4. Implementing Broker

4.1 Learning Objectives

In this chapter, I will detail the central challenges of implementing the *Broker* pattern using the TeleMed systems as a case. I will show the central pieces of the code that handle the challenges. You can find the full code base on FRDS.Broker Library at Bitbucket[1].

4.2 Architectural Concerns

The TeleMed system is architecturally simple, because there is only *one* remote object: the `TeleMedServant` object. This is a simplification that suits the pedagogy of the book: we *take small steps* and present the simple **Broker** implementation first, and only later in Chapter 5 go on into systems with many servant objects. However, it is not uncommon - most distributed systems have a kind of *single point of entry*. Think of services on world wide web - they are hosted on a single URL, like `www.amazon.com` or `www.facebook.com`, and all the internal objects and services are accessed via that single entry point.

Still, the remote servant object must be located on a specific machine on the network. The *Broker* implementation in the FRDS.Broker library rely on internet protocol (IP) and Domain Name System (DNS) to identify servers on the internet. I assume you have a working knowledge of this, so the location of the telemed server could be "telemed.baerbak.com:37111" - on port 37111 on the server with DNS name "telemed.baerbak.com". In the code base, the server defaults to "localhost".

4.3 Domain Layer

At the domain layer we need to implement the **Servant** and the **ClientProxy**, both implementing the **TeleMed** interface. The **TeleMed** interface represents a *Facade* to the tele medicine system.

[1] https://bitbucket.org/henrikbaerbak/broker

```
1  public interface TeleMed {
2
3    /**
4     * Process a tele observation into the HL7 format and store it
5     * in the XDS database tier.
6     *
7     * @param teleObs
8     *          the tele observation to process and store
9     * @return the id of the stored observation
10    * @throws IPCException in case of any IPC problems
11    */
12   String processAndStore(TeleObservation teleObs);
13
14   /**
15    * Retrieve all observations for the given time interval for the
16    * given patient.
17    *
18    * @param patientId
19    *          the ID of the patient to retrieve observations for
20    * @param interval
21    *          define the time interval that measurements are
22    *          wanted for
23    * @return list of all observations
24    * @throws IPCException in case of any IPC problems
25    */
26   List<TeleObservation> getObservationsFor(String patientId,
27       TimeInterval interval);
28 }
```

If you study the interface in the code base, you will find a few more method which I will return to in the REST Chapter.

Developing the **Servant** implementation represents a problem already dealt with in the chapter on Test-Driven Development (TDD) in my original book *Flexible, Reliable Software*[2]. You can find the implementation for `TeleMedServant` in the *telemed.server* package in the *telemed* project.

Next, I will detail the implementation for the client side and next the server side.

[2]Henrik B. Christensen, "Flexible, Reliable Software - Using Patterns and Agile Development", CRC Press, 2010.

4.4 Client Side

ClientProxy

The **ClientProxy** implementation (package *telemed.client* in project *telemed* in the code base) is more interesting, as it is at this level we start the process of converting the method calls to remote calls. Actually, this is quite straightforward as it simply needs to make a call to its associated **Requestor**, which is dependency injected through the constructor.

```
1  public class TeleMedProxy implements TeleMed, ClientProxy {
2
3    public static final String TELEMED_OBJECTID = "singleton";
4
5    private final Requestor requestor;
6
7    public TeleMedProxy(Requestor requestor) {
8      this.requestor = requestor;
9    }
10
11   @Override
12   public String processAndStore(TeleObservation teleObs) {
13     String uid =
14       requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID,
15         OperationNames.PROCESS_AND_STORE_OPERATION,
16         String.class, teleObs);
17     return uid;
18   }
19
20   @Override
21   public List<TeleObservation> getObservationsFor(String patientId,
22       TimeInterval interval) {
23     Type collectionType =
24       new TypeToken<List<TeleObservation>>(){}.getType();
25
26     List<TeleObservation> returnedList;
27     returnedList = requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID,
28               OperationNames.GET_OBSERVATIONS_FOR_OPERATION,
29               collectionType, patientId, interval);
30
31     return returnedList;
32   }
```

The requestor's `sendRequestAndAwaitReply()` is the library's equivalent to the `request()` method of the **Broker** pattern from the previous chapter, but note

that the `location` parameter is missing. As argued above, there is only one physical server computer involved, so this parameter is better provided once and for all instead of in every method call (in the code base, you will find that hostname and portnumber is provided to the **ClientRequestHandler** as constructor parameters.)

So, the remaining parameters are 'objectId', 'operationName', a type parameter, and finally 'arguments'. And - there is only one servant object, so the 'objectId' also becomes a dummy parameter: all incoming messages to the server will go to the server's single TeleMedServant object (also known as a **Singleton**). The actual method to be called is simply a named constant string defined in a **OperationNames** class. One example is:

```
1 public static final String PROCESS_AND_STORE_OPERATION =
2   "telemed-process-and-store";
```

That is, the way clients and servers agree on which method/operation to call is simply by a unique string. As I have done here, it is a good idea to convey both the object type "telemed" as well as the method name "process-and-store" to create a unique, easily identifiable, operation name. The **Invoker** on the server side will of course use this constant string to identify which method to call.

The next argument is the return type, like `String.class`. The requestor's implementation uses Java generic types to know the type of the return value. I will return to that in the next section.

Finally, the `arguments` or the parameter list. Here I pass each method parameter in the same order as on the parameter list. The ordering is important, as the **ClientProxy** and **Invoker** have to agree on in which order parameters appear in the marshalled message. So **Do the same thing, the same way** and respect the ordering.

Looking at the `getObservationsFor` method, you will note that the type for a list of tele observations is a bit of magic, and actually makes a binding to the marshalling library, Google Gson, used in the requestor code. This is an example that boundaries between the roles of the Broker are not easily upheld strictly. The code shown here is also truncated a bit compared to that in the real code base, as it has to handle exceptions when an unknown patient is encountered.

Note how "mechanical" this code is, it is repeating the same template with slight variations in parameters.

Requestor

The requestor's responsibility in turn is to marshall the method call, delegate to the client request handler for send/receive, and demarshall and return the

answer from the remote object.

```
public interface Requestor {
  <T> T sendRequestAndAwaitReply(String objectId,
          String operationName,
          Type typeOfReturnValue,
          Object... arguments);
}
```

I have chosen JSON as marshalling format as it is human readable which is important during the development process and in debugging situations, because it is relatively terse compared to XML, and because I can easily find high-quality JSON libraries to reuse. I have chosen to use the Gson library by Google.

(Fragment in *broker* project: StandardJSONRequestor.java)

```
@Override
public <T> T sendRequestAndAwaitReply(String objectId,
                                      String operationName,
                                      Type typeOfReturnValue,
                                      Object... arguments) {
  // Perform marshalling
  String marshalledArgumentList = gson.toJson(arguments);
  RequestObject request =
          new RequestObject(objectId, operationName, marshalledArgumentList);
  String marshalledRequest = gson.toJson(request);

  // Ask CRH to do the network call
  String marshalledReply =
          clientRequestHandler.sendToServerAndAwaitReply(marshalledRequest);

  // Demarshall reply
  ReplyObject reply = gson.fromJson(marshalledReply, ReplyObject.class);

  // First, verify that the request succeeded
  if (!reply.isSuccess()) {
    throw new IPCException(reply.getStatusCode(),
        "Failure during client requesting operation '"
                + operationName
                + "'. ErrorMessage is: "
                + reply.errorDescription());
  }
  // No errors - so get the payload of the reply
  String payload = reply.getPayload();

```

```
30    // and demarshall the returned value
31    T returnValue = null;
32    if (typeOfReturnValue != null)
33      returnValue = gson.fromJson(payload, typeOfReturnValue);
34    return returnValue;
35  }
```

Note that no TeleMed domain specific roles or objects are used. Thus this is a general purpose implementation that may serve other domains, and I have therefore put it into the *frds.broker* package.

The requestor needs an instance of a **ClientRequestHandler** to make the actual network `send()` and blocking `reply()` calls, and this instance is again constructor injected, as you can see in the constructor declaration.

Inside the `sendRequestAndAwaitReply` method I do the marshalling (lines 7–10). This is done in two steps: first the 'arguments' are marshalled making a JSON array, and next I populate a record type class, `RequestObject`, that just stores the 'objectId', 'operationName', and the JSON array. Finally, the request object is marshalled to produce a JSON string emboding all provided arguments.

Next, I ask the client request handler to send to the server (lines 13–14) and await its reply. Upon reception, it is demarshalled into a `ReplyObject` (line 17), again a simple record type class that only contains a status code and the reply JSON message. The status code is used to report exceptions and error conditions from the **Invoker** on the server side. So, the status code is first tested whether any failures happened on the server which will cause an exception to be thrown (lines 20–26). Next, the actual return value is demarshalled and returned to the client proxy (lines 28 onwards).

RequestObject and ReplyObject

Marshalling and demarshalling is about converting domain objects to byte arrays and back again. However, the **Broker** needs to tack some information along the method calls, like the object id, method/operation name, and the reply needs to convey more than just the return value of the method call, for instance exceptions thrown or error codes.

This is the reason for the two additional roles introduced **RequestObject** and **ReplyObject** in the code above. They are convenience classes of the record/struct/PODO (plain old data type) type: they only have accessor methods and just stores information. They make using marshalling libraries like Gson easier.

ClientRequestHandler

Finally, the **ClientRequestHandler** needs to bind to the operating system and encode a protocol for network communication with the server. In the package *frs.broker.ipc.socket* you will find class `SocketClientRequestHandler` that uses Java socket programming for this. Note that this code is also general and not tied to the TeleMed system itself. Sockets are basically IO streams, so I can write the request object to an output stream (sends the request byte array to the server), and next read the reply from the input stream (receive the reply byte array from the server). The core of this is shown here:

(Fragment in *broker* package: SocketClientRequestHandler.java)

```
1  // Send it to the server (= write it to the socket stream)
2  out.println(request);
3
4  // Block until a reply is received
5  String reply;
6  try {
7    reply = in.readLine();
8
9  } catch (IOException e) {
```

where 'out' is a output IO stream and 'in' is the input stream.

If you review the full code, you will note that I use String instead of byte[] in the code base. Strings are conceptually similar to byte arrays, and easier to handle in the code base and when testing and debugging.

So, I have the three roles implemented and only need to inject the dependencies. In the package *telemed.main* in the *telemed* subproject of the Broker code, you can find client programs that simulate uploading blood pressure and reading values for the last week for a given patient. The core configuration code that injects the dependencies can be found in **HomeClientSocket** and **HomeClientHTTP** and follows the template:

```
1    ClientRequestHandler clientRequestHandler
2         = new SocketClientRequestHandler(hostname, port);
3      Requestor requestor
4         = new StandardJSONRequestor(clientRequestHandler);
5
6      TeleMed ts = new TeleMedProxy(requestor);
```

Note that the location of the server, 'hostname' and 'port', is simply given as parameters to the client request handler.

4.5 Server side

ServerRequestHandler

At the server side, the **ServerRequestHandler** obviously must also bind to the operating system and understand the protocol used by the client request handler.

A socket based implementation that matches the **SocketClientRequestHandler** can be found in package `frs.broker.ipc.socket`. The implementation is rather long, with setting up the socket and error handling taking quite a lot of code, but the central message receiving and processing is shown below

(Fragment in *broker* project: SocketServerRequestHandler.java)

```
1  private void readMessageAndDispatch(Socket clientSocket)
2          throws IOException {
3    PrintWriter out =
4            new PrintWriter(clientSocket.getOutputStream(), true);
5    BufferedReader in = new BufferedReader(new InputStreamReader(
6        clientSocket.getInputStream()));
7
8    String inputLine;
9    String marshalledReply = null;
10
11   inputLine = in.readLine();
12   System.out.println("--> Received " + inputLine);
13   if (inputLine == null) {
14     System.err.println(
15             "Server read a null string from the socket???");
16   } else {
17      marshalledReply = invoker.handleRequest(inputLine);
18
19     System.out.println("--< replied: " + marshalledReply);
20   }
21   out.println(marshalledReply);
22
23   System.out.println("Closing socket...");
24   in.close();
25   out.close();
26 }
```

The basic algorithm is simple: receive the message from the network (line 11), pass it on to the invoker (lines 17) and finally send the reply back to the client (line 21). This socket server is rather verbose for the sake of demonstration.

Invoker

The **Invoker** in turn is responsible for demarshalling the message, determining the right object and method to invoke, call it, and marshalling the return value into a reply. As such, the **Invoker** code is always specific to the domain, and must close match the marshalling format defined in the **Requestor**.

(Fragment in *telemed* package: TeleMedJSONInvoker.java):

```
1  public String handleRequest(String request) {
2    // Do the demarshalling
3    RequestObject requestObject =
4            gson.fromJson(request, RequestObject.class);
5    JsonArray array =
6            JsonParser.parseString(requestObject.getPayload())
7              .getAsJsonArray();
8
9    ReplyObject reply;
10
11   /* As there is only one TeleMed instance (a singleton)
12      the objectId is not used for anything in our case.
13    */
14   try {
15     // Dispatching on all known operations
16     // Each dispatch follows the same algorithm
17     // a) retrieve parameters from json array (if any)
18     // b) invoke servant method
19     // c) populate a reply object with return values
20
21     if (requestObject.getOperationName().equals(OperationNames.
22             PROCESS_AND_STORE_OPERATION)) {
23       // Parameter convention: [0] = TeleObservation
24       TeleObservation ts = gson.fromJson(array.get(0),
25               TeleObservation.class);
26
27       String uid = teleMed.processAndStore(ts);
28       reply = new ReplyObject(HttpServletResponse.SC_CREATED,
29               gson.toJson(uid));
30
31     } else if (requestObject.getOperationName().equals(OperationNames.
32             GET_OBSERVATIONS_FOR_OPERATION)) {
33       // Parameter convention: [0] = patientId
34       String patientId = gson.fromJson(array.get(0), String.class);
35       // Parameter convention: [1] = time interval
36       TimeInterval interval = gson.fromJson(array.get(1),
37               TimeInterval.class);
```

```
38
39      List<TeleObservation> tol =
40              teleMed.getObservationsFor(patientId, interval);
41      int statusCode =
42              (tol == null || tol.size() == 0) ?
43                      HttpServletResponse.SC_NOT_FOUND :
44                      HttpServletResponse.SC_OK;
45      reply = new ReplyObject(statusCode, gson.toJson(tol));
46
47    } else if (requestObject.getOperationName().equals(OperationNames.
48          CORRECT_OPERATION)) {
49      [... handling of other methods removed]
```

The algorithm to determine which object and method to invoke is often termed the **dispatcher** and is sometimes expressed as a role in its own right (Buschmann et al. 2007[3]). Here I have taken the simplest possible approach, namely to make a state machine using a long-winding if-statement. This does not scale well but suffices for our small TeleMed system. In a following chapter, I will discuss and implement the dispatching for multi-object systems.

Actually the `handleRequest` method needs to demarshall in two phases. The first is to get the `objectId`, the `operationName` and the `arguments` as a JSON array (lines 3-7). Upon this first demarshalling, the algorithm has the information to determine which method to call on which object. Note that the 'objectId' is not used anywhere: There is one and only one servant object, 'teleMed', which is created and dependency injected into the **Invoker** in the server's `main()` method. Thus, the TELEMED_OBJECTID on the client side could have been any string.

Thus, it is only the `operationName` that is used to determine which method the client side invoked. For each operation, a further demarshalling of the JSON array must be executed, to get the original parameters. For instance the `TeleObservation` as the first parameter to the `processAndStore` method (lines 24-25).

Again, the algorithm for handling each method is "mechanical": demarshall the provided arguments , do the "upcall" to the servant object, and marshall the return value and error code into a reply (shown below).

Replies from the server need to communicate exceptions happening on the server, and can only do so using marshalled error codes - exceptions cannot by themselves cross machine boundaries. Instead of inventing my own error code system, it is much better to reuse an established one, and I have adopted the HTTP error codes. These are standardized and well described (I will detail them later in Chapter HTTP), and I can reuse constants defined in

[3]Frank Buschmann, Kevin Henney, and Douglas C. Schmidt, "Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing Volume 4 Edition", Wiley, 2007.

common Java HTTP libraries. The code fragment below (which is the end of the 'handleRequest' method) shows how exceptions happening on the server side are caught by the invoker, and converted into HTTP error codes in the reply object.

```
  } else {
      // Unknown operation
      reply = new ReplyObject(HttpServletResponse.
              SC_NOT_IMPLEMENTED,
              "Server received unknown operation name: '"
                      + requestObject.getOperationName() + "'.");
    }

  } catch( XDSException e ) {
    reply =
            new ReplyObject(
                    HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
                    e.getMessage());
  }

  // And marshall the reply
  return gson.toJson(reply);
}
```

The fragment above handles two situations: the first is if the 'operationName' was not known (seems to indicate some kind of marshalling error), and second in case the XDS threw an exception.

So the final piece of the puzzle is the dependency injection and coupling of delegates in the server: Create the Servant, inject it into the Invoker, and finally inject that into the ServerRequestHandler. You will find main programs for the TeleMed server also in package *telemed.main*, below is the configuration fragment from the class `ServerMainSocket`(the binding to the XDS layer has been left out for clarity.)

```
[XDS creation code removed]
TeleMed tsServant = new TeleMedServant(xds);
Invoker invoker = new TeleMedJSONInvoker(tsServant);

// Configure a socket based server request handler
SocketServerRequestHandler ssrh =
        new SocketServerRequestHandler();
ssrh.setPortAndInvoker(port, invoker);

ssrh.start();
```

4.6 Test-driven development of Distributed Systems

I actually used the TeleMed case to develop both the TeleMed case code *as well* as the general implementations of the broker roles, like e.g. the Standard-JSONInvoker, and it was mostly done using test-driven development and automated testing. How? By using test stubs and test doubles (See Sidebar 12.4 in "Flexible, Reliable Software") to *keep focus* on one particular role and associated delegate implementation at a time, so I could *take small steps*—and most importantly: avoid using real network communication.

The overall process was:

- Always get the domain code in place first. Thus I started my TDD process by test-driving the *TeleMedServant* implementation first. After all, if your domain code does not work correctly at the onset, you may easily waste time tracking defects in the large set of code making up the broker delegates only to find a defect in the domain code.
- Next I turn my attention to the client side proxy, and to TDD that I use a test spy as delegate to play the **Requestor** role. These tests I normally do not make comprehensively as the proxy code will be tested intensively later in the process, but they serve the purpose of driving parts of the client proxy code into existence.
- A similar approach can be taken with the **Invoker** role.
- While developing the broker delegates, I simply avoid the IPC layer completely; it is the last thing I turn my attention to. I can do that by introducing a Fake Object **ClientRequestHandler** which calls the Invoker directly.
- Finally, it of course *is* important to do *integration testing* in which the full IPC layer is used.

I will cover aspects of this process below.

It should be noted that developing Broker based distributed systems is *not* just developing a fully mature domain code base first, and then *just add distribution*. More than once in my process of developing the Broker role implementations, I ran into insights that forced me to make minor modifications to the domain code. As an example, a constant source of modification is marshalling of domain objects: marshalling libraries require the existence of a no-argument constructor (the JavaBean standard). When developing the domain code, they are not necessary, but once you have to use them as value objects to be passed between client and server, I had to add these.

Requestor Spy

To help develop the client proxy I use a test spy for the **Requestor** role. Remember that a test spy will record the method calls it receives (and nothing else), so they can later be asserted. In this case I inject a `SpyRequestor` into the `TeleMedProxy`, call the proxy's methods and assert that the requestor gets called with the proper parameters.

```
1  public class TestTeleMedProxy {
2
3    private SpyRequestor requestor;
4    private TeleMed telemed;
5
6    @Before
7    public void setup() {
8      requestor = new SpyRequestor();
9      telemed = new TeleMedProxy(requestor);
10   }
11
12   @Test
13   public void shouldValidateRequestObjectCreated() {
14     // Given a tele observation
15     TeleObservation teleObs1 =
16         HelperMethods.createObservation120over70forNancy();
17     // When the client stores it
18     telemed.processAndStore(teleObs1);
19
20     // Then validate that the proper operation and object id was
21     // provided to the requestor
22     assertThat(requestor.lastOperationName,
23         is(OperationNames.PROCESS_AND_STORE_OPERATION));
24     assertThat(requestor.lastObjectId, is(TeleMedProxy.TELEMED_OBJECTID));
25     // Testing the arguments and the type is tricky, but they will be
26     // covered intensively by other tests later
27
28     [...]
```

The *SpyRequestor* has "spy methods" that allow me to inspect what parameters were passed to its method. Or, rather, as the code above shows, I have simply made the instance variables, like `lastOperationName`, public, so I can read them directly. A spy is just for testing purposes, not part of the production code, so I have been a bit lazy about adhering strictly to good programming doctrine...

Fake Object Request Handlers

A particularly tricky aspect of test-driven development of *Broker* is necessarily the IPC part. First, starting and stopping a server for each individual test case is much slower than normal unit tests. Second, it is also often problematic due to concurrency race conditions in the OS. One such issue is on Linux that spends several seconds to release a port connection, thus if one passed test case has shut down my TeleMed server on port 4567, then when the next test case starts a server on the same port it simply fails with a `java.net.BindException: Address already in use`, as the OS still thinks there is a live connection. Third, the IPC between a client and a server is arguably more in the realm of *integration testing* than in *unit testing*.

So the question is, if there is a way, that you can avoid the IPC layer while you are just getting the other aspects in place: marshalling and demarshalling, object fetch and method dispatch in the invoker, etc.

Actually, this is easy if your architectural toolbox includes *test doubles*! The client and server request handlers are just delegates to get messages from the client to the server and back again, and we can thus replace them with *fake object* test double implementations: Instead of the ClientRequestHandler sending messages to a ServerRequestHandler which in turn invokes the **Invoker**'s 'handleRequest()' method — I can simply make a fake object **ClientRequestHandler** that calls my server side **Invoker** directly.

From the test case, that verifies TeleMed's user stories (`TestStory1.java` in the *TeleMed* project), this is the @Before method which creates the production delegates for servant, invoker, requestor and client proxy roles while using fake object implementations for the client request handler (lines 13-14).

```
1  @Before
2  public void setup() {
3    // Given a tele observation
4    teleObs1 = HelperMethods.createObservation120over70forNancy();
5    // Given a TeleMed servant
6    xds = new FakeObjectXDSDatabase();
7    TeleMed teleMedServant = new TeleMedServant(xds);
8    // Given a server side invoker associated with the servant object
9    Invoker invoker = new TeleMedJSONInvoker(teleMedServant);
10
11   // And given the client side broker implementations, using the local
12   // method client request handler to avoid any real IPC layer.
13   ClientRequestHandler clientRequestHandler =
14       new LocalMethodCallClientRequestHandler(invoker);
15   Requestor requestor =
16       new StandardJSONRequestor(clientRequestHandler);
17
```

```
18    // Then it is Given that we can create a client proxy
19    // that voids any real IPC communication
20    teleMed = new TeleMedProxy(requestor);
21  }
```

As both client and server "live" in the test case, all broker roles are created and configured here. Note that there is no need for a server request handler—instead the client request handler is doubled by an implementation that simply uses local method calls to send the marshalled message right into the server side invoker.

```
1  public class LocalMethodCallClientRequestHandler
2    implements ClientRequestHandler {
3
4    private final Invoker invoker;
5    private String lastRequest;
6    private String lastReply;
7
8    public LocalMethodCallClientRequestHandler(Invoker invoker) {
9      this.invoker = invoker;
10   }
11
12   @Override
13   public String sendToServerAndAwaitReply(String request) {
14     lastRequest = request;
15     String reply = invoker.handleRequest(request);
16     lastReply = reply;
17     return reply;
18   }
```

This allows fast and simple unit testing and development as there are no threads nor any servers that need to be started and shut down. The flip side of the coin is of course, that no testing of the real IPC implementations are made. So, this is the next issue.

Integration Testing IPC

From a reliability perspective, it of course also recommendable to have the IPC layer under automated test control. However, as the *ClientRequestHandler* and *ServerRequestHandler* are necessarily in separate threads, and as both rely on the underlying OS, these tests are subject to less control from JUnit and are thus more brittle. Indeed I have experienced that some of the test cases described below have passed flawlessly using Gradle in the commandline but the very same tests fail when run from within IntelliJ - for no obvious reason. And, these tests are integration tests which are out of the scope of

the present book. Still, the Broker code base contains some integration tests so a short presentation is provided here.

The IPC tests are grouped in the package `telemed.ipc`. A simple test of the socket based CRH and SRH is shown below.

```
@Test
public void shouldVerifySocketIPC() throws InterruptedException {
  // Given a socket based server request handler
  final int portToUse = 37111;
  Invoker invoker = this; // A self-shunt spy
  ServerRequestHandler srh = new SocketServerRequestHandler();
  srh.setPortAndInvoker(portToUse, invoker);
  srh.start();
  // Wait for OS to open the port
  Thread.sleep(500);

  // Given a client request handler
  ClientRequestHandler crh = new SocketClientRequestHandler();
  crh.setServer("localhost", portToUse);

  // When we use the CRH to send a request object to
  // the external socket handler
  RequestObject req = new RequestObject(OBJECT_ID,
      CLASS_FOO_METHOD, MARSHALLED_PAYLOAD);
  ReplyObject reply = crh.sendToServer(req);

  // Then our test spy has indeed recorded the request
  assertThat(lastObjectId, is(OBJECT_ID));
  assertThat(lastOperationName, is(CLASS_FOO_METHOD));
  assertThat(lastPayLoad, is(MARSHALLED_PAYLOAD));

  // Then the reply returned is correct
  assertThat(reply.getStatusCode(),
             is(HttpServletResponse.SC_ACCEPTED));
  assertThat(reply.isSuccess(), is(true));
  assertThat(reply.getPayload(), is(MARSHALLED_REPLY_OBJECT));

  crh.close();
  srh.stop();
}
```

The overall template of the test case is to create first a SRH and next a CRH on the same port on `localhost`. As I only want to verify the communication across the CRH and SRH implementations, the Invoker role is not really interesting and I replace it with a simple Test Spy whose `handleRequest` method just records the parameters and returns a dummy reply:

```
@Override
public String handleRequest(String request) {
  RequestObject requestObj = gson.fromJson(request, RequestObject.class);
  this.lastObjectId = requestObj.getObjectId();
  this.lastOperationName = requestObj.getOperationName();
  this.lastPayLoad = requestObj.getPayload();
  ReplyObject reply = new ReplyObject(HttpServletResponse.SC_ACCEPTED,
    MARSHALLED_REPLY_OBJECT);
  return gson.toJson(reply);
}
```

So the test just sends a dummy request containing some constant values, and validate that A) the Invoker test spy received the proper request object and B) the reply contained the proper values.

The test spy is a *self-shunt* [4], that is, the test case class itself plays the role of the test spy. This is the reason for the line

```
  Invoker invoker = this; // A self-shunt spy
```

in the test case above.

4.7 Using the Broker library

The FRDS.Broker is also available as a Maven Repository library BinTray FRDS.Broker Library[5]. This library can be included in your projects, and provides all the core interfaces and general implementation from the *frds.broker* package.

To include in your Gradle project, just add the dependency

```
dependencies {
  implementation 'com.baerbak.maven:broker:2.0'
}
```

to your 'build.gradle' file.

For your given use case, you need to develop the **ClientProxy** classes as well as **Invoker** implementation(s). The **Requestor**, **ClientRequestHandler**, and **ServerRequestHandler** imlementations of the library can just be reused - or rewritten for your convenience. For instance, I have written CRH and SRH implementations that use Rabbit MQ[6] as IPC layer which provides a lot of benefits such as load balancing and increased architectural availability.

[4] Meszaros, Gerard, "xUnit Test Patterns - Refactoring Test Code", Addison Wesley, 2007.
[5] https://bintray.com/henrikbaerbak/maven/broker
[6] https://www.rabbitmq.com/

4.8 Summary of Key Concepts

It is possible to use *test-driven development* (TDD) to develop most of a *Broker* pattern based implementation of the TeleMed system. The key insight is to use a *fake object* implementation of network layer - doable by making **ClientRequestHandler** that simply calls the server side **Invoker** directly using a local method call. Then only the network implementations need to be under manual test, or tested by integration tests.

The implementation effort of TeleMed showed that of the six roles in the Broker pattern, actually only two of them: **CientProxy** and **Invoker**; contains TeleMed specific code. The remaining four roles are general purpose implementations for the specific choice of marshalling format and network transmission implementation. In the *FRDS.Broker* project, the *frds.broker* package contains implementations of the **Requestor** using JSON as marshalling format, and implementions of the **ClientRequestHandler** and **ServerRequestHandler** using sockets and HTTP libraries. The HTTP implementation will be discussed in the later HTTP chapter.