Beautiful XMPP Testing

Remko Tronçon

At my first job interview, one of the interviewers asked me if I knew what “unit testing” was and whether I used it before. Although I had been developing an XMPP-based instant messaging (IM) client for years, I had to admit that I only vaguely knew what it was, and that I hardly did any automated testing at all. I had a perfectly good reason, though: since XMPP clients are all about XML data, networks, and user interaction, they don’t lend themselves well to any form of automated testing. A few months after the interview, the experience of working in an agile environment made me realize how weak that excuse was. It took only a couple of months more to discover how beautiful tests could be, *especially* in environments such as XMPP, where you would least expect them to be.

Introduction

The *eXtensible Messaging and Presence Protocol* (XMPP) is an open, XML-based networking protocol for real-time communication. Only a decade after starting out as an instant messaging solution under the name *Jabber*, XMPP is today being applied in a broad variety of applications, much beyond instant messaging. These applications include social networking, multimedia interaction (such as voice and video), micro-blogging, gaming, and much more.

In this text, I will try to share my enthusiasm about testing in the XMPP world, more specifically in the *Swift* IM client (`http://swift.im`). Swift is only one of the many XMPP implementations out there, and may not be the only one that applies the testing methods described here. However, it might be the client that takes most pride in beautiful tests.

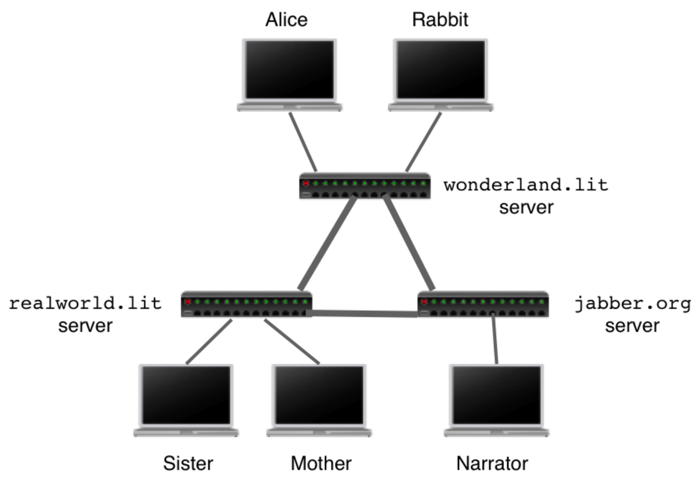
So, what do I consider to be “beautiful testing”? As you probably discovered by now, opinions on the subject greatly vary. My point of view, being a software developer, is that beauty in tests is about the *code* behind the tests. Naturally, beautiful tests look good aesthetically, so layout plays a role. However, we all know that *true* beauty is actually found within. Beauty in tests is about simplicity; it’s about being able to understand what a test (and the system being tested) does with a mere glance at the code, even with little or no prior knowledge about the class or component they test; it’s about robustness, and not having to fix dozens of tests on every change; it’s about having fun both reading *and* writing the tests.

As you might expect, there will be a lot of code in the text that follows. And since I’m taking inspiration from Swift, which is written in C++, the examples in this text will be written in C++ as well. Using a language like Ruby or Python probably would have made the tests look more attractive, but I stand by my point that true beauty in tests goes beyond shallow looks.

XMPP 101

Before diving into the details of XMPP implementation testing, let’s first have a quick crash course about how XMPP works.

Figure 1. The decentralized architecture of the XMPP Network. Clients connect to servers from different domains, which in turn connect to each other.



The XMPP network consists of a series of interconnected servers with clients connecting to them, as shown in Figure 1. The job of XMPP is to route small “packets” of XML between these entities on the network. For example, Alice, who is connected to the `wonderland.lit` server, may want to send a message to her sister, who is connected to the `realworld.lit` server. To do that, she puts her message into a small snippet of XML:

```
<message from="alice@wonderland.lit/RabbitHole"
         to="sister@realworld.lit">
  <body>Hi there</body>
</message>
```

She then delivers this message to her server, which forwards it to the `realworld.lit` server, which in turn delivers it to her sister’s client.

Every entity on the XMPP network is addressed using a *Jabber ID* (JID). A JID has the form `username@domain/resource`, where `domain` is the domain name of the XMPP server, and `username` identifies an account on that server. One user can be connected to the server with multiple instances of a client; the `resource` part of the JID gives a unique name to every connected instance. In some cases, the resource part can be left out, which means the server can route the message to whichever connected instance it deems best.

The small packets of XML that are routed through the network are called *stanzas*, and fall into three categories: *message* stanzas, *presence* stanzas, and *info/query* stanzas. Each type of stanza is routed differently by servers, and handled differently by clients:

- *Message* stanzas provide a basic mechanism to get information from one entity to another. As the name implies, message stanzas are typically used to send (text) messages to each other.

- *Presence* stanzas “broadcast” information from one entity to many entities on the network. For example, Alice may want to notify all of her friends that she is currently not available for communication, so she sends out the following presence stanza:

  ```
  <presence from="alice@wonderland.lit/Home">
    <show>away</show>
    <status>Down the rabbit hole!</status>
  </presence>
  ```

  Her server then forwards this stanza to each of her contacts, informing them of Alice’s unavailability.

- *Info/query* (IQ) stanzas provide a mechanism for request/response interactions between entities, typically used to query or change information on a given entity. For example, Alice could be interested in knowing what client version her sister is using. She therefore sends the following stanza to her sister:

  ```
  <iq type="get" id="aad8a" ❶
      from="alice@wonderland.lit/RabbitHole" to="sister@realworld.lit/Home">
    <query xmlns="jabber:iq:version"/> ❷
  </iq>
  ```

  ❶ The unique identifier of the stanza is used to match an incoming IQ result to the original IQ request.
  ❷ An empty child element (or *payload*) in a specific namespace indicates what type of information is requested (in this case, software version information).

  Upon receiving this request, her sister’s client immediately responds with the name and version of her client software:

  ```
  <iq type="result" id="aad8a" ❶
      from="sister@realworld.lit/Home" to="alice@wonderland.lit/RabbitHole">
    <query xmlns="jabber:iq:version"> ❷
      <name>Swift</name>
      <version>1.0</version>
    </query>
  </iq>
  ```

  ❶ The `id` attribute of the response matches the one from the request.
  ❷ The response’s payload contains the result of the query.

Stanzas carry information in their *payloads*, which are added as child elements of the stanza. For example, a message can have a `body` payload containing the body text of the message. Different types of payloads are handled differently. By using XML namespaces for payloads, the XMPP protocol can easily be extended to support a virtually unlimited amount of information types, without having to worry about conflicting payload element names. For example, many of the early XMPP protocol extensions (including the software version protocol used in the examples) use the `query` payload. By using namespaces such as `jabber:iq:version`, entities know which type of protocol they are dealing with when they receive a `query` payload, and they know how to interpret the payload.

This section only scratched the surface of XMPP, just enough to get you through the rest of this text. If you want to learn more about how XMPP works and what you can do with it, have a look at *XMPP: The Definitive Guide* [XMPP-TDG], or visit `http://xmpp.org`.

Testing XMPP Protocols

One of the important aspects of an XMPP application, be it client or server, is the actual implementation of the XMPP *protocols*. Every XMPP implementation needs to at least implement the XMPP core protocols, as standardized by the IETF in [RFC 3920] and [RFC 3921]. These protocols provide the basic building blocks for XMPP, describing how an XMPP connection is set up, and what you can send over it. On top of the core protocols, the XMPP Standards Foundation created an ever-growing list of *XMPP Extension Protocols* (XEPs). These specifications describe how to extend the core protocol for very specific features, ranging from simple things such as requesting the software version of another client (standardized in [XEP-0092]), up to complex protocols for negotiating audio/video conference calls between clients, transferring files, and so on.

This text focuses on testing the functionality of XMPP protocol implementations, answering questions such as “Does my client correctly respond to incoming requests?”, “Does my client send the right requests at the right time?”, “Can my client handle this specific response on this request?”, and so on. We start out by looking at the most simple request-response protocols, after which we gradually move up the ladder to more complex protocols. While the complexity of the protocols increases, the level at which the tests are written becomes higher at well, moving from very specific unit tests up to full system tests. Although testing is mainly described from the perspective of a client developer, most of the approaches used here apply to server testing as well.

Unit Testing Simple Request-Response Protocols

Many of the XMPP protocols are simple: one side sends an IQ request, the other side receives the request, processes it, and responds with an IQ result. An example of such a simple request-response protocol is the software version protocol illustrated earlier. An implementation of this protocol consists of two parts:

- The *initiator* implementation sends a software version request and processes the corresponding response when it comes in.
- The *responder* listens for incoming software version requests and responds to them.

These implementations are typically implemented locally in one class for the initiator and one for the responder.[1] For example, Example 1 shows how a `VersionResponder` class is instantiated in a client to respond to incoming software version requests. All this class does is listen for an incoming IQ query of the type `jabber:iq:version`, and respond with the values set through `setVersion`. The class uses the central `XMPPClient` class to send data to and receive data from the XMPP connection.

[1] Some implementations are even known to put both responder and initiator implementations in one class. Don’t try this at home, kids!

Example 1. Using `VersionResponder` to listen and respond to software version requests.

```
class MyClient {
  MyClient() {
    xmppClient = new XMPPClient("alice@wonderland.lit", "mypass");
    versionResponder = new VersionResponder(xmppClient);
    versionResponder->setVersion("Swift", "0.1");
    xmppClient->connect();
  }
  …
};
```

Since the implementation of request-response protocols are local to one class, *unit testing* is a good way to test the functionality of the protocol implementation. So, let's see how we can unit test the `VersionResponder` class.

First, we need to make sure we can create an isolated instance of `Responder`. The only dependency the class has is the `XMPPClient`, a class that sets up and manages the XMPP connection. Setting up and managing a connection involves quite some work, and in turn brings in other dependencies, such as network interaction, authentication, encryption mechanisms, and so on. Luckily, all `VersionResponder` needs to be able to do is send and receive data from a data stream. It therefore only needs to depend on a `DataChannel` interface, which provides a method to send data and a signal to receive data, as shown in Example 2. This interface, implemented by `Client`, can be easily mocked in our unit test.

Example 2. Abstracting out data interaction in a `DataChannel` interface. The `XMPPClient` class is a concrete implementation of this interface.

```
class DataChannel {
  public:
    virtual void sendData(const string& data) = 0;
    boost::signal<void (const string&)> onDataReceived; ❶
};
```

❶ The signal `onDataReceived` has one `string` parameter (and no return value). When the signal is emitted, the `string` argument containing the data received will be passed to the connected slot method.

Now that we have all the ingredients for testing our `VersionResponder`, let's have a first attempt at writing a unit test. Example 3 shows how we can test the basic behavior of the responder, using a mock data channel to generate and catch incoming and outgoing data respectively.

Example 3. Testing `VersionResponder` using raw serialized XML data.

```
void VersionResponderTest::testHandleIncomingRequest() {
  // Set up the test fixture
  MockDataChannel dataChannel;
  VersionResponder responder(&dataChannel);
  responder.setVersion("Swift", "1.0");

  // Fake incoming request data on the data channel
  dataChannel.onDataReceived(
    "<iq type='get' from='alice@wonderland.lit/RabbitHole' id='version-1'>"
      "<query xmlns='jabber:iq:version'/>"
    "</iq>");

  // Verify the outcome
  ASSERT_EQUAL(
    "<iq type='result' to='alice@wonderland.lit/RabbitHole' id='version-1'>"
      "<query xmlns='jabber:iq:version'>"
        "<name>Swift</name>"
        "<version>1.0</version>"
      "</query>"
    "</iq>",
    dataChannel.sentData);
}
```

On first sight, this unit test doesn't look too bad: it's relatively short, easy to understand, structured according to the rules of unit testing style, and isolates testing of the protocol from the low-level network aspects of XMPP. However, the beauty of this test is only skin-deep, as the test turns out to be pretty *fragile*. To see this, we need to look at how XMPP implementations generate the response to a request.

Whenever an XMPP client generates an XML stanza, it typically constructs the XML of the stanza by building up a structured document (e.g., using a *Document Object Model* [DOM] API), and then *serializes* this document into a textual XML representation, which is then sent over the network. In Example 2, our test records exactly the serialized XML stanza generated by the responder being tested, and then compares it to a piece of XML that it expects. The problem with this approach is that the same XML element can be serialized in different correct ways. For example, we could have switched the order of the `from` and `type` attribute of the `<iq/>` element, and still have a logically equivalent stanza. This means that the smallest change to the way stanzas are serialized could break *all* tests.

One solution to avoid the fragility caused by XML serialization is to ensure that serialized stanzas are always in *Canonical XML* [XML-C14n] form. By normalizing away non-meaningful properties such as attribute order and whitespace, this subset of XML ensures that two equivalent XML stanzas can be compared in a stable way, thus solving the fragility of our tests. Unfortunately, since XMPP implementations typically use off-the-shelf XML implementations, they often have no control over how XML is serialized, and as such cannot make use of this trick to compare stanzas.

The solution most XMPP implementations take to verify responses is to check the structured DOM form of the response instead of comparing the serialized form. As shown in Example 4, this means that our `VersionResponder` no longer uses an interface to send raw data, but instead depends on a more structured `XMLElementChannel` interface to send and receive stanzas as XML element data structures.

Example 4. Testing `VersionResponder` using the structured XML representation. This test is no longer influenced by changes in the way the XML stanzas are serialized for transferring (e.g., different attribute order, extra whitespace, etc.).

```
void VersionResponderTest::testHandleIncomingRequest() {
  // Set up the test fixture
  MockXMLElementChannel xmlElementChannel;
  VersionResponder responder(&xmlElementChannel);
  responder.setVersion("Swift", "1.0");

  // Fake incoming request stanza on the stanza channel
  xmlElementChannel.onXMLElementReceived(XMLElement::fromString(
    "<iq type='get' from='alice@wonderland.lit/RabbitHole' id='version-1'>"
      "<query xmlns='jabber:iq:version'/>"
    "</iq>"));

  // Verify the outcome
  ASSERT_EQUAL(1, xmlElementChannel.sentXMLElements.size());
  XMLElement response = xmlElementChannel.sentXMLElements[0];
  ASSERT_EQUAL("iq", response.getTagName());
  ASSERT_EQUAL("result", response.getAttribute("type"));
  ASSERT_EQUAL("id", response.getAttribute("version-1"));
  ASSERT_EQUAL("alice@wonderland.lit/RabbitHole", response.getAttribute("to"));
  XMLElement queryElement = response.getElementByTagNameNS(
    "query", "jabber:iq:version");
  ASSERT(queryElement.isValid());
  XMLElement nameElement = queryElement.getElementByTagName("name");
  ASSERT(nameElement.isValid());
  ASSERT_EQUAL("Swift", nameElement.getText());
  XMLElement versionElement = queryElement.getElementByTagName("version");
  ASSERT(versionElement.isValid());
  ASSERT_EQUAL("1.0", versionElement.getText());
}
```

A downside of this test is that it is slightly less appealing than the one from Example 3. For this one test, the fact that the test has become less compact and readable is only a small price to pay. However, suppose now that we also want to test the case where the user didn't provide a version to the version responder, in which case we want to send back "Unknown version" as a version string. This test would in fact look exactly as Example 4, except that the call to `setVersion` will pass an empty string instead of `"1.0"`, and that the test would compare the version to `"Unknown version"`. Needless to say that this is a lot of duplicated code just to test a small difference in behavior, which will only get worse the more complex our protocol is (and hence the more tests it needs).

A first part of the duplication lays in checking whether the responder sends an `<iq/>` stanza of type `result`, whether it is addressed to the sender of the original stanza, and whether the identifier matches that of the request. This part can be easily factored out into a “generic” responder base class, and tested separately.

A second problem with our test is the fact that we need to analyze the structure of the XML to extract the values we want to test. The real underlying problem here is the fact that our tests are testing two things at once: the *logic* of the protocol (i.e., *what* it should respond), and the *representation* of the responses (i.e., *how* the request and response is represented in XML).

To separate the logic from the representation in our test, we adapt our `VersionResponder` to work on a high-level `IQ` data structure, which in turn contains high-level `Payload` data structures representing the payloads they carry. Using these abstract data structures, we can now focus on testing the `VersionResponder`’s functionality, without worrying about how the `IQ` and `Payload` data structures are actually represented in XML. The resulting test can be seen in Example 5.

Example 5. Testing the logic of `VersionResponder`. The actual (XML) representation of the stanzas sent and received by `VersionResponder` are no longer explicitly present in this test, making the test resistant against changes in representation.

```
void VersionResponderTest::testHandleIncomingRequest() {
  // Set up the test fixture
  MockIQChannel iqChannel;
  VersionResponder responder(&iqChannel);
  responder.setVersion("Swift");

  // Fake incoming request stanza on the stanza channel
  iqChannel.onIQReceived(IQ(IQ::Get, new VersionPayload()));

  // Verify the outcome
  ASSERT_EQUAL(1, iqChannel.sentIQs.size());
  const VersionPayload* payload =
    iqChannel.sentIQs[0].getPayload<VersionPayload>();
  ASSERT(payload);
  ASSERT_EQUAL("Swift", payload->getName());
  ASSERT_EQUAL("Unknown version", payload->getVersion());
}
```

The conversion from the `VersionPayload` structure to XML can now be tested independently, as illustrated in Example 6. Although this test still isn’t very attractive, the clutter coming from the representational part no longer impacts the tests for the more important behavioral part of the protocol.

Example 6. Testing the conversion of `VersionPayload` to XML.

```
void VersionPayloadSerializerTest::testSerialize() {
  // Set up the test fixture
  VersionPayloadSerializer serializer;
  VersionPayload payload;
  payload.setVersion("Swift", "1.0");

  // Serialize a payload
  XMLElement result = serializer.serialize(payload);

  // Verify the serialized element
  ASSERT_EQUAL("query", result.getTagName());
  ASSERT_EQUAL("jabber:iq:version", result.getNamespace());
  XMLElement* nameElement = queryElement->getElementsByTagName("name");
  ASSERT(nameElement);
  ASSERT_EQUAL("Swift", nameElement->getText());
  XMLElement* versionElement = queryElement->getElementsByTagName("version");
  ASSERT(versionElement);
  ASSERT_EQUAL("1.0", versionElement->getText());
}
```

In this section, we discussed how to test a simple IQ-based request/response protocol. In our first attempt, we tested the protocol at the lowest level possible, by analyzing the actual data sent over the wire. Subsequent versions tested the logic of the protocol at a higher, more structured level, up to the point where the logic of the responder was tested independently of the actual representation of the data sent over the network. Although it might seem overkill to separate the XML parsing and serializing from the actual data structure for a simple protocol like the one shown here, it makes testing the more complex (multistage) protocols from the next sections a lot cleaner.

Unit Testing Multistage Protocols

So far, the class of protocols we have considered were rather simple: one side sent out a request, the other side responded, and we were done. Although many of the XMPP protocols fall within this category, there are several others that consist of multiple iterations of these request-response cycles. These protocols start by doing a request, and then take subsequent steps based on the response of previous requests. Testing these types of protocols is the focus of this section.

Besides person-to-person conversations, XMPP also allows users to join chat “rooms” to communicate with multiple people at once. Whenever a user wants to join such a *Multi-user Chat* (*MUC* for short), an IM client needs to detect the MUC rooms that are available on a server, and present this list to the server. Obtaining this list requires a chain of multiple *service discovery* (often called *disco* in the XMPP world) requests. For example, let’s assume Alice wants to get a list of all the available rooms on the `wonderland.lit` server. She starts by requesting all the available services of her server, which is done by sending a `disco#items` request to the server:

```
<iq type="get" id="muc-1" to="wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
```

```
</iq>
```

The server then responds with the list of all its services:

```
<iq type="result" id="muc-1"
    from="wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#items">
    <item jid="pubsub.wonderland.lit" />
    <item jid="rooms.wonderland.lit" />
  </query>
</iq>
```

Alice now needs to determine which one of these services provides chat rooms. She therefore sends a `disco#info` request to each service, asking them which protocols they support:

```
<iq type="get" id="muc-2" to="pubsub.wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>

<iq type="get" id="muc-3" to="rooms.wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>
```

The first service responds:

```
<iq type="result" id="muc-2"
    from="pubsub.wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <feature var="http://jabber.org/protocol/pubsub" />
  </query>
</iq>
```

This service seems to support the only PubSub protocol (feature), which is not what Alice was looking for. The second service, however, responds with the following feature list:

```
<iq type="result" id="muc-3"
    from="rooms.wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <feature var="http://jabber.org/protocol/muc" />
  </query>
</iq>
```

Bingo! Now that she found the MUC service, all she needs to do is ask for the list of rooms, which is done using another `disco#items` request:

```
<iq type="get" id="muc-4" to="rooms.wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

This request results in the list of all the MUC rooms on the `rooms.wonderland.lit` server (in this case, a tea party and a room for discussing croquet):

```
<iq type="result" id="muc-4"
    from="rooms.wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#items">
```

```
      <item jid="teaparty@rooms.wonderland.lit" />
      <item jid="croquet@rooms.wonderland.lit" />
    </query>
  </iq>
```

As you can tell from this scenario, a lot of stanzas are going back and forth. Things become even more complex if you take into consideration that every step can result in an error response from the responding entity. Testing this protocol therefore involves multiple tests for determining whether our client can handle every type of response from the server, both successful and unsuccessful. Luckily, because of the high level at which we test our protocols, creating a test for one scenario can be very compact and straightforward. For example, a test for the “happy” error-less scenario described earlier is shown in Example 7.

Example 7. Testing `RoomDiscoverer`.

```
void RoomDiscovererTest::testDiscoverRooms() {
  // Set up the responses
  itemsResponses["wonderland.lit"] = ❶
    DiscoItems("pubsub.wonderland.lit", "rooms.wonderland.lit");
  infoResponses["pubsub.wonderland.lit"] = ❷
    DiscoInfo("http://jabber.org/protocol/pubsub");
  itemsResponses["pubsub.wonderland.lit"] = ❸
    DiscoItems("blogs@pubsub.wonderland.lit", "croquet@pubsub.wonderland.lit");
  infoResponses["rooms.wonderland.lit"] = ❹
    DiscoInfo("http://jabber.org/protocol/muc");
  itemsResponses["rooms.wonderland.lit"] = ❺
    DiscoItems("teaparty@rooms.wonderland.lit", "croquet@rooms.wonderland.lit");

  // Set up room discoverer
  RoomDiscoverer discoverer(channel);

  // Execute room discovery
  discoverer.discoverRooms();

  // Test results
  ASSERT(discoverer.isFinished());
  StringList rooms = discoverer.getDiscoveredRooms();
  ASSERT_EQUAL(2, rooms.size());
  ASSERT(rooms.contains("teaparty@rooms.wonderland.lit"));
  ASSERT(rooms.contains("croquet@rooms.wonderland.lit"));
}
```

❶ Specify the response to a `disco#items` request for the top-level `wonderland.lit` domain. In this case, two items are returned: `pubsub` and `rooms`.
❷ Specify the response to a `disco#info` request for the `pubsub` service. In this case, respond with the namespace of the PubSub protocol.
❸ Specify the items belonging to the `pubsub` service. These are added to test whether `RoomDiscoverer` doesn’t pick up items from non-MUC services.
❹ Respond that the `rooms` service supports MUC.
❺ Specify the list of items (i.e., rooms) of the MUC service.

The test specifies what responses should be sent for both `disco#info` and `disco#items` queries directed to specific JIDs. The `RoomDiscoverer` (which is the class that is responsible for discovering rooms) is then put in action, after which the test checks whether it indeed discovered both MUC rooms (and didn't accidentally include the PubSub service items). Not only is the test simple, but the auxiliary methods used by this test (including the fixture setup and tear down) can be kept very simple as well, as can be seen in Example 8.

Example 8. Setting up the `RoomDiscovererTest` fixture.

```
void RoomDiscovererTest::setUp() {
  channel = new MockIQChannel();
  channel->onSendIQ.connect(bind(&RoomDiscovererTest::respondToIQ, this, _1)); ❶
}

void RoomDiscovererTest::tearDown() {
  delete channel;
}

void RoomDiscovererTest::respondToIQ(const IQ& iq) {
  ASSERT(iq.getType() == IQ::Get);
  if (iq.getPayload<DiscoItems>()) {
    ItemsResponseMap::const_iterator response = itemsResponses.find(iq.getTo());
    ASSERT(response != itemsResponses.end());
    channel->onIQReceived(iq.createResponse(new DiscoItems(response->second)));
  }
  else if (iq.getPayload<DiscoInfo>()) {
    InfoResponseMap::const_iterator response = infoResponses.find(iq.getTo());
    ASSERT(response != infoResponses.end());
    channel->onIQReceived(iq.createResponse(new DiscoInfo(response->second)));
  }
  else {
    FAIL("Unexpected IQ");
  }
}
```

❶ Whenever an IQ is sent, pass it to `respondToIQ`, which will respond to it.

In this section, I showed how you can apply the high level of testing described in the previous section on more complex, multistage protocols. Because the tests aren't cluttered by low-level protocol representational details, the tests can focus on testing the actual logic of the protocol, allowing the number of tests to grow, without compromising the beauty of the protocol test suite.

Testing Session Initialization

By looking at both the single- and multistage request/response protocols from the previous sections, we covered most of the XMPP protocols out there. Although the level of testing for these protocols was already rather high, some protocols are still so complex that even testing at the level of "abstract" payloads results in too much clutter for a beautiful test. These are typically protocols that have a complex state diagram, and possibly even require user input during the process. We therefore bring in a higher level of testing: *scenario testing*.

One of the most complex protocols in XMPP is *session initialization*. Session initialization in an IM client involves creating a connection to the server, negotiating parameters of the connection (e.g., using stream compression for lower bandwidth consumption, encrypting the stream for better security, and so on), and finally authenticating with the server (typically involving sending a username and password to the server). Which parameters to negotiate with the server depends on what features the client and the server support, and also on the user preferences of the client. For example, a server might not support stream encryption; depending on whether the user has stated that he only wants to communicate over an encrypted connection, the client should either report an error or fall back on an unencrypted connection, respectively.

Testing all the possible code paths in session initialization requires a concise way of describing a session initialization scenario. Example 9 shows such a scenario test where the client encrypts the connection. By introducing helper methods describing what the client is supposed to send and what the server would send in response, we can clearly see how the encryption scenario is supposed to happen. It is easy to create scenarios for error conditions such as the server not supporting encryption (as shown in Example 10), and even to test the client's reaction to failing network connections (shown in Example 11). Moreover, creating these helper methods doesn't require all that much code, as they only involve setting expectations and responses on payloads, which can be written at the same level as the sections before.

Example 9. Testing session encryption negotiation.

```
void SessionTest::testStart_Encrypt() {
  Session* session = createSession("alice@wonderland.lit/RabbitHole");
  session->setEncryptConnection(Session::EncryptWhenAvailable);
  session->start();

  sessionOpensConnection();
  serverAcceptsConnection();
  sessionSendsStreamStart(); ❶
  serverSendsStreamStart(); ❷
  serverSendsStreamFeaturesWithStartTLS(); ❸
  sessionSendsStartTLS(); ❹
  serverSendsTLSProceed(); ❺

  ASSERT(session->isNegotiatingTLS());

  completeTLSHandshake(); ❻
  sessionSendsStreamStart(); /* (*) Immediately after the handshake, the
    stream is reset, and the stream header is resent in an encrypted form. */
  serverSendsStreamStart();

  ASSERT(session->isConnected());
  ASSERT(session->isEncrypted());
}
```

❶ Before sending XML elements over the stream, the client initializes the stream by sending an opening `<stream>` tag. All subsequent elements are children of this element. When the connection is closed, the closing `</stream>` tag is sent.

❷ Similar to the client, the server also starts the stream by sending a `<stream>` tag.

❸ Immediately after sending the opening stream tag, the server sends a list of all the features it supports. In this case, it announces support for stream encryption using *StartTLS*.

❹ The client sends a `<starttls/>` element to request the server to encrypt the connection.

❺ The server responds with a `<proceed/>`, indicating that the TLS negotiation (or *handshake*) can start.

❻ Fake a successful TLS handshake.

Example 10. Testing session failure due to the server not supporting encryption.

```
void SessionTest::testStart_ForceEncyptWithoutServerSupport() {
  Session* session = createSession("alice@wonderland.lit/RabbitHole");
  session->setEncryptConnection(Session::AlwaysEncrypt);
  session->start();

  sessionOpensConnection();
  serverAcceptsConnection();
  sessionSendsStreamStart();
  serverSendsStreamStart();
  serverSendsStreamFeaturesWithoutStartTLS();

  ASSERT(session->hasError());
}
```

Example 11. Testing session failure due to a failing connection.

```
void SessionTest::testStart_FailingConnection() {
  Session* session = createSession("alice@wonderland.lit/RabbitHole");
  session->start();

  sessionOpensConnection();
  serverAcceptsConnection();
  sessionSendsStreamStart();
  serverSendsStreamStart();
  closeConnection();

  ASSERT(session->hasError());
}
```

With scenario-based testing, it is possible to test the most complex class of protocols, covering all their corner cases. Although many of the corner cases of each “stage” in such protocols can be tested separately in isolation, scenarios are still needed to test the interaction between the multiple stages of the protocol.

Automated Interoperability Testing

By using unit tests to test our protocols in isolation (without a real network connection to an XMPP server), we were able to test all corner cases of a protocol while keeping our tests clean, simple, fast, and reliable. However, an XMPP client doesn’t live in isolation; its purpose is to eventually connect to a *real* XMPP server and talk to *real* clients. Testing an XMPP client in the real world is important for several reasons. First of all, it allows you to check the functionality of your application at a larger scale than the local unit testing, ensuring that all the components work together correctly. Second, by communicating with other XMPP protocol implementations, you can test whether your interpretation of the protocol specification is correct. Finally, by testing your client against many different XMPP implementations, you are able to ensure interoperability with a wide collection of XMPP software. Unless you are developing a dedicated client to connect to only one specific server, testing interoperability with other clients and servers is very important in an open, heterogeneous network such as XMPP.

Because IM clients are driven by a user interface, testing interoperability between two clients is typically done manually: both clients are started, they connect to a server, an operation is triggered through the user interface of one client, and the other client is checked to determine whether it responds correctly to the operation. Fully automating UI-driven features is very hard.

Testing client-to-server interoperability is somewhat easier than testing client-to-client communication. By creating a small headless test program on top of the client's XMPP protocol implementation, we can test whether the basic XMPP functionality of the backend works correctly, and even test complex protocols such as session initialization in action. For example, consider the test program in Example 12. This program logs into the server, fetches the user's contact list (also called *roster*), and returns successfully if it received the contact list from the server. By running this program, we can test whether most parts of our XMPP client's backend work: network connection, session initialization, stream compression, stream encryption, sending IQ requests, notifications of IQ responses, and so on.

Example 12. A test program to connect to a server and request the roster. The JID and password of the account are passed through the environment.

```
XMPPClient* xmppClient = NULL;
bool rosterReceived = false;

int main(int argc, char* argv[]) {
  xmppClient = new XMPPClient(JID(getenv("TEST_JID")), getenv("TEST_PASS"));
  xmppClient->onConnected.connect(&handleConnected); ❶
  xmppClient->connect();
  return rosterReceived;
}

void handleConnected() {
  GetRosterRequest* rosterRequest = new GetRosterRequest(xmppClient);
  rosterRequest->onResponse.connect(bind(&handleRosterResponse, _1, _2)); ❷
  rosterRequest->send();
}

void handleRosterResponse(RosterPayload*, optional<Error> error) {
  rosterReceived = !error; ❸
  xmppClient->disconnect();
}
```

❶ When connected (and authenticated), call `handleConnected`.
❷ When a response for the roster request is received, call `handleRosterReceived` with the response and status as arguments.
❸ If there was no error, we received the roster properly.

A program similar to the one from Example 12 is run as part of Swift's automated test suite. We use a few different server implementations on every run, passing the test JID and password of each server through the environment. If `ClientTest` fails due to a bug in a protocol implementation, a new unit test is added and the protocol is fixed. If the bug is due to a certain combination of protocols not being handled properly (either by the client or by the server), a scenario test is added to reproduce the scenario, after which either the client bug is fixed or the client is adapted to work around a specific server implementation bug.

When using automated tests like the one just described, a project is of course always limited to testing against the implementations it has access to. Although it is always possible to test against the handful of free XMPP server implementations out there, testing against implementations from commercial vendors isn't always straightforward. To make it easier for the XMPP community to test their implementations against each other, there has been an initiative to create a centralized place (`http://interop.xmpp.org`) that provides access to test accounts on all server implementations out there, including ones from commercial vendors.[2] This initiative paves the way to easy, automated interoperability testing for XMPP projects.

Diamond in the Rough: Testing XML Validity

When testing the functionality of our protocol in the previous sections, we separated the stanza representation from the actual logic of the protocol to improve the focus of our tests. This split made testing the logic of the protocol straightforward and clean. Unfortunately, testing the representational part that converts the abstract stanzas into XML and back is still tedious and error-prone. One of the things that needs to be checked is whether every variation of the payload transforms correctly to a standards-compliant XML element. For example, for the version payload we used earlier, we need to test the representation of a payload with and without a version number. Vice versa, the transformation from XML to a payload data structure needs to be tested for every possible compliant XML element. It would be handy if we could automatically check whether our XML parser and serializer handles all of the possible variations of payloads and stanzas allowed by the protocol standard.

A possible approach for testing XML parsers and serializers is automated *XML Validation*. Every protocol specification published by the XMPP Standards Foundation comes with an *XML Schema*. This schema describes the syntax and constraints of the XML used in the protocol. For example, it specifies the names of the XML elements that can occur in the payload, the names and types of the attributes of these elements, the number of times an element can occur, and so on. Such XML schemas are typically used to test whether a piece of XML is syntactically valid according to the rules specified in the schema. Unfortunately, the XMPP schemas currently serve only a descriptive purpose, and are only used to document the protocol. This is why using the XMPP schemas in automated processes (such as validity checking) is still mostly unexplored terrain. However, there has been interest lately in making normative XML schemas, which would open up some more possibilities for making the tedious testing of XMPP parsing and serialization more pleasant and, who knows, even beautiful!

[2]Unfortunately, this initiative has currently been put on hold for more urgent matters, but it will hopefully be revived soon.

Conclusion

In our quest to create beautiful tests for checking XMPP protocol implementations, we started out by testing simple request-response protocols at the lowest level: the data sent over the network stream. After discovering that this form of testing does not really scale well, we abstracted out the protocol to a higher level, up to the point where the tests used only high-level data structures. By testing protocol behavior on a high level, we were able to write tests for more complex protocols without compromising the clarity of the tests. For the most complex protocols, writing scenarios helped to cover all of the possible situations that can arise in a protocol session. Finally, since XMPP is an open protocol with many different implementations, it's very important to test an XMPP application on the real network, to ensure interoperability with other implementations. By running small test programs regularly, we were able to test the system in its entirety, and check whether our implementation of the protocol plays together nicely with other entities on the network.

The focus of the text has mostly been on testing the protocol functionality in XMPP implementations, as this is probably the most important part of quality control in XMPP. However, many other forms of testing exist in the XMPP world besides protocol tests. For example, performance testing is very crucial in the world of XMPP servers. Simple test scripts or programs like the ones described earlier can be used to generate a high load on the server, and to test whether the server can handle increasing amounts of traffic. In XMPP clients, on the other hand, testing the user interface's functionality is very important. Although automated UI testing is known to be hard, many complex parts, such as contact list representation, can be unit tested in isolation, which can avoid bugs in vital pieces of client code.

Although it's already possible to write simple, clean, and thorough tests for many aspects of XMPP, there's still a lot of beauty in testing waiting to be discovered. If you have suggestions or ideas, or want to help working on improving testing in the XMPP community, feel free to stop by `http://xmpp.org` and join the conversation!

Bibliography

[XMPP-TDG] *XMPP: The Definitive Guide*. Peter Saint-Andre. Kevin Smith. Remko Tronçon.

[RFC 3920] *Extensible Messaging and Presence Protocol: Core*. Peter Saint-Andre.

[RFC 3921] *Extensible Messaging and Presence Protocol: Instant Messaging and Presence*. Peter Saint-Andre.

[XEP-0092] *Software Version*. Peter Saint-Andre.

[DOM] *The W3C Document Object Model (DOM)*. Philippe Le Hégaret.

[XML-C14n] *Canonical XML*. John Boyer.

About the Author

Remko Tronçon is a member of the XMPP Standards Foundation's council, coauthor of several XMPP protocol extensions, and former lead developer of Psi, developer of the Swift

Jabber/XMPP project, and a coauthor of the book *XMPP: The Definitive Guide*. He holds a Ph.D. in engineering (computer science) from the Katholieke Universiteit Leuven. His blog can be found at `http://el-tramo.be`.

About this Article

The original version of this article appeared as a chapter in *Beautiful Testing*.

Copyright

This work is licensed under a Creative Commons Attribution 3.0 License.