mOSAIC API

Java Programming Guide

Author: Georgiana Macariu

eAustria Research Institute

Contents

1 mOSAIC Platform Overview

In this guide we present the main concepts of the mOSAIC API and describe how to develop applications using the Java version of the API. Also we show how to deploy an application in the Cloud. We also present two sample mOSAIC-compliant applications.

This guide is addressed to developers creating Cloud applications which will execute on the mOSAIC platform. They can use this programming guide to understand the concepts and functionality of the mOSAIC API and to learn how to use the API for writing their applications.

1.1 Introduction

mOSAIC is an open source platform aiming to simplify the programming, deployment and management of Cloud applications. For programming Cloud applications, the mOSAIC platform defines the mOSAIC API, a language- and platform-agnostic application programming interface for using multi-Cloud resources. mOSAIC will provide such APIs for developing applications portable across different Clouds using at least two programming languages. Currently, only the Java API is available, but in the near future a Python API will be delivered.

mOSAIC-compliant applications are expressed in terms of Cloud Building Blocks able to communicate with each other.

A *Cloud Building Block* is any identifiable entity inside the Cloud environment. Cloud Building Blocks can be Cloud resources (which are under Cloud provider control) or Cloud Components.

A *Cloud Component* is a building block, controlled by user, configurable, exhibiting a well defined behavior, implementing functionalities and exposing them to other application components, and whose instances run in a Cloud environment consuming Cloud resources. Simple examples of components are: a Java application, or a virtual machine, configured with its own operating system, its web server, its application server and a configured and customized e-commerce application on it.

Overall, mOSAIC promotes several basic and simple principles for developing Cloud applications:

- Applications are developed according to a component-based model.
- Cloud Components should be developed according to an event-driven programming model.
- Communication between Cloud Components takes place using Cloud resources (e.g. message queues) or non-Cloud resources (e.g. sockets).
- In order to ensure portability and scalability, a Cloud Component should communicate directly only with Cloud resources and only indirectly with another Cloud Component through the mediation of Cloud resources like message queues.

Applications developed outside the mOSAIC platform (legacy applications) can also benefit of the platform, although they will not benefit of all features provided by the platform.

1.2 Event-based Programming Model

Programs following the event-based model are organized around event processing. In this model, whenever an operation cannot be completed immediately because it has to wait for an event, it registers a callback function that will be invoked when the event occurs and then returns. The callback function will execute until it hits a blocking operation, at which point it registers a new callback and then returns.

The internal architecture of the mOSAIC API is built according to an asynchronous, event-based programming model and we recommend mOSAIC users to use the same model when writing their applications.

While it may harder to develop your applications using an event-based approach, its advantages overcome its difficulties:

- Events prevent potential bugs caused by CPU concurrency of threads.

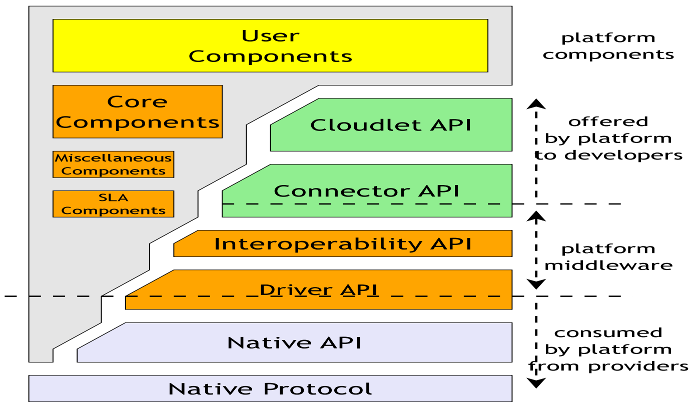

Figure 1.1: mOSAIC API layers

- Performance of event-based programs running under heavy load is more stable than that of threaded programs.
- Event-based programs use a single thread and thus, the programmer does not need to handle data races.

Currently, mOSAIC API is developed in Java, a cross-platform language and the compiled applications can run on any operating system compatible with the Java Virtual Machine (JVM).

In many languages event-based programming uses function or method pointers for registering the event callbacks. In Java there is no support for method pointers, but instead, callbacks are implemented with the help of interfaces which declare the methods that will be called when the event occurs.

1.3 Architecture Overview

The mOSAIC API supports a unified resource representation. Achieving the aim of a unified API for multiple programming languages requires a layered architecture that we present in Figure 1.1. Going up the stack, each layer in the architecture ensures a higher degree of independence from the back-end resource or programming language or programming paradigm used by the resource.

The low level layers - *Native API* and *Driver API* - provide a low level of uniformity: all resources of the same type are exported with the same interface. These layers need not be re-implemented in the different programming language versions of the mOSAIC API. These is due to the *Interoperability API* which provides programming language interoperability.

The upper layers, namely the *Connector* and *Cloudlet APIs* are language dependent and provide the highest degree of resource uniformity. The *Connector API* provides abstractions of Cloud resources, similar to Java JDBC or JDO. You can use this API to interact with resources using an event-based programming paradigm. The *Cloudlet API* is designed for creating Cloud components based on mOSAIC API and also uses the event-based paradigm. This API, defines Cloud components as Cloudlets which are to a Cloud environment what Java Servlets are to a J2EE environment. This programming guide focuses on how to write such Cloudlets.

The developer can write her Cloud components using only the Cloudlet API or she can mix it with Connector API as shown in Figure 1.2. However, the effort required for developing a pure Cloudlet-based application is smaller.

1.4 mOSAIC Cloudlets

In order to write a fully mOSAIC-compliant application, the developer needs to use the Cloudlet API. For such an application, the mOSAIC platform will ensure:

- scalability: the platform will support multiple instances of the same application component and scale well with respect to the increasing number of instances;

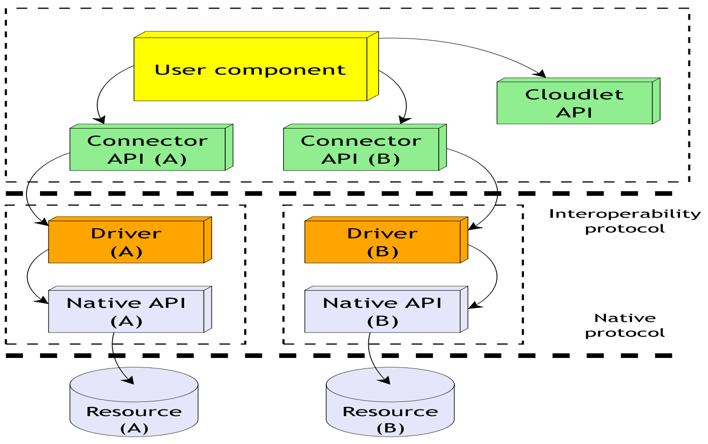

Figure 1.2: User components and mOSAIC API interaction

- fault tolerance: the platform will be able to handle in an as automated as possible way the fault of a component;
- autonomy: the components will be able to run in a Cloud environment, independently from other components.

For these mOSAIC applications, a Cloud Component is a Cloudlet and one or several Containers within which one or more Cloudlet instances execute. A Cloudlet Container hosts a single Cloudlet but there can be several instances of the Cloudlet running in the same Container. The number of instances is under the control of the Container and is managed in order to grant scalability.

Each Container has a unique identifier enabling its identification at runtime. If two Containers host the same Cloudlet, it is possible to distinguish the instances running in one container from the ones running in the other, but it is not possible to distinguish between the instances in the same container. The number of Containers hosting the same Cloudlet is programmable and can be tuned in order to achieve fault tolerance and elasticity. **The functional behavior of the Cloudlet must be independent from the number of Cloudlet instances or Containers.**

At runtime, the platform cannot distinguish between different instances of the same Cloudlet and a request directed to the Cloudlet will be sent to any of the existing instances. Because of this, **it is very important for the request processing to be completely independent of the current state of the Cloudlet instance**. It is recommended to write stateless Cloudlets (in terms of data).

2 mOSAIC API Installation

The easiest way to install and use the mOSAIC API is through the mOSAIC Controller. Using the Controller you can easily start any Cloud resource and Cloud Component. The Controller is available as a package of the mOSAIC Operating System or mOS. mOSAIC OS is a minimal OS based on SliTaz Linux (3.0) and created specially for cloud deployment. More information on mOS is available here: http://mosaic.ieat.ro/trac/wiki/TestingPlatform/mOS.

To install the mOSAIC Controller in mOS you simply run:

```
tazpkg get-install mosaic-node-boot
```

To run the cluster you must run the command:

```
/opt/mosaic-node-boot/bin/run
```

Once the cluster is up and running you can start, manage and stop Cloud Components using the cluster web console available at the address http://localhost:31808/. This is a javascript console which will also be used for deploying your mOSAIC applications as will be shown in Chapter 4. To see how you can use the console, visit http://wiki.volution.ro/Mosaic/Notes/Controller#Access_web_management_console. You can see the output of the cluster using the address http://localhost:31808/log/stream.

For compiling your applications you also need some other mOSAIC API libraries. The source code of all mOSAIC Java platform components is freely available at https://bitbucket.org/mosaic/mosaic-java-platform/.

3 Programming with mOSAIC Cloudlets

mOSAIC-compliant applications are composed of one or several Cloudlets. In this part, we will present the process of writing a Cloudlet. We will look at its structure and give some details about the internal affairs related to Cloudlet execution.

3.1 Cloudlet Implementation

Cloudlets are developed following the event-based programming model. A Cloudlet must react to two categories of events:

1. events related to its life cycle (initialization, destruction), and
2. events related to the Cloud resources used by it.

For each category the mOSAIC API defines special callback interfaces and the Cloudlet must implement these interfaces or extend a pre-defined class.

Life Cycle Callbacks

The code snippet in Listing 3.1 shows the callback interface **ICloudletCallback** in package **mosaic.cloudlet.core** for life cycle related events.

Listing 3.1: Cloudlet life cycle callback interface

```
package mosaic.cloudlet.core;

/**
 * Callback interface for cloudlet life cycle events. All user
 * cloudlets must implement this interface.
 *
 * @author Georgiana Macariu
 *
 * @param <S>
 *            The type of the object encoding the
 *    state of the cloudlet.
 */
public interface ICloudletCallback<S> extends ICallback {

  public void initialize(S state, CallbackArguments<S> arguments);

  public void initializeSucceeded(S state, CallbackArguments<S> arguments);

  public void initializeFailed(S state, CallbackArguments<S> arguments);

  public void destroy(S state, CallbackArguments<S> arguments);

  public void destroySucceeded(S state, CallbackArguments<S> arguments);

  public void destroyFailed(S state, CallbackArguments<S> arguments);

}
```

The first thing to note is that the interface takes as parameter the type of the Cloudlet state. Although, we have said above that Cloudlets are stateless, this is applicable only in terms of data. The API expects the object defining the state of the Cloudlet to contain references to the accessors (handlers) used for accessing Cloud resources. However, the developer can store in this state class any other information as long as the functional behavior of the Cloudlet is independent of this information.

Second observation refers to the parameters of the callback methods. All methods in this interface, and also in the resource related callback interfaces, take two arguments: the state of the Cloudlet and an object embedding the actual data required to processes the event corresponding to the callback method. The type of the second argument must be derived from the **CallbackArguments** class in the **mosaic.cloudlet.core** package. The **CallbackArguments** class defines the **getCloudlet()** method which returns a Cloudlet controller object which can be used later for creating and destroying Cloud resource connectors. Also, using this controller, the developer can access the Cloudlet configuration (see Section 3.2). The controller's interface is described in Listing 3.2.

Listing 3.2: Interface of the Cloudlet controller

```
package mosaic.cloudlet.core;

import mosaic.cloudlet.resources.IResourceAccessor;
import mosaic.cloudlet.resources.IResourceAccessorCallback;
import mosaic.core.configuration.IConfiguration;
import mosaic.core.ops.CompletionInvocationHandler;
import mosaic.core.ops.IOperationCompletionHandler;

/**
 * Interface for cloudlet control operations. Each cloudlet has
 * access to an object implementing this interface and uses it
 * to ask for resources or for destroying them when they are
 * not required anymore.
 *
 * @author Georgiana Macariu
 *
 * @param <S>
 *            the type of the state of the cloudlet
 */
public interface ICloudletController<S> extends ICloudlet {

  IConfiguration getConfiguration();

  /**
   * Initializes the resource accessor for a given resource.
   *
   * @param accessor
   *            the resource accessor
   * @param callbackHandler
   *            the cloudlet callback handler which must handle
   *  callbacks to operations invoked on the accessor
   * @param cloudletState
   *            the cloudlet state
   */
  void initializeResource(IResourceAccessor<S> accessor,
      IResourceAccessorCallback<S> callbackHandler, S cloudletState);

  /**
   * Destroys the resource accessor for a given resource.
   *
   * @param accessor
   *            the resource accessor
   * @param callbackHandler
   *            the cloudlet callback handler which must handle
   *  callbacks to  operations invoked on the accessor
   */
  void destroyResource(IResourceAccessor<S> accessor,
      IResourceAccessorCallback<S> callbackHandler);

}
```

The **ICloudletCallback** interface defines two methods, **initialize()** and **destroy()** which are not actual callback methods. The **initialize()** method is similar to a constructor in a Java class and is called automatically by the Cloudlet Container when the Cloudlet is deployed (see Chapter 4). You can place here any code that should execute immediately after creation of the Cloudlet (e.g. creating the accessors for the resources used by the Cloudlet). If the Cloudlet is successfully initialized, the **initializeSucceeded()** callback method is invoked. Only if the Cloudlet is initialized successfully you can continue and initialize the resources using the controller's **initializeResource()** method. Method **destroy()** is similar to a C++ destructor. This is can be called from your code, but it will also be called automatically by the Container during its shutdown process.

Resource Specific Callbacks

As mentioned above, upon Cloudlet initialization you can specify the resources required by it by creating accessors for each resource. An accessor is an object which handles all operation on the resource using a Connector specific to the resource type.

The current version of mOSAIC API implements Connectors and accessors for two types of Cloud resources:

1. message queues operating according to the AMQP protocol, and
2. key-value store systems.

For each Cloud resource used by the Cloudlet, you must create an accessor which must be stored in the Cloudlet state and you must also implement a specific callback class. Moreover, if the resource is used for storing or exchanging application specific data, you should also provide an encoder (serializer and deserializer) for your data. The reason for this is that connectors and drivers manage your data using byte streams and leave you the task of interpreting these byte streams. Your data encoder must implement the **DataEncoder** interface in the **mosaic.core.utils** package. We recommend to encode your data as JSON objects will are then serialized and deserialized as bytes. We also provide a default encoder **mosaic.core.utils.JsonDataEncoder** which encodes Java Bean objects using JSON which are than serialized.

Using key-value store resources

Listing 3.3 shows the code for creating a key-value store accessor. The accessor constructor requires as parameters the Cloudlet controller, the Cloudlet configuration and the data encoder. Note that the accessor object is stored in the Cloudlet state object. If you need to several key value stores or several buckets of the same key-value store you must create an accessor for each of them.

Listing 3.3: Creating the accessor for a key-value store

```
public static final class LifeCycleHandler extends
    DefaultCloudletCallback<CloudletState> {

  @Override
  public void initialize(CloudletState state,
      CallbackArguments<CloudletState> arguments) {
    ICloudletController<CloudletState> cloudlet = arguments
        .getCloudlet();
    IConfiguration configuration = cloudlet.getConfiguration();
    ...
    state.kvStore = new KeyValueAccessor<CloudletState>(
        configuration, cloudlet, new JsonDataEncoder<DataBean>(
            DataBean.class));
    ...
  }

  @Override
  public void initializeSucceeded(CloudletState state,
      CallbackArguments<CloudletState> arguments) {
    ICloudletController<CloudletState> cloudlet = arguments
        .getCloudlet();
    cloudlet.initializeResource(state.kvStore, new KeyValueCallback(),
        state);
  }
}

public static final class CloudletState {
  IKeyValueAccessor<CloudletState> kvStore;
}
```

The key-value accessor is initialized only after the Cloudlet is initialized (see the **initializeSucceeded()** method above). At initialization, you must specify an instance of your callback for the resource. Several key-value store accessors can share the same callback instance or you can use different instance for each accessor.

Your key-value store callback must implement the **IKeyValueAccessorCallback** interface or extend the **DefaultKeyValueAccessorCallback** class in **mosaic.cloudlet.resources.kvstore** package. Using a key-value store accessor you can store, retrieve and delete values in the store or you can list all keys in a bucket. For each of these operations the callback interfaces defines method which will be called upon their successful or unsuccessful termination (see Listing 3.4).

Listing 3.4: Key-value store callback interface

```
package mosaic.cloudlet.resources.kvstore;

import mosaic.cloudlet.resources.IResourceAccessorCallback;

/**
 * Base interface for key-value storage accessor callbacks. This
 * interface should be implemented directly or indirectly by
 * cloudlets wishing to use a key value storage.
 *
 * @author Georgiana Macariu
 *
 * @param <S>
 *            the type of the cloudlet state
 */
public interface IKeyValueAccessorCallback<S> extends
    IResourceAccessorCallback<S> {

  void setSucceeded(S state,KeyValueCallbackArguments<S> arguments);

  void setFailed(S state,KeyValueCallbackArguments<S> arguments);

  void getSucceeded(S state,KeyValueCallbackArguments<S> arguments);

  void getFailed(S state,KeyValueCallbackArguments<S> arguments);

  void deleteSucceeded(S state, KeyValueCallbackArguments<S> arguments);

  void deleteFailed(S state,KeyValueCallbackArguments<S> arguments);
}
```

Note that the callback methods take an argument of type **KeyValueCallbackArguments**. This extends **CallbackArguments** and contains the data returned by the operations in the accessor or the error messages if the operations were not completed normally.

Using AMQP message queues

A Cloudlet can consume and/or publish messages using an AMQP message queue. If the Cloudlet receives messages from a queue, it will need a **AmqpQueueConsumer** (in **mosaic.cloudlet.resources.amqp** package) accessor. Similarly, if the Cloudlet sends messages to a queue it will need a **AmqpQueuePublisher** (in **mosaic.cloudlet.resources.amqp** package) accessor. Listing 3.5 presents the code for creating the accessors.

Listing 3.5: Creating the accessors for message queues

```
public static final class LifeCycleHandler extends
      DefaultCloudletCallback<CloudletState> {

  @Override
  public void initialize(CloudletState state,
      CallbackArguments<CloudletState> arguments) {
    ICloudletController<CloudletState> cloudlet = arguments
        .getCloudlet();
    IConfiguration configuration = cloudlet.getConfiguration();
    state.consumer = new AmqpQueueConsumer<CloudletState, Data>(
        configuration, cloudlet, Data.class,
        new JsonDataEncoder<Data>(Data.class));
    state.publisher = new AmqpQueuePublisher<Cloudlet.CloudletState, Data>(
        configuration, cloudlet, Data.class,
        new JsonDataEncoder<Data>(
            Data.class));

  }

  @Override
  public void initializeSucceeded(CloudletState state,
      CallbackArguments<CloudletState> arguments) {
    ICloudletController<CloudletState> cloudlet = arguments
```

```
        .getCloudlet();
      cloudlet.initializeResource(state.consumer,
        new AmqpConsumerCallback(), state);
      cloudlet.initializeResource(state.publisher,
        new AmqpPublisherCallback(), state);

    }
  }

  public static final class CloudletState {
    AmqpQueueConsumer<CloudletState, Data> consumer;
    AmqpQueuePublisher<CloudletState, AuthenticationToken> publisher;
  }
```

Similar to the key-value store case, for resource initialization, you must specify an instance of your callback for the resource. Additionally, for message queues, the initialization step must be followed by a register step.

Both types of accessors define **register/unregister** methods. The **register()** method is called after the resources are initialized and is responsible for declaring the exchange and the queue and for registering the Cloudlet as a message consumer (for the consumer accessor only). The **unregister()** method unregisters the consumer from the queueing system.

Your message consumer callback must implement the **IAmqpQueueConsumerCallback** interface or extend the **DefaultAmqpConsumerCallback** class in **mosaic.cloudlet.resources.amqp** package. Using a consumer accessor you can receive messages. When a message for the Cloudlet instance arrives at the Container, the **consume()** method of the callback is called and the message is delivered. The actual message can be retrieved from the **AmqpQueueConsumeCallbackArguments** argument of the **consume()** method. The message can be acknowledged using the **acknowledge()** method. Listing 3.6 implements this process.

Listing 3.6: Message consuming

```
public static final class AmqpConsumerCallback extends
    DefaultAmqpConsumerCallback<CloudletState, Data> {
  ...

  public void consume(
      CloudletState state,
      AmqpQueueConsumeCallbackArguments<CloudletState, Data> arguments) {
    AmqpQueueConsumeMessage<Data> message = arguments
        .getMessage();
    Data data = message.getData();
    ...
    message.acknowledge();
  }

}
```

The message publishing callback must implement the IAmqpQueuePublisherCallback interface or extend the **DefaultAmqpPublisherCallback** class in **mosaic.cloudlet.resources.amqp** package. Using a publisher accessor, the Cloudlet can send messages using the **publish()** method of the accessor. When the operation returns, one of the **publishSucceeded()** or **publishFailed()** methods are called.

3.2 Cloudlet Descriptor

To be able to deploy your Cloudlet in a Container and to configure the Cloud resources used by the Cloudlet, the mOSAIC API requires a Cloudlet configuration file called **Cloudlet Descriptor**. In the current version of the API, the descriptor is a simple properties file whose name must be provided at Cloudlet deployment. The property file must be included in the archive containing the compiled code of the Cloudlet (see Chapter 4).

The properties required by the Cloudlet deployment process are:

- **cloudlet.main_class**: Represents the canonical name (package name + class name) of the Java class implementing the life cycle callbacks (the **ICloudletCallback** interface).

- **cloudlet.state_class**: Represents the canonical name of the Java class implementing the Cloudlet state class.
- **cloudlet.resource_file**: Represents the name of the configuration file containing the properties for configuring the Cloud resources (from now on called Cloudlet resource configuration file). This can be the same file as the one containing these three properties or a different one.

Listing 3.7 presents an example for these configurations. The configurations are valid for the Hello World example in Section 5.1.

Listing 3.7: Cloudlet deployment configuration example

```
# Cloudlet deployment configuration
cloudlet.main_class=mosaic.cloudlet.component.tests.HelloWorldCloudlet$LifeCycleHandler
cloudlet.state_class=mosaic.cloudlet.component.tests.
    HelloWorldCloudlet$HelloCloudletState
cloudlet.resource_file=hello-cloudlet.properties
```

For each resource used by the Cloudlet, the Cloudlet descriptor or the Cloudlet resource configuration file will contain a set of properties specific to the resource type.

For key-value stores the following properties must be set:

- **kvstore.bucket**: Represents the name of the bucket used by the Cloudlet.
- **kvstore.connector_name**: Represents the key-value store connector type to be used. mOSAIC supports simple key-value stores or key-value stores that implement the memcached protocol. The values accepted for this property are **KVSTORE** or **MEMCACHED**.

If the Cloudlet uses more than one bucket, then these three properties must be defined for each bucket. In order to distinct between them, for each bucket you must add a prefix to both properties. For example, Listing 3.8 shows how to configure a Cloudlet which uses two key-value buckets, one of them being in a key-value system which implements the memcached protocol.

Listing 3.8: Cloudlet key-value store configuration example

```
# key-value store configuration for bucket 'a'
bucket_a.kvstore.bucket=a
bucket_a.kvstore.connector_name=KVSTORE

# key-value store configuration for bucket 'b'
bucket_b.kvstore.bucket=b
bucket_b.kvstore.connector_name=MEMCACHED
```

For AMQP message queues the following properties must be set:

- **amqp.exchange**: Represents the name of AMQP exchange.
- **amqp.exchange_type**: Represents the type of AMQP exchange. Possible values are **topic**, **direct** or **fanout**. If this property is not set, the default value is **direct**.
- **amqp.routing_key**: Represents the routing key of consumed or sent messages.
- **amqp.queue**: Represents the name of message queue.
- **amqp.durable**: Indicates whether messages will survive server restart. Possible values are **true** or **false**. If the property is not set the default value is **false**.
- **amqp.auto_delete**: Indicates whether you are declaring an auto-delete queue. Possible values are **true** or **false**. If the property is not set the default value is **true**.
- **amqp.passive**: Indicates whether you are declaring a passive queue (i.e. check if it exists before creating it and raise an error if it does). Possible values are **true** or **false**. If the property is not set the default value is **false**.
- **amqp.exclusive**: Indicates whether you are declaring an exclusive (restricted to this connection) queue. Possible values are **true** or **false**. If the property is not set the default value is **true**.

Depending whether the Cloudlet consumes or publishes to the queue, you must add the “consumer” or the “publisher” prefix. Like in the case of key-value stores, if the cloudlet consumes or publishes to/from more queues, you must add a prefix to all properties in order to distinct between queues. Listing 3.9 shows how to configure a Cloudlet which consumes from two queues, “queue_a” and “queue_b”, and publishes to queue “queue_c”.

Listing 3.9: Cloudlet AMQP message queue configuration example

```
# AMQP consumer configuration from 'queue_a'
queue_a.queue.consumer.amqp.exchange=exchange-consume
queue_a.queue.consumer.amqp.exchange_type=topic
queue_a.queue.consumer.amqp.routing_key=abc
queue_a.queue.consumer.amqp.queue=queue_a
queue_a.queue.consumer.amqp.durable=true
queue_a.queue.consumer.amqp.auto_delete=false
queue_a.queue.consumer.amqp.passive=false
queue_a.queue.consumer.amqp.exclusive=false

# AMQP consumer configuration from 'queue_b'
queue_b.queue.consumer.amqp.exchange=exchange-consume
queue_b.queue.consumer.amqp.exchange_type=topic
queue_b.queue.consumer.amqp.routing_key=cde
queue_b.queue.consumer.amqp.queue=queue_b

# AMQP publisher configuration to 'queue_c
queue_c.queue.publisher.amqp.exchange=exhange-publish
queue_c.queue.publisher.amqp.exchange_type=direct
queue_c.queue.publisher.amqp.routing_key=fgh
queue_c.queue.publisher.amqp.queue=queue_c
queue_c.queue.publisher.amqp.durable=true
queue_c.queue.publisher.amqp.auto_delete=false
queue_c.queue.publisher.amqp.exclusive=false
```

As you have seen in Section 3.1, you can access the properties in the Cloudlet descriptor in the source code of your Cloudlet using the Cloudlet controller (see Listing 3.11).

Listing 3.10: Accessing Cloudlet descriptor in Cloudlet source code

```
public void initialize(CloudletState state,
        CallbackArguments<CloudletState> arguments) {
      ICloudletController<CloudletState> cloudlet = arguments
          .getCloudlet();
      IConfiguration configuration = cloudlet.getConfiguration();
      ...
}
```

Although the configuration object in the example above contains all configuration properties the **IConfiguration** object used when creating your resource accessor must not contain the prefixes that you added to distinguish between resources of the same type and must contain only the properties for that resource. In order to have such a configuration you need to splice the complete configuration as shown in Listing **??**).

Listing 3.11: Accessing Cloud resource configuration in Cloudlet source code

```
public void initialize(CloudletState state,
        CallbackArguments<CloudletState> arguments) {
      ICloudletController<CloudletState> cloudlet = arguments
          .getCloudlet();
      IConfiguration configuration = cloudlet.getConfiguration();

      // configuration for key-value store bucket_a
      IConfiguration kvConfigurationA = configuration
          .spliceConfiguration(ConfigurationIdentifier
              .resolveAbsolute("bucket_a"));
      state.kvStoreA = new KeyValueAccessor<CloudletState>(
          kvConfigurationA, cloudlet, new JsonDataEncoder<String>(
              String.class));

      // configuration for key-value store bucket_b
      IConfiguration kvConfigurationB = configuration
```

```
            .spliceConfiguration(ConfigurationIdentifier
                .resolveAbsolute("bucket_b"));
        ...

        // configuration for AMQP message queue queue_a
        IConfiguration queueConfigurationA = configuration
            .spliceConfiguration(ConfigurationIdentifier
                .resolveAbsolute("queue_a.queue"));
        state.consumerA = new AmqpQueueConsumer<CloudletState, AMessage>(
            queueConfigurationA, cloudlet, AMessage.class,
            new JsonDataEncoder<AMessage>(AMessage.class));

        // configuration for AMQP message queue queue_b
        IConfiguration queueConfigurationB = configuration
            .spliceConfiguration(ConfigurationIdentifier
                .resolveAbsolute("queue_b.queue"));
        ...

        // configuration for AMQP message queue queue_c
        IConfiguration queueConfigurationC = configuration
            .spliceConfiguration(ConfigurationIdentifier
                .resolveAbsolute("queue_c.queue"));
        state.publisher = new AmqpQueuePublisher<CloudletState, CMessage>(
            queueConfigurationC, cloudlet, CMessage.class,
            new JsonDataEncoder<CMessage>(CMessage.class));

  }
```

You can also see how the descriptor can be used in the Cloudlet source code in the Ping-Pong example in Section 5.2.

3.3 Cloudlet Execution

A Cloudlet runs in a Cloudlet Container which is managed by the mOSAIC Software platform. When a Cloudlet is deployed, a Container is created for it. In the Container, the code corresponding to a Cloudlet instance will always execute within that single thread. While the Cloudlet thread is used for processing a request received by it, all requests arriving in the meantime are queued by the container.

The Cloudlet Container completely hides to the developer the execution environment details, leaving the developer to focus on the application behavior. Moreover, since all user code executes in a single thread, the developer does not have to concern about data races and deadlocks, problems which would have been raised if multiple threads would have been used for executing user code.

For the same reason, possible performance problems caused by heavy Cloudlet load are solved by the Container by creating another Cloudlet instance and not by creating other threads to serve the extra requests. If the number of instances in the Container reaches the maximum allowed, the platform may decide to create another Container on another virtual machine.

4 Application Deployment

Once your application is ready, you can deploy it on the mOSAIC platform.

Before deploying your application you must also write an Application Descriptor, specifying the resource types and other components required by your application. This descriptor will be used by the platform to detect if it can provide those resource and if necessary for starting them. In the current version, the descriptor is a JSON-like file with the following structure:

Listing 4.1: Application Descriptor structure

```
{
    "<component-1>" : {
        "type" : "#<type-1>",
        "configuration" : <list of parameters>,
        "order" : <positive-integer>,
        "delay" : <positive-integer-measuring-milliseconds>
    },
    "<component-2>" : {
        "type" : "#<type-1>",
        "configuration" : <list of parameters>,
        "order" : <positive-integer>,
        "delay" : <positive-integer-measuring-milliseconds>
    }
}
```

The semantics of the elements in the descriptor are as follows:

- <**component-n**> is an identifier you associate with the resource or component you want to start.
- **type** identifies the type of resource or component. Currently you may use the following types:
  - **mosaic-components:rabbitmq**: identifies RabbitMQ (message queueing system) servers,
  - **mosaic-components:riak-kv**: identifies Riak key-value servers,
  - **mosaic-components:httpg**: identifies the HTTP gateway. This component receives messages from the HTTP channel and sends out messages on queues. It can be used in order to build up easily HTTP interfaces to mOSAIC applications.
  - **mosaic-components:java-driver**: identifies the Java drivers for message queues and key-value stores.
  - **mosaic-components:java-cloudlet-container**: identifies the Java Cloudlet container.
- **configuration**: is a JSON string containing any parameter required for starting the component.
- **count**: indicates the number of component instances which should be started.
- **order**: specifies the start-up order number of the component. If order=3 then the component will be started third.
- **delay**: specifies the time to wait after starting the component before starting the next component in order.

The application descriptor in Listing 4.2 will start a RabbitMQ server, a Riak server and two drivers, one for the message queue system and one for the key-value store. Finally, it will also start a Cloudlet Container with an instance of a sample cloudlet in it. The Cloudlet code will be downloaded from the URL supplied as the first configuration parameter, while the Cloudlet descriptor is supplied as the second parameter.

Listing 4.2: Application Descriptor example

```
{
    "rabbitmq" : {
        ""type" : "#mosaic-components:rabbitmq",
  "configuration" : null,
  "count" : 1,
  "order" : 1,
  "delay" : 2000
    },
     "riak" : {
        "type" : "#mosaic-components:riak-kv",
  "configuration" : null,
  "count" : 1,
  "order" : 2,
  "delay" : 2000
    },
    "rabbitmq-driver" : {
  "type" : "#mosaic-components:java-driver",
  "configuration" : "amqp",
  "count" : 1,
  "order" : 3,
  "delay" : 2000
    },
    "kv-driver" : {
  "type" : "#mosaic-components:java-driver",
  "configuration" : "kv",
  "count" : 1,
  "order" : 3,
  "delay" : 2000
    },
    "my-cloudlet" : {
  "type" : "#mosaic-components:java-cloudlet-container",
  "configuration" : ["http://localhost:27665/my_cloudlet-jar-with-dependencies.jar","my
      -cloudlet.prop"],
  "count" : 1,
  "order" : 4,
  "delay" : 1000
    }
}
```

While the syntax of the descriptor may change in future versions of the API, its semantics will be kept.

In order to deploy your cloudlets you must follow the steps:

1. Prepare a Java archive jar with your classes and all their dependencies. The Java archive must also include all Cloudlet descriptors. If you are using Maven for building your applications, you can look in the **examples/simple-cloudlets** module which is part of the **mosaic-java-platform** repository, to see how to use Maven for building your archive.

2. Upload the single Java jar such that it is accessible over HTTP somewhere (you could use for example webfsd to make it accessible).

3. Start the mOSAIC Controller on one of your Cloud virtual machines.

4. Submit the application descriptor to the cluster (below we assume the descriptor is in the file *descriptor.json*):

```
curl -X POST -H 'Content-Type: application/json' –data-binary @./descriptor.json
http://<machine-name>:31808/processes/create
```

5 Cloudlet Examples

5.1 Hello World Cloudlet

The first example that we want to expose is a simple "Hello World" Cloudlet. Its simple behavior demonstrates the basic Cloudlet life-cycle. This Cloudlet is described as follows: The "Hello World" Cloudlet starts, and just after being initialized it asks for it's destruction. But during its life-cycle it also logs out all the events it receives. Listing 5.1 presents the code of the Cloudlet.

Listing 5.1: The HelloWorld Cloudlet

```java
package mosaic.cloudlet.component.tests;

import mosaic.cloudlet.core.CallbackArguments;
import mosaic.cloudlet.core.DefaultCloudletCallback;
import mosaic.cloudlet.core.ICloudletController;
import mosaic.core.log.MosaicLogger;

public class HelloWorldCloudlet {
  public static final class LifeCycleHandler extends
      DefaultCloudletCallback<HelloCloudletState> {

    @Override
    public void initialize(HelloCloudletState state,
        CallbackArguments<HelloCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "HelloWorld cloudlet is initializing...");
    }

    @Override
    public void initializeSucceeded(HelloCloudletState state,
        CallbackArguments<HelloCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "HelloWorld cloudlet was initialized successfully.");
      System.out.println("Hello world!");
      ICloudletController<HelloCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.destroy();
    }

    @Override
    public void destroy(HelloCloudletState state,
        CallbackArguments<HelloCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "HelloWorld cloudlet is being destroyed.");
    }

    public void destroySucceeded(HelloCloudletState state,
        CallbackArguments<HelloCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "HelloWorld cloudlet was destroyed successfully.");
    }

  }

  public static final class HelloCloudletState {

  }
}
```

As the mOSAIC API is mainly asynchronous, we import the classes needed to handle the triggered events, which are modeled as callbacks: the Callback interface is used to obtain the context (which Cloudlet) and data (arguments and type) of the triggered event:

- The **CallbackArguments** is a Java class that gives access to the Cloudlet controller and allows one to express (through inheritance) callback specific data.
- The **DefaultCloudletCallback** is a Java class that implements the **ICloudletCallback** interface (the default implementation of the interface just logs events).
- The **ICloudletController** interface provides access to the Cloudlet configuration and is also used for initializing and destroying resource accessors.
- The MosaicLogger class is used for tracing events in the Cloudlet.

The HelloCloudletState class (lines 45-47 in Listing 5.1) is a simple data structure class, that may hold only references to the Cloudlet itself, resource accessors used throughout the Cloudlet, and maybe some other temporary cached values. In this example, the class is empty. From this point on, all code is run only on Cloudlet managed threads and only as reactions to Cloudlet callbacks.

In this example, the Cloudlet reacts only to life-cycle related events, and thus, we implement only the **LifeCycleHandler** extending the **DefaultCloudletCallback**. The implemented callbacks are:

- **initialize(HelloCloudletState state, CallbackArguments<HelloCloudletState> arguments)**: triggered just after the cloudlet is created and registered to a cloudlet container.
- **initializeSucceeded(HelloCloudletState state, CallbackArguments<HelloCloudletState> arguments)**: triggered if the Cloudlet initialization callback succeeded;
- **destroy(HelloCloudletState state, CallbackArguments<HelloCloudletState> arguments)**: triggered when the cloudlet destruction is requested (either from the container or the cloudlet itself); also during this phase any obtained connectors or other objects should be gracefully destroyed;
- **destroySucceeded(HelloCloudletState state, CallbackArguments<HelloCloudletState> arguments)**: triggered after the destruction of the cloudlet succeeded.

As stated earlier, in terms of code, the cloudlet reacts to triggered events by implementing the proper callback interface, one for each category of events. In our example, as we handle only basic cloudlet life-cycle events, we must implement the **ICloudletCallback** interface, which has the methods and their semantic described above, and in order to obtain the desired behavior (initialize then just destroy, but logging all received events), we just log each received event (through the **MosaicLogger** class) and in **initializeSucceeded** we use the Cloudlet controller and ask for Cloudlet destruction.

5.2 Ping-Pong Cloudlets

In this example we present an application containing two Cloudlets, **Ping** and **Pong**, each of them using one or more Cloud resources. When **Ping** starts it sends a message, using an AMQP queue, to **Pong**. The message contains a string key. When **Pong** receives the message, gets from a key-value store the value with the key received from **Ping** and sends it to **Ping**, through another AMQP message queue. After sending the message, the **Pong** Cloudlet destroys itself. When **Ping** receives the value, logs it and then it also destroys itself.

All messages exchanged between Cloudlets are coded as JSON objects. However, this coding is transparent to the developer of the Cloudlet since she can work with Java objects with a structure similar to that of Java Beans. Listing 5.2 lists the code of the Java class representing the message sent by **Ping** to **Pong**, Listing 5.2 contains the code of the Java class representing the message sent by **Pong** to **Ping**, while Listing 5.4 contains the data type of the values stored in the key-value store.

Listing 5.2: PingMessage class

```
package mosaic.cloudlet.component.tests;

public class PingMessage {

    private String key;

    public PingMessage(){

    }

    public PingMessage(String key) {
      this.key=key;
    }

    public String getKey() {
      return key;
    }

    public void setKey(String key) {
      this.key = key;
    }
}
```

Listing 5.3: PongMessage class

```
package mosaic.cloudlet.component.tests;

public class PongMessage {

  private String key;
  private PingPongData value;

  public PongMessage() {

  }

  public PongMessage(String key, PingPongData value) {
    this.key = key;
    this.value = value;
  }

  public String getKey() {
    return key;
  }

  public void setKey(String key) {
    this.key = key;
  }

  public PingPongData getValue() {
    return value;
  }

  public void setValue(PingPongData value) {
    this.value = value;
  }

}
```

Listing 5.4: PingPongData class

```
package mosaic.cloudlet.component.tests;

public class PingPongData {

  private String ping;
  private String pong;

  public PingPongData() {
  }

  public String getPing() {
    return ping;
  }

  public void setPing(String ping) {
    this.ping = ping;
  }

  public String getPong() {
    return pong;
  }

  public void setPong(String pong) {
    this.pong = pong;
  }

  public String toString() {
    return "(" + ping + ", " + pong + ")";
  }

}
```

The **Ping** Cloudlet uses only two AMQP message queues: consumes messages from the “pong-queue” and publishes messages to the “ping-queue”. The details required to connect to these queues are defined in the Cloudlet descriptor (see Listing 5.5).

Listing 5.5: Ping Cloudlet descriptor

```
cloudlet.main_class=mosaic.cloudlet.component.tests.PingCloudlet$LifeCycleHandler
cloudlet.state_class=mosaic.cloudlet.component.tests.PingCloudlet$PingCloudletState
cloudlet.resource_file=ping-cloudlet.properties

queue.consumer.amqp.exchange=pingpong-exchange
queue.consumer.amqp.routing_key=pong
queue.consumer.amqp.queue=pong-queue

queue.publisher.amqp.exchange=pingpong-exchange
queue.publisher.amqp.routing_key=ping
queue.publisher.amqp.queue=ping-queue
```

From the first two lines of the descriptor we find the name of the life cycle events callback class and the name of the state class. The code of these classes as well as the code of the resource callbacks is presented in Listing 5.6.

Listing 5.6: Ping Cloudlet code

```
package mosaic.cloudlet.component.tests;

import mosaic.cloudlet.core.CallbackArguments;
import mosaic.cloudlet.core.DefaultCloudletCallback;
import mosaic.cloudlet.core.ICloudletController;
import mosaic.cloudlet.resources.amqp.AmqpQueueConsumeCallbackArguments;
import mosaic.cloudlet.resources.amqp.AmqpQueueConsumeMessage;
import mosaic.cloudlet.resources.amqp.AmqpQueueConsumer;
import mosaic.cloudlet.resources.amqp.AmqpQueuePublishCallbackArguments;
import mosaic.cloudlet.resources.amqp.AmqpQueuePublisher;
import mosaic.cloudlet.resources.amqp.DefaultAmqpConsumerCallback;
import mosaic.cloudlet.resources.amqp.DefaultAmqpPublisherCallback;
import mosaic.core.configuration.ConfigurationIdentifier;
import mosaic.core.configuration.IConfiguration;
import mosaic.core.log.MosaicLogger;
import mosaic.core.utils.JsonDataEncoder;

public class PingCloudlet {

  public static final class LifeCycleHandler extends
      DefaultCloudletCallback<PingCloudletState> {

    @Override
    public void initialize(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger()
          .info("Ping Cloudlet is being initialized.");
      ICloudletController<PingCloudletState> cloudlet = arguments
          .getCloudlet();
      IConfiguration configuration = cloudlet.getConfiguration();
      IConfiguration queueConfiguration = configuration
          .spliceConfiguration(ConfigurationIdentifier
              .resolveAbsolute("queue"));
      state.consumer = new AmqpQueueConsumer<PingCloudlet.PingCloudletState,
          PongMessage>(
          queueConfiguration, cloudlet, PongMessage.class,
          new JsonDataEncoder<PongMessage>(PongMessage.class));
      state.publisher = new AmqpQueuePublisher<PingCloudlet.PingCloudletState,
          PingMessage>(
          queueConfiguration, cloudlet, PingMessage.class,
          new JsonDataEncoder<PingMessage>(PingMessage.class));

    }

    @Override
    public void initializeSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet initialized successfully.");
      ICloudletController<PingCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.initializeResource(state.consumer,
          new AmqpConsumerCallback(), state);
      cloudlet.initializeResource(state.publisher,
          new AmqpPublisherCallback(), state);
    }

    @Override
    public void destroy(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info("Ping Cloudlet is being destroyed.");

    }

    @Override
    public void destroySucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet was destroyed successfully.");
    }

  }

  public static final class AmqpConsumerCallback extends
      DefaultAmqpConsumerCallback<PingCloudletState, PongMessage> {

    @Override
    public void registerSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet consumer registered successfully.");
    }

    @Override
    public void unregisterSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet consumer unregistered successfully.");
      // if unregistered as consumer is successful then destroy resource
      ICloudletController<PingCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.destroyResource(state.consumer, this);
    }

    @Override
    public void initializeSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      // if resource initialized successfully then just register as a
      // consumer
      state.consumer.register();
    }

    @Override
    public void destroySucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet consumer was destroyed successfully.");
      state.consumer = null;
      if (state.publisher == null) {
        arguments.getCloudlet().destroy();
      }
    }

    @Override
    public void acknowledgeSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      state.consumer.unregister();

    }

    @Override
    public void consume(
        PingCloudletState state,
        AmqpQueueConsumeCallbackArguments<PingCloudletState, PongMessage> arguments) {
      AmqpQueueConsumeMessage<PongMessage> message = arguments
          .getMessage();
      PongMessage data = message.getData();
      String key = data.getKey();
      PingPongData value = data.getValue();
      MosaicLogger.getLogger().info(
          "Ping Cloudlet received key-value pair: (" + key + ", "
              + value + ")");

      message.acknowledge();
    }

  }

  public static final class AmqpPublisherCallback extends
      DefaultAmqpPublisherCallback<PingCloudletState, PingMessage> {

    @Override
    public void registerSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet publisher registered successfully.");
      PingMessage data = new PingMessage("pingpong");
      state.publisher.publish(data, null, "");
    }

    @Override
    public void unregisterSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet publisher unregistered successfully.");
      // if unregistered as publisher is successful then destroy resource
      ICloudletController<PingCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.destroyResource(state.publisher, this);
    }

    @Override
    public void initializeSucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      // if resource initialized successfully then just register as a
      // publisher
      state.publisher.register();
    }

    @Override
    public void destroySucceeded(PingCloudletState state,
        CallbackArguments<PingCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Ping Cloudlet publisher was destroyed successfully.");
      state.publisher = null;
      if (state.consumer == null) {
        arguments.getCloudlet().destroy();
      }
    }
```

```
178
179     @Override
180     public void publishSucceeded(
181         PingCloudletState state,
182         AmqpQueuePublishCallbackArguments<PingCloudletState, PingMessage> arguments) {
183       state.publisher.unregister();
184     }
185
186   }
187
188   public static final class PingCloudletState {
189     AmqpQueueConsumer<PingCloudletState, PongMessage> consumer;
190     AmqpQueuePublisher<PingCloudletState, PingMessage> publisher;
191   }
192 }
```

Because the Cloudlet uses two queues, its state (lines 188–191) contains only the consumer and publisher accessors corresponding to these queues. These accessors are created in the **initialize()** method of the **LifeCycleHandler**. When creating the consumer accessor (lines 34–36) we specify that messages sent to the queue will be encoded a JSON objects by passing a **JsonDataEncoder** object to the accessor. If Cloudlet initialization is successful, the **initializeSucceeded()** callback method will be called. Here we create a callback object for handling the events produced by the consumer and initialize the queue (lines 50-51). The same steps are taken for the publisher accessor.

The **AmqpConsumerCallback** class extends the **DefaultAmqpConsumerCallback** class which implements the **IAmqpConsumerCallback** callback interface. Note that when the **initializeSucceeded(** method of the callback is called we need to register the Cloudlet as consumer (line 98). When a message is delivered to the Cloudlet the **consume()** method of the callback is called. The **AmqpQueueConsumeCallbackArguments** parameter contains the actual data which is retrieved as a **PongMessage** in line 127. The Cloudlet logs the message and then acknowledges the message (line 132). After the acknowledge finishes with success the Cloudlet unregisters itself using the unregister() method of the consumer accessor (line 115) and later destroys the accessor (see method **unregisterSucceeded()** - lines 83–91).

The **AmqpPublisherCallback** class extends the **DefaultAmqpPublisherCallback** class which implements the **IAmqpPublisherCallback** callback interface. Similar, to the consumer case, the Cloudlet also must register as a publisher after the queue is initialized (line 165). After the register finishes, the Cloudlet sends the **PingMessage** to **Pong** (lines 145-146) and when the publish finishes, destroys the resource accessor (lines 180–184).

The **Pong** Cloudlet uses two AMQP message queues: consumes messages from the "ping-queue" and publishes messages to the "pong-queue", and a key-value store. The details required to connect to these resources are defined in the Cloudlet descriptor (see Listing 5.7).

Listing 5.7: Pong Cloudlet descriptor

```
cloudlet.main_class=mosaic.cloudlet.component.tests.PongCloudlet$LifeCycleHandler
cloudlet.state_class=mosaic.cloudlet.component.tests.PongCloudlet$PongCloudletState
cloudlet.resource_file=pong-cloudlet.properties

queue.consumer.amqp.exchange=pingpong-exchange
queue.consumer.amqp.routing_key=ping
queue.consumer.amqp.queue=ping-queue

queue.publisher.amqp.exchange=pingpong-exchange
queue.publisher.amqp.routing_key=pong
queue.publisher.amqp.queue=pong-queue

kvstore.kvstore.connector_name=KVSTORE
kvstore.kvstore.bucket=ping-pong
```

The code of the Cloudlet classes is presented in Listing 5.8.

Listing 5.8: Pong Cloudlet code

```
1 package mosaic.cloudlet.component.tests;
2
3 import mosaic.cloudlet.core.CallbackArguments;
4 import mosaic.cloudlet.core.DefaultCloudletCallback;
```

```java
import mosaic.cloudlet.core.ICloudletController;
import mosaic.cloudlet.resources.amqp.AmqpQueueConsumeCallbackArguments;
import mosaic.cloudlet.resources.amqp.AmqpQueueConsumeMessage;
import mosaic.cloudlet.resources.amqp.AmqpQueueConsumer;
import mosaic.cloudlet.resources.amqp.AmqpQueuePublishCallbackArguments;
import mosaic.cloudlet.resources.amqp.AmqpQueuePublisher;
import mosaic.cloudlet.resources.amqp.DefaultAmqpConsumerCallback;
import mosaic.cloudlet.resources.amqp.DefaultAmqpPublisherCallback;
import mosaic.cloudlet.resources.kvstore.DefaultKeyValueAccessorCallback;
import mosaic.cloudlet.resources.kvstore.IKeyValueAccessor;
import mosaic.cloudlet.resources.kvstore.KeyValueAccessor;
import mosaic.cloudlet.resources.kvstore.KeyValueCallbackArguments;
import mosaic.core.configuration.ConfigurationIdentifier;
import mosaic.core.configuration.IConfiguration;
import mosaic.core.exceptions.ExceptionTracer;
import mosaic.core.log.MosaicLogger;
import mosaic.core.utils.JsonDataEncoder;

public class PongCloudlet {

  public static final class LifeCycleHandler extends
      DefaultCloudletCallback<PongCloudletState> {

    @Override
    public void initialize(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger()
          .info("Pong Cloudlet is being initialized.");
      ICloudletController<PongCloudletState> cloudlet = arguments
          .getCloudlet();
      IConfiguration configuration = cloudlet.getConfiguration();
      IConfiguration kvConfiguration = configuration
          .spliceConfiguration(ConfigurationIdentifier
              .resolveAbsolute("kvstore"));
      state.kvStore = new KeyValueAccessor<PongCloudletState>(
          kvConfiguration, cloudlet, new JsonDataEncoder<PingPongData>(
              PingPongData.class));
      IConfiguration queueConfiguration = configuration
          .spliceConfiguration(ConfigurationIdentifier
              .resolveAbsolute("queue"));
      state.consumer = new AmqpQueueConsumer<PongCloudlet.PongCloudletState,
          PingMessage>(
          queueConfiguration, cloudlet, PingMessage.class,
          new JsonDataEncoder<PingMessage>(PingMessage.class));
      state.publisher = new AmqpQueuePublisher<PongCloudlet.PongCloudletState,
          PongMessage>(
          queueConfiguration, cloudlet, PongMessage.class,
          new JsonDataEncoder<PongMessage>(PongMessage.class));

    }

    @Override
    public void initializeSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet initialized successfully.");
      ICloudletController<PongCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.initializeResource(state.kvStore, new KeyValueCallback(),
          state);
      cloudlet.initializeResource(state.consumer,
          new AmqpConsumerCallback(), state);
      cloudlet.initializeResource(state.publisher,
          new AmqpPublisherCallback(), state);

    }

    @Override
    public void destroy(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info("Pong Cloudlet is being destroyed.");
    }

    @Override
    public void destroySucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet was destroyed successfully.");
    }

  }

  public static final class KeyValueCallback extends
      DefaultKeyValueAccessorCallback<PongCloudletState> {

    @Override
    public void initializeSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger
          .getLogger()
          .info("Pong Cloudlet - KeyValue accessor initialized successfully");
    }

    @Override
    public void destroySucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      state.kvStore = null;
      if (state.publisher == null && state.consumer == null) {
        arguments.getCloudlet().destroy();
      }
    }

    @Override
    public void getSucceeded(PongCloudletState state,
        KeyValueCallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet - key value fetch data succeeded");

      // send reply to Ping Cloudlet
      PongMessage pong = new PongMessage(arguments.getKey(),
           (PingPongData) arguments.getValue());
      state.publisher.publish(pong, null, "");

      ICloudletController<PongCloudletState> cloudlet = arguments
          .getCloudlet();
      try {
        cloudlet.destroyResource(state.kvStore, this);
      } catch (Exception e) {
        ExceptionTracer.traceDeferred(e);
      }
    }

  }

  public static final class AmqpConsumerCallback extends
      DefaultAmqpConsumerCallback<PongCloudletState, PingMessage> {

    @Override
    public void registerSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet consumer registered successfully.");
    }

    @Override
    public void unregisterSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet consumer unregistered successfully.");
      // if unregistered as consumer is successful then destroy resource
      ICloudletController<PongCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.destroyResource(state.consumer, this);
    }

    @Override
```

```java
    public void initializeSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      // if resource initialized successfully then just register as a
      // consumer
      state.consumer.register();
    }

    @Override
    public void destroySucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet consumer was destroyed successfully.");
      if (state.publisher == null && state.kvStore == null) {
        arguments.getCloudlet().destroy();
      }
    }

    @Override
    public void acknowledgeSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      state.consumer.unregister();

    }

    @Override
    public void consume(
        PongCloudletState state,
        AmqpQueueConsumeCallbackArguments<PongCloudletState, PingMessage> arguments) {
      AmqpQueueConsumeMessage<PingMessage> message = arguments
          .getMessage();

      // retrieve message data
      PingMessage data = message.getData();
      MosaicLogger.getLogger().info(
          "Pong Cloudlet received fetch request for key "
              + data.getKey());

      // get value from key value store
      state.kvStore.get(data.getKey(), null);

      message.acknowledge();
    }

  }

  public static final class AmqpPublisherCallback extends
      DefaultAmqpPublisherCallback<PongCloudletState, PongMessage> {

    @Override
    public void registerSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet publisher registered successfully.");
    }

    @Override
    public void unregisterSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      MosaicLogger.getLogger().info(
          "Pong Cloudlet publisher unregistered successfully.");
      // if unregistered as publisher is successful then destroy resource
      ICloudletController<PongCloudletState> cloudlet = arguments
          .getCloudlet();
      cloudlet.destroyResource(state.publisher, this);
    }

    @Override
    public void initializeSucceeded(PongCloudletState state,
        CallbackArguments<PongCloudletState> arguments) {
      // if resource initialized successfully then just register as a
      // publisher
      state.publisher.register();
    }
```

```
222
223     @Override
224     public void destroySucceeded(PongCloudletState state,
225         CallbackArguments<PongCloudletState> arguments) {
226       MosaicLogger.getLogger().info(
227           "Pong Cloudlet publisher was destroyed successfully.");
228       state.publisher = null;
229       if (state.consumer == null && state.kvStore == null) {
230         arguments.getCloudlet().destroy();
231       }
232     }
233
234     @Override
235     public void publishSucceeded(
236         PongCloudletState state,
237         AmqpQueuePublishCallbackArguments<PongCloudletState, PongMessage> arguments) {
238       state.publisher.unregister();
239     }
240
241   }
242
243   public static final class PongCloudletState {
244     AmqpQueueConsumer<PongCloudletState, PingMessage> consumer;
245     AmqpQueuePublisher<PongCloudletState, PongMessage> publisher;
246     IKeyValueAccessor<PongCloudletState> kvStore;
247   }
248 }
```

Beside the queue consumer and publisher accessors for the queues, the state of the **Pong** Cloudlet contains also a reference to the accessor for the key-value store (line 246). Just like the **Ping** Cloudlet, **Pong** Cloudlet also contains callbacks for the queue-related events, with resembling code.

The **KeyValueCallback** class extends the **DefaultKeyValueAccessorCallback** class which implements the **IKeyValueAccessorCallback** callback interface. Beside the **initializeSucceeded()** and **destroySucceeded()**, the KeyValueCallback overrides only the **getSucceeded()** method. This is called when the data asked to the key-value store is returned to the Cloudlet after a previous get request (line 187). Here, the **PongMessage** is built using retrieved data and sent to the **Ping Cloudlet**.