Design

Requirements

Before starting on the implementation, a design has to be made. The design has to take the requirements into account:

- The system has to work with OSGi.
- There has to be an abstraction over the messaging system so the technical dependence is minimized.
- The abstraction should inform the software of the presence of the messaging system.
- The system should be modular so it is not dependent on any specific technology.
- The system should work distributed.
- The system should not have a single point of failure.

Abstraction

An abstraction is usually made using interfaces. These Interfaces can be implemented for various messaging systems. The abstraction should also inform the software of the presence of the messaging system. This is a typical OSGi feature. In OSGi the implemented interfaces are registered as services. These services can be made unavailable in the OSGi framework. Bundles using the services will be informed that they can't use the services anymore.

Modularity

Modularity is another feature of OSGi. There will be components that perform different tasks. A component is dependent on a specific technology but it communicates through API's. Other components don't know about the used technology.

Work distributed

To have the system work distributed it needs to be able to send messages across OSGi frameworks and it needs to know which publishers and subscribers are active in other OSGi frameworks so it can connect to them.

Because Apache Kafka was chosen as the messaging system sending messages across OSGi frameworks is not a problem. All the messages go to and from the Kafka cluster.

A mechanism to know which publishers and subscribers are active in other OSGi frameworks is still needed. There are several options for this mechanism:

- A protocol based on UDP multicast.
- A distributed database

With a protocol based on UDP multicast there would be no dependency on any other system but keeping all the systems in sync would be a challenge. A distributed database would add another dependency but INAETICS already includes a distributed database (etcd) and Kafka relies on Zookeeper. One of these can be used.

Etcd will be used because it is part of the INAETICS stack. If someone wants to use the publish/subscribe system without Apache Kafka it also doesn't need Zookeeper. Other technologies in INAETICS rely on etcd so it is unlikely that etcd will be replaced.

No single point of failure

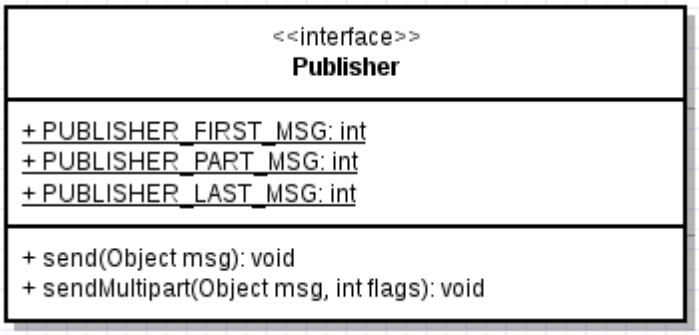
Because the system uses Apache Kafka and etcd there is no single point of failure. If a machine fails the publishers and subscribers on it will of course be gone but it won't affect others. The system will fail if Kafka or etcd is unavailable but these are distributed so they too have no single point of failure.

Implementation

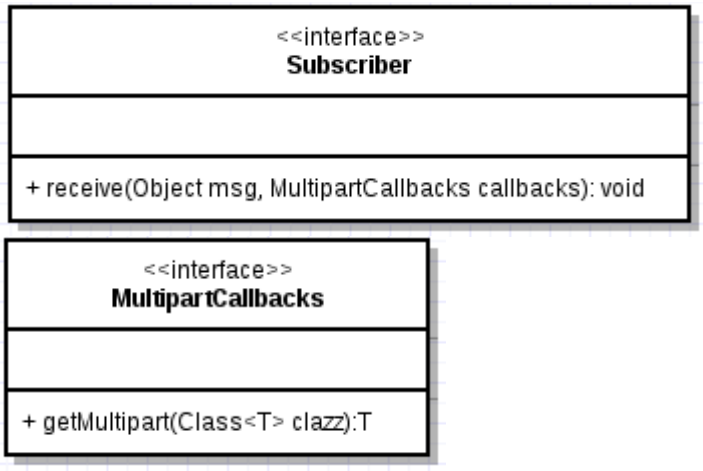
Abstraction

The abstraction will be interfaces for Publishers and Subscribers. These can be implemented for various messaging systems. The interfaces are based on the C implementation. The interfaces are as follows:

Publisher interface:



Subscriber interface:



The Publisher has two methods: send and sendMultipart. The send method sends one object to the messaging system.

The sendMultipart can be used to send multiple objects at once. These objects are grouped into one message on the messaging system. The sendMultipart has an extra argument, an integer with flags. The flags determine if the object is the first of a group, part of a group, the last of a group or any combination thereof. All the objects are send when a last message is send to the Publisher. Sending two first messages without a last message in between, or sending a part or last message without a first message will result in an exception.

The Subscriber only has one method: receive. Receive takes two arguments, an object and a MultipartCallbacks. The object is the object from the Publishers send method or the first message from the sendMultipart. The MultipartCallbacks can be used to access the other objects from the sendMultipart. MultipartCallbacks has a getMultipart method that takes a class as its argument. It will

return a message of that class. This means that a multipart message can only have one object of a class. If you want to send more than one object of a class you will have to wrap them in a container.

Presence of the messaging system.

In OSGi, applications are made out of bundles. Bundles can register services, and other bundles can use those services. The Publishers and Subscribers will be services.

Publishers will be made and registered by the publish/subscribe system. Publishers can have many different configurations and as result will be created when a bundle requests a Publisher, giving the publish/subscribe system the ability to inspect the requesting bundle for additional configuration (e.q. QoS, queue size, technology specific configuration, credentials, etc).

Subscriber services are registered by subscribers; the publish/subscribe system will monitor the registered subscriber services and inspect the bundle for additional configurations. Then it will setup the publish/subscribe system so that it can receive and forward messages.

The publishing bundle will know the messaging system is unavailable when it can't get a Publisher service. The subscribing bundle will not receive any messages when the system is unavailable.

Components

The publish/subscribe system is made out of 4 components. They will be introduced here and explained in more detail later.

The first component is the TopologyManager. The TopologyManager is the heart of the system. It listens to the OSGi framework so it knows when a Publisher service is requested or a Subscriber service is provided. When this happens the request is forwarded to the available PubSubAdmins.

The PubSubAdmin is the second component. It is the only component that knows about the underlying messaging system and therefore is technology dependent. The PubSubAdmin can create Publisher services for the messaging system and connect Subscribers services to it.

A set of key/value properties is given back to the TopologyManager. These properties contain information about the connection made by the PubSubAdmin, like the name of the messaging system, the IP address and port, the topic name etc. The TopologyManager will use DiscoveryManagers to announce the information on the network. A DiscoveryManager is techology dependent and is responsible for distributing information about the available publishers and subscriber through the network.

The last component is the Serializer. The Serializer serializes the objects that are sent to the messaging system to byte arrays. When a message is received it will deserialize the byte array back to an object of the original class.

The following diagram shows all the components:

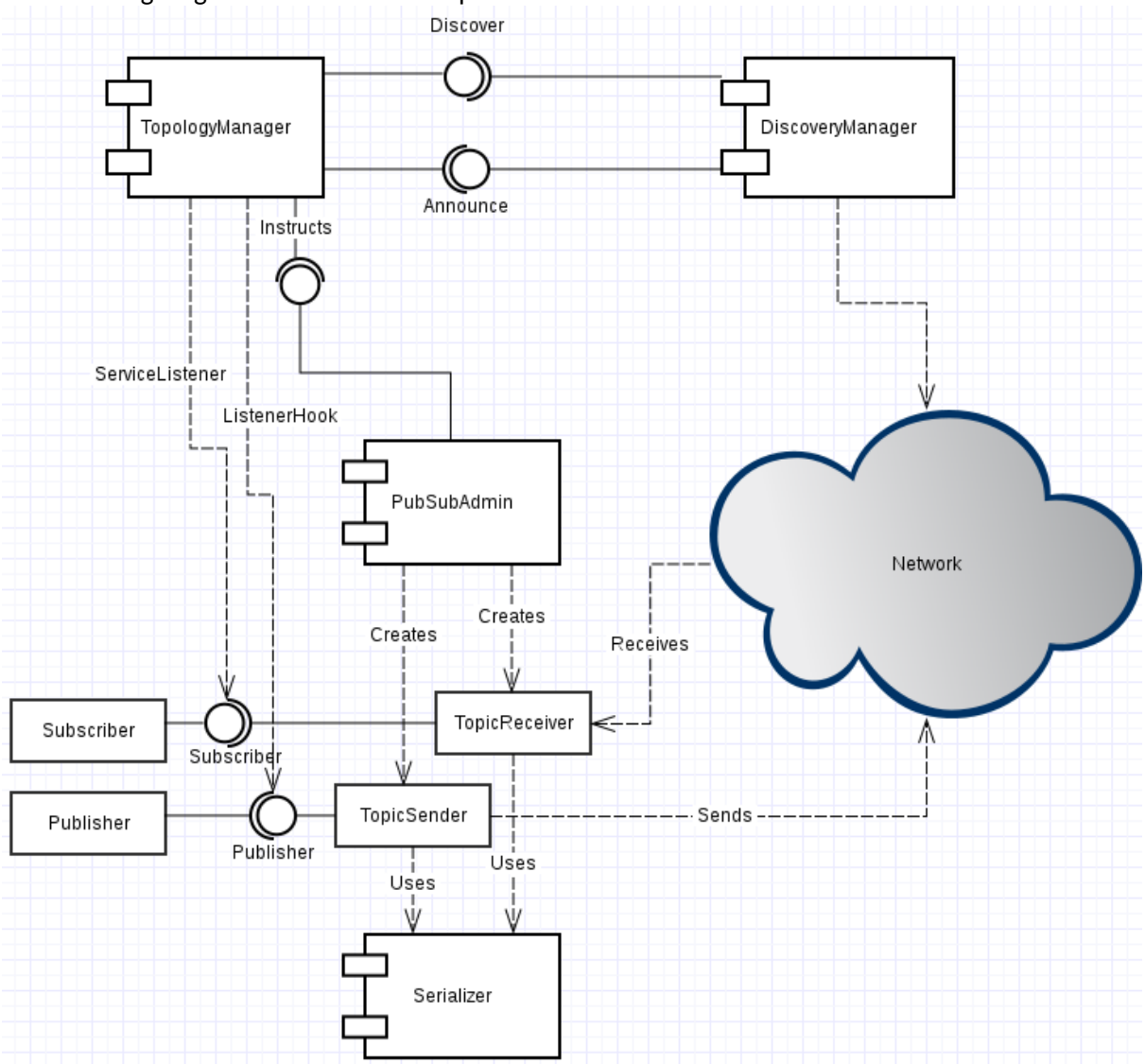


Figure 1: Component diagram of the system

Component composition

It is possible to have multiple PubSubAdmins, DiscoveryManagers and Serializers active at the same time.

If two or more PubSubAdmins are available, the TopologyManager will choose one based on the configuration provided by the bundle that requested the Publisher/Subscriber.

In the case of multiple DiscoveryManagers, the TopologyManager will use all of them.

The publishing bundle decides which Serializer will be used so multiple Serializers do not affect the system, as long as the ones required by the publishing bundles are active.

There can only be one TopologyManager per OSGi framework. All the PubSubAdmins, DiscoveryManagers and Serializers should be active before the TopologyManager starts. Adding and removing components while the TopologyManager is active is not supported.

The TopologyManager

The TopologyManager listens to the OSGi framework so it knows when a Publisher service is requested or a Subscriber service is provided. It does that by implementing a ListenerHook and a ServiceListener.

The ListenerHook is used to find out when a Publisher service is requested. When bundles want to use a service they have to ask the OSGi framework for a reference to the service. Because the framework is dynamic, services can come and go at any time. Bundles need to track the services they're using to make sure the service is still available. This can be done with a ServiceListener or a ServiceTracker.

When a ServiceListener or ServiceTracker is started for a service, the ListenerHooks are called. The ListenerHook receives the requesting bundle and the properties of the requested service. The TopologyManager checks the properties to see if the request is for a Publisher.

If the request is for a Publisher, the TopologyManager will ask all the currently available PubSubAdmins if they can create a TopicSender for the given properties. The PubSubAdmins will return a positive score if they can and a negative score if they can't. The PubSubAdmin with the highest positive score will be asked to create the TopicSender (if there is one). See the chapter about the PubSubAdmin for more information about the scoring process.

A TopicSender is an object that creates the Publisher service. It contains the properties it was given by the PubSubAdmin. The TopologyManager will only create one TopicSender per topic. It is not allowed to have different configurations for the same topic.

The TopicSender registers itself as a ServiceFactory for Publishers of the topic the TopicSender was made for. A ServiceFactory will create unique service objects for a requesting bundle. It will also destroy the service object when it is no longer used.

The TopologyManager will send the properties of the TopicSender to the DiscoveryManagers which will announce it on the network.

When a ServiceListener or ServiceTracker is stopped the ServiceFactory will destroy the Publisher and the ListenerHook is called again. If the TopicSender is not used by any other publishers the TopologyManager will start a Thread that removes the TopicSender after a configurable delay; The delay is added to prevent too much entropy in the system. TopicSenders are not deleted immediately because bundles will sometimes be restarted, which will cause other bundles to restart. All these restarts will result in removing and creating TopicSenders which can be an expensive operation.

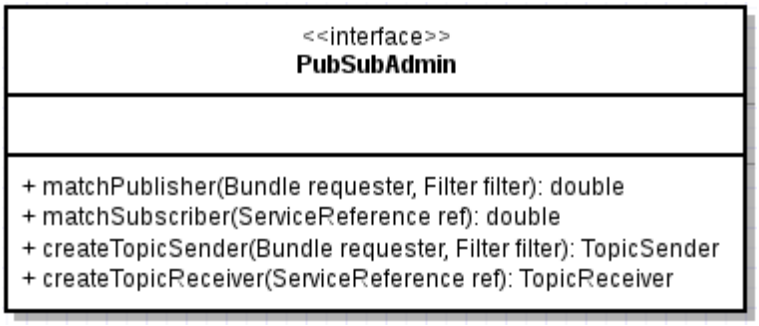
On the Subscriber side a ServiceListener is used. A ServiceListener is called when a service is registered or unregistered. The TopologyManager will check if the service is a Subscriber service and if so create a TopicReceiver the same way a TopicSender is created, announce it and remove it when it is no longer used.

A TopicReceiver is not a ServiceFactory because it does not create Subscribers. The TopologyManager will add Subscribers to the TopicReceiver. The TopicReceiver will call the Subscribers when a message is received.

The TopologyManager will register itself as an EventHandler for the OSGi EventAdmin. The DiscoveryManagers will use the EventAdmin to send events about discovered Publishers and Subscribers on other machines.

The PubSubAdmin

The PubSubAdmin is the component that knows about the underlying messaging system. It implements the PubSubAdmin interface:



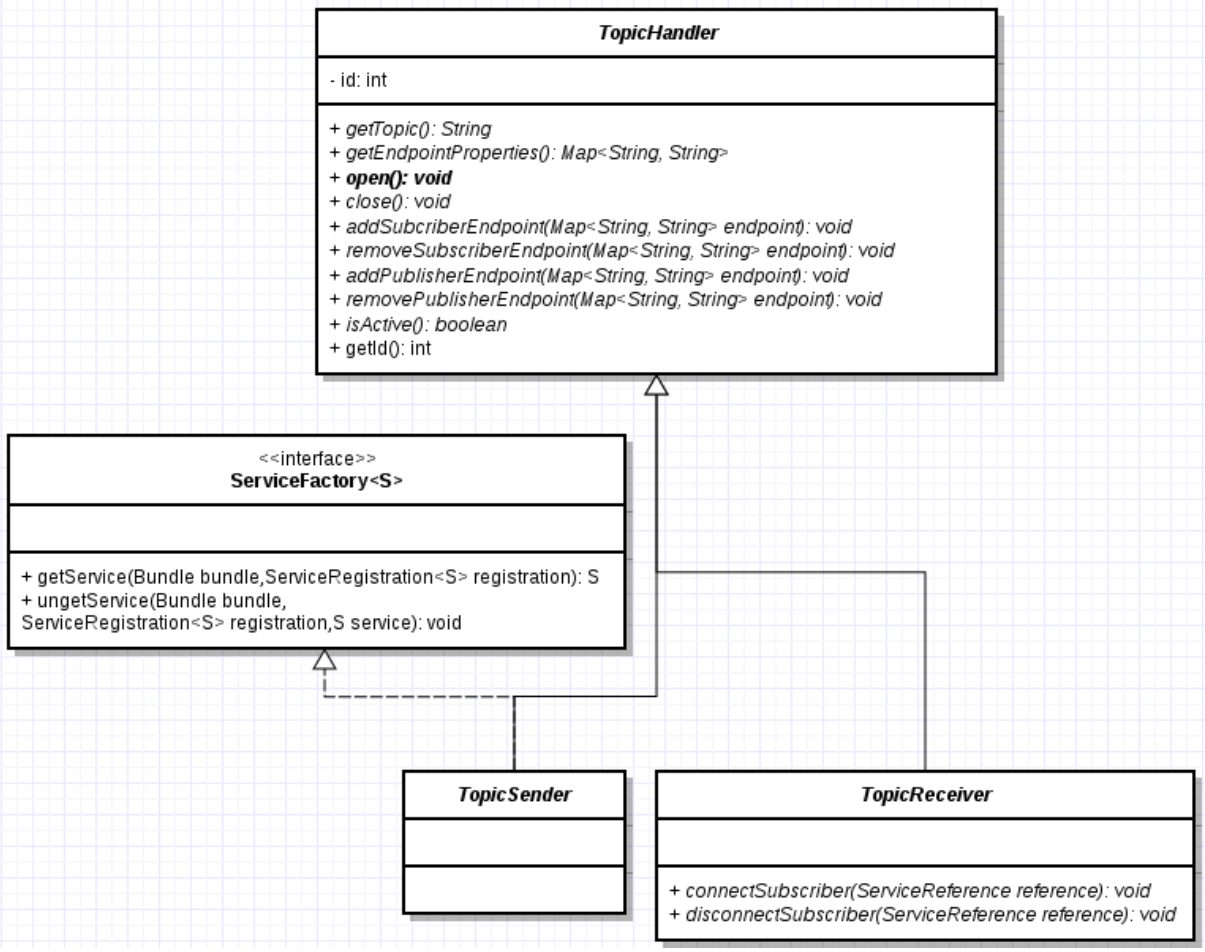
The matchPublisher and matchSubscriber methods are called by the TopologyManager. If the result is positive the PubSubAdmin can handle the Publisher/Subscriber, if it is negative it can not. The TopologyManager will pick the PubSubAdmin that returns the highest positive value (if available).

The PubSubAdmin can use the extender pattern to find configuration files for the requested Publisher/Subscriber. The extender pattern is when a bundle inspects another bundle for files and uses them to extend its functionality. The PubSubAdmin can look in the /pubsub folder for the configuration files. The filename should be the same as the name of the topic and the file extension should be .pub or .sub, or if you have a configuration file for a specific admin give it an extension for that admin. For example the Apache Kafka PubSubAdmin will look for files with the .kafkapub or .kafkasub extension.

With all the configuration information available the PubSubAdmin can determine the score. When implementing the matchPublisher and matchSubscriber methods it is important to take the implementations of the other PubSubAdmins into account. The Apache Kafka PubSubAdmin will add one for every configuration property and two for every Kafka specific property.

This concept can be used to support a different QoS based on the underlying technology. For example Apache Kafka can be used for messages with high importance and messages where subscribers can join after the message was sent. On the other hand for high throughput data a PubSubAdmin based on (Multicast)UDP could be preferable.

If the PubSubAdmin is chosen the createTopicSender or createTopicReceiver method is called. This will create the TopicSender or the TopicReceiver. Both extend the TopicHandler class:



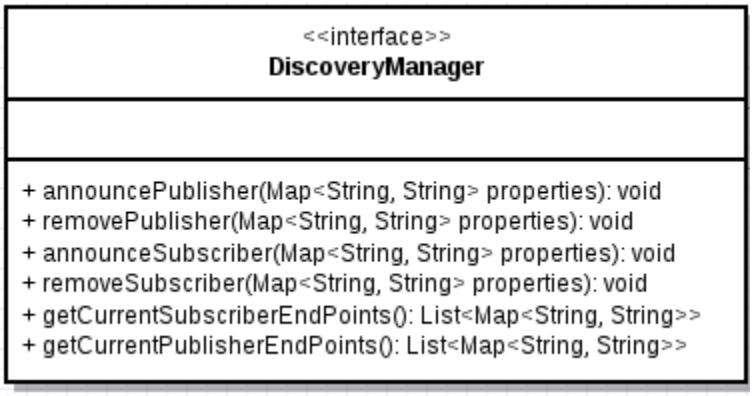
The TopicSenders and TopicReceivers are managed (open,close and add/remove subscribers/publishers) by the TopologyManager. The add/remove endpoint methods are called when a Publisher/Subscriber is discovered or removed on another machine. The isActive method can be used to see if the TopicSender/TopicReceiver is active. If it's not the TopologyManager will remove it. The ID is used to identify the TopicSender/TopicReceiver.

The TopicSender only adds the ServiceFactory interface to create and remove Publisher services.

The TopicReceiver has the connectSubscriber and disconnectSubscriber methods. The TopicReceiver will call the connected Subscriber services when a message is received.

The DiscoveryManager

The DiscoveryManager announces, removes and discovers Publishers and Subscribers on the network. It is up to the DiscoveryManager implementation how to do this (e.g. distributed hashmap, mdns, distributed db, etc). It informs the TopologyManager of the Publishers and Subscribers that are active in other OSGi containers. Every DiscoveryManager has to implement the DiscoveryManager interface:



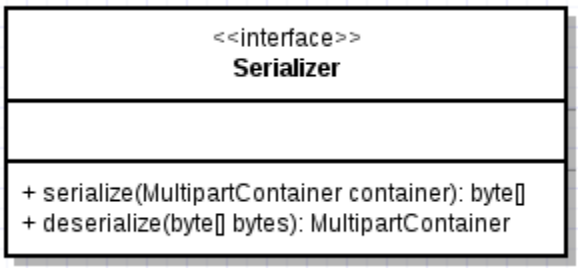
It has announcePublisher and announceSubscriber methods to announce a Publisher or Subscriber. The removePublisher and removeSubscriber methods remove them again. The methods getCurrentSubscriberEndPoints and getCurrentPublisherEndPoints return all currently active Publishers and Subscribers. These methods are called by the TopologyManager.

The DiscoveryManager uses the OSGi EventAdmin to send events when a Publisher or Subscriber is Discovered. The TopologyManager will listen to these events.

When a machine fails the Publishers and Subscribers on that machine need to be removed on the network. The DiscoveryManager should take care of this, for example by using a time-to-live system. The DiscoveryManager needs to announce its Publishers and Subscribers every 10 seconds, if not updated in 15 seconds the entry will be removed.

The Serializer

The final component is the Serializer. The Serializer serializes Java objects to byte arrays and back. The serialized object should be language agnostic, so that support for other languages (e.g. C) is possible. Every Serializer has to implement the Serializer interface:



The Serializer always serializes from and to a MultipartContainer. A MultipartContainer contains a list of Objects and a list of Strings. The Strings represent the classes of the Objects. They are used to deserialize the object to its original class.

Component Implementations

Apache Kafka PubSubAdmin

Apache Kafka is the messaging system chosen for this project. The KafkaPubSubAdmin handles the creation of TopicSenders and TopicReceivers for Apache Kafka.

The KafkaPubSubAdmin has the following design:

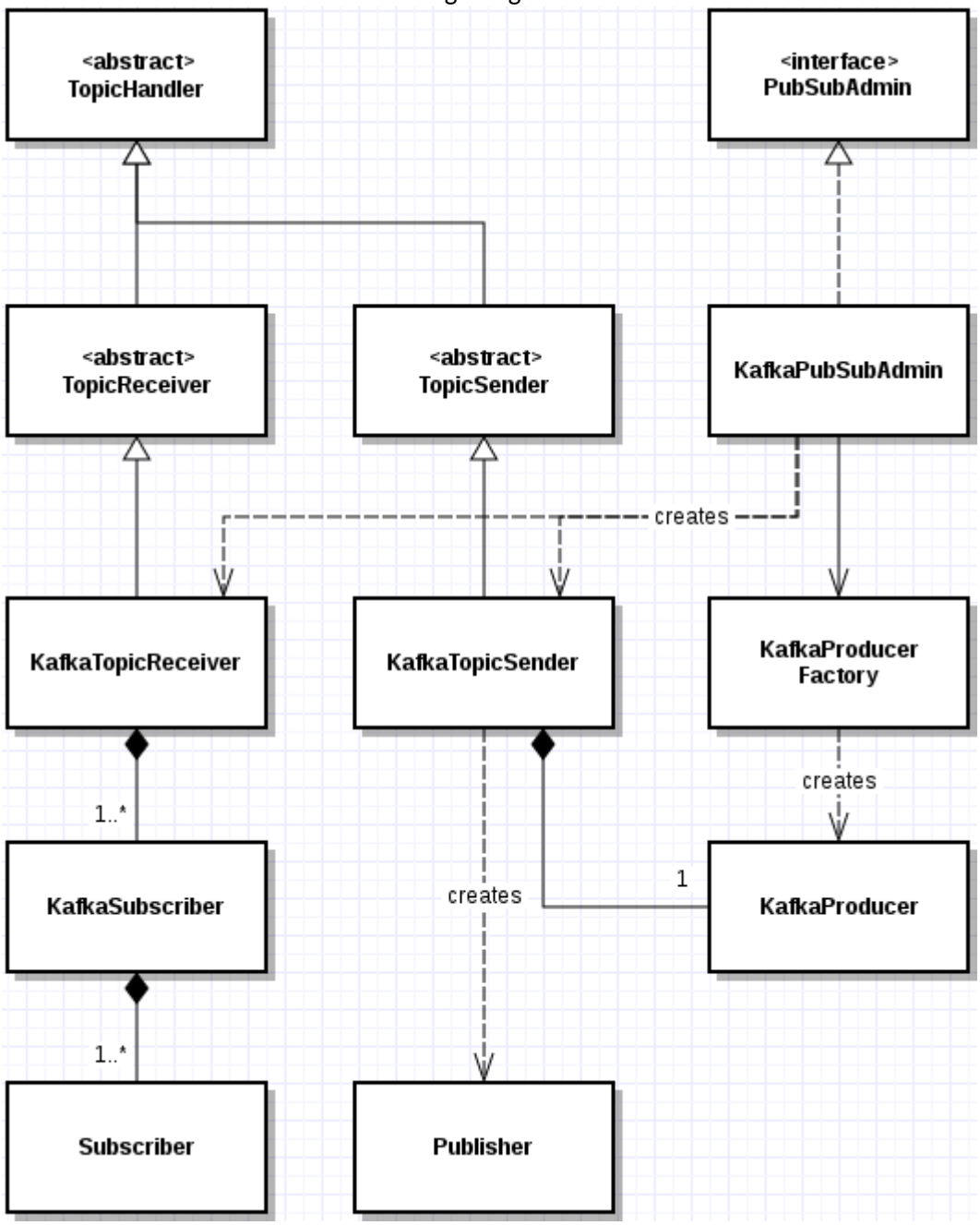


Figure 2: Class diagram of the Apache Kafka PubSubAdmin

The KafkaPubSubAdmin can create KafkaTopicReceivers and KafkaTopicSenders. KafkaTopicReceivers receive messages for a topic. Kafka uses topics itself but a topic in kafka does not map to a topic in the publish/subscribe system. This is because It is possible that multiple Publishers are sending on the same topic with different Serializers. A topic in Kafka will be created for every Serializer.

The KafkaTopicReceiver will make a KafkaSubscriber for every topic in Kafka. The KafkaSubscriber will call the Subscribers when they receive a message.

The KafkaTopicSender needs a KafkaProducer to send messages to Kafka. The KafkaProducer is made by the KafkaPubSubAdmin using a factory. The factory will create a KafkaProducer based on the properties it is given. It will cache the created producers and return the cached instance when a producer with the same properties is requested. This is because Kafka can batch messages to increase performance.

The KafkaTopicSender registers itself as a ServiceFactory to create Publishers.

The Apache Kafka PubSubAdmin will use the extender pattern to get the configuration information when the matchPublisher and matchSubscriber methods are called. It will look for files with the topic name as the filename and ".pub" or ".kafkapub" for publisher configurations and ".sub" or ".kafkasub" for subscriber configurations. To determine the score for the match the Kafka PubSubAdmin will add one for every configuration property and two for every Kafka specific property.

The KafkaPubSubAdmin decides the configuration of the KafkaTopicReceivers and KafkaTopicSenders that it creates. A configuration is a map of key/value pairs. This map is created in 3 steps:

1. The basic configuration is the configuration given by the requesting bundle.
2. The KafkaPubSubAdmin has a map of default configuration values. It will add any configuration whose key is not in the basic configuration.
3. The Apache Kafka library has default values for any configuration still missing.

The default values of the KafkaPubSubAdmin can be set using the Configuration Admin Service. All the producer configs and the old consumer configs listed in the Apache Kafka 0.9 documentation (Kafka 0.9.0 Documentation, 2016) can be used except for the key and value serializers. If a serializer is set the KafkaPubSubAdmin will replace it. This is because serialization is handled by the Serializer component.

To set the default values call the Configuration Admin with a map of keys and values. Prepend the producer keys with "pub:" and the consumer keys with "sub:". The keys and values are not validated. It is up to the user to make sure they are correct. Any exceptions will be logged.

Because of the Publisher/Subscriber abstraction interface it is not possible to set the partition to send your messages to. The partition will be chosen at random.

Etcd DiscoveryManager

The EtcdDiscoveryManager is a DiscoveryManager that uses etcd. Etcd is a distributed key-value store. It allows clients to watch a key. This means that a client will be notified when a key updates. Etcd was chosen because it is already used in INAETICS and it is simple to use.

Keys in etcd have to be unique. Therefor the keys consist of the type (publisher or subscriber), the topic name, the framework UUID and the ID of the TopicHandler. The framework UUID is unique for the running OSGi framework. The key is:

/pubsub/<type>/<topic>/<framework UUID>-<service ID>

The value is a map of key-value properties. These are stored as JSON. The following properties are mandatory:

- service.id: Contains the ID of the TopicHandler
- type: Can be "publisher" or "subscriber"
- pubsub.topic: The topic

The EtcdDiscoveryManager watches etcd for changes and will inform the TopologyManager when a change occurs. It will publish its Publishers and Subscribers with a time to live (TTL). When the TTL

expires etcd will remove the key. The EtcdDiscoveryManager will update the TTL before that happens. This will make sure the Publishers and Subscribers are removed when the machine fails.

The TTL and the TTL refresh rate can be configured using the Configuration Admin Service. The url of etcd can also be configured.

Jackson Serializer

The Jackson Serializer serializes objects to JSON using the Jackson library. In Java there are two widely used libraries to work with JSON, Jackson and GSON. They have similar features and performance is similar. GSON is faster with small objects and Jackson with larger objects. (Jackson vs Gson: A Quick Look At Performance, 2016)

Both could have been used for the Serializer but Jackson was used because we had experience with it.