USAGE AND OPERATION OF WIRELESS SENSOR NETWORK TESTBEDS

Daniel Bimschas
Universität zu Lübeck
Lübeck, Germany
bimschas@itm.uni-luebeck.de

Dennis Pfisterer
Universität zu Lübeck
Lübeck, Germany
pfisterer@itm.uni-luebeck.de

1. Introduction

Recently, experimentally-driven research has become an instrumental tool in designing and optimizing novel networking applications. While simulations are still important tools, they suffer from several imperfections as they make artificial assumptions on radio propagation, traffic, failure patterns, and topologies. Especially in the domain of wireless sensor networks, which are embedded into the environment, applications strongly depend on real-world processes that are often a result of complex interactions and are extremely difficult to model accurately. In order to design robust applications, developers need appropriate tools and methods for testing and managing their applications on real hardware in large-scale deployments. Such tools have been developed by the EU-project WISEBED [Sev08], which provides methods to cope with implementing protocols and applications for heterogeneous networks (cf. Chapter 2.0) as well as an ecosystem of testbeds and accompanying software for conducting experiments.

This chapter introduces the latter and describes requirements for testbeds and WISEBED's architecture (cf. Section 9.2), how to run experiments (cf. Section 9.3), and briefly how to operate testbeds using the WISEBED software (cf. Section 9.4). Section 9.5 concludes this chapter.

2. Requirements and WISEBED Architecture

To realize adequate facilities for research and experimentation several requirements must be identified and corresponding challenges must be met. *Heterogeneity* and *scale* are important to reflect real-world challenges within testbeds. Conducting experiments at large-scale in heterogeneous environments may yield very different results than at small-scale and in homogeneous environments [Exs10]. Deploying a large-scale testbed is costly and requires adequate resources to manage, maintain, and operate it. As this typically overburdens the resources of single institutions, *testbed federation* is vital to achieve large-scale facilities. Another important aspect is collecting performance traces from the testbeds without disturbing running experiments. This requires means for the *out-of-band collection of data*. Out-of-band usually means that devices are connected to a wired infrastructure that forward data to users. Similarly, *out-of-band experiment control* allows users to interact with running experiments, e.g., to explore the parameter space or to influence experiments by sending control messages and data to sensor nodes. While simulations can easily be repeated with predicable outcomes, *reproducibility* is hard to achieve in real-world deployments due to unpredictable interference on the wireless media or external stimuli such as measurements. However, the ability to *repeat* experiments helps to reach statistical soundness.

WISEBED addresses the aforementioned challenges using different means. It provides a generic architecture that inherently supports the federation of multiple physically distributed testbeds over the internet. Each individual testbed can be accessed using a well-defined Web Service API. A so-called *federator* component federates

multiple testbeds and exposes the same API towards clients. This way, a single testbed and a federated one behave identical from a user's point of view. Hence, a set of testbeds providing heterogeneous hardware can be federated to form a single, unifying experimental facility of large scale with heterogeneous hardware. The well-defined API allows clients to access testbeds using the same clients while the internal realization of individual testbeds may be quite different. As a result, an ecosystem of (open-source) clients has evolved that support users in conducting experiments. While single-shot experiments could be launched using web-based clients, more advanced experiment series could be conducted automatically by using scriptable testbed clients.

Using a testbed consists of a three-step procedure: i) *authenticate*, ii) *reserve*, and iii) *experiment*. As depicted in Figure 1, users interact with three different sub-APIs: The SNAA API (Sensor Network Authentication and Authorization API) for authentication, the RS API (Reservation System API) for reservation, and the iWSN API for running experiments.

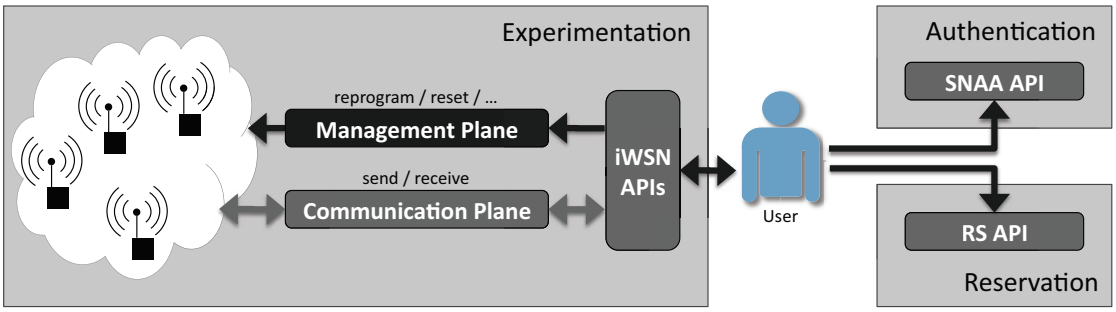

Figure 1: WISEBED Testbed Planes

The iWSN API is split into two planes, the *management plane* and the *communication plane*. The management plane provides functionalities for managing experiments (such as reprogramming and resetting nodes or checking their status) while the communication plane support sending data to nodes and receiving data emitted by nodes via the serial or USB interface. A testbed delivers the output of nodes to clients by invoking a Web Service on the client. Hence, the client must expose a simple service, which implements the *Controller API*. A more detailed description of the several APIs is given in Section 9.3.

In the following, the architecture of single and federated WISEBED testbeds is described (cf. Section 9.2.1). Then, we introduce WISEBEDs concept of describing and addressing individual sensor nodes inside a testbed in Section 9.2.2.

2.1 Testbed Architecture

Figure 2 shows the architecture of single and federated testbeds. Each testbed comprises a so-called *Testbed Server* (TS) that runs the WISEBED APIs (SNAA, RS, iWSN) to expose the testbed's features to clients. At least one sensor node is connected to a wired infrastructure (i.e., to the Testbed Server or to a *gateway* via a serial USB connection). If all nodes are connected to the wired infrastructure, we refer to such a testbed as *wired*. Otherwise, the testbed is called *wireless* as at least one node is *not* connected to a wired infrastructure. Hence, communication with these devices and reprogramming them using an over-the-air programming (OTAP) mechanism is done wirelessly. Depending on the actual implementation, this could mean that trace collection interferes with the running experiment.

Multiple testbeds are federated using a *federator component*. A federation is comprised of a set of testbeds that expose their functionalities as WISEBED-compliant Web Services. To interconnect spatially divided testbeds, WISEBED has developed the concept of *Virtual Links* [BCD+10]. Virtual Links emulate broadcast connectivity between arbitrary nodes by tunneling messages between communicating nodes using the WISEBED APIs.

As shown in the figure, this includes real testbeds as well as simulated networks. For instance, the Shawn network simulator [FKFP07] supports the WISEBED APIs and can hence be part of a federated testbed. Clients can then either connect directly to a single testbed or to the federation using exactly the same software.

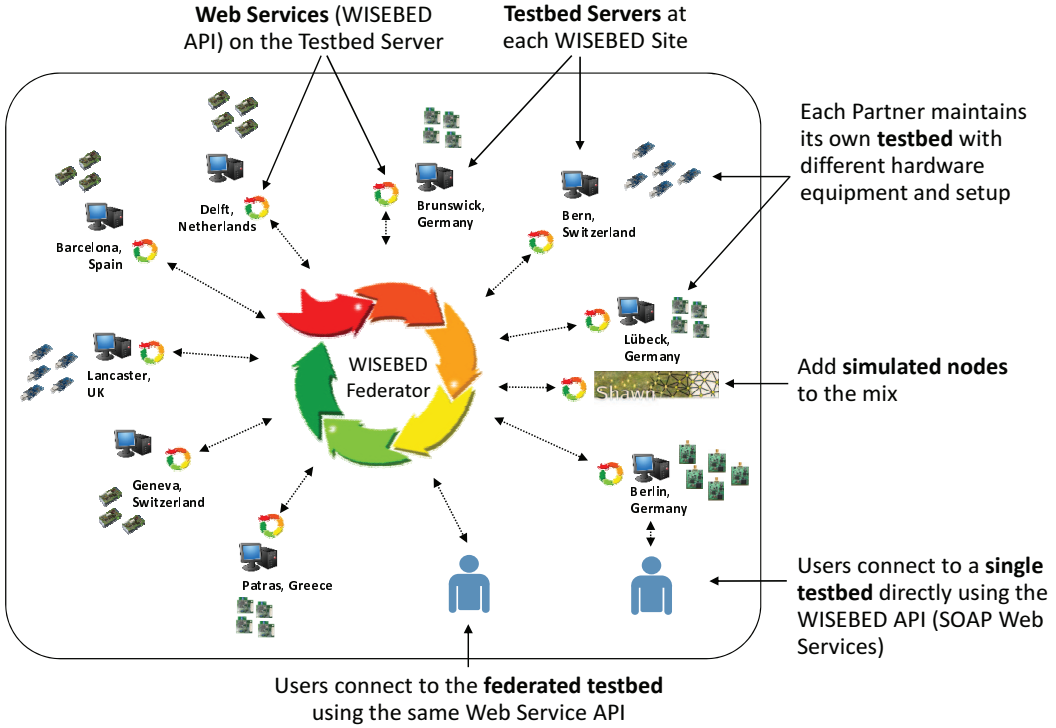

Figure 2: WISEBED Federation Architecture

2.2 Testbed Description and Node Addressing

For clients, it is important to know which nodes are available in a testbed, of what type they are, and what sensors they feature. In WISEBED, each testbed is described using an XML format called WiseML [The09] that contains metadata about a testbed and its nodes. The iWSN API provides a function (`getNetwork`) that returns the WiseML description of a testbed.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<wiseml version="1.0" xmlns="http://wisebed.eu/ns/wiseml/1.0">
    <setup>
        <origin>
            <x>53.833836</x><y>10.704606</y><z>33.0</z><phi>-145.0</phi><theta>0.0</theta>
        </origin>
        <!-- ... -->
        <node id="urn:wisebed:uzl1:0x1bb3">
            <position>
                <x>4.5</x><y>1.0</y><z>1.0</z>
            </position>
            <nodeType>isense</nodeType>
            <description>Processor Jennic JN5139R1, Radio IEEE 802.15.4 2.4GHz</description>
            <capability>
                <name>urn:wisebed:node:capability:pir</name>
                <datatype>integer</datatype><unit>raw</unit><default>0</default>
            </capability>
            <capability>
                <name>urn:wisebed:node:capability:acc</name>
                <datatype>integer</datatype><unit>raw</unit><default>[0,0,0]</default>
            </capability>
        </node>
        <!-- ... -->
    </setup>
</wiseml>
```

Listing 9.1: WiseML based testbed self-description

Listing 9.1 presents an excerpt of the testbed located in Lübeck, Germany. All position declarations inside the document are relative to the origin of the coordinate system defined by the `origin` tag which is described

using GPS coordinates and rotation in degrees. Besides that, the document contains a sequence of `node` tags, each describing a sensor node with its position, human-readable description, and sensors. In WISEBED, sensor nodes are uniquely identified using Universal Resource Names (URN) (`id` attribute of the `node` tag). Nodes in the same testbed share a common prefix. For example, the `urn:wisebed:uzl:0x191` identifies a node with the MAC-address `0x191` in the Lübeck testbed. Sensors of an individual node are described in `capability` tags.

3. Conducting Experiments

This section describes how to conduct experiments on WISEBED-compatible testbeds. Users are free to implement custom clients that invoke the Web Services defined by the WISEBED APIs. However, in most cases using one of the three existing open-source clients will suffice. Two of these are web-based clients (TARWIS [HWAB10] and WiseUI [The10c]) and one is a command-line client. This section focuses on the latter as it provides more freedom to the experimenters. This client is a scriptable, Java-based tool, which is part of the *Testbed Runtime* [The10a] software suite (the WISEBED reference implementation). The code is based on Java classes generated by importing the WISEBED Web Service WSDL files and some helper classes.

To better understand the workflow, we take a closer look at the APIs and their interactions by discussing a sequence diagram and some code listings. As discussed above, a client application conducting an experiment must perform the three basic steps of *authentication*, *reservation*, and *experimentation*. The latter actually comprises an arbitrary number of experiment-specific actions (i.e., API invocations) for programming and interacting with nodes or to create/delete virtual links (cf. Figure 1). Figure 3 depicts all necessary API calls involved in a typical experimentation session.

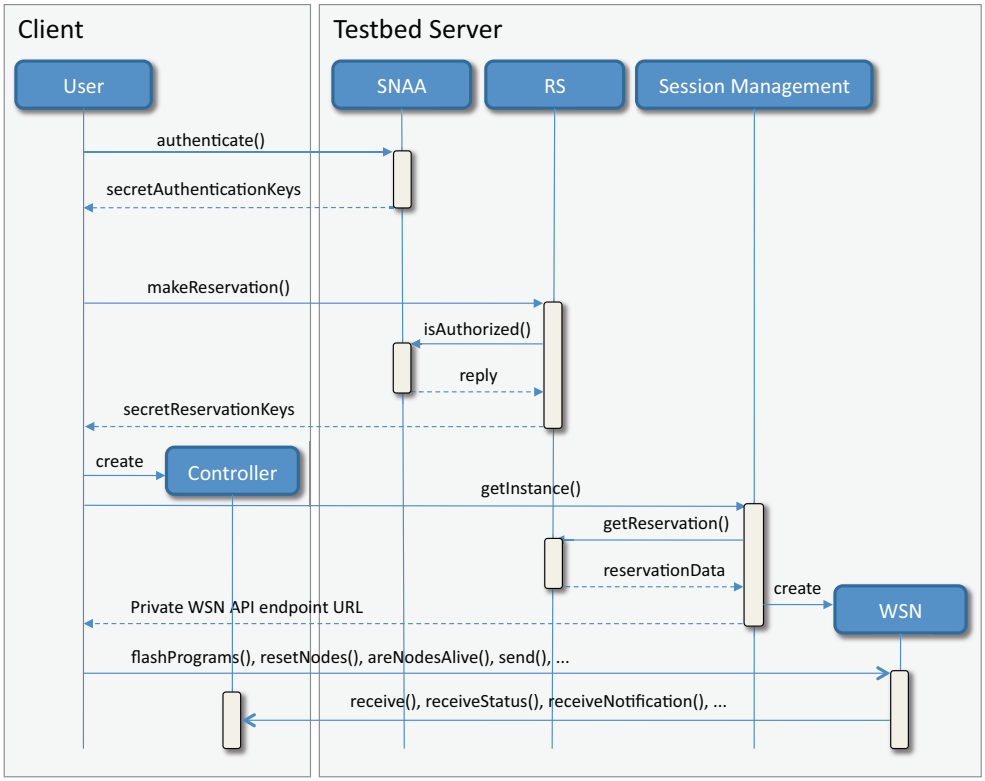

Figure 3: Interaction of the WISEBED APIs

The client first authenticates itself by calling the `authenticate` method at the SNAA. Then it reserves a set of nodes by calling `makeReservation` at the RS, using the secret authentication key obtained in the authentication step. It then starts a Controller Web Service for use as a reverse channel for testbed outputs that will be called by the Testbed Server. The client then calls `getInstance` on the Session Manager who will create and return a private experiment-specific WSN API instance, which is valid for the reservation identified by the secret reservation key obtained in the reservation step. The WSN instance can then be used to interact with the experiment on both

management and communication plane. In the following the individual steps are described in more detail using code snippets.

3.1 Authentication

To authenticate, a user needs an account at the SNAA system, which can be obtained as described on the WISEBED homepage [Sev08]. While each testbed may employ a different SNAA implementation, the WISEBED SNAA uses the single sign-on system Shibboleth [Shi], therefore an account is valid for each testbed in the federation. Users authenticate by sending a Web Service request to the SNAA of the desired testbed (or the federator).

Listing 9.2 shows this process in detail. First, the user retrieves the Web Service client instance interacting with the desired testbed by using the helper function `SNAAServiceHelper.getSNAAService(String endpointUrl)`. The property `urnPrefix` of `AuthenticationTriple` corresponds to the URN prefix that is served by this individual testbed (cf. Section 9.2.2). Authentication credentials are set on an `AuthenticationTriple` object, which is added to a list (for multiple testbeds, more than one instance of `AuthenticationTriple` is required).

If the set of federated testbeds all use the single sign-on solution of the WISEBED federation, the credential are identical for each testbed and only the URN prefixes would differ. This pattern repeats itself in several of the WISEBED API methods as the API is designed to transparently support the federation of arbitrary WISEBED-compatible testbeds, even if their authentication and authorization backend differs from the federations Shibboleth-based single sign-on system.

```
1  SNAA snaaService = SNAAServiceHelper.getSNAAService("http://testbed.wisebed.eu:8890/snaa");
2
3  AuthenticationTriple credentials = new AuthenticationTriple();
4  credentials.setUrnPrefix("urn:wisebed:uzl1:");
5  credentials.setUsername("bimschas@wisebed1.itm.uni-luebeck.de");
6  credentials.setPassword("mysecretpassword");
7
8  List<AuthenticationTriple> credentialsList = new ArrayList();
9  credentialsList.add(credentials);
10
11 List<SecretAuthenticationKey> secretAuthenticationKeys = snaaService.authenticate(credentialsList);
```

Listing 9.2: Authenticating with a testbed

If authorization succeeds, the SNAA API returns a list of `SecretAuthenticationKey` instances. If only a single testbed is used, the list contains only one `SecretAuthenticationKey` instance. The keys are then used to identify ourselves to the reservation system which checks their validity with the authentication system.

3.2 Reservation

A reservation is made using the secret authentication keys returned in the authentication step. WISEBED allows running several experiments in parallel on the same testbed to use large-scale testbeds to the full capacity. However, if researchers want to eliminate side effects from other experiments, the complete testbeds at each involved site should be reserved.

To choose sensor nodes at a site, the WiseML description containing metadata about every sensor node in the testbed (cf. Subsection 9.2.2) is used. This is obtained by invoking `getNetwork()` of the SessionManagement API. Listing 9.3 shows how to filter out the node URNs of nodes of a certain type (here: `isense`). The resulting set `nodeUrns` is used for reservation and experimentation later on.

```
12 SessionManagement smService =
       WSNServiceHelper.getSessionManagementService("http://testbed.wisebed.eu:8888/sessions");
13 String wiseml = smService.getNetwork();
14 List<String> nodeUrns = WiseMLHelper.getNodeUrns(wiseml, "isense");
```

Listing 9.3: Selecting sensor nodes for reservation

After selecting nodes, users use the Reservation System's `makeReservation()` method to reserve the selected nodes for a certain time period. Listing 9.4 shows how this is done by passing the secret authentication keys (received from the SNAA), the list of node URNs and the duration. The Reservation System checks with the SNAA if the user is authorized to perform the reservation, reserves the available nodes, and returns a list of *secret reservation keys*.

```
RS rsService = RSServiceHelper.getRSService("http://testbed.wisebed.eu:8889/rs");

XMLGregorianCalendar from = datatypeFactory.newXMLGregorianCalendar(
    new GregorianCalendar(2011, 04, 12, 15, 00));
XMLGregorianCalendar to = datatypeFactory.newXMLGregorianCalendar(
    new GregorianCalendar(2011, 04, 13, 16, 00));

ConfidentialReservationData reservationData = new ConfidentialReservationData();
reservationData.setFrom(from);
reservationData.setTo(to);
reservationData.getNodeURNs().addAll(nodeUrns);

List<SecretReservationKey> secretReservationKeys = rsService.makeReservation(
    secretAuthenticationKeys,
    reservationData
);
```

Listing 9.4: Reserving sensor nodes

3.3 Experimentation

Once the nodes are reserved and the start time has been reached, users may interact with the testbed via the WSN API to setup, start, and control the experiment. For each reservation, the `getInstance` method of the SessionManagement interface is called with the previously obtained list of secret reservation keys. It returns a private instance of the WSN API that acts as the dashboard of an experiment (cf. Section 9.2).

To enable bidirectional communication between the experiment and the experimenter, the caller needs to provide a controller endpoint URL as an additional parameter. This URL must point to an implementation of the WISEBED Controller API running on the client. Its methods are invoked by the WSN API implementation to send experiment output or asynchronous status updates about ongoing operations (e.g., programming nodes) to the client.

There are already several implementations of the Controller API that ship with Testbed Runtime. For example the aforementioned *scripting client* and a Java-based Web service client. Apart from those, users are free to implement a Web Service that conforms to the Controller API WSDL in any technology and programming language that suits their needs. Although this sounds complex at first, Listings 9.5 and 9.6 shows that it is straightforward to implement the Controller Web service API.

```
@WebService
public class Controller extends ControllerAdapter {
  @Override
  public void receive(@WebParam(name = "msg", targetNamespace = "") final List<Message> messages) {
    for (Message message : messages) {
      System.out.println(message);
    }
  }
}
```

Listing 9.5: Connecting to an experimentation session

The Controller must be started on a public IP address and port so that the WSN implementation on the Testbed Server is able to connect to it.

```
String controllerEndpointUrl = "http://myworkstation.mydomain.com/controller";
Endpoint.publish(controllerEndpointUrl, new Controller());
String wsnEndpointUrl = smService.getInstance(secretReservationKeys, controllerEndpointUrl);
```

Listing 9.6: Connecting to an experimentation session

Listing 9.7 shows how to control an experiment. As mentioned before, users choose from methods of the management plane to reprogram, reset, etc. the sensor nodes and the communication plane to send and receive messages. In the listing, first all nodes are restarted and then a message (*start*) is sent to all nodes.

```
WSN wsnService = WSNServiceHelper.getWSNService(wsnEndpointUrl);

```

```
34 wsnService.resetNodes(nodeUrns);
35
36 Thread.sleep(3000);
37
38 Message startCommand = new Message();
39 startCommand.setBinaryData("start".getBytes());
40 wsnService.send(nodeUrns, startCommand);
```

Listing 9.7: Controlling an experiment

The code listings cover only the most basic aspects of experimentation on the WISEBED platform. There are more features that were omitted here due to space constraints. E.g., assume a special topology in the sensor network is needed for an experiment. The WSN API offers control over link availability between nodes and the formation of virtual links [BCD+10] between nodes which are not in communication range of each other. Using these mechanisms users can create an arbitrary network topology even if nodes are not in the same testbed. This is achieved by a software component on the sensor nodes, which suppresses messages between two nodes if their physical link has been disabled, and by a module in Testbed Server, which controls virtual link message flows between two separate testbeds through the backbone network.

4. Operating Testbeds

The WISEBED technologies and their corresponding open-source implementations can be used to operate private, third-party testbeds. Because they are fully complying with the WISEBED APIS, they can but don't have to participate in the WISEBED federation. This allows reusing the same client software (e.g., to test an experiment on a small-scale before deploying it on larger testbeds) and server software (e.g., to operate a testbed for internal or class-room use).

The WISEBED APIs are designed to be technology independent. This allows deploying different backend implementations ranging from small-scale deployments on a single PC to a full-blown testbed federation. In the simple case, no authentication or reservation may be required and dummy implementations for both SNAA and RS APIs can be used. In the case of a large federation, more complex implementations are required. The RS and SNAA implementations of Testbed Runtime allow pluggable backends to support legacy infrastructures with low administrative overhead to get the system running.

For the SNAA there are two backends available: the *Shibboleth* [Shi] backend used by the WISEBED federation and the *Java Authentication and Authorization Service (JAAS)* backend that itself allows pluggable and extendable components for authentication and authorization such as LDAP or htpasswd files. The RS implementation comes with three pluggable backends: One based on the *Java Persistence API*, supporting virtually all kinds of databases. The second uses Google Calendar as a persistence layer and the third uses in-memory storage that can, e.g., be used for desktop or small-scale testbeds or unit-testing.

All software components are freely available under an open-source license from the project's source code web page [The10a] containing the various RS, SNAA and iWSN implementations. They are ready-to-run and only require little configuration. For more information on the possibilities and how to configure the individual software packages please check the Testbed Runtime project Wiki [The10b].

5. Conclusion

This chapter introduced the architecture of the WISEBED sensor network testbeds and provided some hands-on overview of how clients can access testbeds that are compatible with the WISEBED APIs. The major difference compared to other existing testbeds is that WISEBED offers a well-defined Web Service interface that allows users to implement custom clients or to use existing ones to automate experimentation and to repeat experiments. In addition, the WISEBED federation architecture and the concept of virtual links allow the federation of spatially divided networks and to let the federation appear like a single testbed. Hence, client software can remain unchanged. In addition to using a WISEBED testbed, the available open-source software can be used to operate custom, third-party testbeds, which may be integrated into the larger WISEBED federation. The open architecture of WISEBED and the accompanying implementations have the potential to create a standardized ecosystem for developers, experimenters, and operators.