
# Building Services for the Kumori PaaS

Kumori <info@kumori.systems> v1.0.0., May 2018

# Table of Contents

This document describes how to build service applications to be deployed on ***Kumori***.

# 1. Introduction

***Kumori*** is a Platform as a Service designed to manage the life cycle events of services deployed on it. The goal is to obtain elastic services adapting to varying running conditions with minimal effort and expense on the part of the service provider, letting it focus on the development of the service itself, without getting bogged down by the details of service management tasks.

***Kumori***'s approach is to guide service applications to fit a set of architectural patterns we refer to as the ***Kumori service model.***

In a nutshell, ***Kumori*** service model views a service application as a set of interconnected components, each one playing a different role within the service. Components contain both the code that implements their logic, as well as a description of how they can be connected to other components through their communication channels. Service applications contain descriptions of how its components are actually connected through connectors with particular semantics, as well as a description of how they can be connected to other service applications through their communication channels.

The rest of the document is organized as follows. We start with an overview description of ***Kumori***'s service model, including what a service is in ***Kumori*** and what its structure is.

Afterwards we describe how to define a component structurally to fit within the service model, to continue with a description of how components are connected together into a topology describing the service application. The definition of service application connections is also introduced.

Once we are done describing the structure of service applications and components, we continue with a description of the API ***Kumori*** exposes for component developers. In particular we present the life cycle events to be taken care of by the component's code, to finally delve into the channel-related API.

Appendices show details of the runtime environment in which component code will be run.

# 2. Services and Service applications

In the ***Kumori*** model a Service is, typically, a long-running, stateful computation obtained as the result of executing a set of software components previously deployed within suitable computing environments, offering functionality accessible through some communication channel, and providing guarantees about how these functions are delivered.

The long-running nature of services lead them to accrue "state" which must have some durability and consistency properties in order to satisfy the semantics of the functions they make available. This characteristic makes them difficult to manage except in the simplest situations.

Many of today´s services use HTTP channels to be accessed, usually offering a mix of web page (HTML5) content serving (basic HTTP model), and request/reply REST calls initiated by client

software (which more and more often resides within the downloaded HTML page)

Simple services can be built as a three-layer application: the front-end takes care of interacting with clients of the service. A back-end takes on state-persistence duties, being often implemented with some sort of database technology.

More complex systems require more sophistication in the form of a richer set of components interconnected forming a more complex topology, with each component specializing in particular tasks. What's more, some of those components may already exists, needing only to be integrated into a wider solution.

Service scalability has always been a main worry of designers of applications destined to become services. The worry focuses on the ability that the whole system has to adapt to load changes while fulfilling the guarantees of the service. A trivial form of adapting to higher loads is providing more powerful computing environments where the programs execute (more memory, cpu, faster drives,...). This way of dealing with higher demands, known as *vertical scalability* has the inconvenient of not scaling itself: there is a technological limit on the amount of resources a particular executing program can receive from its environment.

Scalable service design must, thus, take into account adapting to the load using another approach: replicating some of the components, using a technique referred to as *horizontal scalability*. While replication must be used to ultimately achieve scalability, applications using replication must be designed carefully to avoid the pitfalls of inconsistent state across different replicas of the same component.

Finally, long-lived services must also deal with failures of its components. Failures are a fact of life on any system, computing systems included. Failures may produce total or partial loss of the state associated with the component that fails. The whole service must be able to deal with this situation, as failure of one component can also potentially affect established communication links and in-transit messages.

The ***Kumori*** platform's goal is to help service application designers to cope with the challenges of designing service applications to deploy them as scalable services capable of elastically adapting to the changing circumstances of their environment (load changes, failures, upgrades).

## 2.1. The *Kumori* service model

***Kumori***'s high level definition of what a service is fits that given earlier on. Its service model includes a structural description of a service application, as well as the semantics of deployment.

In ***Kumori*** we need to consider the following elements:

*Service Component (a.k.a. Component)*

An autonomously deployable piece of software, adhering to a particular life-cycle directed by ***Kumori***.

*Role*

A particular occurrence of a component within a *Service Application*, fulfilling a specific purpose.

*Channel*

A communication pathway of various types associated to the definition of a component.

*Channel protocol*

The protocol that runs through a channel.

*Configuration Resources*

Resources, organized as a a list of named and typed entities, declared by *Service Component*'s and *Service Application*'s. Resources can carry data accessible by the code in components. Resources are intermediated by the platform, and may carry extra information to be used by the platform/runtimes.

*Configuration Parameters*

Purely application-dependent "`data`" needed by components and the service application, organized as a list of named and typed entities, declared by *Service Component*s and *Service Application*s.

*Types*

A description of the characteristics of the various elements used in ***Kumori*** (parameters, resources, runtimes, ...). Expressed as versioned URIs with a concrete structure.

*Runtime*

Software stack expected by a component's software to run.

*Service Application*

A ***Service Application*** is the specification of a set of **Roles**, each one implemented by a *Service Component*, related through a *Topology* linking those roles through the channels of their respective components by means of connectors.

*Channel Connector (a.k.a. Connector)*

Service model elements joining channels from components within a *Service Application*. ***Kumori*** defines its own set of connectors with its channel-binding restrictions and semantics.

*Role Instance*

The result of executing the component associated with a Role on top of the runtime that component requires.

*Service*

The result of deploying a *Service Application*. This deployment typically involves the creation of several Roles Instances for each one of the roles declared in the service, configured according to their declared configuration, and the interconnection of their channels, according to the model of the service.

*Topology*

How *Service Component*s are wired together through channels and connectors.

In what follows we are going to discuss each one of those elements in detail.

### 2.1.1. Identifying *Kumori* model elements

In ***Kumori***'s service model, many of the above elements must have a unique name identifying them. ***Kumori*** uses URIs as a means to specify unique identifiers for elements that must be uniquely named.

All URIs used to globally identify elements of ***Kumori*** follow a similar structure, schematized as follows

`eslap://<domain>/<elementtype>/<name>/<version>`

Where `<elementtype>` signals the kind of element being named. Currently ***Kumori*** requires unique id's for the following elements

`component`
: Used to name *Basic Component*s. See the components section.

`service`
: Used to name *Service Application*s. See the Composing a *Service Application*: topology section.

`runtime`
: Used to name a runtime kind. See the Component programming section.

`protocol`
: Used for protocol names. See the Channel protocol section.

`parameter`
: Used for parameter type names. See the Configuration settings section.

`manifest`
: Used by ***Kumori*** itself to name each one of the manifest specifications it makes available for the rest of elements.

`resource`
: Used for resource type names. See the Configuration settings section.

`channel`
: Used for channel type names. See the Channels section.

Next, `<domain>` must be a domain name belonging to the author of the component. The rest of the URI should uniquely identify the element within this domain. `<name>` is an arbitrarily deep path name. It works as the generic name of the element within its domain.

Finally, `<version>`, specifies which variation of the element is being named. `<version>` is a triplet of integers following the semantic versioning conventions:

```
<version> == <Major>_<Minor>_<Revision>
```

The `<version>` URI component is required as most elements will mutate in time and it is

important to be able to capture the relationships between them to make policy decisions on upgrades.

Thus, an example of a URI identifying a component would be:

```
eslap://my.domain.com/component/maincomp/1_2_0
```

Elements are defined within manifests, JSON files following a structure like this

```
{
  "spec": "http://eslap.cloud/manifest/<elemnttype>/<version>", (1)
  "name": "eslap://<domainname>/<elementtype>/<elementname>/<version>",
(2)
  ...
}
```

(1) Specification version to be found in the manifest

(2) ID of the element being defined.

There is a set of builtin elements provided by ***Kumori*** whose manifests are publicly available. Furthermore, some elements can be defined by ***Kumori*** users, notably, *Service Application*'s and *Basic Component*'s, but also other parameter types (derived from existing ones) as well as runtimes and protocols (also derived from existing ones). More about this in the appendices.

## 2.2. Components: specifying microservices

A component should be seen as an autonomously runnable executable. As such, out of a component it is possible to obtain many different running component instances. Each *Component Instance* runs in isolation of any other *Component Instance*, however, a component typically needs functionality it does not implement in order to carry out its function. We refer to such functionality as its *dependency set*.

Also, a component is barely useful if it cannot offer functionality others can access. We refer to this functionality as the *provided function set*. Obtaining and providing functionality requires communication between components. Consequently, our model for components makes component instances capable, a priori, of communicating through *channels* of various characteristics.

In ***Kumori***, a *Service Component* is designed to be included within a *Service Application*, playing a *role* that makes it capable to interact with other included components in the service, each playing its own *role*. The main structural elements of a *Service Component* are thus its channels, which each of its instances expects to have available when run.

In addition to communication channels, components need access to configuration. As with channels, *Component Instance*s expect to be able to access the configuration settings declared by the component.

*Service Component*s are specified by means of a manifest that must provide the component's

identifier, as well as its channels and configuration settings.

### 2.2.1. Channels

A component can have one or more communication channels through which messages can be sent, or received or both. In ***Kumori***, communications between components are message based, where each message is structured as a collection of segments. The entire collection is atomically sent and received.

A component instance is injected with instances of its declared channels properly configured, and valid for the whole lifespan of the component's instance.

Each channel has a name and a type. The name of the channel is part of the definition of the component and is local to that definition. ***Kumori***'s runtime API uses channel names to let component code retrieve the channel object and send/receive messages through it.

Finally, a channel's type indicates how the channel can be used from within the component. It also restricts the kinds of connectors (see later) to which it can be connected. ***Kumori*** currently defines the following channel types:

*request*

Messages can be sent and received through this channel, following a request/reply pattern. A component with this kind of channel can only receive messages as responses to previous request messages sent through it. The `req` channel should be declared by components that request functions from others.

*reply*

This is the complement of the `req` channel. Thus messages can also be sent and received. In this case, sending is only allowed as a response to a previous request message.

*send*

A channel with this type can only be used to send messages. No message can be received through a channel of this type.

*receive*

A channel with this type can only be used to receive messages, not being possible to send any message through it.

*duplex*

Channels with this type allow unrestricted sending and reception of messages. As we will see, `duplex` channels can be connected through `complete` connectors, that need destination information to be provided with each send. Thus `duplex` channels expose the membership of the channels attached to a connector, and make that available to components with such channels.

#### Provided/Depended channels

Component channels are further classified within a component as *provided* and *required* channels. *Provided* channels must be used to access the *provided function set* of the component, while *required* channels work more like an implementation detail of a component, allowing it to access its dependency set. Thus, they indicate what external functionality is needed by the component to

implement its function.

#### Channel protocol

Optionally, channels can be associated with a named protocol element. This association is, for now, merely informative, although it can be used by tooling to infer potential trouble spots.

### 2.2.2. Configuration settings

A component declares the set of configuration settings it expects its instances to get when they start executing.

In ***Kumori*** we divide settings into *resources* and *parameters*.

#### Configuration resources

Configuration resources represent entities that must be mediated by ***Kumori***. They are usually subjected to some form of restriction, and oftentimes, contain information that is used by the platform itself, or the runtimes of the components.

Configuration resource types are defined only by ***Kumori***, and some of their information is made available through the ***Kumori*** API to the components finally consuming them.

To create or allocate a resource, users must detail its characteristics in a manifest, and give it a unique name which will be used to refer to it.

For example, a `persistent volume`:

```
{
  "spec" : "eslap://eslap.cloud/resource/volume/persistent/1_0_0",
  "name" : "eslap://mydomain/resources/volume/mypersistent",
  "parameters" : {
    "size":"100"
  }
}
```

Similarly, a `vhost` resource can be created like this:

```
{
  "spec" : "eslap://eslap.cloud/resource/vhost/1_0_0",
  "name" : "eslap://mydomain/resources/vhost/wwwmywebsitecom",
  "parameters" : {
    "vhost": "www.mywebstite.com"
  }
}
```

#### Predefined resources

***Kumori*** provides a set of predefined resources for every component. They cannot be declared on

a component, although initial values for some of them can be specified on deployment.

*__cpu*

Amount of CPU units given to an instance of a component. Each CPU unit corresponds to a specific amount of raw compute power.

*__memory*

Amount of main memory units needed by an instance of a component. Each unit corresponds to a certain amount of physical RAM plus a fraction of that value of swap.

*__ioperf*

Amount of I/O performance units available to an instance of a component. Each IOperf unit corresponds to a specific disk bandwidth rate and to a specific rate of disk operations per second (IOPS) the instance can perform. **The way disk performance is configured is temporary, and will be improved in future releases.**

*__bandwidth*

Maximum rate (in Mbps) of data transmission through network interfaces.

The above set of resources is applicable to each instance of a component, and it is conceivable that each instance gets different values for those resources, depending on scaling decissions made by ***Kumori***.

Actual unit values are described in Resource units definition and implications.

Besides the above, ***Kumori*** deals with collective properties of a *Service Component*, that is, properties of the set of its instances:

*__instances*

Number of instances of a given component to be maintained running.

*__maxinstances*

Maximum number of instances of a given component allowed to be running simultaneously.

*__resilience*

Number of failures needed to take down all instances of a component. The resilience is specified by levels which indicate various types of failures by likelihood (e.g. normal data center failures, vs whole datacenter failure, vs whole region failure, etc...).

The three properties are correlated, as the number of instances must be sufficient to cover the level of resilience demanded.

**Configuration parameters**

Configuration parameters are named pieces of application data which can vary on each deployment of a service. Each configuration parameter has a type, specified by a type name, constructed out of a URI with element type `parameter`. When not specified, the type is assumed to be a valid JSON object.

Currently ***Kumori*** provides a set of predefined types that developers can use directly. In addition,

developers can define types derived from the ones provided. See the appendices for a relation of the available types.

***Kumori*** toolchain can use these declarations to verify sanity, as well as to support user interactions in tools.

### 2.2.3. Runtime

Finally, each component needs to declare which runtime it needs so that it can be run properly. The runtime determines the characteristics of the execution environment an instance can expect to run under.

Runtimes are named elements in ***Kumori***, with an element type of `runtime`. It is possible to define additional runtime types derived from the existing ones. ***Kumori*** provides a set of predefined runtimes, presented in the appendices.

### 2.2.4. Defining *Service Component*s: the Component manifest

All the above elements come together when building a component declaration within the component Manifest. The component Manifest is a `JSON` file, whose structure is demonstrated by the following example component definition.

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0", ①
  "name": "eslap://some.domain/component/fe/0_1_1", ②
  "code": "fe-code-blob", ③
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0", ④
  "channels": {
    "provides": [{ ⑤
      "name": "entrypoint",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0", ⑥
      "protocol": "eslap://some.domain/protocol/message/http/0_1_0", ⑦
    }],
    "requires": [{ ⑧
      "name": "sessiondb",
      "type": "eslap://eslap.cloud/channel/request/1_0_0",
      "protocol": "eslap://some.domain/protocol/message/dict/2_3_2"
    }, {
      "name": "sitedb",
      "type": "eslap://eslap.cloud/channel/request/1_0_0",
      "protocol": "eslap://some.domain/protocol/message/data/2_0_0"
    }]
  },
  "configuration": { ⑨
    "resources": [{ ⑩
      "type": "eslap://eslap.cloud/resource/volume/volatile/1_0_0", ⑪
      "name": "site", ⑫
      "properties": { ⑬
        "best_effort": true
      }
    }, { ⑭
      "type": "eslap://eslap.cloud/resource/volume/persistent/1_0_0",
      "name": "forever"
    }],
    "parameters": [{ ⑮
      "name": "database",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0",
      "default": "site"
    }, {
      "name": "prefix",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "profile": { ⑯
    "threadability": "*"
  }
}
```

① Manifest specification version.

(2) Component name.

(3) Name of the *blob bundle* containing the executable component pieces to be deployed.

(4) Runtime declaration. This is the runtime under which this component has been designed to run. In this case, the NATIVE runtime.

(5) Provided channels section of the manifest. This is an array of channel descriptions. All channels in here define what kind of function this component provides.

(6) Type of the channel.

(7) Protocol to be run over the channel. This is optional in current version.

(8) Required channels section. These channels are required for the component to operate properly, requesting services from others, or obtaining information.

(9) Configuration section. With two parts, one for resources, another for data parameters.

(10) Resources: array. Each must be specified/reserved by deployment. The runtime takes care of handling their "installation". The application may access their properties through the API.

(11) Type of first resource: a volatile volume. The definition of this kind of resource carries several properties. One is the efforts made to maintain the same volume in case of instance restart.

(12) Name of resource. We can use this name in the API to access properties of the volume. In addition, volumes have a mountpoint property. When not provided, it is computed from this name, and accessible to the application code.

(13) We fix some of the parameters associated to the resource type. In this case we declare it to be best_effort: the platform will try not to lose the state in case of failures/upgrades.

(14) Another volume resource. This one is of type persistent.

(15) Data configuration parameters.

(16) Performance-related properties. To be handled by the platform.

In the appendices we show the `json-schema` for a component's manifest.

## 2.3. Composing a *Service Application*: topology

Service applications are obtained making autonomous Components play different *Roles* within the *Service Application*, and linking the channels of those roles via *Channel Connectors*.

Specifying a service application requires specifying its topology (essentially a labelled graph), which in ***Kumori***'s case translates to providing the set of elements we present in what follows.

*Service Roles*

The runnable parts of a *Service Application* are its roles. Each role is placed in a *Service Application* to interact in a certain way with the rest of the roles of the service, or, possibly with other actors outside the service. Any given Role is implemented by a *Service Component* capable of providing its functionality. Thus, a Role adopts the properties of the *Service Component* implementing it, in particular its channels and configuration parameters.

Roles in a *Service Application* are provided with names local to that *Service Application*. Those names distinguish roles from each other.

Finally, note that a role can be multiply instantiated within a running service. Each *Role Instance* will be the result of instantiating the role's associated component with the set of parameters derived from the service configuration, and with its channels connected according to the model representing the service application.

*Service Configuration*

It serves the same purpose as the configuration we considered for *Service Component*s. Thus, we have two sets of configuration entities, resources and parameters. The types of resources and parameters supported are the same, and they receive names local to the service application.

A *Service Application* must provide a means of propagating these configuration elements to each one of its roles. The current version of ***Kumori*** allows resources to be directly propagated to resources in roles. For parameters ***Kumori*** relies on scripts that must be provided within a *Service Application* bundle, transforming the service parameters into role parameters. This is described in more detail in section Putting it all together: The *Service Application* Manifest.

*Channel Connectors*

Roles within a *Service Application* cooperate to implement a service application. To do so they need to interact with each other. The *Service Application* specifies how they can interact by means of *Channel Connector*s linking together channels from the roles.

A service connector establishes a relationship among a set of channels from different roles in a *Service Application*. The *Channel Connector* type determines what types of channels can be related by it and what the semantics are as a result when instances of the roles affected try to communicate via their related channels.

***Kumori*** currently specifies three kinds of connectors:

*Load Balancer* (**LB**)

A load balancer connector is designed to relate a set of `request` channels with a set of `reply` channels. Request channels must appear in the `depended` channel section of the component implementing the role supplying the channel. On the other hand, the `reply` channels must be specified in the `provides` channel section of the corresponding components.

Its semantics are simple. When a role instance sends a message through one of its `request` channels attached to the LB connector, the connector selects one of the instances of the roles supplying `reply` channels attached to the connector. ***Kumori*** then delivers the message to that instance through the relevant `reply` channel supplied by its role. When the instance receiving the request sends back a reply, it should do so through the same channel it received the request through, and the LB connector delivers it to the requester role instance through the `channel` it used to send the request.

At this point in time, ***Kumori*** does not allow specifying policies as to how channels and instances should be selected to receive the request message.

*Publish-Subscribe* (**PS**)

A Publish/Subscribe connector brings together *provided* `send` channels and *required* `receive` channels. The current ***Kumori*** semantics for this connector are also simple.

Messages sent through one of the attached *send* channels are broadcasted to all attached *receive* channels (thus all role instances will have that message delivered to them through the corresponding attached channel).

***Kumori*** currently allows the specification of filters for messages that must be delivered through a particular *receive* channel. They are discussed in a later section.

*Complete Connector (**FC**)*

A complete connector links `duplex` channels from roles in a service. As explained earlier, messages sent through a duplex channel must indicate the destination, which must be one of the channel/instance combinations from roles providing a channel to the connector. In order for sender instances to be able to address the messages they send through a duplex channel, the Complete Connector interacts with the channel to notify changes in the available destinations set.

It is up to each instance to handle the complexities of communication among all instances with channels attached to the connector.

*Service channels*

As with components, service applications must expose their offered service, for end users of the service or for other services. Also like components, services will oftentimes require functionality from other services, and will need to establish a *Service Level Agreement* (SLA) to obtain it.

***Kumori*** allows a *Service Application* to declare its *entry* and *dependency* points by means of *channels*.

We have seen channels in the context of describing components. At the service level, channels behave similarly. The difference is that the service manifest must internally link those channels through connectors to internal component channels, as components handle ultimately communications.

# 2.4. Built-in services

***Kumori*** supplies a series of predefined services that service application developers can readily use with their own services. These services are made available to complete the functionality of the platform.

Currently there is only one built-in service: the HTTP Inbound (also referred to as SEP, for Service Entry Point). The SEP allows incoming HTTP connections from the internet to services deployed on ***Kumori*** through a **reply** channel that the service must expose implementing the `message/http` protocol.

## 2.4.1. The HTTP Inbound service

This service provides the means to expose a domain name through which ***Kumori*** deployed services can be reached using the `http` protocol. Usage of this service will consist on configuring the domain for which requests should be served. The SEP exposes a *required* `request` channel implementing the `message/http` protocol. Through this channel the SEP can be linked to a customer-deployed service exposing a corresponding `reply` channel implementing the

message/http protocol or a derived one.

The SEP service takes care of primary load balancing, ending the http connection, for which it should be suitably configured according to its pseudo-manifest that we present here:

```
{ (1)
  "spec": "http://eslap.cloud/manifest/service/1_0_0",
  "name": "eslap://eslap.cloud/services/http/inbound/1_0_0", (2)
  "channels": { (3)
    "requires": [{
      "name": "frontend",
      "type": "eslap://eslap.cloud/channel/request/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }]
  },
  "configuration": {
    "resources": [{
      "name": "server_cert", (4)
      "type": "slap//eslap.cloud/resource/cert/server/1_0_0"
    }, {
      "name": "vhost", (5)
      "type": "eslap://eslap.cloud/resource/vhost/1_0_0"
    }],
    "parameters": [{
      "name": "TLS", (6)
      "type": "eslap://eslap.cloud/parameter/boolean/1_0_0",
      "default": "false"
    }, {
      "name": "clientcert", (7)
      "type": "eslap://eslap.cloud/parameter/boolean/1_0_0",
      "default": "false"
    }]
  }
}
```

(1) Note that this is a simplified manifest. Any detail pertaining to the implementation of the component is left out, as the platform itself is providing it.

(2) The service name is provided by ***Kumori*** itself.

(3) We can see that only required channels are specified. Requests must arrive to this service via HTTP. The service balances requests and passes them to the service connected through its required channel.

(4) Note that the server certificate is specified as a resource. To be used only when TLS is enabled. Passing it as a resource is a good practice helping to insure secrecy of its contents.

(5) Note also that the `virtual host`, is passed as a resource too.

(6) The TLS flag indicates whether https termination must be performed.

(7) Flag indicating whether the client must authenticate via a certificate. Only applies if TLS is true.

Note that in this example we introduced a simplified manifest for the built-in service. This is natural, as ***Kumori*** has no need for implementation-related details.

# 3. Putting it all together: The *Service Application* Manifest

We are now ready to put together the different pieces we have seen so far and provide a complete structural description of a service application.

We have already seen many of the elements we should take into consideration. We now will introduce some missing pieces having to do with the interface presented by the service to the outside world.

To do this, we consider a toy example of a service application. This application integrates two components: a front-end, serving HTML5 content, and serving HTTP requests; and a session db, storing session-related data for sharing among all instances of the front-end.

This basic *Service Application* needs access to a data base, which is not provided from within the service, generating a dependency that must be satisfied from the outside.

The structure of the service is somewhat simplistic, as the session db role does not provide a means to coordinate the contents among its potentially several instances.

The manifest of the front-end component has been shown in section Defining *Service Components*: the Component manifest. The SEP service's manifest has been shown in section The HTTP Inbound service and the next listing shows the manifest for the second component, the session db.

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://some.domain/component/sessiondb/0_1_1",
  "code": "sessiondb-code-blob",
  "runtime": "eslap://eslap.cloud/runtime/native/1_0_0",
  "channels": {
    "provides": [{
      "name": "dictapi",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://some.domain/protocol/message/dict/2_3_2"
    }]
  },
  "configuration": {
    "resources": [{
      "type": "eslap://eslap.cloud/resource/volume/volatile/1_0_0",
      "name": "dict",
      "properties": {
        "best_effort": true
      }
    }],
    "profile": {
      "threadability": "1"
    }
  }
}
```

The *Service Application* manifest describing its topology can be finally shown here. This kind of manifest shares some elements with that of a component. In particular it also declares configuration and endpoints. Beyond these similar elements, it also needs to declare roles, connectors and the rules to follow to propagate configuration to the roles.

```
{
  "spec": "http://eslap.cloud/manifest/service/1_0_0", (1)
  "name": "eslap://eslap.cloud/service/example/0_0_1",
  "code": "xSDDFjjnj ...ASAS",
  "configuration": {
    "resources": [{ (2)
      "name": "volatile_disk",
      "type": "eslap://eslap.cloud/resource/volume/volatile/1_0_0"
    },{
      "name": "session_disk",
      "type": "eslap://eslap.cloud/resource/volume/volatile/1_0_0"
    }],
    "parameters": [] (3)
  },
  "roles": [{ (4)
    "name": "fe",
    "component": "eslap://some.domain/component/fe/0_1_1",
    "resources": { (5)
```

```
      "site": "volatile_disk"
    },
    "parameters": { ⑥
      "prefix": "mydeploy"
    }
  }, {
    "name": "sessions",
    "component": "eslap://some.domain/component/sessiondb/0_1_1",
    "resources": {
      "dict": "session_disk"
    }
  }],
  "channels": { ⑦
    "provides": [{
      "name": "service",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }],
    "requires": [{
      "name": "db",
      "type": "eslap://eslap.cloud/channel/request/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }]
  },
  "connectors": [{ ⑧
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "endpoint": "service"
    }],
    "provided": [{
      "role": "fe",
      "endpoint": "entrypoint"
    }],
  }, {
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "role": "fe",
      "endpoint": "sessiondb"
    }],
    "provided": [{
      "role": "sessions",
      "endpoint": "dictapi"
    }]
  }, {
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "role": "fe",
      "endpoint": "sitedb"
    }],
    "provided": [{ ⑨
      "endpoint": "db"
```

```
      }]
    }]
  }
```

(1) The manifest type is now `service`.

(2) Resources needed by the components behind the service's roles must only be declared when sharing of the same resource among roles is needed. Otherwise, in order to simplify manifest structure, resources needed by components will be directly assigned in the service management tools (deployment/maintenance).

(3) In this case, no paramters are used. The mechanism will be described next.

(4) Section that declares roles. Role declaration consists on assigning the role a name, and associating it with a component, for which we use the component's URI identifier.

(5) Note that within each role we can directly assign the resources its associated component needs from those declared in the service. Service-level resources not appearing here, will not be available to the instances of the role.

(6) Also note that at this moment we can fix the values of configuration parameters for the particular role. For instance, in case we used the same code for different roles, with behaviors determined by a switch, this would be the place to set the value of such switch.

(7) Channels are declared similarly to how we do it for components. There are provided and required channels. They should be linked via connectors with internal role channels or other services channels (for example SEP channels).

(8) The connectors section declares those connectors used to link together endpoints from different roles in the service among themselves or with endpoints at the service level. It is an array of channel connectors, as described in section [s-service-connectors]. Each connector is specified by its type, and by a pair of collections of channel endpoints, the `depended` collection (those channels in the `requires` collection of their component), and the `provided` collection (likewise, channels in the `provides` collection of their component).

Each channel is specified by the role providing it, plus the local name of the endpoint in the component associated to the role.

(9) In the case of a channel declared by the *Service Application* itself, the role name is omitted. In this case we should take into account that the endpoints *required* by the service are *provided* to the internal roles for connection.

Likewise, endpoints provided by the service are actually *required* towards the internal roles of the service.

## 3.1. Configuration propagation

In the previous example we have seen that a *Service Application* can define configuration as a component does. In this configuration we can have service-level resources, potentially shared by many roles in the service, as well as configuration parameters. In both cases, specification is identical to that used for components.

Shareable resources declared in the *Service Application* must be directly linked to those roles making use of them. Once the resources are actually assigned to a service resulting from the deployment of the *Service Application*, then they are propagated to the instances of the

components implementing the roles to which they have been linked in the *Service Application* manifest.

### 3.1.1. Propagating configuration parameters

Configuration parameters are pure data, and they are made sense of only by the application code.

Each component declares its own set of parameters, and can provide default values. Component code can access these parameters via the runtime API provided by ***Kumori***.

At the service level, other parameters can also be declared for the *Service Application*, the intention being that the parameters needed by all the components be derived from the service-level parameters, which begs the question of how are those component parameters actually computed out of the values provided when the service is in deployment.

The procedure is actually quite simple, and depends on accompanying a *Service Application* with a set of nodejs modules, one per role declared. Thus, if a role with name `RNAME` is declared in the *Service Application*, then there is a nodejs module with name `RNAME.coffee` exporting a function, which we refer to as `FRNAME`. This function takes a JSON object in and returns a JSON object out. The input JSON object has an entry per configuration parameter declared at the service application. The JSON object returned has an entry per configuration parameter declared in `RNAME`'s component.

If no `nodejs` module exists associated to the role, then we assume `FRNAME(X)==TREE(RNAME)`, where `TREE(RNAME)(X) == X.RNAME`. That is, if propagation is uncomplicated, the service declares as one of its configuration parameters one with a role's name, and a JSON type, without needing to provide the nodejs module. On deployment the JSON object is filled with the parameters needed for that role.

Let's assume that at deployment time, the set of *Service Application* parameters have been assigned values either by directly specifying them or by taking the defaults in the manifest of the *Service Application*. Let us refer to this set of parameters as `SCP`.

Then, parameter `P` from role `R` associated to component `C` will receive value `V` if and only if:

1. `P` has been fixed in the *Service Application* manifest, with value `V`
2. Else, `FR(SCP).P==V`
3. Else, `FR(SCP).P==undefined`, and the default value for `P` in `C` is `V`

It is an error to leave any parameter with no assigned value

# 4. Deployment

Given a *Service Application*, we can convert it into a *service* by deploying it on a ***Kumori*** stamp.

Initial deployment is specified via the *deployment manifest*. The deployment manifest refers to the *Service Application*, and provides initial configuration for the service and its roles.

To illustrate its components let us show a potential deployment manifest for the *Service Application* we just explored.

```
{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",
  "servicename": "eslap://eslap.cloud/service/example/0_0_1", (1)
  "configuration": { (2)
    "resources": {
      "volatile_disk": "300", (3)
      "session_disk": "150"
    },
    "parameters": {} (4)
  },
  "roles": { (5)
    "fe": { (6)
      "resources": {
        "__instances": 2,
        "__cpu": 2,
        "__memory": 3,
        "__ioperf": 3,
        "__iopsintensive": true,
        "__bandwidth": 100,
        "__resilience": 2
      }
    },
    "sessions": {
      "resources": {
        "__instances": 1,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      }
    }
  }
}
```

(1) A deployment manifest MUST refer to the service application being deployed.

(2) The manifest must include values for the service-level configuration.

(3) Most resources must be referenced by resource token, assigned to the resource after registration of the resource. In this case, however, the resource is volatile disk space, which is directly represented.

(4) Service has no parameters. It there were, they would simply be given.

(5) The roles section spells out properties for each one of the roles in the *Service Application*, thus, for each role, we must have an entry with its name.

(6) For any given role, all its intrinsic resources must be specified.

## 4.1. Bundles

In previous sections we have presented the elements of ***Kumori***'s service model and how they are represented with manifests.

In at least two cases, *Service Application* and *Service Component*, it is necessary to present not only the manifests, but also some extra information. In the case of a *Service Application* we need to provide the nodejs modules that help propagate configuration parameters. In the case of a *Service Component* we need to provide the actual code of the component, and so on.

User-definable elements need to be registered onto ***Kumori*** before they can be used in a deployment. To facilitate this process, ***Kumori*** accepts *bundles* encapsulating all the material necessary to register an element with ***Kumori***.

A *bundle* is nothing more (and nothing less) than a directory structure where the top directory contains a file with name *Manifest.json*. We can then talk about a *component* bundle, when its top level `Manifest.json` file is that of a component. A Service Application bundle is one with its top level `Manifest.json` file containing a *Service Application* manifest, *et cetera*.

Besides the above mentioned manifest, a Bundle can recursively nest other bundles, forming a tree of bundles. One type of bundle that cannot contain any more bundles is what we refer to as a *blob bundle*. Blob bundles are designed to carry code or other non-interpreted material. The manifest for a blob bundle is simply,

```
{
  "spec" : "http://eslap.cloud/manifest/blob/1_0_0",
  "name" :"fe-code-blob", (1)
  "hash" : "xSDDFjjnj ...ASAS" (2)
}
```

(1) Free string, but must be unique within its immediate containing bundle.

(2) If the blob is presented as an archive, its SHA1.

The blob bundle itself must contain either one archive (the blob) in one of the packaging formats admitted by the platform or one folder (the contents of the blob).

Even though in our description nesting of bundles is arbitrary, it is often advisable to directly nest only those bundles carrying elements which can be referred from the element in the nesting bundle. E.g., a *Service Application* bundle nesting *Service Component* bundles, but not the other way around.

Finally, it may be occasionally useful to group several bundles within a folder, so that related elements are properly organized. This situation is easily detected by the absence of a *Manifest.json* file within the grouping folder. In those cases, bundle processing ignores the grouping folder and processes its contents as if they were in its parent folder (we could say that non-bundle folder structure is flattened when processing).

Bundles can be packaged in many ways. ***Kumori*** can accept zip-packaged bundles, directory-packaged bundles (as subdirectories of other bundles), and git-packaged bundles. Other formats may be supported in the future.

Some bundles need "code" blobs. These must be included within a *blob bundle*. Code-carrying *blob bundles* must be unambiguously associated with the bundles referring to them. This match is achieved via the `code` attribute in their manifest, which must match the `name` of the *blob bundle* containing its code.

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://some.domain/component/fe/0_1_1",
  "code": "fe-code-blob",
  ...
}
```

### 4.1.1. Referencing git bundles

As mentioned above, it is possible to reference git bundles. This is done via a `Bundles.json` file with the following format:

```
{
  "spec" : "http://eslap.cloud/manifest/bundles/1_0_0",
  "bundles" : [{
    "repository" : "https://gitlab.somewhere.far/user/project.git", (1)
    "branch"     : "branchName",
    "subdir"     : "optionalSubdirectory"
  },{
    "site" : "http://some.domain/somearchive.zip"                   (2)
  }]
}
```

(1) We can refer to https-based accessible repositories. In this case the repository must be public.

(2) Bundles can also be referred by a URL to a zip file.

When a `Bundles.json` file is found within a bundle, it is processed bringing the directories/packages referenced from those repos as bundles themselves.

### 4.1.2. *Service Application* bundles

*Service Application*s must carry with them nodejs modules to compute parameter propagations to the roles within the application. As explained earlier, each role declared within a *Service Application* should have a module with its name (suffixed with .coffee) that can be run to get a function computing the target parameter values for the role's component.

***Kumori*** expects those modules (all of them) to be in the code bundle of the *Service Application*. What is more, all dependencies needed by those functions should be part of that code bundle.

# 5. Registering elements with a *Kumori* stamp

As mentioned earlier, elements must be registered in ***Kumori*** before they can be used in a deployment. This implies registering them on the stamp.

In ***Kumori*** registration is done by using the REST API of the `Admission` service.

## 5.1. Using the *Kumori* admission API

***Kumori*** exposes a REST API that can be used to register elements to the stamp. Authentication through this interface is token-based, where the token must have been obtained through the user portal.

The user wishing to send a bundle for registration (or to produce a deployment) must send a POST `bundles` on the `Admission` URL publicized by the stamp provider, tipically `https://stamp-of-interest.kumori.cloud/admission`, and stream a file which must be either a `Bundles.json` file or a packaged bundle file (zip).

## 5.2. Linking services

A *Service Application* declares a set of dependency channels. These channels should be bound to specific services on deployment. Establishing such binding will be mediated by ***Kumori***, and will involve establishing the terms of the SLA with the providing services, which may need to be elastically re-scaled to cope with the terms of the provisioned SLA.

The current ***Kumori*** version allows establishing links between services without any SLA negotiation.

Service dependencies are established through the usage of the admission API of ***Kumori***. This API can be used to establish and take down dependencies between services. The parameters should simply reference the endpoints involved in taking the dependency.

Services must be written to support interruptions in the depended services. As mentioned earlier, failures will occur. This includes failures in the services depended upon. While such failure may mean a service cannot be provided while the fault exists, service writers should avoid at all costs permanent service shutdown in the event of a dependency failure.

# 6. Component programming

In the previous sections we have explained the elements that constitute our service model. Missing from it were aspects about how code is actually run within it.

As mentioned earlier, roles are associated with components, which, on deployment, are instantiated multiple times. Each role instance receives the configuration of its role, making them indistinguishable *ab-initio* by others in the same service. In some cases, it will be possible to also use id-like data to specify a particular instance to send a message to.

***Kumori*** base support is provided over `nodejs`. ***Kumori***'s `NATIVE` runtime provides a `nodejs` environment to run application code into, and the API is available from this environment. Despite this, it is entirely possible to run non-nodejs code within this environment, as besides nodejs itself, it includes a full linux stack, with 0MQ and access to loopback interfaces.

***Kumori*** does not automatically bring any dependencies of the component's code, even if it is purely based on nodejs. Components have the choice to pull any extra dependencies from internet-open sites, as internet connections to public IPs are available by default. However it is not advisable to proceed this way because it makes startup time unpredictable.

The full definition of the software stack available on each *runtime* is provided in the appendices.

# 6.1. Native component programming: the API

## 6.1.1. Initialization

The NATIVE runtime executes ***Kumori*** *Service Component* instances within Docker containers. The particular linux flavor of the container is described in the appendices, on top of which we have added ***Kumori***'s runtime software and facilities, including `nodejs` and `0MQ`.

A `blob bundle` providing the code for a component under the NATIVE runtime, must be a nodejs module exporting a class extending the `Component` class, defined within the `component` module available in the NATIVE environment. Class `Component` has the following signature:

```
class Component
    constructor: (      (1)
        @runtime,       (2)
        @role,          (3)
        @iid,           (4)
        @incnum,        (5)
        @localData,     (6)
        @resources,     (7)
        @parameters,    (8)
        @dependencies,  (9)
        @offerings      (10)
    )     ->

    run: ->                                 (11)
        #
    shutdown: ->                            (12)
        #
    reconfig: (resources, parameters) ->    (13)
        # changed configuration
        true # if we can take the reconfig
module.exports = Component
```

(1) The constructor returned is used to create an instance of the component object representing the component and pass deployment-derived information and an object representing the platform.

(2) The `runtime` object, through which the component can interact with the platform. We will discuss it in depth later on.

(3) The `role` to which this instance belongs. This is a string with the name.

(4) The `Instance Id` ***Kumori*** assigns to this instance. This is a unique value during the lifetime of a service. It is a string.

(5) The *incarnation number* of the instance. It indicates how many times the instance has been started.

(6) The path where to store local data. All components get this path even though they do not declare a volatile volume: this is it, the default volatile volume for which no effort is made in case of failures. Note that component instance restarts may destroy the data in this location, thus it is unwise to rely on its persistence accross failures.

(7) The set of resource objects assigned to this instance. This is organized as a dictionary, whose keys are the names of the declared resources for which data can be queried.

(8) The set of configuration parameters, organized as a dictionary with their names as keys.

(9) The set of **requires** channels organized as a dictionary, where the keys are the requires channel names, and the values are *channel objects*, which we will analyze later on.

(10) The set of **provides** channels. A similar dictionary to the one for `requires` channels.

(11) Method invoked to start execution of the instance.

⑫ Method invoked to warn that the instance is going to be stopped. The instance should take whatever action it deems necessary to save state, and save work or avoid data loss or loss of service quality.

⑬ Change of configuration parameters/and or resources. Returns ***true*** if the reconfiguration can be taken without problem. Otherwise, the instance is restarted.

***Kumori*** will perform a `require './component'`, loading the module representing the component code. Then it will instantiate the class passing the arguments presented above. It will complete initialization by calling the `run` method on the object instance just created.

The above simple protocol forms the base to instantiate any kind of component on ***Kumori***.

The rest of the interaction is, except for programmed shutdowns, initiated by the component itself via the `runtime` object received during construction.

### 6.1.2. The Runtime object

When a component is instantiated, ***Kumori*** creates an instance of the `runtime` object and passes it to the component's code.

The `runtime` object offers a set of services to interact with ***Kumori***. The `runtime` class has the following signature:

```
class RuntimeAgent
    ping: ->                                    ①
    createChannel: (requestHandler) ->          ②
    getMemberShip: (channelId) -> ③
```

① The method ping must be used by the instance to periodically signal its good health to ***Kumori***. At the very least, ping must be used to indicate the event loop is not blocked. It is recommended that ping is called once every second.

② The `createChannel` method returns a dynamic ***Reply*** channel ready to be used. When the `requestHandler` parameter is present, it is expected to have a `handleRequest` method, which will be invoked when messages arrive through the channel. If the parameter is not present, the component will need to override the `handleRequest` method of the returned channel in order to serve requests.

③ Returns a promise resolved to a list of available destinations for the channel passed as parameter. This applies only to ***Duplex*** channels. Each item of the list is an object including destination instance id, destination endpoint and destination service.

### 6.1.3. The channel API

Channel objects offer a set of methods that *Component Instances* can use to communicate with other *Component Instances*. Different chanels offer different methods.

*Request/Reply* **channels**

Request/Reply channels are used to perform direct requests to other components in the service.

Typically a Request channel is declared as a dependency to a component, whereas a Reply channel is declared as an offering. Request/Reply channels can pass along references to dynamic Reply channels created using `createChannel` method. On arrival, dynamic Reply channels get transformed into Request channels.

A Reply channel is any class satisfying the following specification:

```
class Reply
    handleRequest: (message, channels) -> ①
```

① Method to execute when a request is received, where *message* is an array of message parts, and *channels* is an array of Dynamic channels objects.
Returns a promise resolved to a [message, channels] array.

A component uses a Reply channel by simply defining the handleRequest method on the channel object it receives, overriding the default implementation of the runtime.

If something happens, the channel emits an `error` event with the following signature:

```
reply_error_handler = (error) ->    ①
  ...
```

① An `Error` object extended with `code` and `originalMessage` attributes. The `code` attribute indicates what happened and the `originalMessage` indicates which message produced the error. Currently, Reply channels only generates `EDESTINATIONUNAVAILABLE` error events.

A Request channel is any class satisfying the following specification:

```
class Request
    sendRequest: (message, channels, config) -> ①
```

① Sends the request to the target of the channel, where *message* is an array of message parts, *channels* is an array of Dynamic channel objects, and *config* is a dictionary to customize the request management (currently with just a *timeout* key, with the maximum time to resolve the request).
Returns a promise resolved to a [message, channels] array

Note that the method returns a ***Promise***. Promises are used in this setting to facilitate pipelining of request between component instances. In the above API's, *message* is an array of message parts, and *channels* is an array of dynamic channels being passed around.

If a request message can not be forwarded to any replyer, the promise is rejected with an error containing one of the following error codes:

- `EDESTINATIONUNAVAILABLE`: a replier cannot be found.
- `TIMEOUT`: reply not received.
- `ESENDFAILED`: unexpected failure sending the request.

- `ESENDABORTED`: request aborted by the platform.

*Receive* **channels**

A *receive* channel is an event emitter, issuing the ***message*** event. The event carries with it two parameters: the first is a `message`, and the second optionally carries a dictionary of dynamic channels. Besides the events it emits, *receive* channels expose subscription methods, as shown in the portion of its class definition below.

```
class Receive
  subscribe: (topic) ->     (1)
  unsubscribe: (topic) -> (2)
```

(1) Adds a new subscription. The channel will only emit messages matching one of the topics included in the channel subscription list. If the topic list is empty, all messages will be emitted.

(2) Removes an existing subscription. The channel will stop emitting messages associated to this topic unless the topics list gets empty. In that case, all messages will be emitted.

The signature of a `message` event handler is as follows

```
receive_message_handler = (message, channels) ->    (1)
  ...
```

(1) `channels` is optional.

*Send* **channels**

A *send* channel implements a ***send*** method accepting a message as its first parameter, and, optionally, a dictionary of dynamic channels created by the sender, as well as a topic string.

```
class Send
  send: (message, channels, topic) -> (1)
```

(1) Sends a message, where `message` is and array of request parts, `channels` is an array of chanels and `topic` is the topic of the message.
Note that both `channels` and `topic` are optional parameters.

If destinations are not found for a message, the channel emits an `error` event. The event handler has the following signature:

```
send_error_handler = (error) ->    (1)
  ...
```

(1) An `Error` object extended with `code` and `originalMessage` attributes. The `code` attribute contains `EDESTINATIONUNAVAILABLE` code.

*Duplex* **channels**

*Duplex* channels allow both sending and receiving messages asynchronously, combining the functionality of *send* and *receive* channels into one, and augmenting it with the ability to distinguish the destination.

Thus, *duplex* channels issue *message* events, as do *receive* channels, with its two parameters (`message` and `channels`), augmented with another parameter, `sender`, containing a string identifying the instance sending the message. This is the signature of a handler:

```
duplex_message_handler = (message, sender, channels) ->    ①
  ...
```

① `channels` is optional.

They also implement a ***send*** method, accepting also a message parameter, and a second parameter indicating the instance to receive the message, among the set of allowed target instances.

If a message cannot be delivered to the destination instance, the channel emits an `error` event with the following signature:

```
duplex_error_handler = (error) ->    ①
  ...
```

① The `error` parameter is an `Error` object with extra `code` and `originalMessage` attributes. `code` contains `EDESTINATIONUNAVAILABLE` and `originalMessage` is a pointer to the undelivered message.

Duplex channels also support a ***getMembership*** method, returning the list of instance ID's that can be used as targets of ***send***'s. When connected via *complete* connectors, the list is built out of the set of destination channels attached to it.

```
# A duplex channel class can send a receive messages.
class Duplex
  send: (message, target, channels) ->    ①
  getMemberShip: () -> ②
```

① Sends `message` to the instance identified by `target`, optionally passing a dictionary of created dynamic channels, `channels`.

② Returns a promise resolved to a list of available destinations that can be reached through the channel via `send` messages. Each item of the list is an object including destination instance id, destination endpoint and destination service.

Finally, a duplex channel also emits the event ***changeMembership***, issued when the set of destination instances attached to the *complete* connector changes as a consequence of a change in the deployment.

```
duplex_changeMembership_handler = (members) ->    ①
  ...
```

① `members` contains the list of available destinations that can be reached through the channel via `send` messages.

### 6.1.4. Dynamic channels

Dynamic channels are explicitly requested by a component's code to the ***Kumori*** runtime dynamically.

currently, ***Kumori*** supports requesting dynamic channels of type *reply*, while *duplex* will be supported in future releases.

Dynamic channels are not declared  in the configuration specification of the component, but they can be passed through an ordinary channel to one or more receiving instances, thus, they can be considered part of the "protocol" of the declared channels over which they are carried.

When a *reply* channel is transmitted, the receiver gets a corresponding *request* channel. This establishes a dedicated connection between the instances participating in this exchange, where the receiver can initiate requests on the sender via that *request* channel it received.

Dynamic channels have the following lifecycle properties:

1. They do not survive its creator's restart. That is if the instance creating a dynamic channel is restarted, having its incarnation number incremented, all dynamic channels it created disappear. On restart, thus, an instance must resend its dynamic channels.
2. Also, when an instance is restarted, all dynamic channels it received in its previous incarnation are no longer available to it. Thus the protocols passing dynamic channels must take into account the possibility that the receiver be restarted losing that dynamic channel, and the ability to request whatever services it was scheduled to receive through it. This means that protocols transmitting dynamic channels must operate in a way allowing re-generation of those channels.
3. Passing multiple times the same dynamic channel to the same connected instance results in the receiver receiving exactly the same channel every time. No new *Request* (or *Duplex*) channels are created in the receiver.

## 6.2. HTTP support in *Kumori*

A large percentage of services interact with their users via the HTTP protocol. As seen in earlier sections, ***Kumori*** provides a builtin service, the ***HTTP Inbound***, that must be configured if the service needs to accept external HTTP requests.

***HTTP Inbound*** creates instances of the builtin component ***SEP***, whose mission is to balance load and proxy an IP endpoint declared at the service application level, expecting connections to transport the HTTP protocol. The ***SEP*** itself is externally load balanced via standard mechanisms available for this purpose on the internet.

***SEP*** applies a minimal transformation of the HTTP protocol to push it through its *request* channel,

through a provided channel of the service, to a component working as the web server. ***SEP*** is able to support websockets by using dynamic channels with the actual front end component.

In order to allow programmers to use familiar interfaces, ***Kumori*** provides a module, ***httpMessage*** handling the **message/http** protocol emitted by ***SEP***, providing a server with an interface like that provided by the builtin **httpServer** module in nodejs. ***httpMessage*** includes full support for websockets.

```
# component.coffee
#
Component   = require 'component'
http        = require 'http-message'   (1)

module.exports = class MyComponent extends Component
  ...
  run: () ->
    channel = @offerings[name]
    server  = http.createServer @handleRequest
    server.listen channel    (2)
    ...
```

(1) Requiring the module provides the component program with an httpServer-like function.

(2) Instead of binding the server to a port, we link it to one of our channels, the one receiving `message/http` communications.

In the example above, once we have created the `server`, we can use it as an `httpServer` within frameworks such as `express`:

```
# component.coffee
#
Component   = require 'component'
http        = require 'http-message'

module.exports = class MyComponent extends Component
  ...
  run: () ->
    app = express()
    app.use ...
    channel = @offerings[name]
    server = http.createServer app
    server.listen channel
    ...
```

## 6.3. Other supported languages for the API

API access for other languages is planned. Please contact us if you would like to know more.

# 7. Fitting legacy code

The *NATIVE* runtime in ***Kumori*** is based on `nodejs` + zmq (0MQ on nodejs). The instance initialization sequence depends on having the proper nodejs module on the top-level directory of the code bundle implementing a component.

However it does not imply that all components must be nodejs based. It only implies that the module ***Kumori***'s native runtime expects must exist, and that direct interactions with the rest of the API (channels, runtime) must be ultimately carried out by nodejs code loaded on initialization (i.e., the **component.coffee** module).

As mentioned earlier, the NATIVE runtime is more than just nodejs/zmq, it is a full fledged linux distribution. This gives us many possibilities to actually run software not based on `nodejs`. The following are some of the possibilities:

*Link to binary libraries*

This is trivially achieved by including the binaries of those libraries in the code bundle. If they are to be used from within a nodejs-based program, the bridge lib should have been preferably compiled previously (on a linux environment like the one we supply), and included within the code bundle. This is a very simple variation on having a purely nodejs-based program.

*Include arbitrary executables*

As we can include binary libraries to be used from within node, we can also include arbitrary binary executables within the code bundle. When doing so we only need to ensure that the software stack they require can be found within the environment. If any additional libraries are necessary, they should be included within the code bundle.

When taking this approach, from the `component` module we will launch a process executing the binary. In the simplest cases, the executable "communicates" only through the file system and standard i/o. In those cases, the standard nodejs api gives us all the tools we need to launch and pass data to and from the process launched.

*Long-lived non-nodejs servers*

In other cases, the program being launched is long-lived, probably containing most of the logic of the component, and needs to communicate with other roles in the service. Depending on the case, the launched process can simply coordinate actions with the **component** module using one of the standard communication mechanisms available on a Linux system: sockets (unix or ip), pipes... Additionally, 0MQ can be used when the legacy code either uses it or easily adapts to using it.

This case can be supported by ***Kumori*** directly when runtimes for other language environments are provided.

*Legacy servers (even based on nodejs)*

There will be cases when we just have legacy software, coded to interact through IP channels, whose code cannot be easily adapted to ***Kumori***'s API. In this case ***Kumori*** itself provides support in terms of a **proxy** module available in the NATIVE runtime. The component needs to be properly defined according to the elements of the service model presented earlier (channels, settings,...), and the code needs to be bundled properly. However, the **component** module only plays a simple **adaptation** role: it makes use of an

appropriate ***Kumori*** **proxy** module to handle communications duties, and launches the legacy server, properly configured. In the sections that follow we provide details on how to proceed on these cases.

## 7.1. Legacy web server support

If we happen to have a web server that we cannot adapt to our environment, using directly the **httpMessage** module to interact directly through a SEP, we can still use that server untouched by using a mixed approach.

1. Use the **component** module as an adapter to launch the web server configured from the settings of the component, and to wait for connections on `localhost:80`
2. Let the **component** use the **httpMessage** module as explained in HTTP support in ***Kumori***, obtaining a `server` object connected to a channel.
3. Use the **httpProxy** module available in ***Kumori*** repositories to establish a very light proxy to `localhost:80`

Following with our earlier example:

```
# my-component.coffee
#
Component   = require 'component'
http        = require 'http-message'
httpProxy   = require 'http-message-proxy'      (1)
child       = require 'child-process'           (2)

module.exports = class MyComponent extends Component
  ...
  run: () ->
    [command,parameters] = @computeServerParameters() (3)
    child.spawn command, parameters                   (4)
    channel = @offerings[name]
    server  = http.createServer()
    server.listen channel
    httpProxy server, 'localhost', 80                 (5)
    httpProxy.on 'error', (err) => ...                (6)
    ...
```

(1) We need the `httpProxy` module provided by ***Kumori***.

(2) In addition, we also use the `child_process` module, to launch the actual server process.

(3) Assuming we have a `computeServerParameters` method, execute it to find the server executable name and the parameters it should be started with. We assume it has access to the configuration of the component.

(4) Launch the actual server. We should make this more sophisticated to enable sensical ping production.

(5) Start the httpProxy. From now on, HTTP goes directly to the legacy server.

⑥ httpProxy is an event emitter; it only emits *error* events.

## 7.2. Adapting Legacy servers: the TCP proxy facility

In the previous section we showed the special case of adapting an existing web server. In general, we may have situations in which a component functionality is actually provided by an existing server program, expecting access to IP networks, and such that it is either costly, impractical or outright impossible to carry out any sort of change in their code.

As was the case of a web server, the component should be properly specified, with the channels that make sense to how it is to be connected, and the configuration needed for semantically consistent set up of the legacy server.

Besides this configuration, however, we will need a way to convert the IP-based configuration directly supported by the legacy server to interact with other roles within the deployed service.

Just as in the case of the web server, the approach will require the component module to act as an adapter, launching the legacy server to interact with the `localhost` network interface, through adequate network ports.

Unlike the case of the legacy web server, we now can have an arbitrary protocol being spoken by the legacy server, thus we need a "neutral" solution. For this case ***Kumori*** provides a **proxy-tcp** module that can be used by the adapter/component to tunnel IP protocols through ***Kumori*** channels transparently to the legacy server code.

Unlike the approach in Legacy web server support, the channel protocol knows nothing about the higher level protocol supported by the server, limiting itself to ship the bytes being pushed back and forth by legacy pieces of software.

The following shows an example of how to generally set up this kind of legacy support.

```
# my-component.coffee
#
Component   = require 'component'
ProxyTcp    = require('proxy-tcp').ProxyTcp          (1)
child       = require 'child-process'

module.exports = class MyComponent extends Component
  ...
  run: () ->
    [server, parameters, channels] =
@computeServerParametersAndChannels() (2)

    @proxy = new ProxyTcp @iid, @role, channels      (3)

    @proxy.on 'ready', (@bindIp) =>                  (4)
      @startLegacyServer server, bindIp, parameters

    @proxy.on 'error', (err) =>                      (5)
      @processProxyError err

    @proxy.on 'change', (data) =>                    (6)
      @reconfigLegacyServer server, bindIp, parameters, data

    @proxy.on 'close', () =>                         (7)
      @stopLegacyServer server, parameters

  shutdown: ->
    @proxy.shutdown()                                (8)
  ...
```

(1) We now require the generic **proxy-tcp** module.

(2) Assuming method **computeServerParametersAndChannels** returns as in Legacy web server support the server program and parameters to pass to it. But, in addition, it returns an object relating the component's channels to the legacy server ports/connections/bindings.

(3) **ProxyTcp** object initialization requires the role and ID of the instance, and the list of channels (with additional information) to be proxied.

(4) **ProxyTcp** object issues an event when it is ready to process requests, providing the local IP address to be assigned to the legacy server. For **proxy-tcp** to function properly, legacy server must be bound to that IP.

(5) **ProxyTcp** object issues an event when an error occurs.

(6) **ProxyTcp** object issues an event, during its life cycle, when any change occurs that should result in a reconfiguration of the legacy server.

(7) **ProxyTcp** object issues an event when it's closed.

(8) **ProxyTcp** object finalization.

While method **computeServerParameters** is straightforward to write, needing only to know how

to start the legacy server, writing method **computeServerParametersAndChannels** will require knowing how to configure the **ProxyTcp** object too.

### 7.2.1. Configuring the ProxyTcp object

**ProxyTcp** object initialization requires an object relating the component's channels to the legacy server ports/connections/bindings.

This object is a dictionary whose key is the channel name, and values contains:

- A reference to the channel object
- TCP port to be proxied
- In case of duplex channels, its operating mode (bind/connect)

Example configuration for a **ProxyTcp** that proxies four channels:

```
{
  'myDuplex1': {
    channel: myDuplex1,
    port: 9100,
    mode: 'bind'
  },
  'myDuplex2': {
    channel: myDuplex2,
    port: 9100,
    mode: 'connect'
  },
  'myRequest3': {
    channel: myRequest3,
    port: 9200
  },
  'myReply4': {
    channel: myReply4,
    port: 9200
  }
}
```

### 7.2.2. Ready event

When **Proxy** object is ready to process requests, a *ready* event is emitted,

Data associated with this event is the local IP address to be used by the legacy server.

Typically, this IP address is used when starting the legacy server with duplex/bind or a reply channels. Typically, this information is not used with duplex/connect or a request channels.

```
@proxy.on 'ready', (bindIp) =>
  @_startLegacyServer bindIp, ...
```

### 7.2.3. Error event

**ProxyTcp** object can issue error events, basically during the creation of internal connections initializing the proxy.

```
@proxy.on 'error', (err) =>
  ...
```

### 7.2.4. Change event

During its life cycle, **ProxyTcp** object emits change events when any change occurs, which may result in a reconfiguration of the legacy server.

Data associated with this event will vary depending on the channel that caused it.

```
@proxy.on 'change', (data) ->
  @reconfigLegacyServer server, bindIp, parameters, data
```

*Request*

When a request channel is proxied, a TCP port is opened in a local IP address. A change event is issued when **ProxyTcp** is ready and listening on this port. An event is issued too, when the port is closed (this happens when the instance is shutting down, so usually an action on the legacy server is not required). Event data contains parameters that legacy server could need to be reconfigured:

- Listening (true/false)
- Channel name
- IP
- Port

For example:

```
{
  channel: 'myRequest3',
  listening: true,
  ip: ip:'127.0.0.7',
  port: 9300
}
```

*Reply*

Never issues change events.

*Duplex*

When the set of instances attached to the *complete* connector (duplex channels) changes, **ProxyTcp** issues a change event. Event data is a list of current members with the information

that the legacy server could need to be reconfigured:

- Channel name
- Instance ID
- IP
- Port

For example:

```
{
  channel: 'myDuplex1',
  members: [
    {iid:'A_10', ip:'127.0.0.7', port:9100},
    {iid:'A_11', ip:'127.0.0.8', port:9100},
    {iid:'A_12', ip:'127.0.0.9', port:9100}
  ]
]
```

### 7.2.5. Close event

After **ProxyTcp.shutdown()** method is invoked, a *close* event is emitted when operation is finished.

```
@proxy.on 'close', () =>
  ...
```

# 8. General service programming tips

Writing software for a cloud service with elasticity in mind is a bit different from other software-writing activities.

A service typically accumulates state over time that is relevant to how it behaves when requests arrive. That is one reason why failures in any part of the service that wipe out state may have critical consequences to the service functionality.

But wiping the state is not the worst that may happen to it. Some failures may leave the state of a store in such a shape that it cannot be accessed without fixing it. File systems, for instance, are prone to this kind of trouble: if the file system has been changed, but the metadata does not properly reflect those changes, the state of the file system metadata is inconsistent, and no normal use can be resumed. A similar situation can happen if other stores are involved.

Failures are not either the only source of trouble with state in services. Services must ensure a proper level of performance is achieved while in operation, and this must often be achieved by replicating components, or, using ***Kumori***'s verbiage, by creating several instances of components.

In many cases, the set of instances must act transparently as one. That is, its clients do not know anything about the replication, nor care about it: they just want to receive the service... However,

to achieve this degree of transparency it is necessary to ensure that whatever state changes the client may infer from the server are properly propagated among the set of replicas, or clients do not change of replica during a *stateful* session (stickyness).

Service application programmers must be aware of this issues and take proper steps to ensure service state handling fits the purported intent of the service.

# 8.1. A state sharing scenario: handling sessions

As mentioned above, one problem with replication is sharing state among the replicas. One of the best known examples is the case of session state in web applications when several replicas of the server are created.

In this case a simple solution, requiring minimal effort is to have a component holding the shared state, and connect the web server component with this session state component. This is what we showed in our example service presented in an earlier section.

# 8.2. State and persistence

Programmers used to write desktop, or even typical server programs usually think of the underlying file system as a durable store. This illusion is actually maintained through replication: a backup system helps ensure a degree of durability deemed sufficient in many cases. When circumstances demand stronger guarantees, the replication is actually carried out at the file system layer, ensuring that write requests, when carried out, are properly replicated.

When programming for a cloud environment these expectations must be carefully checked, to avoid making disastrous mistakes. A typical virtualized environment from one of the available IaaS provides compute VMs attached to some sort of volume in which a file system is available. While the VM exists, everything may behave as usual, and a program written for a classical server environment keeps on behaving properly. However, taking down the VM without any other action, may ensure that all the state of that VM is lost. One way in which a VM can be taken down is due to failures in the hardware itself, rendering a node in a data center useless, and the data stored in its disks unreachable.

Failures are to be expected in a data center environment, and a service programmer MUST code for failures. This means that if a service wants to guarantee some level of availability, it must code understanding that failures can wipe out data stored in a local volume.

## 8.2.1. Instantiation and component replicas

In ***Kumori*** component instances are created anew on initial service launch, and on reconfigurations, when more instances are needed to sustain the load of the service without sacrificing performance.

Earlier on we described the instantiation process. From it, each component instance has access to a default low performance volatile volume to store their working data. What volatile means in ***Kumori*** is that the data stored in them will be lost if any kind of failure (be it accidental or induced by operator actions) takes down the instance: the system is not going to take exceptional measures to protect its contents.

To avoid this, in ***Kumori*** we can take one of the following approaches.

### 8.2.2. State durability: volatile volumes

Component developers may request volumes with a durability higher than the one provided by the default volume made available to every instance.

A Component's specification can request a more durable volume from ***Kumori***. Besides the pure volatile volume we explained above, a component may request a volatile volume with a *best effort* level of guarantee. What this means is that in the case an instance fails, when ***Kumori*** goes to recover it, it tries to do so on the same node it was executing before the failure, and with the same volatile volume. This way, the previous state can be accessed by the new instance incarnation.

### 8.2.3. State durability: persistent volumes

Finally, a component can request a *persistent* volume resource. Such volumes are provided by a ***Kumori*** built-in service, operating as a distributed partition service with triple redundancy, exposing those volumes with a well established network-based volume access protocol (transparent beneath ***Kumori***). Changes to the volume are carried out before acknowledged, ensuring their durability without any extra work on the part of the component coder.

Typical implementations of *persistent* volumes do not allow them being mounted on more than one replica, thus, when a component gets this type of volume, it will get only one replica. In case of replica failure, the volume is made available to the recovered replica.

This is the most expensive kind of volume, and the one with a potentially larger hit on performance, as ensuring durability with the needed underlying redundancy at the block level has its price.

### 8.2.4. State durability: external services

We can rely on an external store service for durability. The approaches will vary depending on the type of external store.

Blob stores can be used in a very simple way using an approach like this: before an instance starts doing useful work, it accesses the blob store, and brings a blob containing its initial state.

When a shutdown is programmed, the changes in the state are shipped to the external blob store service.

In the event of an non-programmed failure, all changes would be lost, as nobody will care about shipping the altered state to any external store.

To avoid this total loss, partial changes can be shipped periodically to this external service.

In the case the external service is some sort of database, the interaction with it will typically based on some sort of transaction notion. Thus, state changes needing durability will not be commited before the database actually commits them.

## 8.3. Failures

Besides interfering with the state of a service, failures actually can take down the service itself.

Replication comes to the rescue again: ensuring that more than one instance of critical components exists can help guarantee the whole service keeps on working despite failures, without any detectable loss of service continuity.

However, all the replication in the world can still be useless if no attention is being payed to the correlation of failures. That is, many different instance failures may be caused by the same event, thus being correlated by it. If the event is sufficiently likely, having all replicas in those instances is useless from the point of view of guaranteeing service continuity.

As we do not expect service writers to understand the many different failure modes a PaaS may suffer, ***Kumori*** provides a high level approach to this. It allows each component to specify the degree of *resilience* it should get, and this is expressed as a basic resource of a component, with an integer value indicating how many failures (of a sufficiently likely kind) should be needed to fully wipe out all instances of the component. The default is 1.

Currently ***Kumori***'s implementation considers only one kind of abstract failure (the "sufficiently likely kind"). In the future it may be useful to actually specify more levels of failure for this resource parameter.

Note that values of the *resilience* resource higher than 1 will always require several instances of the component to be up concurrently.

# 9. Advanced component development

## 9.1. Defining runtimes

***Kumori*** provides a set of predefined runtimes that can be used to run application code directly on them. Additionally, it is also possible to create new custom runtimes by extending them.

Typically, a runtime is the combination of an Operating System that meets ***Kumori***'s requirements and a software stack available to the application code. For example, ***Kumori***'s NATIVE runtime referred to earlier on is currently based on Ubuntu OS, with a basic software stack installed on top of it, mainly "`nodejs`", 0MQ and some ***Kumori*** platform libraries.

If application code needs additional software libraries to run, and it is not feasible or it is very inconvenient to bundle them with the code of the component, a new custom runtime can be defined by taking an existing runtime as a base, and extending it by installing any required software. The new runtime will be registered in ***Kumori***, and the application configured to be run with it.

This course of action is also advised when many different components need a set of libraries, binaries or nodejs modules beyond what is provided in the native runtime.

Runtimes are maintained by ***Kumori*** as Docker image blobs. Thus, the general procedure to define a runtime will be to extend an existing one, following these steps:

1. Choose a base runtime from ***Kumori***'s runtimes repository.
2. Download the runtime bundle for the chosen runtime. This bundle will include a docker image that can be extended.
3. Import the base runtime image to a suitable Docker repository (usually on the developer's machine)
4. Customize the image via one of these methods:
    a. Dockerfile: building a new Docker image derived from the base, or
    b. Manually: creating a Docker container from the imported image, installing any desired software on it and committing the container to a new Docker image
5. Create a manifest for the new runtime
6. Generate a runtime bundle with the help of ***Kumori***'s runtime generation tool
7. Register the new runtime bundle on ***Kumori*** platform

A runtime generation tool is available as a part of ***Kumori***'s toolbox, to facilitate the handling of Docker images and exporting only the necessary layers to the runtime bundle to be registered.

Note that derived runtimes, even from the NATIVE runtime, can alter significantly the initialization sequence and the way ***Kumori*** services and API are made available to components running under the new runtime. It will be necessary to document and inform about those aspects in case alterations are introduced.

## 9.2. Examples

The manifest of a custom runtime that derived from the NATIVE runtime, with the addition of library *XYZ* to the software stack, could look as follows:

```
{
  "spec": "http://eslap.cloud/manifest/runtime/1_0_0",
  "name": "eslap://some.domain/runtime/nodejs-libxyz/1_0_0",
  "derived" : {
    "from" : "eslap://eslap.cloud/runtime/native/1_0_0"
  },
  "sourcedir": "/eslap/component",
  "entrypoint": "/eslap/runtime-agent/scripts/start-runtime-agent.sh",
  "metadata": {
    "description": "My custom runtime",
    "os_name": "Ubuntu",
    "os_version": "14.04",
    "os_release": "Trusty Tahr",
    "software": {
      "libxyz": {
        "version": "1.2.3"
      }
    }
  }
}
```

Using ***Kumori*** toolbox, it is straightforward to retrieve an existing runtime for extending it. For the runtime defined in the above manifest, the Docker image can be installed locally by running:

```
/home/user/kumori-sdk/tools/runtime-tool.sh install -n
eslap://eslap.cloud/runtime/native/1_0_0
```

Once the command finishes, a Docker image called `eslap.cloud/runtime/native:1_0_0` should be available in the local Docker, which can then be extended.

### 9.2.1. Image extension via Dockerfile

Assuming that the only additional software library needed by the component is `libxyz`, the base runtime can be extended with a simple Dockerfile:

```
FROM eslap.cloud/runtime/native:1_0_0

MAINTAINER MyCompany Ltd. <maintainer@my.company>

RUN apt-get update && apt-get install libxyz
```

A new Docker image can the be built from the Dockerfile, by running this command from the Dockerfile directory:

```
docker build -t some.domain/runtime/nodejs-libxyz:1_0_0 .
```

All standard Docker image building options and commands can be used in the extension process.

### 9.2.2. Image extension via container commit

Alternatively, a runtime image can be extended by running a container from the base image, installing the software from the inside and then committing the container state to a new Docker image.

This extension mechanism requires the base image to have a linux stack installed, to be able to run commands on it. For example, BARE runtime can't be extended via container commit.

```
$ docker run --entrypoint /bin/bash -it
eslap.cloud/runtime/native:1_0_0

root@332cf02b5e10:/#                                # INSIDE THE NEW
CONTAINER
root@332cf02b5e10:/# apt-get install libxyz
  [...]
root@332cf02b5e10:/# exit                           # EXITING THE NEW
CONTAINER

$ docker commit -m 'Kumori NATIVE runtime customization' -a
'maintainer' 332cf02b5e10 some.domain/runtime/nodejs-custom-
libxyz:1_0_0
```

## 9.3. Runtime bundle generation

Once the image for the new custom runtime is ready locally, it must be exported, processed and packed in a bundle, along with its manifest, so it's ready to register on ***Kumori***. This process is carried out with the help of the `runtime-tool` of the SDK:

```
/home/user/kumori-sdk/tools/runtime-tool.sh bundle -i
some.domain/runtime/nodejs-libxyz:1_0_0 -m Manifest.json
```

Local image tags will be ignored and only used to locate the image, since the generated bundle will contain a properly tagged Docker image matching the runtime name specified in the runtime manifest, as expected by ***Kumori***.

This will result in a zip bundle file as expected by ***Kumori***'s admission service.

# Appendix A: Balancing incoming IP requests

***Kumori*** can handle requests to the same service coming to different IP addresses. This ability can be leveraged to load balance arriving requests through the usual method of registering multiple IPs for the same domain, and leaving the decision of which particular IP to contact to the client of

the DNS service.

A second level balancing is performed the SEP component. When a SEP gets requests it will balance them among all the service instances connected to it, although stickiness is maintained.

# Appendix B: Resource units definition and implications

*__cpu*

Each CPU unit corresponds to the raw compute power of an Intel Avoton C2750 CPU core or equivalent.
This compute power is usable across all CPU cores available to the instance so that the usage is not limited to a restricted set of them, although all the compute power could be available in a single core depending on `threadability` value. This approach allows a better resource exploitation by multithreaded applications.
For example, a hypothetic instance that happens to be deployed in a node with 8 CPU cores, each of them twice as powerful as Intel Avoton ones, with a single CPU unit assigned to it, will be able to use at the same time:

- 50% of 1 core
- 25% of 2 cores
- 10% of a core, 25% of a second one, and 15% of a third one
- ...or any other combination that sums 50% of a core.

As would be expected, if the requested amount of CPU units is not enough for a component, it is possible that it won't be capable of processing requests on a timely basis, so the application will run slow. Conversely, a very high value will probably result in unused resources.

*__memory*

Each memory unit guarantees 256MB of physical RAM plus a fraction of that value of swap. The swap fraction might change in the future, and is currently set to 1/2, that is 128MB.

Linux kernel will kill processes trying to exceed this limit. Therefore, the chosen value must be enough for the component processes to start and run in normal operation. Higher than needed values will result in unused resources.

As a recommendation, swap memory should not be accounted when deciding how many memory units have to be requested for a component. Instead, its use should be reserved for sudden increases of memory usage, so to avoid processes being killed.

*__ioperf*

Each IOperf unit corresponds to 25MB/s disk bandwidth rate and 50 IOPS.
**The way disk performance is configured is temporary, and will be improved in future releases of the platform.**

A single IOperf unit should be enough for most applications. Only in case of observing high `iowait` (time spent by the CPU waiting, i.e. doing nothing, for I/O operations to complete) values, it is advisable to request more IOperf units.

# Appendix C: Parameter types

## C.1. Boolean

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/boolean/1_0_0",
  "typespec" : {
    "type" : "boolean"
  }
}
```

## C.2. Integer

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/integer/1_0_0",
  "typespec" : {
    "type" : "integer"
  }
}
```

## C.3. JSON

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/json/1_0_0",
  "typespec" : {
    "type" : "object"
  }
}
```

## C.4. List

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/list/1_0_0",
  "typespec" : {
    "type" : "array"
  }
}
```

## C.5. Number

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/number/1_0_0",
  "typespec" : {
    "type" : "number"
  }
}
```

## C.6. String

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/string/1_0_0",
  "typespec" : {
    "type" : "string"
  }
}
```

## C.7. VHost

```
{
  "spec": "http://eslap.cloud/manifest/parameter/1_0_0",
  "name": "eslap://eslap.cloud/parameter/vhost/1_0_0",
  "typespec" : {
    "type" : "object",
    "properties" : {
      "domain" : {"type" : "string"},
      "port"   : {"type" : "integer"}
    }
  }
}
```

# Appendix D: Predefined runtimes

## D.1. Bare

The Bare runtime contains no base OS. It is provided for users to derive other runtimes from it.

```
{
  "spec": "http://eslap.cloud/manifest/runtime/1_0_0",
  "name": "eslap://eslap.cloud/runtime/bare/1_0_0",
  "agent": "eslap://eslap.cloud/runtime-agent/1_0_0",
  "codedir" : "/eslap/component",
  "entrypoint" : "/eslap/runtime-agent/scripts/start-runtime-agent.sh",
  "metadata" : {
    "description" : "ECloud Bare Runtime (no base OS)",
    "os_name" : "",
    "os_version" : "",
    "os_release" : "",
    "software" : {
      "ecloud-runtime-agent" : {
        "version" : "1.0.0"
      },
      "nodejs" : {
        "version" : "4.3.2"
      }
    }
  }
}
```

## D.2. Alpine

Alpine Linux 3.3 based runtime with preconfigured code and entrypoint. This is an alternative runtime used to execute native components.

```
{
  "spec": "http://eslap.cloud/manifest/runtime/1_0_0",
  "name": "eslap://eslap.cloud/runtime/alpine/1_0_1",
  "agent": "eslap://eslap.cloud/runtime-agent/1_0_0",
  "sourcedir": "/eslap/component",
  "entrypoint": "/eslap/runtime-agent/scripts/start-runtime-agent.sh",
  "metadata": {
    "description": "ECloud Alpine Linux 3.3",
    "os_name": "Alpine Linux",
    "os_version": "v3.3",
    "os_release": "stable",
    "software": {
      "ecloud-runtime-agent": {
        "version": "1.0.0"
      },
      "openssh-client": {
        "version": "7.2p2"
      },
      "nodejs": {
        "version": "4.3.0"
      },
      "zeromq": {
        "version": "4.1.3"
      },
      "libsodium": {
        "version": "1.0.7"
      }
    },
    "layerId":
"745201b861b7944a4276f42c58a4c663aa7b053d49ca2aefe0f85154693e0f29"
  }
}
```


## D.3. Native

Ubuntu 16.04 based runtime with preconfigured code and entrypoint. This is the 1.1.x runtime series used to execute native components.

```
{
  "spec": "http://eslap.cloud/manifest/runtime/1_0_0",
  "name": "eslap://eslap.cloud/runtime/native/1_1_1",
  "agent": "eslap://eslap.cloud/runtime-agent/1_0_0",
  "sourcedir": "/eslap/component",
  "entrypoint": "/eslap/runtime-agent/scripts/start-runtime-agent.sh",
  "metadata": {
    "description": "ECloud Ubuntu 16.04",
    "os_name": "Ubuntu",
    "os_version": "16.04",
    "os_release": "Xenial Xerus",
    "software": {
      "ecloud-runtime-agent": {
        "version": "1.0.0"
      },
      "openssh-client": {
        "version": "7.2p2"
      },
      "nodejs": {
        "version": "4.3.2"
      },
      "libzmq5": {
        "version": "4.1.4"
      },
      "libsodium": {
        "version": "1.0.8"
      }
    },
    "layerId":
"78896f55bb59121bf564490200820df4dd62e0b580a2af34b536f084da8eb3d2"
  }
}
```


# Appendix E: Manifest schemas

## E.1. Blob

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/blob/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/element.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/blob/1_0_0"]
      },
      "name": {
        "type": "string",
        "description" : "Not a URI. Uniquenes requirements are within
its containing bundle"
      }
    }
  }],
  "description" : "Blobs are not registered by themselves. They form
part of an enclosing bundle"
}
```

## E.2. Channel

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/channel/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/channel/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/channel.json"
      }
    }
  }]
}
```

## E.3. Component

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/component/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/component/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/component.json"
      },
      "runtime": {
        "$ref": "http://eslap.cloud/schemas/uris/runtime.json"
      },
      "comfiguration": {
        "$ref": "http://eslap.cloud/schemas/configuration.json"
      },
      "channels": {
        "$ref": "http://eslap.cloud/schemas/channels.json"
      },
      "profile": {
        "type": "object",
        "properties": {
          "threadability": {
            "type": "string",
            "pattern": "^((\\d+\\+?)|\\*)$"
          }
        }
      }
    },
    "required": ["runtime", "channels"]
  }]
}
```

## E.4. Connector

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/connector/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/connector/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/connector.json"
      }
    }
  }]
}
```

## E.5. Deployment

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/deployment/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/element.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/deployment/1_0_0"]
      },
      "name" : {
        "type": "string"
      },
      "servicename" : {"$ref" :
"http://eslap.cloud/schemas/uris/service.json"},
      "comfiguration": {
        "type" : "object",
        "properties" : {
          "resources" : {
            "type" : "object",
            "additionalProperties" : {"$ref" :
"http://eslap.cloud/schemas/uris/resource.json"}
          },
          "parameters" : {
            "type" : "object",
            "additionalProperties" : {"$ref" :
"eslap://eslap.cloud/parameter/json/1_0_0"}
          }
        }
      },
```

```
      "roles" : {
        "type" : "object",
        "patternProperties" : {
          "^\\w+$" : {
            "type" : "object",
            "properties" : {
              "resources" : {
                "type" : "object",
                "patternProperties" : {
                  "^__\\w+$" : {}
                }
              }
            },
            "required": ["resources"]
          }
        },
        "additionalProperties" : true
      }
    },
    "required": ["servicename", "roles"]
  }]
}
```

## E.6. Parameter

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/parameter/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/parameter/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/parameter.json"
      },
      "typespec" : {"$ref" : "http://json-schema.org/schema"}
    }
  }]
}
```

## E.7. Protocol

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/protocol/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/protocol/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/protocol.json"
      },
      "parent": {
        "$ref": "http://eslap.cloud/schemas/uris/protocol.json"
      }
    }
  }]
}
```

## E.8. Resource

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/resource/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/resource/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/resource.json"
      },
      "parameters" : {"$ref" :
"http://eslap.cloud/schemas/parameters.json"}
    }
  }]
}
```

## E.9. Runtime

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/runtime/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/runtime/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/runtime.json"
      },
      "parent": {
        "$ref": "http://eslap.cloud/schemas/uris/runtime.json",
        "description" : "Parent runtime on which this one relies"
      },
      "codedir"    : {"type" : "string"},
      "entrypoint" : {"type" : "string"},
      "parameters": {
        "$ref": "http://eslap.cloud/schemas/parameters.json"
      }
    }
  }]
}
```

## E.10. Service Application

```
{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://eslap.cloud/manifest/service/1_0_0",
  "allOf": [{
    "$ref": "http://eslap.cloud/schemas/namedelement.json"
  }, {
    "properties": {
      "spec": {
        "enum": ["http://eslap.cloud/manifest/service/1_0_0"]
      },
      "name": {
        "$ref": "http://eslap.cloud/schemas/uris/service.json"
      },
      "comfiguration": {
        "$ref": "http://eslap.cloud/schemas/configuration.json"
      },
      "channels": {
        "$ref": "http://eslap.cloud/schemas/channels.json"
      },
      "roles" : {
        "type" : "array",
```

```
        "items" : {
          "type" : "object",
          "properties" : {
            "name" : {"type" : "string"},
            "component" : {
              "anyOf" : [
                {"$ref" :
"http://eslap.cloud/schemas/uris/component.json"},
                {"$ref" :
"http://eslap.cloud/schemas/uris/service.json"}
              ]
            },
            "resources" : {
              "type" : "object",
              "additionalProperties" : {"type" : "string"},
              "description" : "Simple maps from service resources to
role resources"
            },
            "parameters" : {
              "type" : "object",
              "additionalProperties" : {"$ref" :
"eslap://eslap.cloud/parameter/json/1_0_0"},
              "description" : "Simple way to fix constant values for
role parameters"
            }
          },
          "required" : ["name","component"]
        }
      },
      "connectors" : {
        "type" : "array",
        "items" : {
          "type" : "object",
          "properties" : {
            "type" : {"$ref" :
"http://eslap.cloud/schemas/uris/connector.json"},
            "depended" : {"$ref" : "#/definitions/endpoint_set"},
            "provided" : {"$ref" : "#/definitions/endpoint_set"}
          },
          "required" : ["depended"]
        }
      }
    },
    "required": ["channels", "roles", "connectors"]
  }],

  "definitions" : {
    "endpoint" : {
      "type" : "object",
      "properties" : {
        "role" : {"type" : "string"},
        "endpoint" : {"type" : "string"}
```

```
      },
      "required" : ["endpoint"],
      "description" : "When no role is provided, the endpoint is the
service's"
    },
    "endpoint_set" : {
      "type" : "array",
      "items" : {"$ref" : "#/definitions/endpoint"}
    }
  }
}
```