BFH (Bern University of Applied Sciences), CH-2501 Biel, Switzerland

UniBoard Architecture Specification

Version 0.1

Eric Dubuis

August 13, 2013

Revision History

Revision	Date	Author(s)	Description
0.1	August 13, 2013	Eric Dubuis	Initial draft.

Contents

1 Introduction and Architectural Goals **4**

2 Layering **5**
2.1 Generic Service 5
2.2 Specific Services 6
2.3 The Layers 7
2.4 Replication 8
2.5 Namespaces 9

3 Implementation Hints **11**
3.1 Access to the Layer Beneath 11

1 Introduction and Architectural Goals

This document presents the architectural specification of UniBoard, a (rather) universal public bulletin board. Specific to this architecture specification is that it defines the *standalone*, monolithic as well as the *distributed* version of the public bulletin board.

To achieve this, the following architectural goals have been in mind:

- The same specification for the architecture of the *standalone* and the *distributed* variant of UniBoard.
- Establishment of a layered system of components allowing to achieve the desired functionality.
- Plug-in facility to switch from the standalone variant to the distributed one.

Note that the desired properties of UniBoard have been specified elsewhere (Ref. to be provided).

2 Layering

We introduce here the layers of UniBoard. Unless noted otherwise, a statement applies for both, the standalone and the distributed variant of the board.

2.1 Generic Service

In order to achieve upmost flexibility, all the layers of UniBoard adhere to the very same *generic* interface, simply called *Service* (namespaces in text omitted for brevity). That is, each layer (indirectly) implements this interface.

Specification of the *Service* interface:

```
package ch.bfh.uniboard.core;

public interface Service {

    public Response post(Request request) throws UniBoardException;
    public Result get(Query query) throws UniBoardException;

}
```

> *Note.* Above name of namespace as well as those mentioned bellow need agreement.

Not only the interface is rather generic, but also the provided and returned data containers *Request*, *Response*, *Query*, and *Result*. It is assumed that these abstractions are refined for interlayer communication. As a consequence, *downcast* operations will be needed that may restrict the freedom of plugging any two layers to each other. (More elaboration on this topic to be provided.)

The meaning of the data abstractions can be summarized as follows:

- *Request:* Data that originates from an upper layer or, ultimately, from an application for a *post* operation, that is given to a lower layer. Each receiving layer inspects the data and, if data is according to the expected format and content, processes it and ultimately sends it to the layer beneath, perhaps by adding additional information (except the bottom-most layer).

- *Response:* Data that is computed in a *post* operation by a processing layer and given to an upper layer or, ultimately, to the application. Each receiving layer (or the application) inspects the data and, if data is according to the expected format and content, processes it and ultimately sends it to the upper layer or the application, perhaps by adding additional information (except the application).
- *Query:* Data that originates from an upper layer or, ultimately, from an application for a *get* operation, that is given to a lower layer. Each receiving layer inspects the data and, if data is according to the expected format and content, processes it and ultimately sends it to the layer beneath, perhaps by adding additional information (except the bottom-most layer).
- *Result:* Data that is computed in a *get* operation by a processing layer and given to an upper layer or, ultimately, to the application. Each receiving layer (or the application) inspects the data and, if data is according to the expected format and content, processes it and ultimately sends it to the upper layer or the application, perhaps by adding additional information (except the application).

  Note. It is not yet decided how the data containers will be concretized. One possibility is through sub-classing, another one is through generic maps (or lists).

The abstraction of the data containers are:

```
package ch.bfh.uniboard.core;

public abstract class Request {

}

public abstract class Response {

}

public abstract class Query {

}

public abstract class Result {

}
```

Note. Names of data containers need agreement.

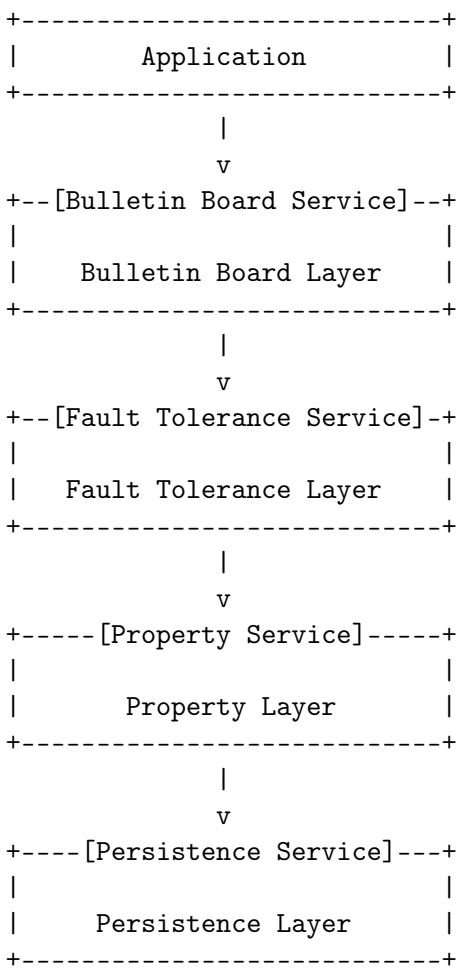
2.2 Specific Services

In order to implement the desired properties of the (distributed) public bulletin board, a number of concrete services are required:

- *Bulletin Board Service* This is an abstract view of UniBoard. Different type of technologies may be used for implementing this service in order to access UniBoard. However, web service technology (WSDL, SOAP) will be the primary choice.
- *Fault Tolerance Service* This service provides fault tolerance. It may or may not include *Byzantine* fault tolerance.
- *Property Service* This service provides additional properties required by the bulletin board. Its realization is composed of a series of property implementation, all adhering to the very same *Service* interface, exploiting the pluggable layered architecture.
- *Persistence Service* This service provides the view to the persistence storage of the postings sent to UniBoard.

2.3 The Layers

The layers of UniBoard can be sketched as follows (nice drawing, perhaps in UML, to be provided).

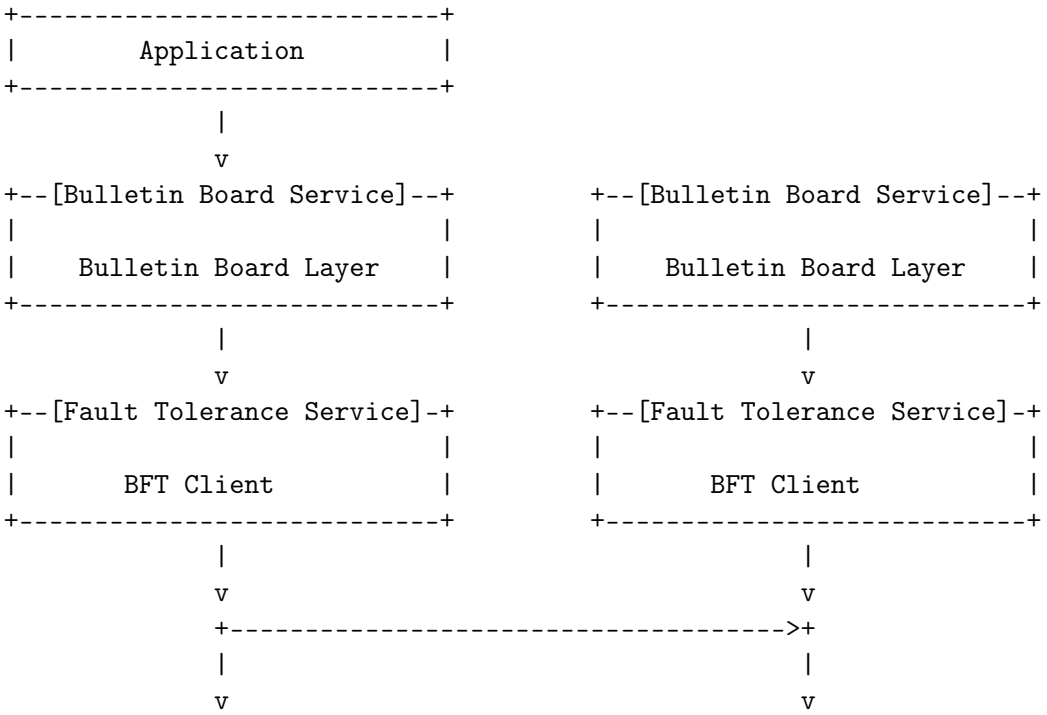


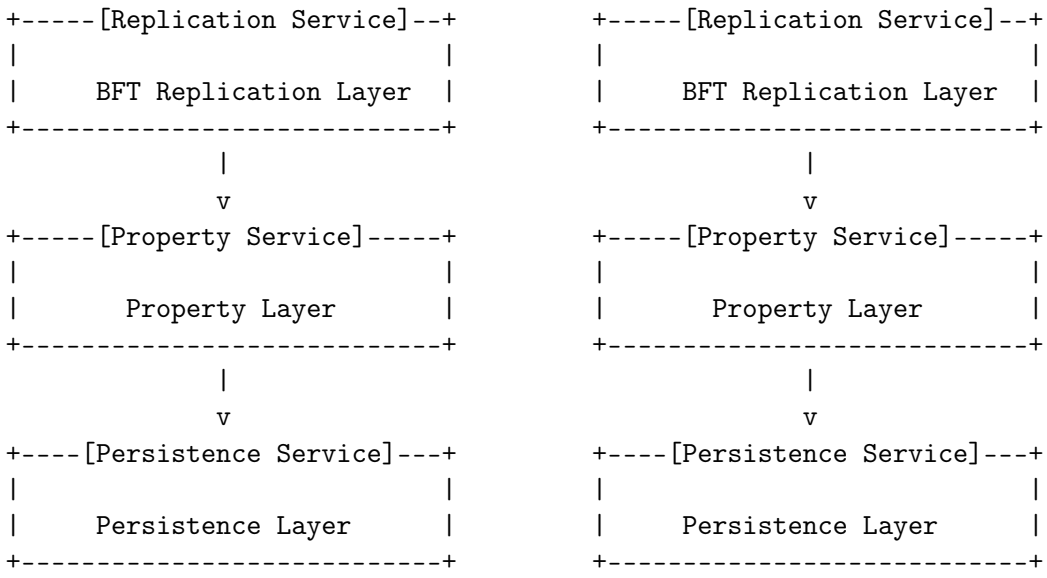
A few notes are in place:

- The identifiers between brackets "[" and "]" denote service access points. Each service access point refines the above described *Service* abstraction.
- A layer, e.g., the *Property* layer, can be decomposed into further layers, all implementing the *same Service* abstraction. Bulletin Board properties such as *authentication* or *authorization* can be implemented as separate layers (not shown in the figure above).
- If a layer is not needed (such as the Fault Tolerance Layer for the standalone case) then there are two possibilities:
  - Provide a null implementation, i.e., an implementation that purely delegates all requests to the layer beneath.
  - Do not configure it, i.e., the upper layer directly accesses the next layer with functionality.
- Several different implementations may exist for the same layer.
- The interlayer communication is supposed to be within the same address space. Hence, specific technology given within the programming framework at hand can be used. (The use EJB technology is foreseen, however.)

2.4 Replication

To achieve (Byzantine) fault tolerance, replication must be introduced. The above layered structure and a corresponding adaption yield the layering for the case having replication:





The communication between the BFT client and the BFT replicas is an implementation detail and, therefore, not discussed here. (It involves yet another layered architecture and specialized protocols, and it can used whatever communication technology.)

2.5 Namespaces

The following namespaces shall be used (subject to discussions):

The core service abstraction:

```
package ch.bfh.uniboard.core;

public interface Service ...
public abstract class Request ...
public abstract class Response ...
public abstract class Query ...
public abstract class Result ...
```

The bulletin board service:

```
package ch.bfh.uniboard.bulletinboard.service;

public interface BulletinBoardService extends Service ...
```

An implementation of the bulletin board service:

```
package ch.bfh.uniboard.bulletinboard.bean;

@Stateless
public class BulletinBoardServiceBean implements BulletinBoardService ...
```

The fault tolerance service:

```
package ch.bfh.uniboard.faulttolerance.service;

public interface FaultToleranceService extends Service ....
```

An implementation of the fault tolerance service:

```
package ch.bfh.uniboard.faulttolerance.bftclient;

@Stateless
public class BFTFaultToleranceServiceBean implements FaultToleranceService ...
```

The property service:

```
package ch.bfh.uniboard.property.service;

public interface PropertyService extends Service ....
```

An implementation of a property service:

```
package ch.bfh.uniboard.property.authorization;

@Stateless
public class AuthorizationServiceBean implements PropertyService ...
```

The persistence service:

```
package ch.bfh.uniboard.persistence.service;

public interface PersistenceService extends Service ....
```

An implementation of the persistence service:

```
package ch.bfh.uniboard.persistence.existdb;

@Stateless
public class ExistDBPersistenceServiceBean implements PersistenceService ...
```

3 Implementation Hints

Some implementation hints are given here.

3.1 Access to the Layer Beneath

In order to achieve upper-most flexibility, is is mandatory to access the lower layer service by using the *generic* service interface. For example:

```
pacage ch.bfh.uniboard.bulletinboard.bean;

@Stateless
public class BulletinBoardServiceBean implements BulletinBoardService {
    @EJB
    private Service lowerLayer; // injected by the container

    public Response post(Request request) {
        // downcast Request object to expected specialized
        // request object
        if (request instanceof BulletinBoardServiceRequest) {
            BulletinBoardServiceRequest newRequest =
                (BulletinBoardServiceRequest) request;
            ...; // do some logic here, perhaps modifying newRequest
            Response response = lowerLayer.post(newRequest);
            return response;
        } else {
            // unexpected data container, cannot process
            throw new UniBoardException(...);
        }
     }
 }
```

The injection performed by the container can be configured at deploy-time. If EAR deployment units are used then the original deployment descriptors or annotations can be overridden upon configuring the EAR deployment unit.

TODO: Sevi, provide configuration sample.