aggregates all three output ports.

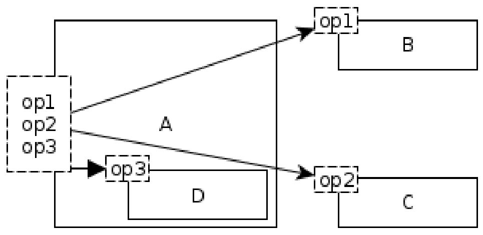

Figure 1: Aggregation and embedding

`Figure 1` shows how the communication flow is. A client sends a request to service A with three operations. Op1 is forwarded to service B, op2 forwarded to service C and op3 forwarded to service D via their respective output ports (`Listing 6`).
Notice that the aggregator service completely abstracts away from the fact that output port D embeds an independent service. Aggregators only deal with output ports and do not care how the service is structured.
Using aggregators allows Jolie to build large, distributed systems with very

low coupling between modules.

**Courier**

Courier sessions allow the composition of services independently from the context they belong to. Courier session can add extra procedures (for example authentication) to any service, without having any effect on the service itself.

Couriers rely on aggregation and allow the aggregator to handle a special session for the incoming request. Couriers allow services to overload operations, and add extended data to the request.

Listing 7: Jolie courier

```
1  type AuthenticationData: void { .key: string }
2
3  interface extender AuthInterfaceExtender {
4         RequestResponse: *( AuthenticationData )( void )
5         OneWay: *( AuthenticationData ) }
6
7  outputPort outputPort_1 {
8         Location: "socket://localhost:8991"
9         Protocol: sodep
10        Interfaces: Interface_1 }
11
12 inputPort inputPort_1 {
13        Location: "socket://localhost:8009"
14        Protocol: http
15        Aggregates: outputPort_1 with AuthInterfaceExtender }
16
17 courier inputPort_1 {
18        [ interface Interface_1( request )( response ) ] {
19        // some code
20        forward outputPort_1( request )( response ) } }
```

`Listing 7` shows an example of a courier sessions in Jolie.

`AuthInterfaceExtender` is a special type of interface which extend an existing interface. E.g if an interface has a request with the type int, the request will be extended to contain `AuthenticationData` in addition to the original request.

Notice the *with* keyword (`Listing 7:15`). The with keyword allows the input port to aggregate an output port and at the same time, extend the interface of the aggregated output port with the given interface extender (AuthInterfaceExtender).

When the request is received by `inputPort_1`, the courier session starts. The courier handles some code (this could for example be authentication), and then the request, without the extended data, is forwarded to the aggregated output port.

# 3 Methodology

## 3.1 Technical analysis & implementation

This section will describe and analyse the technical contributions of the thesis.
The section is divided into four sub sections. First there is a description of JSOA in its current state and how it works. Followed by the technical changes made to JSOA, and the underlying Jolie code in order to get JSOA to run on Windows. The third section describes and analyses the C# integration. This section finishes with an analysis and implementation of the authentication.

### 3.1.1 JSOA

Before discussing the technical changes that have been made to the JSOA framework, a short description on how JSOA works under the hood will make it easier to describe and understand the changes that have been made.
This section is divided into two parts. The first part will explain how JSOA is structured and what options there are for extending it. The second part will show the message flow when sending requests to JSOA.

**JSOA structure**

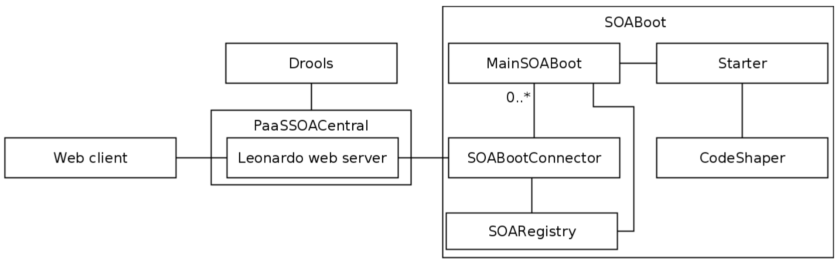

Figure 2: JSOA architecture

As seen in `Figure 2` the architecture of JSOA can be roughly divided into three services (excluding the web client). These services are: Drools, PaaSSOACentral and SOABoot. This section will explain briefly what each service does, and how they communicate with each other.

**Drools**
When JSOA starts, the first module to load is Drools.

Drools has two main features. The first is to load some libraries which JSOA requires in order to run. The second is event logging. Every time an event happens somewhere in JSOA, it is sent to Drools. Drools also contains a database which meta information about all running services, such as name of deployed input and response time treshold.

**PaaSSOACentral**
PaaSSOACentral is the central module of JSOA. After Drools has loaded, PaaSSOACentral is the next thing to start.
When the main PaaSSOACentral is started, it starts up the Leonardo[9] web server. In addition to the web server, PaaSSOACentral also starts up the SOARegistry, which is used for registering meta information of services (e.g., input and output ports), and the SOABoot connector. The SOABoot connector is used for connecting the Leonardo web server to the main SOABoot. Leonardo acts as a front end to the SOABoot service. When a service is uploaded to JSOA and the "addService" request is received, the web server validates the request, and forwards it to the SOABoot connector. The SOABoot connector fetches meta information for the service from the SOARegistry, and forwards the request to the SOABoot.

**SOABoot**
SOABoot is the core service in regards to handling deployed services. When a service is uploaded, started, stopped or deleted, SOABoot is in charge of carrying it out.
As seen in `Figure 2`, SOABoot is divided into five main sub services. MainSOABoot is the main SOABoot service, SOABootConnector is connecting the PaaSSOACentral with the SOABoot and SOARegistry is registering various meta information regarding services. Starter is the service starter service, when a service is stared on JSOA, the Starter is in charge of executing the start commands. The CodeShaper is a sub service of the Starter. When JSOA starts a service, various Jolie service files are generated which JSOA uses to monitor and interact with the service. These files include a surface file, and a wrapper file.
In addition to these sub service, SOABoot also connects to a database where every service that is uploaded to JSOA is registered. This data base contains name and status of the service, email address of the service administrator and file name and extension of the uploaded service file.

**Heterogeneity**
Notice that in `Figure 2` that PaaSSOACentral coordinates zero or more SOABoots. This is what makes JSOA excel at extending the framework to a

large distributed system, since the developer can easily scale the framework by adding additional SOABoots.
As stated above, PaaSSOACentral is the main module of JSOA. When a SOABoot is started, it is attached to the PaaSSOACentral, and when a service is uploaded to JSOA, it is attached to a specific SOABoot.
`Figure 3` shows a visual representation of how PaaSSOACentral, SOABoot and services are connected.

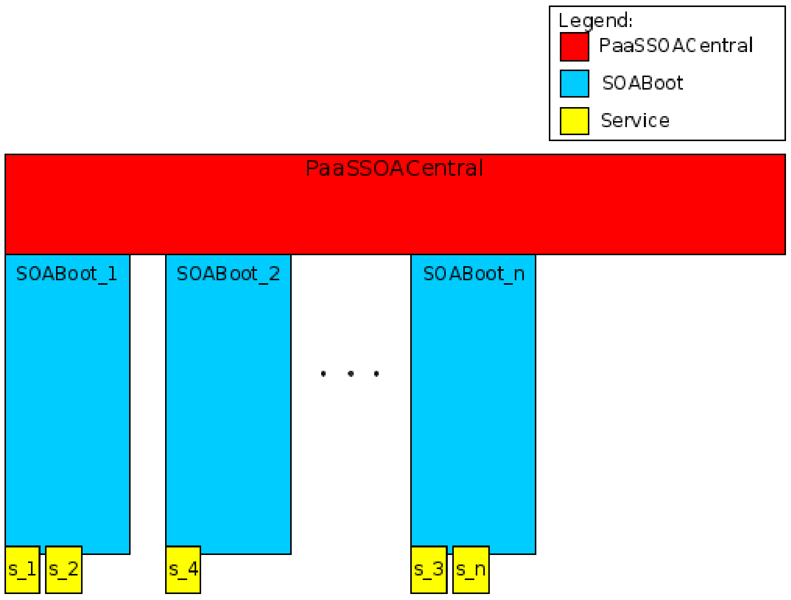

Figure 3: JSOA with multiple SOABoots

When a new SOABoot instance is created, it is connected to a single PaaS-SOACentral. When a service is uploaded to JSOA, it is connected to one SOABoot. A single SOABoot can manage many services.
Each SOABoot can be thought of as a domain. So if `Figure 3` was a web-site, with multiple web services running, each SOABoot can be considered a domain or sub domain of the web site, or one SOABoot could even be a whole different web site that is hosted on the same server.

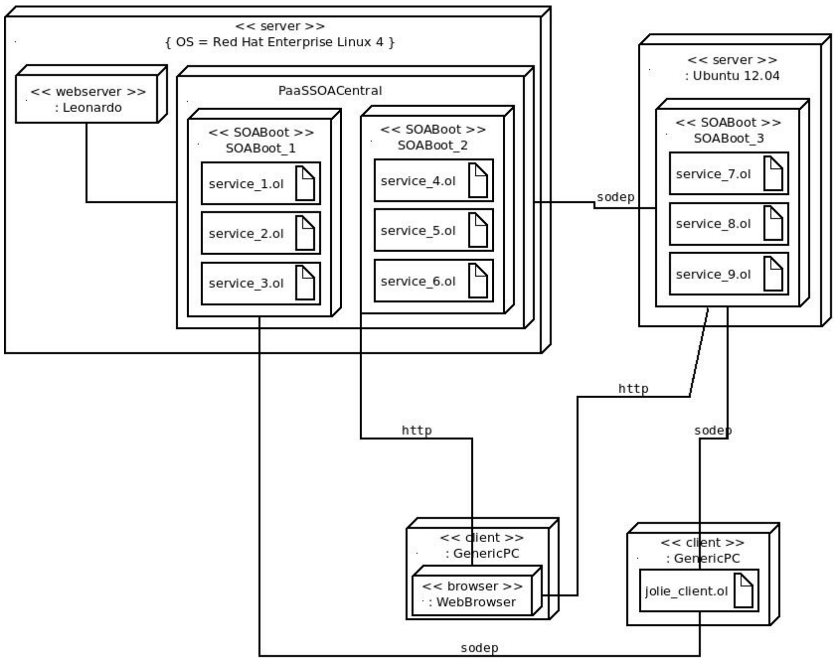

Figure 4: JSOA deployment diagram

`Figure 4` shows a possible deployment diagram for JSOA.
As shown in the diagram, the Red Hat Linux server runs PaaSSOACentral and has two SOABoots running with three services running on each SOABoot. The Ubuntu server has one SOABoot running, with three services running. The SOABoot on the Ubuntu server is connected to the PaaSSOACentral of the Red Hat server.
The two clients interact with the services on both servers, via their respective SOABoots.

This diagram shows how scalable JSOA is. Imagine that the SOABoots on the Red Hat server allows developers to deploy services written in Python, while the Ubuntu server allows the running of Jolie and Java services. One could add another SOABoot, either on one of the existing servers or a new server, which allows the running of software written in PHP and Ruby. Each of these SOABoots could share the same authentication procedure or alternatively they could all have their own.
The diagram gives a good picture on how well JSOA handles scalability, with the ability to build large distributed frameworks, while still coordinating it all from a central location.

**Adding a service**

There are two ways a user can upload a service to JSOA. The first way is to upload a single Jolie (<servicaName>.ol) file. The second way is to use a `jap` file.

A jap file essentially is a jar file which is recognized by JSOA. The jap file can contain a single Jolie service file, or it can contain a more complex service with multiple folders and files. If there is more than one file, or the main Jolie service does not have the same name as the jap file, JSOA requires that a manifest file is present.

Listing 8: Jap file manifest

```
Manifest-Version: 1.0
X-JOLIE-Main-Program: main.ol
X-JOLIE-Options: Options here
```

`Listing 8` shows how a jap manifest is structured.

The first line is just auto generated information. Line two tells JSOA where the main Jolie service file is found (e.g., main.ol). Line three is for additional options, e.g., command line arguments for the main Jolie service.

When a jap file is uploaded to JSOA, it first looks for the manifest file in order to find the main jolie service, this service is then registered in JSOA. If there is no manifest, JSOA tries to find the main service with the same name as the jap file.

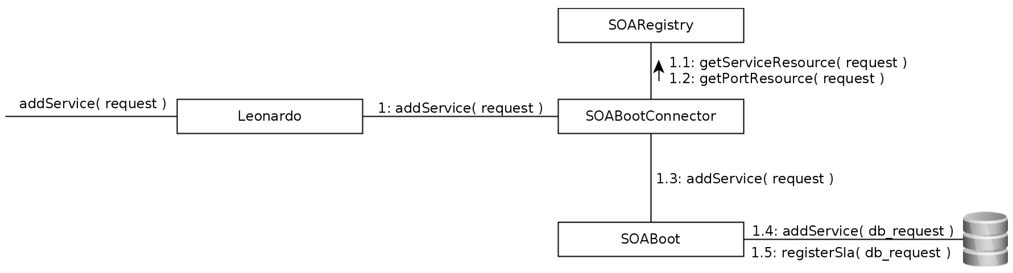
SOARegistry
1.1: getServiceResource( request )
1.2: getPortResource( request )
addService( request )
Leonardo
1: addService( request )
SOABootConnector
1.3: addService( request )
SOABoot
1.4: addService( db_request )
1.5: registerSla( db_request )

Figure 5: Add service

`Figure 5` shows a communication diagram, describing what happens when a user uploads a service to JSOA.

The first step is that the web client sends the uploaded file, along with the name of the service and the email address of the administrator, to the Leonardo web server. The web server checks the uploaded file to see if it is valid. Checks are made to make sure that the file type is valid and has the correct extension. After the file has been confirmed to be valid, Leonardo parses the file to see if the syntax is correct and all required attributes (e.g.,

input port and main function) of the service are in place. If any of these tests fail, a FaultException is sent back to the client.
Assuming that all tests have passed, Leonardo creates a request, based on the uploaded service, and sends it to the SOABootConnector.

The SOABoot connector readies the service for deployment. Service related resources are fetched from the SOARegistry (e.g location and domain of the SOABoot). The SOABoot parses the input file in order to the get input/output ports of the uploaded service, and sets the location (domain) according to the location/domain of the SOABoot.
With this information in place, a new request is created and sent to the main SOABoot.

The main SOABoot is where the service is actually added to JSOA.
The first thing that happens is that the SOABoot checks if the service exists. Since JSOA requires that services have unique names, SOABoot checks the service database to see if a service already exists with the same name as the newly uploaded service. Assuming that it does not exist, SOABoot creates a working directory for containing the uploaded file, and writes the uploaded file to the directory. After this, a few more checks are made to make sure the service file is valid and does not contain any error.
Assuming the file is error free, the next step is to register the service in the database. The service is added to the service database of the SOABoot. Some additional information is also inserted into the SLA database of the Drools.
Finally, a console directory is created for the service. This directory is used for logging interactions with the new service.

## Starting / deploying a service

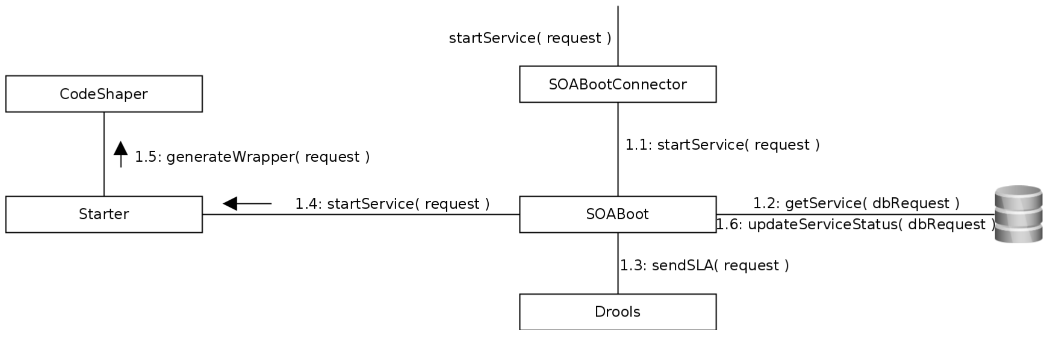
startService( request )
SOABootConnector
CodeShaper
1.1: startService( request )
1.5: generateWrapper( request )
1.4: startService( request )
Starter
SOABoot
1.2: getService( dbRequest )
1.6: updateServiceStatus( dbRequest )
1.3: sendSLA( request )
Drools

Figure 6: Start service

As `Figure 6` shows, the communication flow of the start service action is very similar to when adding a new service (`Figure 5`).
The web client sends start command to the SOABoot connector. The SOA-Boot connector, like with the add service, fetches information about the service from SOARegistry (e.g., ports), and sends the request to the main SOABoot.

SOABoot again makes various tests to see if the service is valid.
Information about the service is fetched from the database, and the service file is parsed by the SOABoot. The parsing of the file includes setting the location of the input ports, based on the SOABoot domain. Updated information about the service is sent to Drools.
After this, the request is sent to the main starter which handles the actual work on deploying the service.

The startService on the main starter first creates a deployer directory for the service. This directory contains all files related to the deployed service as long as the service is running (the directory is deleted after the service is stopped, and recreated when the service is restarted). The main starter then starts to "unwrap" the service. Here there are two options. The first is if the service is a Jolie file (e.g., <serviceName>.ol file) or the service is a jap file. If the service is a jap file, the first thing that the starter does is copy the jap file into the newly created deployer directory where the jap is extracted. The next step is to find the main Jolie file. This has been specified in the manifest file, so the starter parses the manifest file in order to find the main Jolie file, along with some extra (optional) parameters (e.g., command line arguments for the main Jolie file).

After these steps, the rest is handled like a regular Jolie file.

At this stage, JSOA writes a new file based on the uploaded service. This file is the upload service, with the addition of a few input ports and operations which allow JSOA to interact with the service. These operations include a monitor which allows JSOA to get statistics and consoles from the running service.
After writing the "monitor-file" JSOA generates a "surface" file. This file is basically a modification of the uploaded service. The changes are made to make it easier for JSOA to fetch input ports and interfaces by altering the names and merging all interfaces to a single interfaces. This surface file is what is exposed to the clients.

After the surface, JSOA generates a wrapper file for the service. The wrapper file contains input- and output ports which allows JSOA to interact with the running service. In addition, the wrapper file contains some additional functions, the "shutDown" is an example. The wrapper file is in a way the central file of the service. JSOA communicates with the wrapper file in order to interact with the service and the monitor file. The wrapper file is also the entry point for when clients are using the service. The main input port of the wrapper file embeds the uploaded service, thus allowing clients to interact with the service via the wrapper file.

Having generated these files, the service is registered in the SOARegistry. JSOA then fetches the input ports to be deployed and sets the location of the input port based on the current SOABoot domain.

The actual deployment phase relies on executing shell[10] commands.
The first command is to generate joliedoc[3] for the service. Joliedoc generates an html files which contains the API for that particular service.
JSOA then creates a shell script which is in charge of starting the service as well as creating log file for the service.
Finally, the shell file is executed, and the service is running.

After the service has started, the starter returns back to the main SOABoot. The SOABoot updates the database with information that the service is now running. An entry is also added to the Drools database, informing it about the deployed input port.

[3]Example of joliedoc for the "console.iol" service:
http://www.jolie-lang.org/?top_menu=documentation&sideMenuAction=jsl/Console

**Stopping a service**
Stopping a running service on JSOA is more or less the opposite of starting a service. The web client sends a stop request to the SOABootConnector, which validates the request and passes it on to the main SOABoot.
The main SOABoot fetches the service meta data from the SOARegistry (e.g., ports). JSOA then sets the location of the Admin input port of the wrapper file (the one which is in charge of shutting down the service) and executes the shutDown command.

After shutting down the service, the service (along with its ports) are removed from the SOARegistry.
When the service is remove from the registry, the deployment files and directory are removed. The stop event is sent to Drools to inform that the service has been stopped, and the SOABoot database is updated with information about the new state of the service.

### 3.1.2 JSOA on Windows

As stated in the problem description, the first contribution to the project is to port JSOA to the Windows platform. Since JSOA is based on Java, the foundation was already in place for porting it to Windows.
Before JSOA can run on Windows, some changes need to be made. Most of the changes are not related to JSOA, but with the underlying Jolie code. The first change is regarding Windows file paths.

**Issue #1: Windows file paths**
JSOA is written for the Linux platform, and on Linux the standard file path uses forward slashes (`/some/path/`) but on Windows the default path uses backward slashes (`C:\some\path\`). When the Jolie interpreter is parsing a file path, an exception is thrown because of an illegal index in the file path (e.g., the backslash).

The issue appears when JSOA tries to access some files, and the Java code will not accept the backslashes.

Listing 9: Windows file path issue

```
// Linux ( original )
String filename = new File(argsList.get( i )).getCanonicalPath();
// Windows work around 1
String filename = new File(argsList( i )).toURI().toString();
```

```
5 // Windows work around 2
6 filename = filename.replace( "\\", "/" );
```

As seen in `Listing 9`, there are two ways to solve this issue. The first solution is to get the URI of the file, and use the URI as the file path (line 4). The second solution is simply to replace the backslashes with forward slashes (line 6).

The second solution is used for two reasons. The first is that it is very simple, and does not need to alter the existing code but just add a single line of code after the original. The second is that this solution also works if given only a file path (e.g., a String object, not a File object). If the first solution was used then first required to create a file based on the file name, and then implement the solution.

One of the drawbacks of this solution is that the original file path might contain backslashes (as an escape character for white spaces for example). This should not happen, because Java deals with the white spaces, but only extensive testing, which lies outside of the scope of this thesis, can verify if this will happen or not.

**Issue #2: Jap relative file paths**
The second issue is related to relative file paths and jap files.
Jolie services, like most other programming languages, use so called "include" statement to import code from other files in to the current file. An example of an include statement which includes a file from its parent folder is presented in `Listing 10`

Listing 10: Jolie include statement

```
include "../someFile.iol"
```

In a regular Jolie service, this does not pose any problems, even on Windows. The problem is when using JSOA jap files.

`Figure 7` shows the structure of a file system. Current folder is the folder that we are currently working in. Jap file is the jap file that is uploaded to JSOA. Inside the jap file are two Jolie files, jolie_file1 and jolie_file2 and an additional folder folder 1. Inside folder 1 are two additional Jolie files, jolie_file3 and jolie_file4.

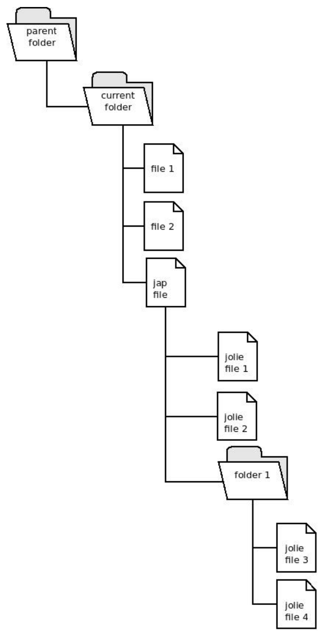

Figure 7: Jap file include issue

The jolie_file3 starts with some include statements:

Listing 11: Include statements

```
1 include "jolie_file4"
2 include "../jolie_file1"
```

The first line executes as it should, since the included file is in the same directory as the source file. However, the second line poses an issue.
Ordinarily, the statement "../" "means go back one folder". In Windows, a jap file is not considered to be a folder and therefore produces unexpected results. Instead of going from Folder 1 and back to Jap file to look for the file to include, it assumes that we are in the "Current folder" and therefore goes back to the "parent folder" to look for the file to include. Since the target file doesn't exist in the Parent folder, a null pointer exception is raised.
In order to get the correct file to include, the parser first needs to get the absolute path of the source file (jolie_file3). When the absolute path is found, the parser will then create new absolute paths to the include files.

The code snippet in `Listing 12` creates absolute file paths based on the relative file paths in the include statements.
The path (line 2) is the absolute path to the Jolie file
(e.g., `C:\some\path\jolieService.ol` ), while the filename (line 3) is the filename to include (e.g., `../someOtherJolieFile.ol`).
Lines 4-5 convert the path to work with JSOA (replace backslashes with forward slashes). Lines 6 - 11 are creating the actual absolute path. As long as the filename starts with "../", it is removed from the filename, at the same time the last folder is removed from the path. Finally the filename is appended to the path, creating the absolute path to the include path.
Lines 20 - 34 do the same thing, only when the include file starts with "./" (e.g., this folder).

Listing 12: Fix for jap file include paths

```
1   if(filename.startsWith( "../") ) {
2           String tmpPath = path;
3           String tmpFilename = filename;
4           if(!tmpPath.contains( "/" ) && tmpPath.contains( "\\" ))
5                   tmpPath = tmpPath.replace( "\\", "/" );
6           while(tmpFilename.startsWith( "../" )) {
7                   tmpFilename = tmpFilename.substring( 2 );
8                   if(tmpPath.endsWith( "/" ))
9                           tmpPath = tmpPath.substring(0,
                                tmpPath.length() -1);
10                  tmpPath = tmpPath.substring(0, tmpPath.lastIndexOf(
                       "/" ));
11          }
12          String tmpUrl = new StringBuilder()
13                  .append( tmpPath )
14                  .append( tmpFilename ).toString();
```

```
        try {
                url = new URL(
                        tmpUrl.substring( 0,4 ) +
                        tmpUrl.substring( 4 ));
        } catch(Exception exn) { }
        } else if(filename.startsWith( "./" )) {
                String tmpPath = path;
                String tmpFilename = filename;
                if(!tmpPath.contains( "/" ) && tmpPath.contains(
                    "\\" ))
                        tmpPath = tmpPath.replace("\\", "/");
                tmpFilename = tmpFilename.substring(1);
                if(tmpPath.endsWith( "/" ))
                        tmpPath = tmpPath.substring(0,
                            tmpPath.length() -1);
                String tmpUrl = new StringBuilder()
                        .append( tmpPath )
                        .append( tmpFilename ).toString();
                url = new URL(
                        tmpUrl.substring( 0,4 ) +
                        tmpUrl.substring( 4 ));
}
```

**Issue #3: Jolie built-in includes**
This issue arises when a service is deployed, and JSOA parses the uploaded file for the input ports of the service. As stated earlier, JSOA generates a wrapper file for the uploaded service. When generating this file, JSOA takes advantage of some libraries that are built-in to Jolie.
When parsing "regular" include statements (such as the one in `Listing 13`) the, MetaJolie[4] class first checks the current directory to see if the file is found. If not, then it checks the built-in Jolie library. The issue here is that the default Jolie installation directory on Linux (e.g., /opt/jolie/include/) is hard coded into the MetaJolie parser.

Listing 13: Jolie regular include statement

```
include "console.iol"
```

Since this is a fundamental bug with Jolie, and therefore outside the scope of this project, a temporary fix is put in place by adding the default Windows installation directory to the MetaJolie parser. This will of course cause

[4]MetaJolie is part of the Jolie installation. The MetaJolie class is in charge of the meta information of services, one of which is the include statements

the same problem if Jolie is installed in any directory other than the default, but this will be fixed in a later version of Jolie so the problem will be resolved.

### Issue #4: Shell commands

The final thing which needs adapting to the Windows platform is the execution of services and various commands.
As stated in `section 3.1.1` the Starter executes shell commands when starting services. These commands need to be fitted for the Windows command prompt. When starting a service, JSOA creates a bash[5] script which is in charge of starting the services, as well as creating directories and log files. In addition to starting services, JSOA uses the shell for extracting the content of jap files and generating API documentation for the service.
`Listing 14` shows the differences between creating and executing start commands in Linux and Windows.
Lines 1 - 11 show how the shell (Linux) commands are created and executed, while lines 13 - 23 show the same thing on the command prompt (Windows).

Listing 14: Linux & Windows starter commands

```
1  // create shell command
2  with( shell_file ) {
3         .filename = current_dir + "/run.sh";
4         .content = "#! /bin/sh\nnow=\"$(date
             +\"%y%m%d%H%M%S\")\"\nlogfile=../../consoles/" +
             service_name + "/$now\"_log.txt\"\njolie " + session +
             "_wrapper.ol > $logfile 2>&1" };
5  writeFile@File( shell_file )();
6
7  // execute shell command
8  cmd = "sh";
9  cmd.args = "run.sh";
10 cmd.workingDirectory = current_dir;
11 exec@Exec( cmd )( result )
12
13 // create Windows command
14 with( shell_file ) {
15        .filename = current_dir + "/run.bat";
16        .content = "set now=%date%\nset logfile=..\\..\\consoles\\"
             + service_name + "\\%now%_log.txt\njolie " + session +
             "_wrapper.ol > %logfile% 2>&1" };
17 writeFile@File( shell_file )();
18
```

[5]Born again shell. Referring to the Linux default shell.

```
// execute windows command
cmd = "cmd";
cmd.args = "/c start run.bat";
cmd.workingDirectory = current_dir;
exec@Exec( cmd )( result )
```

In addition to adapting the shell commands to run on Windows, new commands needed to be added in order to start the C# service.
When the starter is extracting the jap file, additional code needed to be added in order to find the C# service file.

Listing 15: Find .exe file in jap

```
1  undef( rq );
2  rq.filename = current_dir + "META-INF/MANIFEST.MF";
3  readFile@File( rq )( manif );
4  undef( split_req );
5  split_req = string( manifest );
6  contains_req = split_req;
7  contains_req.substring = "X-JOLIE-CSHARP-PROGRAM:";
8  contains@StringUtils( contains_req )( contains_resp );
9  if( contains_resp ) {
10         split_req.regex = "X-JOLIE-CSHARP-PROGRAM:";
11         split@StringUtils( split_req )( split_res );
12         split_req = split_res.result[ 1 ];
13         split_req.regex = "\n";
14         split@StringUtils( split_req )( split_res );
15         trim@StringUtils( split_res.result[ 0 ] )( cSharpFile );
16         winServiceFilename = current_dir + cSharpFile }
```

`Listing 15` shows the additional code needed to find the .exe file. Lines 1 - 8 reads the file and split it by the ”X-JOLIE-CSHARP-PROGRAM:” string, which has been specified in the manifest. If it finds anything, lines 9 - 16 split the string further to find the filename of the .exe file.
If the C# service is found, it needs to be started when the Jolie proxy is started.

Listing 16: Start C# service

```
// Create and start the c# application
   exists@File( winServiceFilename )( winServiceExists );
   if( winServiceExists ) {
      undef( cmd );
      cmd = "cmd";
      cmd.args = "/c start " + winServiceFilename;
```

```
        cmd.workingDirectory = current_dir;
        exec@Exec( cmd )( result ) };
```

`Listing 16` shows how the C# service is started. Notice that it is almost identical to starting a Jolie service.

**Issue #5: Empty lines in services**
One final note regarding JSOA on Windows.
When a Jolie service is uploaded to JSOA. it is required that the service file does not contain any empty lines. The reason for this is that when Jolie is parsing the service file, it considers empty lines as end of file, and stops reading.
There seems to be an issue with how Windows and Java file readers work together (since this problem does not exist on Linux). To counter this issue, it requires that service developers to not have any empty lines in their Jolie services, at least until the Jolie/Java parser is fixed.

### 3.1.3 C# Integration

The second part of the first contribution to the project is to add the ability to run services written in C# to run on JSOA and integrate C# to communicate with Jolie.
In order to integrate Jolie with C#, there are three things that need to be implemented into C#: Jolie values, the Sodep protocol and input/output ports. This section will go through the implementation of all three, along with examples on how they are used.

**Values**
Jolie types are tree like structures (see section 2.2). The types have a root value (which may be empty) and an arbitrary number of children. The children have a value (again, can be empty) and potentially more children.

Listing 17: Jolie type

```
type Person: void {
        .firstName: string
        .lastName: string
        .address: void {
                .streetName: string
                .streetNumber: int
                .zipCode: int
                .city: string } }
```

`Listing 17` shows an example of a person type in Jolie.
The root value (Person) is void and contains three children, firstName (string) lastName (string) and address (void). Address has four children, streetName (string), streetNumber (int), zipCode (int) and city (string).
The structure of the Jolie types makes it very easy for creating large, complex types. The underlying Value implementation however, is slightly more complicated.

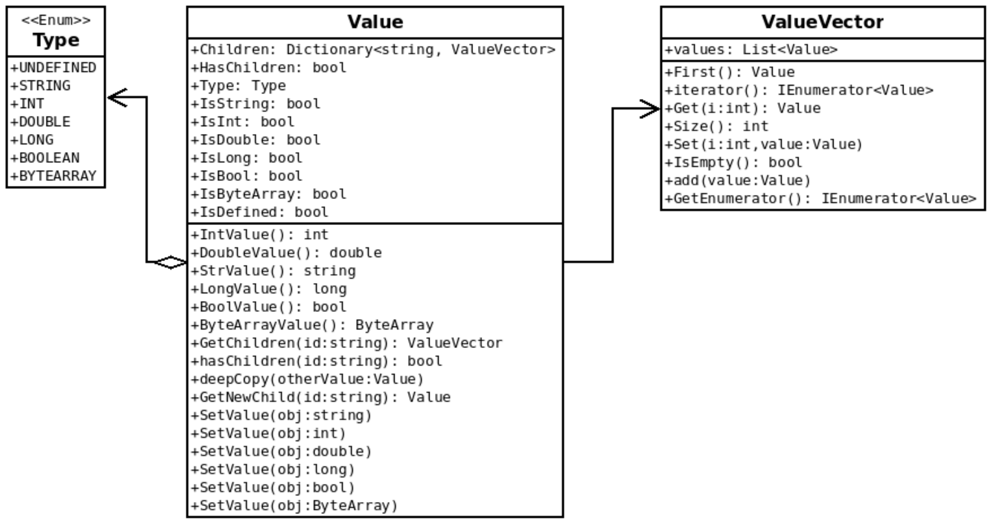

Figure 8: Excerpt from C# class diagram

`Figure 8` shows part of the class diagram for the C# integration. As the figure shows, Value has a `type`, this is the type of the data (e.g string, int etc.) The Value also contains a dictionary containing the children of the Value. In the example in `Listing 17`, this would be firstName, lastName and address. The dictionary does not contain the child value directly but contains a ValueVector which in turn contains a list of values. The reason for the ValueVector containing a list instead of a Value object, is that in Jolie, any data type is a dynamic array (see `section 2.2`).
The rest of the operations on the value objects are used for various checks and to get and set the data of the value. In addition, Value contains a method for copying the Value into another value object (deepCopy[6]).

[6]This is not used in the C# implementation, but it is used in the Jolie. If the C# implementation ever will run natively with Jolie (embedded) this will be used

Listing 18: Person type in C#

```
public class Address {
        public string StreetName { get; set; }
        public int StreetNumber { get; set; }
        public int ZipCode { get; set; }
        public string City { get; set; }

        public Address(Value value) { ValueToObj(value); }

        public void ValueToObj(Value value) {
                this.StreetName =
                    value.GetChildren("streetName").First().StrValue;
                this.StreetNumber =
                    value.GetChildren("streetNumber").First().IntValue;
                this.ZipCode =
                    value.GetChildren("zipCode").First().IntValue;
                this.City =
                    value.GetChildren("city").First().StrValue; }

        public Value ObjToValue() { /* omitted */ }
}
public class Person {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Address { get; set; }

        public Person(Value value) { ValueToObj(value); }

        public void ValueToObj(Value value) {
                this.FirstName =
                    value.GetChildren("firstName").First().StrValue;
                this.LastName =
                    value.GetChildren("lastName").First().StrValue;
                this.Address = new
                    Address(Value.GetChildren("address").First()); }

        public Value ObjToValue() {
                Value retval = new Value();
                ValueVector fNameVec = new ValueVector();
                fNameVec.add(new Value(FirstName));
                ValueVector lNameVec = new ValueVector();
                lNameVec.add(new Value(LastName));
                ValueVector aVec = new ValueVector();
36                aVec.add(Address.ValueToObj());
37                retval.Children.Add("firstName", fNameVec);
38                retval.Children.Add("lastName", lNameVec);
39                retval.Children.Add("address", aVec;
40                return retval; } }
```

`Listing 18` shows how the person type in `Listing 17` is implemented in C#.
Each Jolie Value object[7] contains two methods. One to convert Jolie values to C# object, and one to convert C# objects to Values. In the example above, the Person type is created by first getting the child from the dictionary with the key ”firstName” (line 25). Note that they key is the same as the variable name in the Jolie implementation. Since `value.GetChildren` returns a ValueVector, it then needs to get the first element in the *values* list of the ValueVector, and finally extract the string value.
The rest of the object is created the same way. Note however that when creating the `Address` object, the child Value of the original Value object, containing the ”address” Value, is sent to the Address object (line 27) to handle in the same way as explained here.

When converting from C# object to Value, the procedure is the same but the other way around (line 29 - 40). Create a ValueVector, and add the Value to the values list of that ValueVector. Like before, the Address value is created by delegating the conversion to the Address object which then creates the required structure.
When finished with all ValueVectors, they are all added to the Values children.

Listing 19: Using C# Values

```
1 Person p = new Person(value);
2 p.ValueToObj(value) // is called implicitly in the constructor.
3 Value v2 = p.ObjToValue();
```

`Listing 19` shows how simple it is to convert a Value to a C# object (line 1 - 2) and vice versa (line 3).

When implementing the Value, the ByteArray needed to be implemented as well. The ByteArray basically is just a wrapper class around a regular ByteArray, which contains some helper methods to make it easier to interact with.

[7]”Jolie Value object” refers to a Jolie Value in C#

**Sodep**
In order to make the communication between C# and Jolie as smooth as possible, the Sodep protocol needed to be implemented in C#. Since the Sodep protocol uses CommMessages to transfer data, naturally, the CommMessage needed to be implemented first.
A CommMessage is basically a wrapper around a message sent via Sodep. CommMessage contains five fields:

- **Id**
  The id of the CommMessage
- **OperationName**
  This is the operation the CommMessage is calling to invoke. In C#, this could be translated to be a method call at the destination.
- **ResourcePath**
  The resource path is used for redirection in Jolie[11]. Since this feature is not used in JSOA at all, this is not implemented in C#.
- **Value**
  The value is, as described earlier in this section, the actual data that is sent with the CommMessage.
- **FaultException**
  The FaultException is the default exception used in Jolie. In C#, it is derived from the Exception[12]. FaultException contains a name (fault name) and a value.

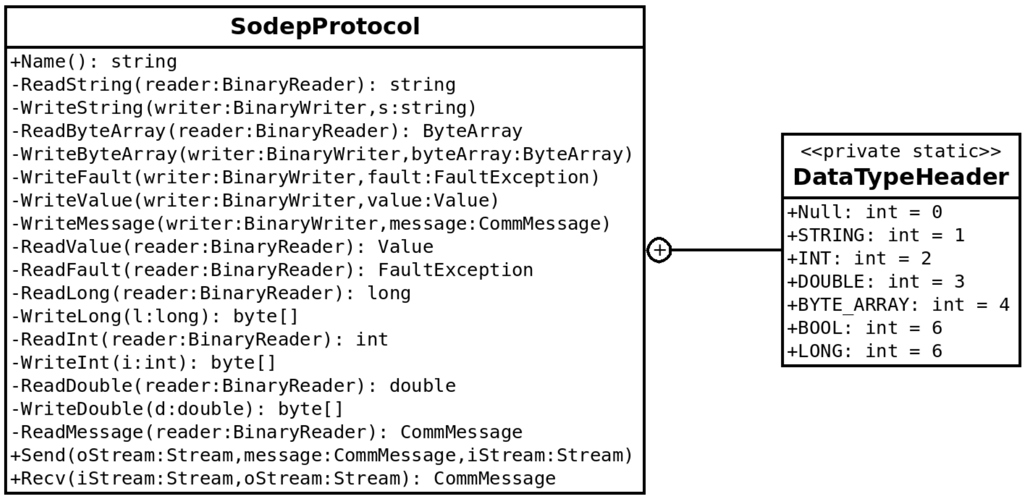
SodepProtocol
+Name(): string
-ReadString(reader:BinaryReader): string
-WriteString(writer:BinaryWriter,s:string)
-ReadByteArray(reader:BinaryReader): ByteArray
-WriteByteArray(writer:BinaryWriter,byteArray:ByteArray)
-WriteFault(writer:BinaryWriter,fault:FaultException)
-WriteValue(writer:BinaryWriter,value:Value)
-WriteMessage(writer:BinaryWriter,message:CommMessage)
-ReadValue(reader:BinaryReader): Value
-ReadFault(reader:BinaryReader): FaultException
-ReadLong(reader:BinaryReader): long
-WriteLong(l:long): byte[]
-ReadInt(reader:BinaryReader): int
-WriteInt(i:int): byte[]
-ReadDouble(reader:BinaryReader): double
-WriteDouble(d:double): byte[]
-ReadMessage(reader:BinaryReader): CommMessage
+Send(oStream:Stream,message:CommMessage,iStream:Stream)
+Recv(iStream:Stream,oStream:Stream): CommMessage
<<private static>>
DataTypeHeader
+Null: int = 0
+STRING: int = 1
+INT: int = 2
+DOUBLE: int = 3
+BYTE_ARRAY: int = 4
+BOOL: int = 6
+LONG: int = 6

Figure 9: Sodep protocol class overview

The Sodep protocol has two public methods, Send and Recv, along with multiple private helper methods (`Figure 9`). The public methods are used for sending and receiving CommMessages, while the private methods help to write the various data to the stream.
In addition, the SodepProtocol class also contains the private class DataTypeHeader, this is used for determining what data type to read/write next (string, int etc).
Writing string values is pretty straight forward. The string is converted to a byte array[8]. The length of the byte array is then sent to the stream, followed by the byte array itself. The receiving end is pretty similar, first read the length of the byte array, then read the byte array, and finally convert it to its string representation.
Dealing with number types (int, double and long) is slightly more complicated. Usually it is pretty simple to read and write number types since the BinaryReader/Writer have read/write methods for all these data types. However, since Sodep (the Jolie implementation) deals in big endian[13] and C# deals in little endian, there are some issues. Without going into too much detail about endianness, what it does is that they read/write bytes in opposite order. Because of this, the data would not make any sense on the receiving end, and the reader would crash. To counter this, a couple of helper methods are implemented.

[8]Byte array here is a regular C# byte array and not the Jolie implementation

Listing 20: Little- big endian Int32 converter method

```
1  private int ReadInt(BinaryReader reader) {
2          byte[] arr = reader.ReadBytes(4);
3          Array.Reverse(arr);
4          return BitConverter.ToInt32(arr, 0); }
5
6  private byte[] WriteInt(int i) {
7          byte[] arr = BitConverter.GetBytes(i);
8          Array.Reverse(arr);
9          return arr; }
```

`Listing 20` show how the data is converted. When the data comes in, the bytes are read into a byte array (line 2). The byte array is then reversed (line 3) and then converted to its data type (line 4). When sending data the opposite is done.

Most of the SodepProtocol methods are pretty simple. What they do is read and write data to the NetworkStream using Binary readers/writers. The most complex of the read/write methods, are the read/write value methods. `Listing 21` shows how the WriteValue method is implemented.

Listing 21: WriteValue excerpt

```
private void WriteValue(BinaryWriter writer, Value value) {
        try {
        if (value.IsString) {
                writer.Write((byte)DataTypeHeaderId.STRING);
                WriteString(writer, value.StrValue); }
        else if( ... ) // Rest of write data methods ommited
        Dictionary<string, ValueVector> children = value.Children;
        LinkedList<KeyValuePair<string, ValueVector>> entries =
                new LinkedList<KeyValuePair<string, ValueVector>>();
        foreach (KeyValuePair<string, ValueVector> entry in
            children) {
                entries.AddLast(entry);
        }
        writer.Write(WriteInt(entries.Count));
        foreach (KeyValuePair<string, ValueVector> entry in
            entries) {
                WriteString(writer, entry.Key);
                writer.Write(WriteInt(entry.Value.Size()));
                foreach (Value v in entry.Value)    {
                        WriteValue(writer, v);
19                  }
20          }
21          } catch (IOException e) { }
22  }
```

The WriteValue first sends the data type of the root value to the stream (string in the example), followed by the actual data[9] (line 4 - 5). When the data is sent, all children of the values are fetched and added to a list and the amount of children in sent to the stream (line 7 - 13).
Finally the method loops through all the children, sending their key (variable name) followed by the amount of children (line 15 - 16). Each child Value is then sent by recursively calling the WriteValue method (line 17 - 19).
The procedure on the receiving end is basically the reverse of the writing (see `appendix A` for the ReadValue implementation).

**Input and output ports**

In an attempt to stay true to the Jolie structure, input port and output ports are implemented. Implementation of the ports is not critical for the communication with Jolie, but in order to simplify the services, and to save developers from having to implement communication media (e.g., sockets) for each service, the ports are implemented.

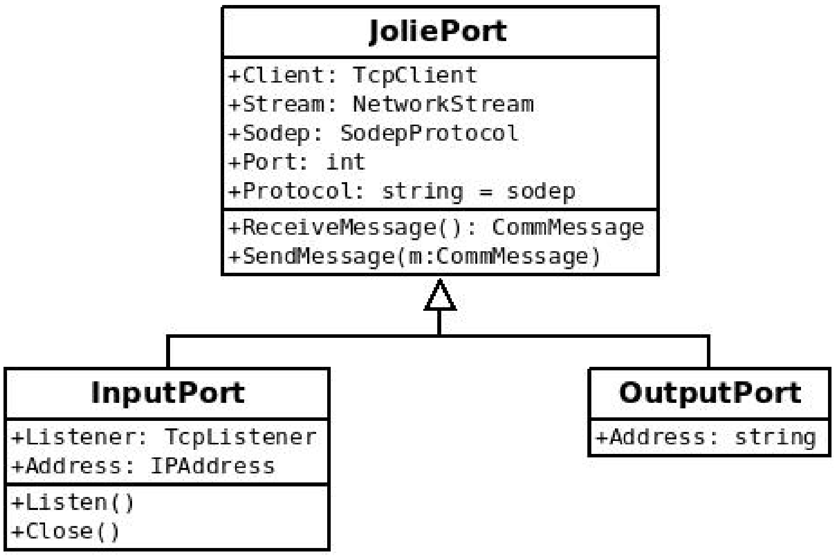

Figure 10: Input and output port class diagram

[9]The sending of the other data types have been omitted in order to save space

As Figure 10 shows, the ports are very simple.
The base port, JoliePort, contains all the properties and methods which are common for both ports. Client is used for sending data, via its Stream. SendMessage and ReceiveMessage are used for sending and receiving CommMessages via the SodepProtocol. The Port specifies which port number the JoliePort will use. Protocol is only used for returning the name of the protocol used, this is pretty redundant in its current state, but if the ports will be extended to support multiple protocols, this will become relevant.
InputPort contains two additional properties and methods. Listen() and Close() are used for starting and stopping the inputPort. The Listener listens for incoming connections on the port number specified. The Address property is used for setting the ip address of the input port. In this implementation, the ip address is always local but this might change in the future.
The OutputPort contains a single property, namely address. The address is used to specify the location of the InputPort which the output port communicates with.

Listing 22: Example usage of input/output port in C#

```
public JoliePort inputPort = new InputPort(8000);
inputPort.Listen();
CommMessage m = inputPort.Recv();
// ... do something with CommMessage

public JoliePort outputPort = new OutputPort("localhost", 8000);
CommMessage m2 = new CommMessage( ... );
outputPort.send(m2);
```

These simple construct make it a lot easier for developer to handle communication with Jolie via the Sodep protocol, as can be seen in Listing 22. Depending on the service, listening for incoming request might a bit more complex than this (see section 3.2 for a more thorough example) but essentially, Listing 22 shows all that is required of the service developer to handle communication with Jolie using input and output ports.

### C# interface / abstract class

In order to make life easier for developers, some interfaces and abstract were created. The interface IJolieValue, was created for the data objects. The interface has two methods, to convert a Value to C# object, and vice versa. The abstract class JolieServiceBase is created in order to aid developers in developing Jolie services. This abstract class contains all the methods which

a service requires to communicate efficiently with Jolie. At this time, there are only two methods, Start() and ShutDown(). The Start() method is used for "starting" the input port, and handling incoming requests (see `section 3.2` for a detailed example). The ShutDown() is, as the name states, used for shutting down the C# service.
As the C#-Jolie library will grow, and additional functionality is added, this abstract class will grow.

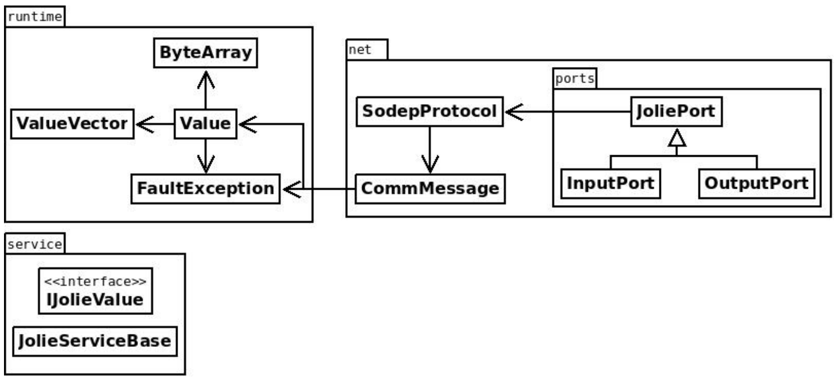

Figure 11: Jolie C# class diagram

`Figure 11` shows a complete class diagram of the C# implementation of Jolie. Properties and methods are omitted for clarity.

### 3.1.4 Authentication

As stated in the problem description, the second contribution the project is to implement authentication to JSOA. Currently, JSOA does not offer any form of authentication, so any form of authentication is tied to each individual service.

When implementing the authentication, the preferable method is that the uploaded service shall not be affected by the authentication in any way. To put it in other words, the authentication must be entirely based on the framework, completely independent of the running service.

There were a few options considered on how to implement the authentication.
The first option is to implement it in the Sodep protocol. The way this would be done is to add authentication data (e.g username and password or some authentication token) to the CommMessages sent via the Sodep protocol.

This data could then be handled by the main inputport of JSOA before being forwarded to the relevant service.
The main upside of this solution is that the developer would not have to worry about the authentication, since it all happens on a lower level. One of the drawbacks of using this method is that it limits JSOA to only use the Sodep protocol. Since JSOA (Jolie) supports multiple communication protocols[8], this would severely limit JSOA. To counter this, authentication data could be added to every protocol which JSOA supports, but doing so is a lot of work, which in turn is hard to maintain. Another drawback is that it makes the protocol a lot more complex which can cause problems and maybe performance issues as well.

The other method discussed (the one used) is to use courier sessions[10]. Courier sessions allow the extension of existing types and handling of those extensions independently of the services' handling of the original request. For example if a request from a client is sent to a service running on JSOA only contains an int, using a courier session, authentication data can be added to the request and handled before handling the actual request. After handling the authentication, the field is removed from the request (going back to the original request) and the request forwarded to the intended service.
`Listing 23` and `24` show how the request looks like with and without authentication data on the client side.

Listing 23: Simple Jolie data

```
type Request: void { .x: int }
```

Listing 24: Simple Jolie data with authentication key

```
type Request_auth: void { .x: int .key: string }
```

`Listing 7` in `Section 2.2`, shows how the courier session looks like on the service side. The example in `Listing 7` shows that the input port aggregates an output port with an `Interface extender`. This allows the input port to add the authentication data to the incoming request. The courier session then handles the authentication data, removes it, and forwards the request to the output port.

As seen in `Figure 12` and `Figure 13` the authentication has added one layer of indirection.[11] In `Figure 13`, a request is sent from the client to the service

[10]See `section 2.2` for more details on courier sessions
[11]Note that the courier is actually attached to the services wrapper but has been shown

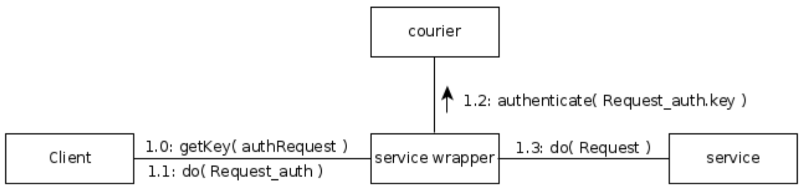

Figure 12: JSOA communication diagram with authentication

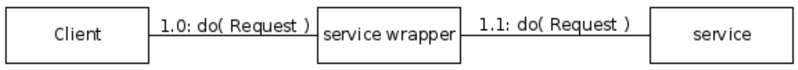

Figure 13: JSOA communication diagram without authentication

wrapper, and is then forwarded to the service, which then does the work and returns through the same path.
`Figure 12` contains additional operations. First the client sends a "getKey" request to the service wrapper. The service wrapper validates the request to see if the user has access or not (usually username and password). If valid, the service wrapper returns an authentication key to the client. The client then sends a "do" request to the service wrapper. Unlike `Figure 13` here the client sends an extended request (Request_auth in the example) instead of a regular request. The extended request is the same as the regular, but with added authentication data (see `Listing 23 and 24`). When the extended request is received by the wrapper, the courier session starts. The courier session checks if the key is valid. If the key is valid, the courier session removes the authentication data from the extended request (making it a regular request) and forwards the regular request to the service. If the key is invalid, the courier returns a fault exception and the operation terminates.

Only a few lines of code are required to implement this solution. Since the service wrapper (which is the main entry point for the running service) is generated at runtime, the courier session, along with the authentication data and the interface extender, is added to the code shaper, which generates the wrapper file.

Listing 25: Adding courier to service wrapper

```
[ prepareWrapper( request )( response ) {
/* ... */
```

separately in the diagram for clarity.

```
3  wrapper = wrapper + "type AuthenticationData: void {\n";
4  wrapper = wrapper + ".key: string \n";
5  wrapper = wrapper + "}\n";
6  wrapper = wrapper + "interface extender AuthInterfaceExtender {\n";
7  wrapper = wrapper + "RequestResponse: *( AuthenticationData )(
       void ) \n";
8  wrapper = wrapper + "OneWay: *( AuthenticationData ) \n";
9  wrapper = wrapper + "}\n"; */
10 /* ... */
11 wrapper = wrapper + "interface __WrapperInterface {\n";
12 /* ... */
13 // Add authentication to the wrapper interface
14 wrapper = wrapper + "getKey( string )( string )\n";
15 /* ... */
16 wrapper = wrapper + "inputPort Admin {\n";
17 /* ... */
18 wrapper = wrapper + "Interfaces: __WrapperInterface\n";
19 wrapper = wrapper + "Aggregates: Monitor, " + request.input[
       deployed_input[ 0 ] ].name.name + " with AuthInterfaceExtender";
20
21 wrapper = wrapper + "courier Admin {\n";
22 wrapper = wrapper + "[ interface " + request.input[
       deployed_input[ 0 ] ].name.name + "Interface( request )(
       response ) ] {\n";
23 wrapper = wrapper + "if( request.key == \"auth_valid\" ) {\n";
24 wrapper = wrapper + "forward " + request.input[ deployed_input[ 0
       ] ].name.name + "( request )( response )\n";
25 wrapper = wrapper + "}\n}\n}\n";
26
27 wrapper = wrapper + "main {\n";
28 /* ... */
29 wrapper = wrapper + "[ getKey( username )( key ) {\n";
30 wrapper = wrapper + "if( username == \"username\" ) {\n";
31 wrapper = wrapper + "key = \"auth_valid\"\n";
32 wrapper = wrapper + "} else {\n";
33 wrapper = wrapper + "key = \"auth_invalid\"\n";
34 wrapper = wrapper + "}\n";
35 wrapper = wrapper + "} ]{ nullProcess }\n";
```

`Listing 25` shows how the authentication is implemented. Lines 3 - 9 define the AuthenticationData and the InterfaceExtender. The interface extender extends all requests with the authentication data. Line 14 adds the ”getKey” method to the __WrapperInterface. On line 19, the Admin input port (which

is the main input port to the uploaded service) aggregates the deployed service input port (which has been converted to an output port by JSOA) with the interface extender. What this does is to extend every incoming request with the interface extender.
Lines 21 - 25 defines the actual courier session. Note that on line 22 *request.input[ deployed_input[ 0 ] ].name.name* is the name of the interface for the service. The courier checks if the key, which is the extended authentication data, matches a certain valid key (”auth_valid” in the example). If it does match, the request is forwarded to the aggregated output port (line 24).
Lines 29 - 35 add the ”getKey” method to the main function, which allows the client to authenticate and get the valid key. Since the example is just a proof-of-concept, ”getKey” just checks if the username matches the string ”username”. In reality, this is where the actual authentication takes place (check if username and password match some database entry for example).

Using this method has a lot of advantages.
The complexity of the authentication algorithm can easily be extended or changed. All that needs changing is the courier session and the ”getKey” operation on the service side.(`Listing 25`:29-35).
Another upside with this method is that it is completely tied to the SOABoot. If a developers wishes to extend the JSOA framework further, using a different authentication method, all that is needed to be done is to edit the SOABoot, and connect another instance of the SOABoot to the PaaSSOACentral.
As mentioned before, the major upside of using this method, is that it is completely independent of the deployed service. This allows the developer of a service to implement it any way he wants without being concerned about authentication. The only part which needs to be aware of the authentication, is the client connecting to the service.

As with most software solutions, there are drawbacks. One of the drawbacks is that every service uploaded to this instance of SOABoot is required to use the same authentication procedure. Even if a service is deployed which is not intended to have any authentication, it will have to handle it. This can be countered by coding the key into the client. Doing this makes it seem like there is no authentication (from the users perspective) even if there is.
Even if every service is tied to this authentication, there is no stopping the developers of the services to add additional authentication to the service.

## 3.2 Running example

The third and final contribution is to develop a proof-of-concept service, written in C#, which will be used to evaluate the changes made to JSOA. The service will consist of a C# service, a Jolie proxy which will handle the communication between JSOA and the C# service, and a client[12] written in Jolie.

Listing 26: TwiceRequest and TwiceResponse in Jolie

```
type TwiceRequest: void { .x: int }
type TwiceResponse: void { .y: int }
```

The example service is divided into two parts. The first part deals with simple data. It takes a TwiceRequest and returns a TwiceResponse. `Listing 26` shows the Jolie implementation of the TwiceRequest and TwiceResponse. The service will take the number from the request, double it, and return it to the Jolie client.

The second part of the service demonstrates how the C# service can handle complex data by simulating a database application. The service takes a Person request (`Listing 17` shows the Jolie implementation of the Person type) and allows the client to get, insert or delete persons from a database.[13] When the C# service starts, it first initializes the input port by specifying the port number to listen on (`Listing 27`). The actual service is then started, in this example it is started on a new thread.

Listing 27: Initialize C# service

```
public class Program {
        public static int port;
        public static void run() {
                Twice t = new Twice(port);
                t.Start(); }

        public static void Main(string[] args) {
                port = 9998;
                Thread th = new Thread(new ThreadStart(run));
                th.Start(); } }
```

[12]See `appendix B`

[13]The service does not use an actual database but a list of Person objects to simulate database behaviour.

After the input port has been initialized, it starts to listen for incoming connections.
When a request is received, the service enters a switch. The cases of the switch is the operation name of the incoming CommMessage. Each operation name corresponds to a method in the C# service (`Listing 28`).

Listing 28: C# service start method

```
public override void Start() {
  while (true) {
    if (InputPort.Listener.Pending()) {
      InputPort.Listen();
      try {
        CommMessage m = InputPort.ReceiveMessage();
        if (m != null && m.Value != null) {
          switch (m.OperationName.ToLower()) {
            case "twice":
              handleTwice(m);
              break;
            case "insert":
              handleInsert(m);
              break;
            case "get":
              handleGet(m);
              break;
            case "delete":
              handleDelete(m);
              break;
            case "shutdown":
              handleShutDown();
              break; } }
            InputPort.Close(); }
            catch (Exception) { } } } }
```

`Listing 29` shows what happens when the "get" request is received in the input port.
First the Jolie value is converted to a C# object, `Listing 18` shows how the conversion between Jolie value and C# objects is implemented.
When the value is converted, the service does what ever work it is designed to do. In this example, it gets a Person from a database.
When the operation is done, the object is converted into a Jolie Value, and returned back to the client, in a CommMessage, via the Sodep protocol.

Listing 29: Handle get method

```
private void handleGet(CommMessage m) {
        PersonDB db = new PersonDB();
        Person person = new Person(m.Value);
        PersonResponse resp = new PersonResponse(db.Get(person));
        InputPort.SendMessage(new CommMessage(m.Id,
            m.OperationName, m.ResourcePath, resp.ObjToValue(),
            m.IsFault ? m.Fault : null)); }
```

As can be seen from this example, the communication with Jolie is very simple to implement once the Jolie-C# library is in place. The only thing the developer needs to worry about to implement (regarding the Jolie communication) is the switch in the start method (`Listing 28`).
Note that even though there is no sign of any authentication, this service uses the authentication discussed in `section 3.1.4`.

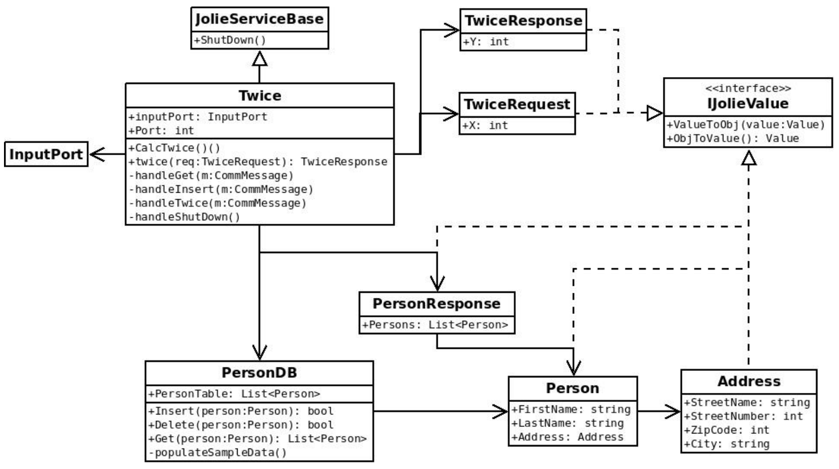

Figure 14: Example service class diagram

`Figure 14` shows a complete class diagram of the example service.
Before the C# service is ready to upload to JSOA, a Jolie proxy needs be implemented. This proxy handles the communication between JSOA and the C# service.

Listing 30: Jolie-C# proxy

```
type Person: void { ... }
type PersonResponse: void { ... }
type TwiceRequest: void { ... }
type TwiceResponse: void { ... }
interface twiceInterface { ... }
interface personInterface { ... }
outputPort proxy {
           Location: "socket://localhost:9998"
           Protocol: sodep
           Interfaces: twiceInterface, personInterface }
inputPort twiceIP {
          Location: "local"
          Protocol: sodep
          Aggregates: proxy }
main { linkIn( nothing ) }
```

`Listing 30` shows the proxy for the C# service example[14]. Note that the implementation of the types and interfaces have been omitted in order to save space, but the implementation is matching the types and methods in the C# service.
How this works is that the output port proxy, is the output port which communicates with the C# service (note that the Location is the same as the one in the C# service). The input port is how JSOA communicates with the service. The Location of the input port is set by JSOA.
With the proxy in place, the service is ready to be uploaded to JSOA.
When uploading the service, all related files need to be put inside a jap file.

Listing 31: Updated jap file manifest

```
Manifest-Version: 1.0
X-JOLIE-Main-Program: proxy.ol
X-JOLIE-CSHARP-PROGRAM: TwiceService.exe
Comment_X-JOLIE-Options: Options here
```

`Listing 31` shows the manifest required for the jap file in this example. Compared to the manifest file in `Listing 8`, there is one extra line. This line *(X-JOLIE-CSHARP-PROGRAM: TwiceService.exe)* tells JSOA where to find the main C# service file. Note that the main Jolie file in this manifest file is the proxy service.

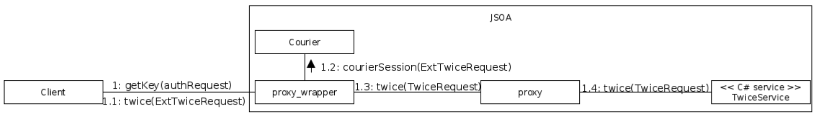

Figure 15: Example service communication diagram

[14]"linkIn( nothing )" inside the main function just means that it does nothing, but just forwards the request

Figure 15 shows a communication diagram of the example service deployed on JSOA. Notice that compared to figure 12, there is an extra layer of indirection, namely the proxy.

## 3.3 Benchmarking

The second part of the third contribution is to evaluate the JSOA framework in order to evaluate potential performance changes after the extra layers of indirection have been added because of the C# service and the authentication (see figure 12 and 15).
The full tables with all the results can be found in the SVN repository mentioned in the preface.

### Benchmarking methodology

In order to do the evaluations, some benchmarking needs to be done. This is done by uploading two versions of the same service, one written in Jolie and one written in C# with a Jolie proxy.

Listing 32: Jolie implementation of the TwiceService

```
type TwiceRequest:void { .x: int }
type TwiceResponse: void { .y: int }
interface twiceInterface {
       RequestResponse: twice( TwiceRequest )( TwiceResponse ) }
inputPort mainIP { /* ... */ }
main { twice( req )( resp ) { resp.y = req.x * 2 } }
```

Listing 32 shows the Jolie implementation of the service used for the benchmarking. The service takes a TwiceRequest, doubles the number (x) and returns a TwiceResponse.

The actual benchmarking is divided into two parts, first both the Jolie and C# services test how much time difference there is when sending requests which are authenticated every time, compared to just authenticating the first request[15]. This test simulates a client sending a single request, and then closes. The second part simulates a client that connects to the service, fetches the authentication key and stores it for all additional requests.

The second test is to measure the time difference between the calling the Jolie service compared to the C# service. Since the C# service has one

[15]Authentication in this context refers to the "getKey" request. The courier session is evaluated on every request regardless

extra layer of indirection compared to the Jolie service (`figure 12` and `15`), the intuition is that the C# service should take a little longer.

Listing 33: Benchmark client

```
1  main {
2          /* .. prepare file for result export */
3          request.x = 55;
4          for( i = 0, i < 100, i++ ) {
5                  getCurrentTimeMillis@Time()( startTime );
6                  getKey@twiceOP( "username" )( request.key );
7                  twice@twiceOP( request )( response );
8                  getCurrentTimeMillis@Time()( endTime );
9                  time = endTime - startTime;
10                 timeArray[i] = time;
11                 undef( startTime ); undef( endTime ); undef( time );
12                 undef( response )
13         };
14         /* .. write file with results */
15 }
```

`Listing 33` shows the client used for the benchmarking. Note that the types, interface and output port have been omitted for clarity (see `appendix B` for the full implementation).
The client sends 100 requests to the service and times how long it takes to make each request. These results are then written to a file to be processed later.
Notice that on line 6, the "getKey" call is inside the loop. In the second part of the test (where the request is evaluated only once), the "getKey" is outside the loop.

When all tests are finished, both for the Jolie and C# services, the results are processed. The processor finds the time difference between the Jolie and the C# service, as well as the request with authentication and the one without. In addition to finding the difference, the average time it takes to make a request is calculated.

Listing 34: Result processor excerpt

```
public static void main( String[] args ) { ... }
public static String compare( int[] arg1, int[] arg2, int[] arg3 )
{ ... }
public static double getAverage( int[] arr ) { ... }
```

```java
public static void writeFile(String filename, String content)
{ ... }
```

Listing 34 shows an excerpt of the result processor. It takes two parameters, the result files from two of the tests mentioned above, along with an optional third parameter which is the name output file for the processed results.
The Compare method takes three arrays. The first two arrays are the results from the input files which are inserted into an array in the main method. The third array is the combined result. The getAverage method calculates the average time it takes to send a request to the service. The final method, writeFile, is optional. If an output file has been specified, this method writes the results to the output file.

**Benchmarking results**
This section will analyse the results of the benchmarking.
It should be noted that the actual time it takes to handle a request might not be 100% accurate. The reason for this is that there are multiple factor which can affect the performance. For one thing, the server runs locally so any time it takes to send the request via the network (internet) is not taken into account. The second thing is that there are multiple other processes running on the same computer which might use some resources, causing the results to be incorrect. However, what is interesting in these tests is the difference between the two cases studied. Since both cases are run on the same computer, under the same circumstances, these test should give a fair picture of what the difference in performance looks like.

**Authentication vs no authentication**
The first benchmark measures how long it takes to handle a request when the request is authenticated every time, compared to when the request is authenticated once, and then the same authentication key is used in the following requests.

| No auth (ms) | Auth (ms) | Compared (no auth - auth) (ms) |
|---|---|---|
| 9 | 37 | -28 |
| 4 | 3 | 1 |
| 4 | 4 | 0 |
| 9 | 4 | 5 |
| 5 | 3 | 2 |
| 4 | 3 | 1 |
| 4 | 3 | 1 |
| 6 | 6 | 0 |
| ... | ... | ... |
| avg. 3.7 | avg. 3.69 | avg. 0.01 |

Table 1: Excerpt of benchmark results of Jolie service authentication vs no authentication

`Table 1` shows an excerpt of the results of the first test for the Jolie service. As seen in the third column (combined) there is very little difference between the two. Sometimes the request with authentication is slightly faster and sometimes the other is faster.
As it also can be seen by the average, the request with authentication is 0.01 ms faster, which in practice means that they are identical.
Notice that the first request of the second column it takes 28ms longer than the one in the first column. This is because Jolie closes the connection after some time has passed and it takes a bit longer to open the connection again. This can be seen throughout the full table.

| No auth (ms) | Auth (ms) | Compared (no auth - auth) (ms) |
|---|---|---|
| 30 | 39 | -9 |
| 5 | 5 | 0 |
| 10 | 4 | 6 |
| 9 | 5 | 4 |
| 7 | 5 | 2 |
| 5 | 8 | -3 |
| 4 | 5 | -1 |
| 7 | 7 | 0 |
| ... | ... | ... |
| avg. 3.49 | avg. 3.8 | avg. -0.31 |

Table 2: Excerpt of benchmark results of C# service authentication vs no authentication

`Table 2` shows an excerpt of the results of the first test for the C# service. Much like with the Jolie service, there is very little difference. The first column (no authentication) is slightly faster on average (0.31 ms). This is such a small number that in practice, there is no real difference.

Based on these two tests, it can be concluded that even if the authentication has added an extra layer of indirection. and requires the client to make one additional request for the authentication, there is no loss in performance.
In practice, when the service is hosted on a remote server, there might be a difference since the extra request takes some time to send to the server via the network (internet).

**Jolie service vs C# service**
The second benchmark measures the time difference when using Jolie service compared to C# services. This test requires that each request is authenticated.

| Jolie service (ms) | C# service (ms) | Compared (jolie - C#) (ms) |
|---|---|---|
| 40 | 38 | 2 |
| 3 | 5 | -2 |
| 4 | 3 | 1 |
| 4 | 4 | 0 |
| 4 | 5 | -1 |
| 4 | 2 | 2 |
| 3 | 5 | -2 |
| 6 | 6 | 0 |
| ... | ... | ... |
| avg. 3.81 | avg. 3.33 | avg. 0.48 |

Table 3: Excerpt of benchmark results of Jolie service vs C# service

`Table 3` shows an excerpt of the results of the final test, the comparison between Jolie services and C# services.
This test is a little trickier. On average, the C# service is a little faster (0.48 ms) but looking at each request separately, the Jolie service is faster in most of the requests.
The reason for C# being faster on average, is that the C# service keeps the connection open, while the Jolie service closes the connection after some time, and it needs to be opened again before handling the request.
Based on this test, it can be concluded that when not taking in consideration the time it takes to open the connection, the Jolie service is faster, but sending multiple requests, which require the connection to reopened from time to time, the C# service is faster, even if it has an extra layer of indirection.

# 4 Conclusion

This thesis set out to create a PaaS cloud framework which is platform independent, allows the deployment of services written in multiple programming languages, especially the larger, commercial programming languages. The framework needed to be scalable in the sense that support for additional programming languages could be added in the future. Finally the framework needed to automatically handle some non-functional aspects, such as authentication.

The work shown in this thesis has been built on the existing PaaS framework JSOA[4].

JSOA has been extended to run natively on the Windows platform, taking it one step further towards becoming completely platform independent.
JSOA has also been extended to support services written in the C# programming language. Multiple Jolie constructs have been implemented in C#; Values, input and output ports and the Sodep protocol are noteworthy examples. By this implementation, JSOA now offers support for running services written in C#.

Authentication has also been added to the JSOA framework. The actual authentication algorithm is in a prototype state (e.g., only a static check of a username is made), but the foundation is in place for extending it to production state. This authentication allows service developers to develop services without having to consider authentication into the service architecture.

Evaluations showed that there is no noteworthy performance loss caused by the added layers of indirection caused by the authentication and the C# implementation. Although it should be noted that these evaluations took place on a regular laptop, with the framework running locally.
Extensive evaluation of JSOA would be a thesis in itself, but based on the comparison between Jolie and C# services, both with authentication and without, there is no noteworthy difference in performance.

## 4.1 Limitations

Because of time constraints, there are some limitations and minor issues with the JSOA framework.
JSOA offers the ability to get the surface definition to a service, which is used for developing clients to interact with the service. In its current state,

the surface definition does not show any signs of the required authentication data, which results in an error when trying to connect to the service. It still is possible to add this authentication data to the client, but unless client developers know the structure of the authentication in JSOA, they have no way of knowing that it is required.
This is issue is easily fixed and will be fixed in the future.

## 4.2 Future work

Extending JSOA is a never ending story. There are always new functionalities, and support for additional platforms and programming languages which can be added, while those existing can be improved. The following section names a few additions which are interesting for the future.

**Automatic generation of the C# interface**
As it is implemented now, every C# service need to handle the communication with JSOA manually. Although the implementation of input ports and output ports have made it really simple, this can still be improved further. Automatic generation of the interface could easily be implemented by adding some annotations to the C# library which then can be parsed, and generated automatically.

**Automatic generation of the Jolie proxy**
Every C# service running on JSOA requires a Jolie proxy service which acts as an intermediary between the C# service and JSOA. This proxy does nothing except forwarding requests from JSOA to the C# service.
Adding a parser and a code generator, this proxy service could easily be automatically generated when the service is deployed. However, this requires that the parser has access to the C# source code, or at least a strictly defined API of the C# service.

**Port to OSX platform**
Porting JSOA to Windows is one step in making the framework runnable on every major platform (JSOA is already running on Linux).
The next step would be to port JSOA to the Mac OSX platform.
Since OSX is based on Unix, there should not be too much work to do, but as with the Windows implementation, there might still be some work to do.

**Support for additional programming languages**
With the addition of C#, JSOA now supports deployment of services written in Jolie, Java, JavaScript and C#. Adding additional programming lan-

guages to the list of supported languages would be the next step. Using the methods used in this thesis (implement Sodep, Jolie values and input/output ports) support for additional programming languages can be added to the framework.

**Add compilers**
In its current state, JSOA accepts compiled C# services. Extending JSOA to allow developers to uploaded source files, which can be compiled on the cloud framework could be a huge improvement. Having this option, version control systems (e.g., Git, SVN) can be added to JSOA, making life easier for service developers.

**Add domain to the authentication, allowing the deployment of multi-instance services**
Adding domains to the authentication procedure allows for the deployment of multi-instance service on JSOA. This would allow developers to upload multi-instance software to the framework, and have JSOA handling the routing for each instance.

## References

[1] Microsoft Azure website. http://azure.microsoft.com/en-us/

[2] Amazon EC2 website. https://aws.amazon.com/ec2/

[3] Google developers. Google app engine. https://developers.google.com/appengine/?csw=1

[4] JSOA website. http://sourceforge.net/projects/jsoa/

[5] NIST website. Cloud computing definition. http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf

[6] Jolie website. http://www.jolie-lang.org/

[7] Java api. Socket listener. http://download.java.net/jdk7/archive/b123/docs/api/java/net/ServerSocket.html

[8] Jolie website. Supported protocols. http://www.jolie-lang.org/?top_menu=documentation&sideMenuAction=protocols/introduction

[9] Leonardo website. http://sourceforge.net/projects/leonardo/

[10] Indiana University website. In Unix, what is the shell? http://kb.iu.edu/data/agvf.html

[11] Jolie website. Redirections. http://www.jolie-lang.org/?top_menu=documentation&sideMenuAction=architectural_composition/redirection

[12] Microsoft MSDN. Exception API. http://msdn.microsoft.com/en-us/library/system.exception.aspx

[13] Webopedia. Big- little-endian. http://www.webopedia.com/TERM/B/big_endian.html

[14] Jolie website. Locations. http://www.jolie-lang.org/?top_menu=documentation&sideMenuAction=locations/introduction

# A Sodep ReadValue

Listing 35: Sodep ReadValue method

```
private Value ReadValue(BinaryReader reader)
{
try {
  Value value = new Value();
  byte b = reader.ReadByte();
  switch (b) {
    case DataTypeHeaderId.STRING:
      value = new Value(ReadString(reader)); break;
    case DataTypeHeaderId.INT:
      value = new Value(ReadInt(reader)); break;
    case DataTypeHeaderId.LONG:
      value = new Value(ReadLong(reader)); break;
    case DataTypeHeaderId.DOUBLE:
      value = new Value(ReadDouble(reader)); break;
    case DataTypeHeaderId.BYTE_ARRAY:
      value = new Value(ReadByteArray(reader)); break;
    case DataTypeHeaderId.BOOL:
      value = new Value(reader.ReadBoolean());
      break;
    case DataTypeHeaderId.NULL:
      break;
    default:
      break;
  }
  Dictionary<string, ValueVector> children = value.Children;
  string s;
  int n, i, size, k;
  n = ReadInt(reader);
  ValueVector vec;
  for (i = 0; i < n; i++) {
    s = ReadString(reader);
    vec = new ValueVector();
    size = ReadInt(reader);
    for (k = 0; k < size; k++) {
      vec.add(ReadValue(reader)); }
    if (!children.ContainsKey(s)) children.Add(s, vec);}
    return value; }
        catch (Exception e) { return null; } }
```

# B Example service client

Listing 36: Example service client

```
include "console.iol"
include "time.iol"
include "file.iol"
type TwiceRequest:void { .x: int .key: string }
type TwiceResponse: void { .y: int }
interface twiceInterface {
         RequestResponse: twice( TwiceRequest )( TwiceResponse ),
            getKey( string )( string )
         OneWay: shutdown( void ) }
outputPort twiceOP {
          Location: "socket://localhost:12001"
          Protocol: sodep Interfaces: twiceInterface }
main {
     tmpFilename = args[0]; filename = "result/" + tmpFilename +
        ".txt";
     exists@File( filename )( fileExists );
     if( fileExists ) {
        delete@File( filename )( isDeleted )
     };
     timeArray = "";
     content = "";
     request.x = 55;
     for( i = 0, i < 100, i++ ) {
          getCurrentTimeMillis@Time()( startTime );
          getKey@twiceOP( "username" )( request.key );
          twice@twiceOP( request )( response );
          getCurrentTimeMillis@Time()( endTime );
          time = endTime - startTime;
          timeArray[i] = time;
          undef( startTime ); undef( endTime ); undef( time );
          undef( response )
     };
     for( j = 0, j < #timeArray, j++ ) {
          content = content + timeArray[j] + "\n"
     };
     fileRequest.filename = filename;
     fileRequest.content = content;
     writeFile@File( fileRequest )()
}
```