Complete Dynamic Multi-cloud Application Management

Project no. 644925

Innovation Action

Co-funded by the Horizon 2020 Framework Programme of the European Union

Call identifier: H2020-ICT-2014-1

Topic: ICT-07-2014 – Advanced Cloud Infrastructures and Services

Start date of project: January 1st, 2015 (36 months duration)

Deliverable D6.2

Specification of Interfaces for Brokering, Deployment, and Management

Due date:	31/08/2015
Submission date:	31/08/2015
Deliverable leader:	SixSq
Editors list:	C. Loomis (SixSq)

Dissemination Level

☒	PU:	Public
☐	PP:	Restricted to other programme participants (including the Commission Services)
☐	RE:	Restricted to a group specified by the consortium (including the Commission Services)
☐	CO:	Confidential, only for members of the consortium (including the Commission Services)

List of Contributors

Participant	Short Name	Contributor
Interoute S.P.A.	IRT	
SixSq Sàrl	SIXSQ	C. Loomis
QSC AG	QSC	
Technische Universitaet Berlin	TUB	M. Slawik
Fundacio Privada I2CAT, Internet I Innovacio Digital A Catalunya	I2CAT	J. Aznar, E. Escalona
Universiteit Van Amsterdam	UVA	Y. Demchenko, M. Zivkovic
Centre National De La Recherche Scientifique	CNRS	

Change history

Version	Date	Partners	Description/Comments
0.5	23/08/2015	SixSq	First complete version for review.
0.6	24/08/2015	SixSq	Changes based on feedback from SixSq personnel and collaboration partners.
0.7	27/08/2015	SixSq	Integrate comments on previous version.
1.0	31/08/2015	SixSq	Integrate final comments.

Table of Contents

List of Figures

List of Tables

Executive Summary

This document specifies the interfaces that will be used for the deployment and management of cloud applications as well as the brokering interface for selecting the optimal cloud services for each application.

Within the CYCLONE architecture, SlipStream handles all of the cloud application deployment and management operations. Cloud Application Developers and Operators will directly use the SlipStream web interface or REST API to trigger management actions.

Previously, the project adopted a strategy where the current SlipStream application model will be used in the short-term with an incremental migration towards the CIMI standard in the longer-term. This strategy also defines the deployment and management interfaces, as both the SlipStream and CIMI models are tightly associated with well-defined APIs. Consequently, the SlipStream API (and its evolution towards CIMI) will be used for the deployment and management of cloud applications. The API is documented on the site http://ssapi.sixsq.com and the details of CIMI can be found in the specification from DMTF.

To provision cloud resources for applications, SlipStream must create a deployment plan. This is done when first deploying the application and again whenever resource re-allocations of the application are triggered. Currently SlipStream users manually select the cloud service to use for each application component. With CYCLONE, SlipStream will be extended to call an external brokering service to select cloud services based on user-defined policies. CYCLONE will provide two implementations of the brokering service: one based on the SlipStream Service Catalog and the other based on the Open Service Compendium.

The brokering service API, defined in this document, consists of a single HTTPS GET request with a JSON message body containing the user's requirements. The service will return a standard HTTP response with an appropriate status code and JSON body with the matching cloud services.

In the interest of practicality and to allow the CYCLONE use case representatives to test the brokering service as soon as possible, an incremental implementation plan is proposed. Three phases have been identified with the following goals for each:

1. Demonstrate that all of the components of the brokering system are available to CYCLONE users, although with very limited selection capabilities.
2. The goal of the second phase is to pass the application topology to the brokering service and to converge on a common query language and core schema for the two brokering service implementations.
3. Expand the supported application characteristics and to demonstrate the use of third-party information.

When completed, SlipStream through the brokering service will allow users to place application components based on a wide range of characteristics including resource availability, resource accessibility, performance, legal constraints, pricing, and application topology.

The incremental implementation strategies for the deployment and management interfaces as well as the brokering service, will provide a working platform for evaluation by the project's use cases rapidly.

1. Introduction

Figure 1 shows the lifecycle of a cloud application including its creation, deployment, and management. The following describes each stage of the lifecycle in more detail:

1. **Define Application**: The Cloud Application Developer, using the CYCLONE application model, defines the components (e.g. servers, storage, services) of the application and how those components interact (e.g. network connectivity). The application description may also include policies to select which cloud services to use for the application. The Cloud Application Developer stores the application description within SlipStream.
2. **Create Deployment Plan**: When the Cloud Application Operator chooses to run an instance of a defined application, SlipStream first creates a deployment plan, determining exactly which resources will be provisioned for the application. Currently, the Cloud Application Operator explicitly identifies which Cloud Service Provider to use for each application component; interactions with a brokering service (see Section 3) will allow Cloud Service Providers to be selected based on policies defined by the Cloud Application Developer and/or Operator.
3. **Provision Resources**: Once the deployment plan has been created or updated, SlipStream realizes the application by provisioning resources on the selected Cloud Service Provider(s) and configuring the application components.
4. **Monitor Application**: Once the application has been deployed and configured, SlipStream enters a monitoring phase, periodically checking the status of the provisioned resources. Currently, this includes only the state of the virtual machines comprising the application. The monitoring will be extended to include virtual machine load characteristics as well as application-specific metrics. The desired metrics for an application can be specified through the CYCLONE application model.
5. **Update Deployment Plan**: The load of an application will change over time. Using the monitoring information, the Cloud Application Operator can decide (manually or via a defined policy) to change the resource allocation for a running application. SlipStream supports both horizontal and vertical scaling of the resources. Once the deployment plan has been updated, SlipStream will re-provision resources for the application (step 3). If policies are specified, the brokering service will be consulted to define the acceptable cloud services.
6. **Terminate Application**: When the Cloud Application Operators decide that cloud application instances are no longer useful, they can terminate them, freeing the resources associated with the application.

Within CYCLONE, realizing this lifecycle requires interacting with the deployment and management interfaces (web interface and REST API) of SlipStream. Advanced features where the deployment plan is determined by Cloud Application Developer/Operator policies additionally require interaction with the brokering service. The following chapters describe these interfaces.

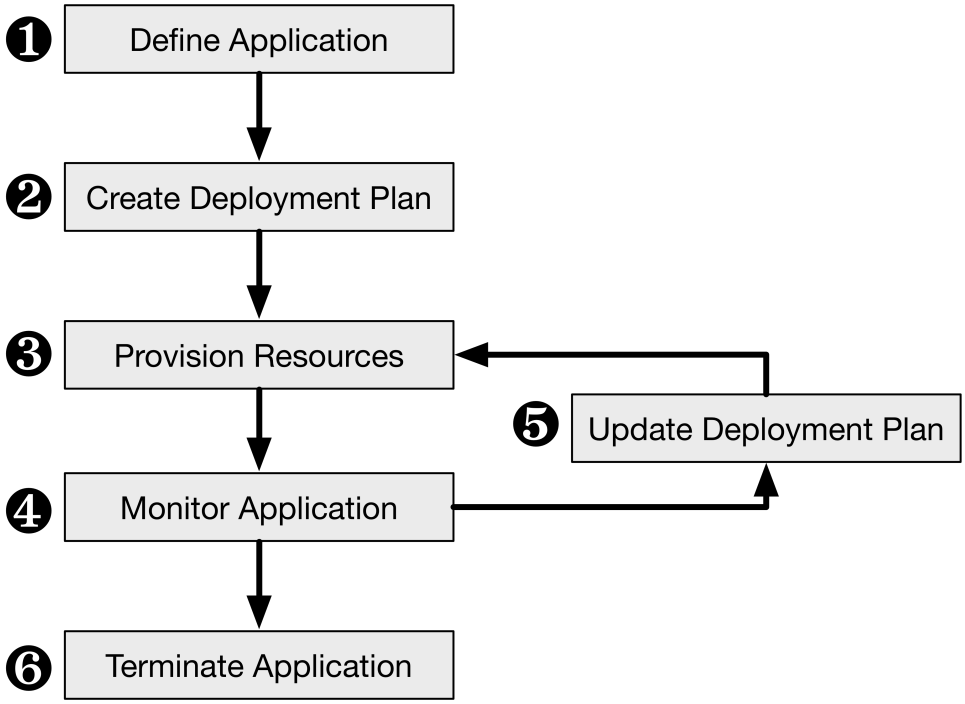

Figure 1: Cloud Application Workflow

2. Deployment and Management

Within the CYCLONE architecture, SlipStream handles all of the cloud application deployment and management operations. Cloud Application Developers and Operators will directly use the SlipStream web interface or REST API to trigger management actions.

2.1. Interfaces

In a previous document (CYCLONE, 2015), the CYCLONE model for cloud applications was defined. That document provided requirements for the application model and evaluated several alternative models against those requirements. Based on that analysis, the project adopted a strategy where the current SlipStream model will be used in the short-term with an incremental migration towards the CIMI (DMTF, 2015) standard in the longer-term.

This strategy essentially defines the deployment and management interfaces as well, given that both the SlipStream and CIMI models are tightly associated with well-defined APIs. Practically, this strategy means that the SlipStream API (and its evolution towards CIMI) will be used for the deployment and management of cloud applications. The majority of the SlipStream API features are also available through the web interface.

Details of the current SlipStream API can be found on the SixSq documentation website (SixSq, 2015). The API documentation is kept up-to-date with the latest release of the SlipStream software. For cloud resource descriptions that follow the CIMI model, the CIMI specification (DMTF, 2015) is the definitive guide.

The current SlipStream API allows Cloud Application Developers to define complex, multi-machine applications and to make them available to Cloud Application Operators. They can in turn use those definitions to deploy and to manage application instances on multiple cloud infrastructures. The API also allows the Cloud Application Operator to scale a running application either horizontally—by adding or removing virtual machines—or vertically—by changing the CPU, RAM, or disk space of a running virtual machine. The migration towards CIMI will provide users with a richer set of cloud resources (in particular related to network resources) and mechanisms to specify and consume monitoring metrics.

2.2. Limitations

The defined cloud application lifecycle and the adopted APIs will allow Cloud Application Operators to make changes to running cloud applications. However, there are a couple of limitations on what types of changes can be managed through the CYCLONE components and what types of changes are possible.

2.2.1. Migration

For long-running applications, the ability to migrate application components from one cloud provider to another, for example to provide clients with lower-latency connections or to minimize the costs by moving to a cheaper provider, might be convenient. However, in the face of significant, perhaps insurmountable, technical barriers (e.g. the difficulty in moving VM state between providers or the uncertain conversion of VM formats), CYCLONE will not provide this feature.

Even without direct support of migration, CYCLONE will permit dynamic changes to the geographic distribution of application resources, through the re-deployment of services. By taking advantage of the horizontal scaling capability, Cloud Application Operators can provision new, redundant application components (presumably on a different provider and/or in another geographic region). Once the new components have started, the older components can be terminated. This allows the Cloud Application Operator to optimize their applications without running into the technical barriers from a pure migration. To take advantage of this however, the cloud application must use an architecture that supports horizontal scaling.

2.2.2. **Updates**

The adopted application models and associated APIs concentrate intentionally on the provisioning and management of cloud resources. They do not touch upon keeping the configuration of the application or of the underlying operating system up-to-date with, for example, security patches.

There are a vast number of standard tools, such as YUM, Puppet, or Chef, for managing such updates. Cloud Application Developers should take advantage of those tools for long running applications rather than relying on CYCLONE features for this.

Alternatively, Cloud Application Developers can use the horizontal scaling features to upgrade individual components of an application incrementally. Again presuming that the Cloud Application Developer has used an application architecture that is capable of live replacement of components.

3. Brokering

The formal definition of a broker is a person who acts as an intermediary in arranging marriages, negotiating agreements, etc. For CYCLONE, this is primarily a matchmaking service that allows cloud service providers to describe their offerings and allows application developers (or users) to specify their resource requirements. Given these inputs, the brokering service compares the application requirements against the cloud service characteristics to return a (prioritized) list of acceptable services for application component placement.

Possible application requirements span a wide range of functional and non-functional characteristics:

- **Resource Availability**: Does a cloud service provide the required computing resources (e.g. CPU, RAM, bandwidth, or data) for an application component?
- **Resource Accessibility**: Can a user access the resources of a particular provider? For example, does the user have an account and does the resource request fall within applicable quotas?
- **Performance**: Does a cloud service meet the application's minimum requirements for availability, reliability, network latencies, or other performance characteristics? Users may require that these metrics come from third-parties to ensure their objectivity.
- **Legal Constraints**: Certain types of data (e.g. medical or financial) often come with heightened legal requirements. The brokering may need to select providers based on their legal jurisdictions and/or third-party certifications (e.g. ISO certifications or credit card processing authorization).
- **Pricing**: The cost of running an application may also be an important characteristic to consider when placing application components.
- **Application Topology**: For redundancy, failover, or performance, the application may require that certain components are co-located or not co-located on the same cloud infrastructure.
- **Service Level Agreement (SLA):** Specific information from providers and consumers about the guaranteed levels of performance and support for cloud services.

Selecting cloud service providers based on these characteristics will nearly always involve tradeoffs between cost and performance. As these tradeoffs can only be made by the Cloud Application Developer and Cloud Application Operator, CYCLONE must allow them to provide a selection policy per application component that will then be evaluated by SlipStream when creating the deployment plan (see Figure 1).

3.1. Brokering Service Interactions

Before defining the interfaces for the CYCLONE brokering service, it is instructive to see how that service will interact with the other services and actors within the CYCLONE ecosystem. Figure 2 shows a sequence diagram of the interactions required for using the brokering service for selecting the cloud services used within an application deployment. As the diagram shows, there are two interactions between the brokering service and the other services:

- SlipStream will call the brokering service with a set of policies (or queries) to obtain a list of acceptable Cloud Service Providers for each application component.
- The brokering service will asynchronously obtain information from Cloud Service Providers (and potentially other actors) to create an internal database of Cloud Service Provider characteristics.

Unlike for Cloud Service Providers, SlipStream will **not** use a plug-in architecture for interacting with different brokering service implementations; instead the brokering service implementations are expected to support the CYCLONE brokering interface defined below.

The interactions between the brokering service implementation and the Cloud Service Providers (or other actors supplying information) are details of the implementations themselves and do not affect how Cloud Application Operators deploy and manage their cloud applications. Consequently, these details are not relevant to CYCLONE and are not specified by the project.

Future work will extend these brokering facilities to include the exchange and negotiation of Service Level Agreements and other advanced features. A future deliverable (D6.3), scheduled for September 2016, will describe the extensions necessary for the brokering service to provide those advanced features.

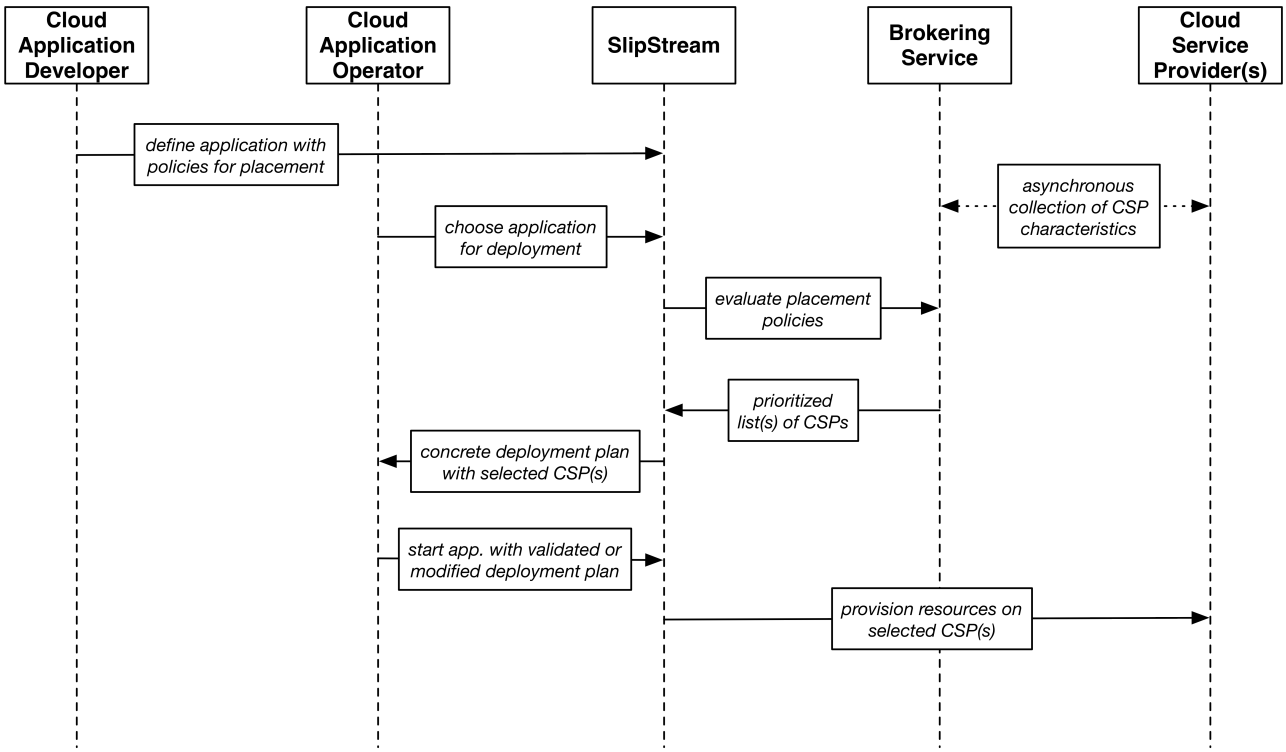

Figure 2: Sequence Diagram of Brokering Interactions

3.2. Requirements

Based on discussions with the developers and use case representatives, we have identified the requirements (listed in Table 1) for the brokering system. These requirements essentially boil down to the constraints that the broker must provide a general query scheme and that the service descriptions must be

both human and machine readable. To cover the full range of selection criteria, the brokering system must be able to incorporate information from external sources.

In keeping with the agile and iterative development processes used by the project, we expect that the requirements will evolve as the brokering service implementations advance and feedback from the use case representatives comes in.

Table 1: Brokering Requirements

ID	Title	Description
1	Machine Readable Service Description Format	CYCLONE consists of a set of interoperating tools that will produce and process the service descriptions. Consequently, the format MUST be a standard format (XML, JSON, etc.) that can be easily processed in a wide variety of programming languages.
2	Human Readable Service Description Format	Humans will need to read the service descriptions to understand provider offerings and to prepare appropriate queries (policies). The service description format SHOULD be easily readable by humans.
3	Endorsement Format	The format used for service endorsements (e.g. certifications, availability metrics, etc.) MUST be the same format as the service description format.
4	Machine Readable Service Query Format	The service query format (policy) may be created or altered by the CYCLONE tools. The policy format MAY be easily readable from a wide variety of programming languages, either it can be a well supported standard or written according to a well defined grammar.
5	Human Readable Service Query Format	Humans as well as tools will be creating the service queries (policies), so the format and grammar MUST be easily readable by humans.
6	Flexible Service Description Schema	The characteristics of the underlying cloud services may vary from service to service (e.g. one based on virtual machines and another based on containers) and over time as new services appear. Consequently, the schema of the service description MUST be flexible.
7	Graceful Handling of Missing or Incorrect Information	For many reasons (e.g. a field isn't appropriate for a service, a tool/human creates an erroneous entry), the service descriptions may not contain information expected by a particular query or may have provided the information in an incompatible format (e.g. string instead of an integer). The matchmaking process MUST respond to malformed queries/descriptions in a well defined manner and MUST accept them without crashing.
8	Join Service Descriptions and Endorsements	For matching on non-functional characteristics, the query language MUST be able to join information from service descriptions and from third-party endorsements.
9	Ordering of Results	A particular service query may return more that one that

		matches the given requirements. The query language SHOULD allow multiple matches to be ranked or ordered. This allows user preferences to be taken into account.
10	Logical Operations	The query language MUST support logical AND and OR operations. Additional logical operations (e.g. XOR) MAY also be supported.
11	Arithmetic Operations	The query language MUST support basic arithmetic operations (add, subtract, multiply, divide) for integer and floating values. Other arithmetic operations MAY also be supported.
12	String Matching	The query language MUST support string equality and inequality operations.
13	String Regular Expressions	The query language SHOULD support string matching based on regular expressions.
14	Array Operations	The query language SHOULD support array operations, for example, be able to define filters on the inclusion and exclusion of values within an array.
15	Service Level Agreement	The brokering implementations must facilitate the exchange of Service Level Agreements in standard or customary formats.

3.1. Brokering Service API

Matchmaking and brokering functionality is sufficiently complex that those deploying CYCLONE may want to have alternate brokering services that emphasize different features or that make different performance tradeoffs. Those people deploying CYCLONE may even want to provide their own, customized implementation of the brokering service. To facilitate use of different brokering service implementations, CYCLONE specifies a simple interface between SlipStream and the brokering service.

The brokering service API consists of a single HTTPS GET request with a JSON message body. The service will return a standard HTTP response with an appropriate status code and JSON body with the results.

The request to the brokering service will be an HTTP GET request on the endpoint of the brokering service. The body of the request will be in JSON format and will contain information about the application topology and resource requirements as well as a selection filter or query with additional requirements (see Figure 3). Complex applications may use different policies for different application components, consequently the request format allows named queries associated with each application component to be evaluated in a single request.[1]

The response from the brokering service must return the HTTP status code 200 and contain a JSON body that follows the example in Figure 3. For each query, a list of maps describing the selected services is returned. Each map contains the “cloud-service-uri” key and may also optionally contain other keys. If an individual query returns multiple results, then the results must be ordered from most to least desirable, as

[1] The query syntax will initially be the native query syntax of the brokering service implementations. Based on feedback from the users and the developers, the query syntax will eventually be specified as part of the API, allowing true interoperability between the brokering service implementations.

specified in the application component's selection query. If a query matches no services, then an empty list must be returned. The returned strings are the unique identifiers for each cloud service.

If any of the queries are malformed, then the brokering service must return the HTTP status code 400. The body of the response must be a JSON document following the error response format in Figure 3. For invalid queries, the value must be a string with an informative error message. For valid queries, the list of service identifiers (properly ordered) must be returned as usual.

The service may also return other appropriate error codes (e.g. 401: authorization required). The body of the response must be an informative error message provided in plain text.

The service must use the HTTPS protocol. The brokering service may require authentication of the clients. If so, the authentication scheme must be compatible with automated use of the API.

Request

```
{
  "system": { … CIMI system resource … },
  "machine_1": "selection query",
  "machine_2": "selection query",
  "machine_3": "selection query"
}
```

Normal Response
HTTP Status 200

```
{
  "machine_1": [
                 { "cloud-service-uri": "id_1", … },
                 { "cloud-service-uri": "id_2", … }
               ],
  "machine_2": [
                 { "cloud-service-uri": "id_3", … }
               ],
  "machine_3": []
}
```

Error Response
HTTP Status 400

```
{
  "machine_1": [
                 { "cloud-service-uri": "id_1", … },
                 { "cloud-service-uri": "id_2", … }
               ],
  "machine_2": "clear error message",
  "machine_3": []
}
```

Figure 3: Brokering Service Request and Response Formats

3.2. Brokering Service Implementations

To demonstrate that alternate implementations can be used, the collaboration will develop complete working brokering services built using two existing prototypes from within the CYCLONE consortium. Given the functionality required of the brokering service, the final implementations are likely to have a similar architecture with common functional blocks.

3.2.1. Existing Prototypes

3.2.1.1. SlipStream Service Catalog

The **SlipStream Service Catalog** is not available in the open-source Community Edition. It is a feature of the Enterprise Edition, which is, however, available to the CYCLONE collaboration. The current implementation

is quite simple: it allows cloud providers to provide detailed descriptions of their offerings (including service locations, capabilities, and pricing) that users then evaluate to select appropriate cloud services.

The SlipStream roadmap already includes features 1) to allow users to query (rather than just list) the service descriptions, 2) to incorporate certification information from the Cloud Security Alliance, and 3) to allow users to evaluate the costs of an application on different services. These additional features will go a long way towards covering the desired range of functional and non-functional characteristics for service matchmaking.

Note that the matchmaking functionality for provisioning (e.g. evaluation of user quotas for accessible cloud services) is a core feature for both the Community and Enterprise Editions. As obtaining the necessary (real-time) information requires the user's cloud credentials, the provisioning matchmaking must remain inside SlipStream and cannot be outsourced to an external service.

3.2.1.2. Open Service Compendium

The **Open Service Compendium** (Slawik, Zilci, Knaack, & Küpper, 2015) was developed within the TRESOR project by TUB. It contains descriptions of different classes of cloud offerings (e.g., IaaS and cloud storage) available for application deployment. It provides Cloud Application Developers and Operators with the functional and non-functional properties of those platforms, as well as a comparison of their costs. It provides a simple Domain Specific Language (DSL) for the service descriptions and for querying the collection of service descriptions (Slawik & Küpper, 2014).

3.2.2. Common Characteristics

While the details of the particular brokering service implementations do not impact the interface between SlipStream and the brokering services, there are common functional blocks that will have to be implemented by any complete brokering service:

- **Cloud Service Provider Database**: A collection of formatted information concerning Cloud Service Providers, including, for example, the types of resources provided, commercial terms (pricing, etc.) for using those resources, capacity, third-party certifications, and metrics.
- **Information Updater**: All of the information can vary over time, although probably at very different rates; consequently, the brokering implementation must also have mechanisms for updating the information regularly, including mechanisms for service providers to update their own information.
- **Matchmaker**: This will take the information provided by SlipStream (from both the application description and from the user) to select acceptable offers from the Cloud Service Providers. In the case that multiple offers are acceptable, then the matchmaker will rank the offers based again on the information passed from SlipStream.

Given the need for a common brokering service interface and for these common functional blocks, the different implementations will likely have very similar architectures are will differ only in the choice of technologies and the range of different types of CSP characteristics that are supported.

4. Implementation Plan

The ultimate goal is to provide two implementations of the CYCLONE brokering service (one developed from the SlipStream Service Catalog prototype and another based on the Open Service Compendium from TUB) that cover the widest possible set of cloud service characteristics and that provide the interface defined in the previous chapter.

In the interest of practicality and to allow the CYCLONE use case representatives to test the brokering service as soon as possible, an incremental implementation plan is proposed.

4.1. Phase 1: Basic Prototype

The goal of the first phase is to demonstrate that all of the components of the brokering system are available to CYCLONE users, although with very limited selection capabilities.

- Modify SlipStream to allow an external brokering service to be configured. The SlipStream system administrator will provide the endpoint and credentials (if necessary) in the SlipStream configuration.
- The SlipStream Service Catalog will be re-implemented following the CIMI specification. The initial query language will be the "filtering" language defined within the CIMI specification.
- A query resource for the SlipStream Service Catalog will be provided that implements the brokering service interface, defined above.
- An interface to the Open Service Compendium will be implemented that follows the brokering service interface, defined above.

Note that the query strings in the initial phase will be the native query formats for the two implementations: a CIMI filter for the SlipStream Service Catalog and a DSL query for the Open Service Compendium.

Note also that this initial version will not pass the application topology (the CIMI "System" resource) to the brokering service. Only the selection filter(s) per machine will be sent and processed.

The two implementations are only expected to implement selections for a limited subset of the application characteristics listed above.

4.2. Phase 2: Common Interface

The goal of the second phase is to pass the application topology to the brokering service and to converge on a common query language and core schema for the two brokering service implementations.

A possible choice of a common query language is SPARQL (W3C, 2013). SPARQL works well with evolving schemas, supports common formats like JSON and XML, allows the querying of third-party information (if properly formatted in RDF), and has mature implementations available in many languages, notably Java for

SlipStream and Ruby for the Open Service Compendium. Moreover, W3C has published this as a recommendation, the W3C's equivalent of a standard.

However, SPARQL is mainly built for querying data and its construction capabilities are quite limited. If the matchmaking results have to be constructed, that is, if prices have to be calculated based on input variables and if different service options and variants have to be selected and their impact on the service offering evaluated, then SPARQL will not be sufficient and other means will have to be evaluated.

- Modify SlipStream to pass application topology information to the brokering service.
- Perform a technical evaluation of SPARQL and other possible query languages.
- Create a minimal common schema using the existing schemas as starting points.
- Modify the SlipStream Service Catalog implementation to support the selected query language.
- Modify the Open Service Compendium implementation to support the selected query language.

The API for the brokering service remains the same, although the query strings will now be in the common query language and the application topology information will be provided.

4.3. Phase 3: Expanded Application Characteristics

The goal of the third phase is to expand the supported application characteristics, demonstrating that information such as third-party certifications or metrics can be used.

- Define what application characteristics will be supported for each implementation.
- Define what third-party information will be available for service brokering. This may include, for example, certifications from the Cloud Security Alliance or metrics from an external monitoring service.
- Define common schemas for application characteristics shared between the implementations.

Note that the application characteristics supported by each implementation may not be identical.

5.Summary

This document specifies the interfaces that will be used for the deployment and management of cloud applications as well as the brokering interface for selecting the optimal cloud services for each application.

Within the CYCLONE architecture, SlipStream handles all of the cloud application deployment and management operations. Cloud Application Developers and Operators will directly use the SlipStream web interface or REST API to trigger management actions.

Previously, the project adopted a strategy where the current SlipStream application model will be used in the short-term with an incremental migration towards the CIMI standard in the longer-term. This strategy also defines the deployment and management interfaces, as both the SlipStream and CIMI models are tightly associated with well-defined APIs. Consequently, the SlipStream API (and its evolution towards CIMI) will be used for the deployment and management of cloud applications. The API is documented on the site http://ssapi.sixsq.com and the details of CIMI can be found in the specification from DMTF.

To provision cloud resources for applications, SlipStream must create a deployment plan. This is done when first deploying the application and again whenever resource re-allocations of the application are triggered. Currently SlipStream users manually select the cloud service to use for each application component. With CYCLONE, SlipStream will be extended to call an external brokering service to select cloud services based on user-defined policies. CYCLONE will provide two implementations of the brokering service: one based on the SlipStream Service Catalog and the other based on the Open Service Compendium.

The brokering service API, defined in this document, consists of a single HTTPS GET request with a JSON message body containing the user's requirements. The service will return a standard HTTP response with an appropriate status code and JSON body with the matching cloud services.

In the interest of practicality and to allow the CYCLONE use case representatives to test the brokering service as soon as possible, an incremental implementation plan is proposed. Three phases have been identified with the following goals for each:

1. Demonstrate that all of the components of the brokering system are available to CYCLONE users, although with very limited selection capabilities.
2. The goal of the second phase is to pass the application topology to the brokering service and to converge on a common query language and core schema for the two brokering service implementations.
3. Expand the supported application characteristics and to demonstrate the use of third-party information.

When completed, SlipStream through the brokering service will allow users to place application components based on a wide range of characteristics including resource availability, resource accessibility, performance, legal constraints, pricing, and application topology.

The incremental implementation strategies for the deployment and management interfaces as well as the brokering service will provide a working platform for evaluation by the project's use cases rapidly.

References

CYCLONE. (2015, July 20). *Complex Application Description Specification.* (C. Loomis, Ed.) Retrieved August 20, 2015, from CYCLONE Project: http://www.cyclone-project.eu/assets/images/deliverables/Complex%20Application%20Description%20Specification.pdf

DMTF. (2015, March 20). *Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol.* Retrieved August 20, 2015, from Distributed Management Task Force, Inc.: http://dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0c.pdf

SixSq. (2015, August 20). *SixSq API*. Retrieved August 20, 2015, from SixSq Documentation: http://ssapi.sixsq.com

Slawik, M., & Küpper, A. (2014). A Domain Specific Language and a Pertinent Business Vocabulary for Cloud Service Selection. *Economics of Grids, Clouds, Systems, and Services* (pp. 172-185). Springer International Publishing.

Slawik, M., Zilci, B. I., Knaack, F., & Küpper, A. (2015). The Open Service Compendium. Business-pertinent Cloud Service Discovery, Assessment, and Selection. *To be presented at GECON 2015. Preprint available at http://arxiv.org/abs/1508.06119* .

W3C. (2013, March 21). *SPARQL Query Language for RDF*, 1.1. Retrieved August 20, 2015, from W3C: http://www.w3.org/TR/sparql11-overview/

Acronyms

API	Application Programming Interface
CIMI	Cloud Infrastructure Management Interface
CAD	Cloud Application Developer
CAO	Cloud Application Operator
CAU	Cloud Application User
CSP	Cloud Service Provider
DMTF	Distributed Management Task Force
IaaS	Infrastructure-as-a-Service
IT	Information Technology
OSC	Open Service Compendium
PaaS	Platform-as-a-Service
REST	Representational State Transfer
RDF	Resource Description Framework
SaaS	Software-as-a-Service
SLA	Service Level Agreement
VM	Virtual Machine

<END OF DOCUMENT>