Knipp RRI Toolkit
Overview

Table of Contents

Document History

Version	Date	Author	Comments
1.0	2005-04-29	Klaus Malorny	Initial creation.
1.1	2005-05-26	Klaus Malorny	Updated section 2.1, added sections 2.6 and 4.2.
1.2	2005-08-12	Klaus Malorny	Updated section 1.2
1.3	2006-07-19	Klaus Malorny	Updated properties example (3.2), added section 4.3
1.4	2019-05-03	Klaus Malorny	Minor updates of text and graphics

1. Introduction

1.1. Purpose

The toolkit implements an interface to DENIC's Real-time Registry Interface (RRI) for the Java programming language. It represents DENIC requests and responses, contact and domain data as well as messages as Java objects, allowing an easy access to the registry functionality. For the communication, both key-value and XML representations are supported at the user's discretion. The toolkit supports TCP communication via SSL/TLS encryption. As the communication interface is thread-safe, it enables multi-threaded access to the registry without effort.

1.2. Prerequisites

The toolkit makes use of Java 8 constructs. As such, Java 8 or a later version of Java is required to compile and use the toolkit. If there is a need to rebuild the library, Apache Ant 1.6 or later is required.

Although the toolkit fully supports Internationalized Domain Names (IDN), it does not provide any means for the conversion between the IDN and its Punycode representation by itself. While Java itself provides means for the conversion in the form of the `java.net.IDN` class, it should be noted that at the time of writing (at least up to Java 12), this class only supports the IDNA2003 standard. However, DENIC uses the IDNA2008 standard, which behaves differently especially in the handling of the "German double s" character. As such, it is recommended to use a third party library for this purpose. The ICU library[1] is known to work well, but other libraries may also exist and be suitable.

In case that you explicitly include the XML parser Apache Xerces in the class path of your Java runtime environment, it may be required to update it to version 2.7 at least, which fully supports all of the JAXP 1.3 classes.

1.3. Licensing

The toolkit is published under the GNU Lesser General Public License 2.1. This license allows the modification of the toolkit as well as the use of the toolkit for commercial applications. Please refer to the license for details.

2. Concepts

2.1. Requests and Responses

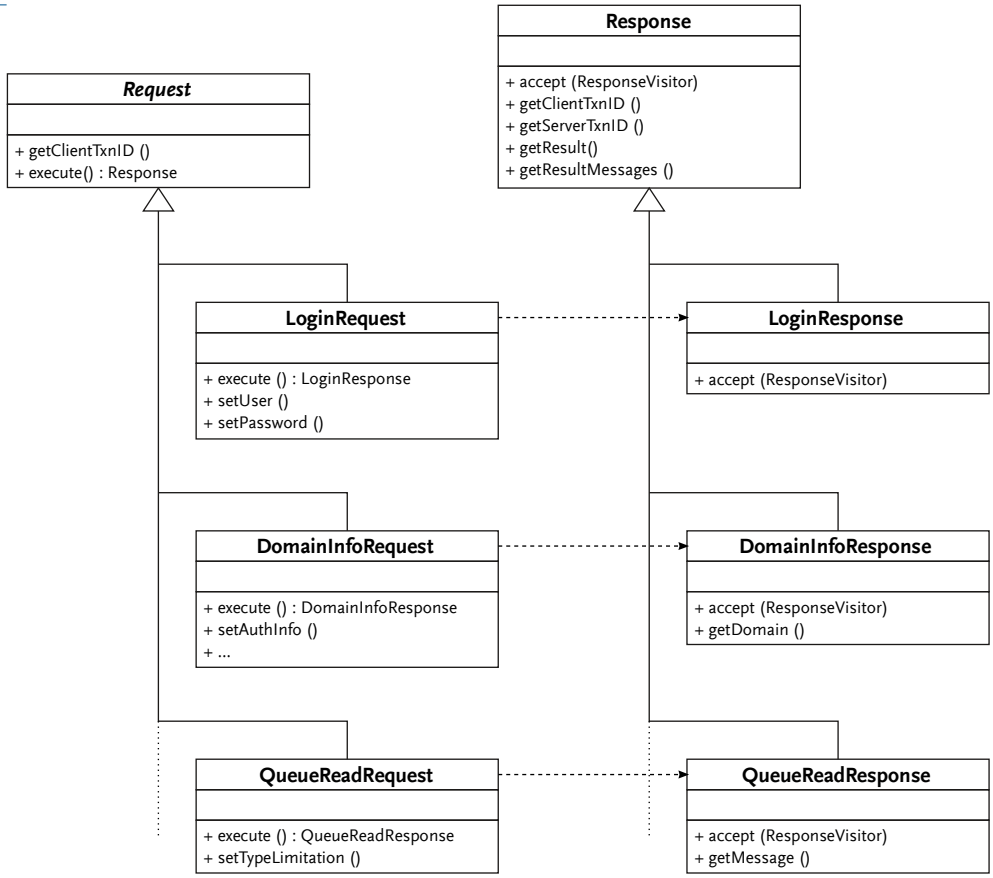
Each individual request and response is implemented as a separate class. They share common base classes. There is a one-to-one correspondence between the request and response classes to have a full symmetry, although most DENIC requests do not return any additional data besides the standard information (transaction IDs, result, result messages).

1 `http://site.icu-project.org/`

While this concept is straight-forward, two notes must be added: First, to allow object-oriented access to the different `Response` subclasses, the class implements the well-known *Visitor* pattern in conjunction with the `ResponseVisitor` interface.

Second, the `Request` class has a method called `execute`, which sends itself to the given `Port` instance and returns a `Response` object. Each subclass overrides this method to return an instance of the corresponding `Response` subclass.

Basic Principle of the Request and Response class hierarchies



2.2. Message Classes

For each message type a corresponding subclass of `Message` exists. The `Message` class defines a `getType` method that returns the actual type of the message. Also, the class implements the Visitor pattern.

2.3. DomainData and ContactData Classes

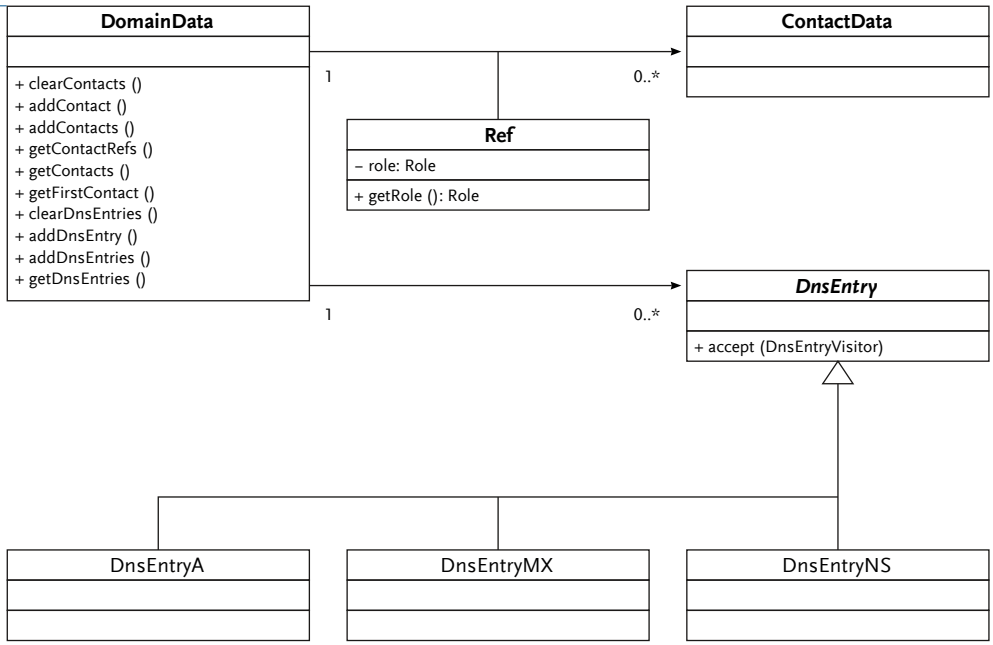
The `DomainData` and `ContactData` classes are used to store domain resp. contact information for both requests and responses. This allows data to be read from the registry via an Info request, modified and sent back to the registry via an Update request.

In some cases, the `ContactData` class contains nothing more than the handle of the contact. This is intentional.

The `DomainData` class uses an inner class (`Ref`) to associate the contacts with their roles. It also manages instances of the `DnsEntry` class and its subclasses to represent the DNS configuration of the domain.

The `DnsEntry` class implements the Visitor pattern also.

DomainData, ContactData and Related Classes



2.4. RegistryChannel and ChannelFactory Interface

These two interfaces are used for the abstraction of the communication channel to the registry. The `RegistryChannel` contains send and receive methods on a binary level. The `ChannelFactory` is an interface to a class that is capable of creating new channels.

`TcpChannelFactory` is a class which implements the `ChannelFactory` interface for TCP connections. The class takes the initialization parameters from an instance of the `TcpParams` class.

2.5. Port Interface

Since the `RegistryChannel` operates on a binary basis and also does not provide means to be accessed from multiple threads in parallel, a different "high-level" interface called `Port` exists that is meant to be used by the application developer. It serves as a facade to the conversion process from the request objects to the binary data that is sent to the registry as well as the reverse direction from the received binary data to the response objects. In addition, it takes care about parallel access from multiple threads, ensuring that each thread receives the response that matches the given request.

Two implementations of `Port` exist: `SinglePort` and `PooledPort`. `SinglePort` allocates a single `RegistryChannel` instance from the given factory and uses

this for the communication. `PooledPort` is a more complex implementation. It manages a pool of multiple connections to the registry. New channels are created and discarded as needed. In contrast to `SinglePort`, `PooledPort` also performs the login and logout at the registry. If more threads are requesting a communication with the registry than channels are available, the threads are put into a FIFO queue until a channel becomes available or a new one has been created and prepared (i.e. a login has been performed).

2.6. Error Handling

The toolkit provides an additional enumeration `ErrorType`, which simplifies the handling of the result messages included in the response. For all known error codes, a corresponding enumeration value exists. The method `getErrors` allows to create a set of all error codes that appear in the response. The `test` method allows to test for a small set of errors and returns the first matching one. It reflects the programming style that typically a certain number of error conditions are expected and handled (like that the domain is already registered on a CREATE request) while other errors are handled in a generic way. With the help of the `filter` method, the actual `ResultMessage` instances that contain certain error codes can be determined. By using the helper classes `ErrorPattern` (along with its subclasses) and `ErrorMapper`, the result messages can be filtered individually, e.g. to filter all result messages related to the zone checking as a part of the error reporting to the user. The use of these classes is optional, the developer can operate directly on the `Response`, `ResultMessage` and `ErrorCode` classes if required.

2.7. Inner Working

Several other classes represent the “inner working” of the toolkit. The `Codec` class is responsible for the conversion between the Java objects and the binary representation that is used at the communication level. It contains both all-in-one methods and methods that perform the conversion steps individually.

Both key-value and XML responses sent by the registry are not completely self-describing, i.e. just by looking on the response it is not always possible to tell the type of the originating request. Because of this, the creation of response objects has not been integrated into the codec, but delegated to a separate interface named `ResponseFactory`. A default implementation, the singleton `DefaultResponseFactory` class, derives the class either from the request object, or, if it is not available, heuristically by a set of rules.

`KVList` and `KVMultiList` are helper classes to store the key-value representation during the conversion, while the `XmlBuilder` and `XmlHelper` classes contain methods to simplify the creation and parsing of XML documents.

3. Using the Toolkit

3.1. Required Steps

To make use of the toolkit, the following steps are required:

- creating a channel factory
- creating a port
- creating and sending requests, receiving and processing responses
- shutting down (optional)

3.2. Creating a Channel Factory

As mentioned above, the toolkit implements a SSL/TLS encrypted TCP connection to the registry. In the context of SSL/TLS, the toolkit operates as a client. It has the duty to verify the server certificate it receives from the server. As the test environment of DENIC uses a self-signed server certificate, you must setup a key store that contains this certificate and specify this in the configuration of the TCP connection, otherwise the connection will not be established. The toolkit also allows the specification of a client certificate, but at the moment, the DENIC registry does not make any use of it.

The first step is to create an instance of the `TcpParams` class. While you can setup the various parameters individually, it contains a convenience method that takes all parameters from a given Properties instance.

Initialization of a TcpParams instance

```
Properties props = new Properties ();
props.load (new FileInputStream ("test.properties"));
TcpParams params = new TcpParams ();
params.setupFromProperties (props,  "tcp.");
```

A typical property file could contain the following data:

Typical Contents of the Properties File

```
tcp.server.name: rri.test.denic.de
tcp.server.port: 51131
tcp.trust.keystore.path: trust.jks
tcp.trust.keystore.password: trustMe
tcp.trust.verifyname: no
```

Creating the factory:

Factory Creation

```
ChannelFactory factory = new TcpChannelFactory (params);
```

3.3. Creating a Port

Both `Port` implementations optionally accept an instance of the `Codec` class. This allows the selection between key-value representation and XML representation on the protocol level, as well as the use of a custom `ResponseFactory`. If not specified, a default codec is created that uses XML representation. To create a port, use:

SinglePort

```
SinglePort port = new SinglePort (factory);
```

or, alternatively,

PooledPort

```
PooledPort port = new PooledPort (factory, null, "DENIC-60-USER",
  "myPassword", 0, 1, 10);
```

For the `SinglePort` solution, a login has to be performed at first before the port is actually usable for operation.

3.4. Working with the Port

Once the port has been set up properly, it can be used for doing requests. The following example shows the query for a given contact:

Request Example

```
ContactInfoRequest req = new ContactInfoRequest ("DENIC-60-TEST");
ContactInfoResponse resp = req.execute (port);

if (resp.isSuccess ())
{
  System.out.println (resp.getContact ());

} else
{
  System.out.println (“request failed”);
}
```

3.5. Shutting Down

The `PooledPort` implementation has a special method called `shutdown` that performs a logout and a close on all channels in the pool. The call does not return before this has been completed, which may take some seconds.

4. Miscellaneous

4.1. Test Class

The `Test` class currently contained in the distribution is not an official part of the package. It represents a small command interface which takes command line arguments, creates a request from it and sends it to the registry. Due to its nature it does not provide much convenience and creates a lot of debugging output.

The first argument must always be the name of a properties file. It must contain the following properties:

`tcp.`*<key>*`:` *<param>*	see TcpParams.setupFromProperties details; prefix is "tcp."
`user:` *<username>*	the user name for the login at DENIC
`password:` *<password>*	the password for the login at DENIC

`xml: true | yes | no | false`	if true/yes, data is sent to DENIC in XML format, if false/no, data is sent to DENIC in key/value pair format

The second parameter is the name of the request to send to the registry. Use "`help`" to view a list of all supported requests and their options.

4.2. DebugChannelFactory & DebugPort Classes

These two classes provide simple means to print the interaction between the application and the registry to the console. They are merely for testing purposes. Similar approaches may be used by the application developer to implement logging or recording of the communication in order to allow a later review in case of problems.

4.3. Idle TCP Connections

DENIC closes, like other registries, connections that haven't been used for a while. Unfortunately, this cannot be detected within Java except by actually reading from or writing to the socket. Additionally, it has been observed that some firewalls drop their contexts about the connection some time after having monitored the single side close, making them unable to respond to the close of the client, once it detects the closed host connection. On some client platforms, this caused the JVM to hang in the `close()` call for several minutes, until the TCP stack declared the connection as really closed. This is a no-go for production environments.

To solve this problem, two measures have been taken. First, the `TcpChannelFactory` can be configured to monitor the created connections and proactively closes those connections that have been idle for a while. To enable the feature, the `setIdleTimeout` method has to be called on `TcpParams` or the `timeout.idle` property has to be set for the `setupFromProperties` method. Please note that the `SinglePort` implementation cannot reestablish a connection by itself, therefore it is not recommended to use it together with the idle timeout.

The second measure is that the `PooledPort` does not fail on an I/O exception during a write attempt to the socket. Instead, it drops the used channel and requests another one from the internal pool until it succeeds. If the pool runs dry, a new channel is allocated and a login is performed, like under normal conditions. However, if this fails, too, the failure is reported to the caller.