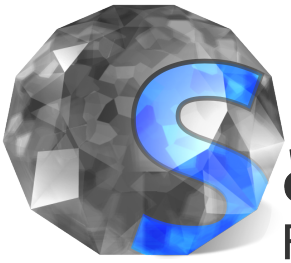

Safir SDK Core

For Truly Distributed Systems

User's Guide

2025-11-11

Lars Hagström (lars.hagstrom@consoden.se)
Anders Widén (anders.widen@consoden.se)
Joel Ottosson (joel.ottosson@consoden.se)

Contents

Preface

The purpose of this document is to provide a comprehensive guide to using the services in Safir SDK Core. The reader should have general knowledge about object-oriented programming. Some knowledge of distributed real-time systems programming is also desirable, but not essential.

This document is delivered as part of Safir SDK Core with version tag "".

How to read

This document is intended to be used both as an introduction to new users of the Safir SDK Core, and as a reference for those who have used the SDK for a long time. Because of this the level of the different sections vary quite a lot. Some sections are marked as advanced (with an ^{adv} tag to them), to signal that they cover advanced topics that can be skipped by newcomers.

Acknowledgements

This document is a combination of several other documents, so some parts of the text have different authors, or is based on texts authored by other people.

More specifically the section on Safir.Utilities.Foreach was written by Stefan Lindström, and the appendix about the example applications was written by Petter Lönnstedt. Most of the text on the basic Dob services is based on the original Dob Software User's Guide written by Jörgen Johansson.

Versions of this document

To obtain the latest version, please visit http://safirsdkcore.com/.

Licences and Copying

works) without releasing your code under the GPL license too. If you want to do this you need to obtain (and pay for...) a commercial license from Saab AB (contact information at http://www.saabgroup.com).

For more information on the GPL v3 license, see http://gplv3.fsf.org/.

Portions of the source code has other licenses and other copyright holders. These are listed in the file source package file build/packaging/debian/copyright.

Chapter 1

What is Safir SDK Core?

Safir SDK Core is a middleware and platform for creation of distributed soft real-time systems. It is Scalable, Reliable, Portable, and last but not least, it is Open!

Safir SDK Core is based on modern architectural principles and has a solid foundation in more than 20 years of experience of development of distributed systems at Saab AB.

1.1 Provided services

Safir SDK Core is mainly aimed at providing data distribution for distributed real-time systems and information systems, but there are a few other services included in Safir SDK Core, mostly because they are needed internally. Table 1.1 contains a summary of the services provided by Safir SDK Core.

Safir SDK Core consists of a number of *components*, some of which have cryptic four-letter acronyms as names. Each component provides one or more services. This document tries not to use the cryptic acronyms, but rather talks about the services provided, but it is good to know about them.

Table 1.1: Components and Services in Safir SDK Core

Component/Service	**Name**	**Provided Service(s)**
Low Level Utility Functions	Lluf	Low level functions that are not provided by Boost.
N/A	Logging	Send log messages to the Safir Log mechanism.
N/A	Distribution	Low level communication mechanism.
Distributed Objects Type system	Dots	The type system used for the distributed objects.
Distributed Objects Service	Dose	Inter-process and inter-computer distribution.
N/A	Control	Application launching and monitoring (currently limited implementation).
Dob Utility Functions	Douf	Utility functions for applications using the Dob.
Software Reports	Swre	Debug trace logging.
Dob Persistence	Dope	Persistent storage of Dob entities.
N/A	Foreach	Functionality for operating on multiple entity instances.

When we talk about *the Dob* what is really meant is the services provided by Dots, Dose and Dope, i.e. distributed objects with optional persistence.

Component dependencies and the layered architecture of Safir SDK Core is illustrated in Figure 1.1.

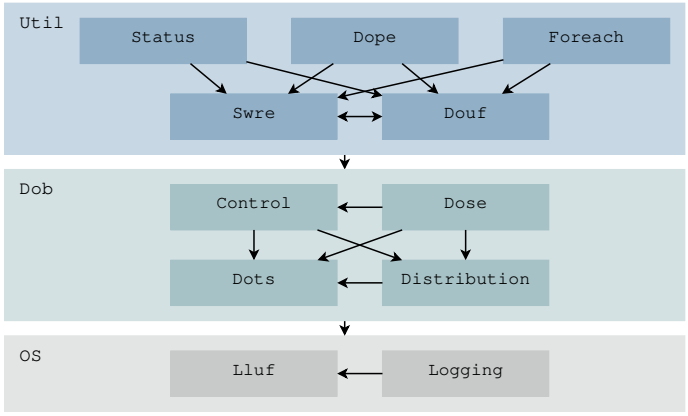

Figure 1.1: Safir SDK Core component dependencies

Again, to be able to use Safir SDK Core you do not need to understand or like these acronyms, but knowing that they are there may make some things in the SDK easier to use.

1.2 Why "Core"?

Saab sells another product Safir SDK, which is not open-source, which consists of Safir SDK Core plus a number of services that are very useful when building systems for both the civilian and military markets.

The additional services that Safir SDK provides include Alert handling, Application and Node redundancy control, communication over low-bandwidth and low connectivity media (e.g. radios of different kinds), track handling and correlation etc.

Contact Saab AB for more information (contact information at http://www.saabgroup.com).

1.3 Prerequisites / Dependencies

Safir SDK Core uses the *Boost* library to provide low level functionality and operating system independence wrappers. See http://www.boost.org/ for more information.

Safir SDK Core also uses some other open source libraries, such as Qt for C++ widgets and CMake for building. See the build instructions for more information.

1.4 Supported Platforms

Safir SDK Core supports both the Windows and Linux platforms on the x86 and x86_64 platforms. It also successufully builds and runs on ARM Linux, but we have not tested this extensively.

Obviously it is possible to combine all these platforms into one system and use Safir SDK Core for communication (after all what use is a middleware if it isn't platform independent).

An application should, written correctly using Boost and Safir SDK Core, be portable to all the supported platforms with no or very little effort.

The compilers that we use for building and testing for the C++ code are GNU gcc (version 9 or later), Microsoft Visual Studio (2015, 2017, 2019 and 2022). For C# code we use Visual Studio (2015, 2017, 2019 and 2022) or Mono (version 6.8 or later). Any Java 17 (or later) compiler should be able to compile the Java code (we've used Temurin for Windows and plain OpenJDK for Linux). We strive to make all our code standards compliant, so other versions and compilers *should* work.

There is more information about platform support on the Safir SDK Core wiki.

Chapter 2

What is the Dob?

This chapter describes what the Dob does and a little bit about why it does it the way it does.

2.1 What does the Dob do?

The Dob provides several distribution mechanisms needed to create distributed real-time systems. It is possible to interface with the Dob from several languages. Currently we provide C++, C#, and Java interfaces.

The main "selling points" include:

- Completely transparent addressing on single or multi-node systems.
- Supports *both* request/response and publish/subscribe idioms.
- Easy-to-use Persistency.
- Support for redundancy and hot-standby.
- Language independent inter-process and inter-computer communication.
- Supports use of a modular information model.

2.1.1 A bit about the Dob architecture

From the user perspective the Dob itself consists of two parts that together provide the object distribution mechanism. These two parts are the type system (Dots), which provides the language independent types and manages the definitions of the objects to distribute, and the distribution mechanism (Dose), which does the actual data disfstart an d sttribution and synchronisation.

To use the Dob an application includes the language specific interface part of the Dob interface into the application. This language specific interface in turn uses language independent libraries (written in C++) to manage the objects and to communicate with other parts of the Dob.

To run the Dob there is an executable, "safir_control", that must be running. This executable is responsible launching and monitoring the process that manages the distribution of data between applications and nodes in the system.

Within a node all data distribution is done through shared memory, and between the nodes it is UDP/IP communication (with a reliable protocol on top, to guarantee delivery).

2.2 Which problem does the Dob solves?

The Dob is appropriate for distribution of data between applications (in the same, or another, computer) in real-time and information systems, since it:

- Has no infinite queues.
- Is event driven (no polling).
- Is asynchronous (no RPC, no blocking calls).
- Designed to provide bounded latency.

The Dob implements a distributed object cache, making it possible to either read object information synchronously from shared memory, or to subscribe to object changes.

The Dob provide services so that applications (possibly written in different languages) running on Windows and Linux platforms can communicate transparently.

2.3 Quick intro

The Dob provides three distribution mechanisms; Messages, Services and Entities. These three fulfil different needs in a distributed real time system, and have different characteristics and provide different guarantees.

This section describes these, and then goes on to describe how the objects are defined.

2.3.1 Messages

Messages are data that any application can subscribe to and any application can send. Messages do not have owners in any useful sense. When an application sends a message, the Dob forwards it to all subscribers of that message.

No record is kept of messages, i.e. they are not stored in the Dob in any way. So once the message has been sent there is no way of getting hold of it again.

Messages do not have guaranteed delivery. If some application cannot keep up with the rate of messages it will miss messages. If you need guaranteed delivery, messages are not what you are looking for.

2.3.2 Services

A Service has one or more handlers (known as a Service Handler) to which any application (known as a Requestor) can send service requests. For each service request that is sent, a response is received. The response is sent by the Service Handler, and should indicate the result of the operation (i.e. success or failure, with or without result data). If the service handler does not send a response within a reasonable amount of time (configurable, see Section 6.4), the Dob will send a timeout response to the requestor.

Service requests and responses are guaranteed to be delivered. And you are guaranteed a response if you have sent a request, even if it may be a timeout response if the handler is overloaded.

2.3.3 Entities

An entity is a *class* of which there can be objects (known as instances) that are stored in the Dob and has one (and only one) owner. Only the owner is allowed to modify the object. Any application can subscribe for a entity instances, which means that it will receive updates whenever the instances are changed. Applications can also send requests (very similar to the service requests above) to the entity owner asking it to change something in the object.

Entity requests and responses are guaranteed to be delivered. And you are guaranteed a response if you have sent a request (this may be a timeout response, just as for service requests, as described above). Entity updates are guaranteed to reach all subscribers, with one important caveat; subscribers are not guaranteed to see all intermediate states of an entity. E.g. if an application misses one update the next update will look as if both things changed at once.

2.3.4 Defining the information model

The Dob allows each application/component that wants to provide a Dob interface (as in a Service, Message or Entity that other applications/components can use) to contribute to the information model by specifying its objects in xml files, which are built into the information model.

The xml files are used to generate language interfaces, that any application that wishes to use an object can include and use.

2.3.4.1 Defining objects

New types are defined by inheritance from a number of predefined classes. The classes for the distribution mechanisms are Safir.Dob.Message, Safir.Dob.Service, Safir.Dob.Entity, Safir.Dob.SuccessResponse and Safir.Dob.ErrorResponse.

All classes are defined off-line using xml in a file called a *dou*-file (named after its extension; ".dou"). These dou-files are then used to generate and compile language specific interfaces to that specific class. Applications use these generated interfaces for operating on the objects.

The dou-files are also used by the Dob to create binary chunks (usually called *blobs*) from the classes that can be sent to other nodes over a network. The dou-file information is of course also needed to interpret these blobs. The dou-files can also contain parameters for the applications. These parameters are read by the Dob at start-up, so no recompilation is required for parameter changes.

When a member is added to an object it is only necessary to regenerate/recompile the interface, and not the applications that use it. An application that is built against a previous version of the object will still work correctly (although it, of course, is unaware of the new member).

Here is an example of a dou-file for a simple entity (containing one 32-bit integer):

A simple dou-file

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <summary>This is an example entity</summary>
  <name>MyEntity</name>
  <baseClass>Safir.Dob.Entity</baseClass>
  <members><member>
    <name>TheNumber</name>
    <type>Int32</type>
  </member></members>
</class>
```

This is described in greater detail in the next chapter.

Chapter 3

The type system

One "half" of the Dob consists of a type system that is used to describe and manage the types in the Dob. This is needed to be able to create objects that are language inter-operable and to have types that it is possible to use in heterogeneous systems (e.g different nodes having different processor architectures and operating systems).

All Safir systems are based around types defined using the Dob type system.

A Dob object is made up of *members*, that can be simple or complex types (and collections of these).

The type system also provides functionality for defining run-time parameters (values are loaded at start-up) and properties, that provide a common interface for accessing members of different classes.

3.1 The simple types

These are the simple types that class members can have, either as single members or in collections (it is also possible to put objects inside objects, which is described below).

Table 3.1: The simple Dob type system types.

Type	Description
Boolean	As defined in each supported language.
Enumeration	Routines for conversion to/from strings are generated.
Int32	4 byte signed integer.
Int64	8 byte signed integer.
Float32	4 byte floating point value.
Float64	8 byte floating point value.
TypeId	Reference to a Dob class or enumeration.
ChannelId	A Message channel identifier (see Section 3.1.2).
HandlerId	A Service or Entity handler identifier (see Section 3.1.2)
InstanceId	An Entity instance identifier (see Section 3.1.2).
EntityId	An aggregate of a TypeId and an InstanceId. A reference to an Entity instance
String	Unicode string. Size is defined in number of Unicode characters.
Binary	Binary data. An arbitrary sequence of bytes.

3.1.1 SI types

The Dob type system also has definitions for the SI units. This makes it easier to specify what values are expected in objects. If a member is called "Angle" it should have a type of Radian32; this makes it easy for object users to know what to expect from

the member. If it had been defined as a Float32 the users would have to find out from somewhere else what the expected contents are supposed to be; should there be radians, degrees, gons or mils in there?

The SI unit types are all based on Float32 or Float64 respectively:

Table 3.2: Types for SI units.

4 byte type	**8 byte type**
Ampere32	Ampere64
CubicMeter32	CubicMeter64
Hertz32	Hertz64
Joule32	Joule64
Kelvin32	Kelvin64
Kilogram32	Kilogram64
Meter32	Meter64
MeterPerSecond32	MeterPerSecond64
MeterPerSecondSquared32	MeterPerSecondSquared64
Newton32	Newton64
Pascal32	Pascal64
Radian32	Radian64
RadianPerSecond32	RadianPerSecond64
RadianPerSecondSquared32	RadianPerSecondSquared64
Second32	Second64
SquareMeter32	SquareMeter64
Steradian32	Steradian64
Volt32	Volt64
Watt32	Watt64

3.1.2 Hashed types

The InstanceId, ChannelId and HandlerId types are *hashed types*. They can be defined either as a string or as a number. If defined as a number the number will be used as the value, but if defined by a string, the string is hashed and the hash is used as the value.

If a string is used to define hashed type, the string will be included in the type, but is only meant to be used for reference (the hash value is used for all built-in operations). There is a method on all the hashed types, `RemoveString`, that removes the string to save space/bandwidth.

3.1.3 Binary type

Binary is a simple type that can be used for binary data of any size. When the Binary type is used in parameters and xml or json serialized objects, the data is encoded to base64 format.

3.2 Flags on members

Each member inside a Dob object has two flags associated with it. One *IsNull* flag and one *IsChanged* flag. The purpose of the IsNull flag is to allow all members to, apart from their normal values, have a state where they're not set or unknown. The IsChanged-flag allows the Dob and applications to signal *intent as well as content* when transmitting data, for example indicating what has changed in an entity between two subscription responses.

All members have methods to access these flags, `IsNull` and `IsChanged`, and the flags can be manipulated using the `SetChanged` and `SetNull` methods.

The change flags are described in greater detail in Section 3.11.1 and Section 4.6.

3.3 Items and structs

Apart from simple types such as integers and strings and collections of these, classes can also contain other classes. These complex types or "contained classes" are usually called *Items* (due to the fact that they should inherit from Safir.Dob.Item).

An Item works exactly the same way as any other Dob class apart from the fact that it is not possible to send it to another application without putting it inside an object that is distributable (i.e. a Message, Entity, Service or Response). An item has change flags and null flags just like any other member, and all of the item's members have change flags and null flags too.

All these flags mean that there is a certain overhead to items, which in many cases is undesirable. And also, it doesn't always make sense to have null and change flags on all members of items. For example, in a Position type, there is no point in setting the Latitude member to null. Either the position as a whole is null or it is a valid position. For this purpose *Structs* (inheriting from Safir.Dob.Struct) were introduced. Currently they behave exactly like an Item, but in a future Dob version they will be optimised so as to remove all the flags and overhead.

This means that there are some limitations to what you should do to a Struct. Do not use IsChanged or IsNull on any members of a struct. Instead check IsNull and IsChanged on the whole struct. Do not inherit from a user-defined struct (structs will not support inheritance). In the current Dob this will of course work, but once the optimisation is introduced your code will break.

3.4 Collections

The Dob type system supports three kinds of collections. *Arrays*, *Sequences* and *Dictionaries*.

An array is a fixed-length integer-indexed collection of simple or complex members. Each index has its own IsNull and IsChanged flag.

A sequence is a variable-length collection of simple or complex members. There is only an IsChanged flag on the whole collection, i.e. not on individual elements. The change flag will be set whenever a value is added, removed or replaced. IsChanged/SetChanged on sequences containing objects are be recursive, just as you would expect. Sequences do not have a real IsNull flag, but instead `IsNull()` is equivalent to the sequence being `empty()`, and calling `SetNull()` is the same as calling `clear()`.

A dictionary is a variable-length collection of key-value pairs. The keys can be any simple type (with some exceptions, i.e. Boolean and floating point types) and the values can be simple or complex members. Each value has its own IsNull and IsChanged flags, and the collection itself has an IsChanged flag that will be set whenever a value is added or removed.

3.4.1 A note on collections in Java

Due to the nature of function overloading in Java the operations for adding elements to dictionaries and sequences in Java differ slightly from the other languages. The `put()` function expects a *Container*, not a value, which makes it rather difficult to work with. Unfortunately it is not possible to overload this function in a sensible way, so instead there is another operation `putVal()` (or `putObj()` for objects). Use this operation to not have to worry about the containers.

Note

Use `putVal(...)`/`putObj(...)` instead of `put(...)` when adding elements to collections in Java.

3.5 Parameters

Apart from members a class can also contain constant parameters. These parameters are not compile-time constants, instead they are read from the dou-files by the Dob when it starts and applications can ask for the values by a simple function call. This means that all that is needed to change a parameter is to change its value in the dou file, or to add a copy of the dou file, with modifications, in an override directory, as described in Section 6.1.3) and restarting the Dob and all the applications that use the Dob (note that running the code generation, as described in Section 3.9, will overwrite these changes).

Parameters are defined by a name, a type and a value. Here is an example of a parameter definition (it is cut out of its context, but it goes in the dou-file at the same level as the *members*-tag).

Example of parameter definitions

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <summary>An example parameter file.</summary>
    <name>Capabilities.MyParameters</name>
    <baseClass>Safir.Dob.Parametrization</baseClass>
    <parameters>
        <parameter>
            <name>MyStringParameter</name>
            <type>String</type>
            <value>Safir SDK Core, for truly distibuted systems</value>
        </parameter>
        <parameter>
            <name>MyInt32Parameter</name>
            <type>Int32</type>
            <value>25</value>
        </parameter>
        <parameter>
            <name>MyFloat32Parameter</name>
            <type>Float32</type>
            <value>3.14</value>
        </parameter>
        <parameter>
            <name>MyBooleanParameter</name>
            <type>Boolean</type>
            <value>True</value>
        </parameter>
        <parameter>
            <name>MyWeekdayParameter</name>
            <type>Capabilities.MyWeekdayEnum</type> <!--assumes this type has been defined ↩
                in another dou-file-->
            <value>Monday</value>
        </parameter>
        <parameter>
            <name>MyTypeId</name>
            <type>TypeId</type>
            <value>Safir.Dob.Item</value>
        </parameter>
        <parameter>
            <name>MyInstanceId</name>
            <type>InstanceId</type>
            <value>myNamedInstance</value> <!-- name or hash are valid, same sytax for ↩
                ChannelId and HandlerId -->
        </parameter>
        <parameter>
            <name>MyEntityId</name>
            <type>EntityId</type>
            <entityId>
                <name>Safir.Dob.Entity</name> <!-- name of an entity type -->
                <instanceId>theOnlyInstance</instanceId>
            </entityId>
        </parameter>
        <parameter>
            <name>MyBinaryParameter</name>
            <type>Binary</type>
            <value>SGVsbG8gV29ybGQ=</value> <!-- base64 encoding of the ascii bytes "Hello ↩
                World" -->
        </parameter>
```

```
    </parameters>
</class>
```

This defines a number of parameters of different types. For example the parameter MyInt32Parameter that is a 32 bit integer of value 25.

It is recommended to keep most parameters in pure parameter classes, suffixed Parameters, that inherit from Safir.Dob.Parametrization. These classes should not contain any members. The reason for this is to make it easier to know where parameters can be found. The Dob does not enforce this recommendation in any way.

String members are normally trimmed, i.e. leading and trailing whitespace is removed. This behaviour can be changed by using the attribute `xml:space="preserve"`.

Parameter values can contain CDATA sections and use character references to specify characters by their numeric codes, e.g. `&#89;`.

3.5.1 Parameter arrays

A parameter can also be an array of values, as shown below, where one array parameter of strings and one array parameter of entityIds are defined with two values each.

A parameter array

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <summary>An example parameter file.</summary>
    <name>Capabilities.MyParameters</name>
    <baseClass>Safir.Dob.Parametrization</baseClass>
    <parameters>
        <parameter>
            <name>StringParameter</name>
            <type>String</type>
            <array>
                <value>Safir(R)</value>
                <value>(R)rifaS</value>
            </array>
        </parameter>
        <parameter>
            <name>EntityIdParameter</name>
            <type>EntityId</type>
            <array>
                <entityId>
                    <name>Safir.Dob.Entity</name>
                    <instanceId>one</instanceId>
                </entityId>
                <entityId>
                    <name>Safir.Dob.Entity</name>
                    <instanceId>two</instanceId>
                </entityId>
            </array>
        </parameter>
    </parameters>
</class>
```

Array indexing starts at 0 when accessing the values from code.

In previous versions of Safir SDK Core the array parameter syntax was a bit more bulky. For backward compatibility the old syntax is still supported but is considered deprecated. The example below shows a parameter declaration in the old format.

Old syntax array parameter (deprecated)

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <summary>An example parameter file.</summary>
    <name>Capabilities.MyParameters</name>
    <baseClass>Safir.Dob.Parametrization</baseClass>
    <parameters>
        <parameter>
            <name>StringParameter</name>
            <type>String</type>
            <arrayElements>
                <arrayElement>
                    <value>Safir(R)</value>
                </arrayElement>
                <arrayElement>
                    <value>(R)rifaS</value>
                </arrayElement>
            </arrayElements>
        </parameter>
    </parameters>
</class>
```

3.5.2 Dictionary parameters

A parameter can also be a dictionary of key/value pairs, as shown in the example below.

A dictionary parameter of Strings mapped to Float64

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <summary>An example parameter file.</summary>
    <name>Capabilities.MyParameters</name>
    <baseClass>Safir.Dob.Parametrization</baseClass>
    <parameters>
        <parameter>
            <name>StringsMappedToDoubles</name>
            <type>Float64</type>
            <dictionary keyType="String">
                <entry>
                    <key>Hot</key>
                    <value>65.4321</value>
                </entry>
                <entry>
                    <key>Cold</key>
                    <value>-12.3456</value>
                </entry>
            </dictionary>
        </parameter>
    </parameters>
</class>
```

3.5.3 Objects in parameters

Parameters can also contain whole Dou-defined objects, not just the basic types. Below is an example from `Safir.Dob.NodeParame` where there is in fact an array of items in a parameter.

A parameter array

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <parameters>
       ...
        <parameter>
            <name>Nodes</name>
            <type>Safir.Dob.NodeDefinition</type>
            <array>
                <Safir.Dob.NodeDefinition>
                    <NodeName>My Server</NodeName>
                </Safir.Dob.NodeDefinition>
                <Safir.Dob.NodeDefinition>
                    <NodeName>My Client</NodeName>
                </Safir.Dob.NodeDefinition>
                <Safir.Dob.NodeDefinition type="Safir.Dob.NodeDefinitionSubtype">
                    <!-- type attribute is needed since we're inserting derived type -->
                    <NodeName>My Special Client</NodeName>
                    <AnotherValue>5</AnotherValue>
                </Safir.Dob.NodeDefinition>
            </array>
        </parameter>
    </parameters>
</class>
```

Each index in the array contains the xml serialization of an instance of the `Safir.Dob.NodeDefinition` item (which currently only contains one member, the node name). Of course this can be done in non-array parameters as well, just leave out all the array stuff.

One interesting feature is that you can put any item that *derives* from `Safir.Dob.NodeDefinition` into this array (that is the reason for the redundant specification of the `type` attribute in the objects). The type attribute is only needed when you want to put an instance of a derived type into the member or array element, otherwise it is redundant.

3.5.4 Environment variables in parameters

For parameters it is also possible to use environment variables in the parameter value.

The syntax for environment variables is `$(ENVIRONMENT_VARIABLE_NAME)`, and there can be several environment variables in one parameter. An example use is shown below.

Environment variable in parameter.

```
<parameter>
    <name>MyParameterWithEnv</name>
    <type>String</type>
    <value>The HOME environment variable points to $(HOME).</value>
</parameter>
```

Remember that all parameters are loaded *when the first application that uses parameters starts*, so any environment variables set after that time will not be seen by the type system environment variable expansion.

There is also support for *special variable* expansion using the same syntax as is described in Section 6.1.4. Special variables have built in magic in order to make them easy to use, for example when specifying operating system paths in parameters.

3.6 Create routines

Create routines allow the designer of a Dob class to define custom routines for creating commonly used instances of that class.

Create routines are similar to constructors. The members to be given as parameters to the routine, and those to be fetched from parameter definitions are defined. For example:

A create routine definition.

```
<createRoutines>
    <createRoutine>
        <summary>Create a position with dummy altitude.</summary>
        <name>Position</name>
        <parameters>
            <member>Latitude</member>
            <member>Longitude</member>
        </parameters>
        <values>
            <value>
                <member>Altitude</member>
                <parameter>
                    <name>Safir.Geodesy.Position.DummyAltitude</name>
                </parameter>
            </value>
        </values>
    </createRoutine>
</createRoutines>
```

Will result in generated code like:

Generated C++ create routine

```
/**
 * Create a position with dummy altitude.
 */
static PositionPtr CreatePosition
    (const Safir::Dob::Typesystem::Si64::Radian Latitude,
     const Safir::Dob::Typesystem::Si64::Radian Longitude);
```

Using the `CreatePosition` method will allow the user to create a two-dimensional Position object using only one line of code, instead of having to first create the object, and then set the three members to correct values (Position is a Struct, so it doesn't have the null flag for its members, hence the dummy position is used to signal that it is a two-dimensional position).

The Altitude member will be set to the value specified in the `Safir.Geodesy.Position.DummyAltitude` member.

The Position class also supplies a create routine for a three-dimensional position, but the two-dimensional one is a better example, since it uses a parameter.

From version 5.0 of Safir SDK Core it is also possible to write member values in place instead of referencing parameters the way it's done above with the Altitude member. The example could then be written like this instead:

In place altitude value.

```
<createRoutines>
    <createRoutine>
        ...
        <values>
            <value>
                <member>Altitude</member>
                <value>0</value> <!-- No need to declare a dummy parameter -->
            </value>
        </values>
    </createRoutine>
</createRoutines>
```

3.7 The syntax - putting it all together

As mentioned above classes are defined by creating an Xml-file called a dou-file. The dou-file describes the class and is used to generate the interface code used by the components to interface the Dob. All dou-files have two mandatory fields that describe the name of the class and its base class. The *name* field contains both the name of class and the namespace the class is located in, and the *baseClass* field contains the base class (and its namespace) to inherit from.

In the example below the name is Vehicle and the Vehicle class is located in the namespace Vehicles which is located in the namespace Capabilities. The class is referenced by other classes as Capabilities.Vehicles.Vehicle. The Vehicle class is an Entity (that resides in the Safir.Dob namespace).

Start of a dou-file class declaration

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <summary>Definition of vehicle entity</summary>
    <name>Capabilities.Vehicles.Vehicle</name>
    <baseClass>Safir.Dob.Entity</baseClass>
```

A note on namespaces

The top-level namespace *Safir* is reserved for Dob classes belonging to Safir and shall not be used for classes not belonging to Safir.

One of the main purposes of Safir SDK is to promote reusable code. Due to this Saab recommends **not** using project names as top-level namespaces (or indeed any part of the namespace), since this will cause problems when reusing those components in another project.

Saab uses *Capabilities* as the top-level namespace for reusable components built on Safir. Non-top-level namespaces should not be component or project specific names but rather service oriented names describing the functionality of the classes in that namespace. As an example the class `Capabilities.Vehicles.Vehicle` is a class in a reusable component for handling vehicles.

Arrays are declared by adding the field `arraySize` to an element or constant as shown in the example below.

Array member declaration

```
    <member>
        <name>Type</name>
        <arraySize>10</arraySize>
        <type>Int32</type>
    </member>
```

Sequences are declared by adding the field `sequence` to an element or constant as shown in the example below.

Sequence member declaration

```
    <member>
        <name>Type</name>
        <type>Int32</type>
        <sequence/>
    </member>
```

Dictionaries are declared by adding the field `dictionary` to an element or constant as shown in the example below. The key type is specified using the `keyType` attribute.

Declaration of dictionary of strings mapped to 32 bit integers

```
    <member>
        <name>Type</name>
        <type>Int32</type>
```

```
        <dictionary keyType="String"/>
    </member>
```

For strings the maximum length in number of Unicode characters can be defined using the `maxLength` tag.

String member declaration

```
    <member>
        <name>Callsign</name>
        <type>String</type>
        <maxLength>10</maxLength> <!-- Optional -->
    </member>
```

Exceeding the maxiumum string length (i.e. setting a string that is too long) will cause an exception to be thrown *when the object is serialized*, e.g. when a Message is transmitted. If no maximum length is specified the strings can be Very Long™.

The string lengths and array sizes can also be specified with parameters. The tags used for this is arraySizeRef and maxLenghtRef, and the parameter name must be fully qualified (full namespace and class name).

Parameters and members

```
    <parameters>
        <parameter>
            <name>ArraySize</name>
            <type>Int32</type>
            <value>10</value>
        </parameter>
        <parameter>
            <name>StringLength</name>
            <type>Int32</type>
            <value>10</value>
        </parameter>
    </parameters>
    <members>
        <member>
            <name>Type</name>
            <arraySizeRef>
                <name>Capabilities.Vehicles.Vehicle.ArraySize</name>
            </arraySizeRef>
            <type>Int32</type>
        </member>
        <member>
            <name>Callsign</name>
            <type>String</type>
            <maxLengthRef>
                <name>Capabilities.Vehicles.Vehicle.StringLength</name>
            </maxLengthRef>
        </member>
```

Of course, as mentioned in Section 3.5 it is recommended that parameters are placed in separate parameter classes.

It is possible - indeed it is even recommended - to add comments to most fields in dou-files since these comments will be put into the generated code in a style that is appropriate for the specific language.

A commented member

```
        <member>
            <summary>This is a callsign.</summary>
            <name>Callsign</name>
            <type>Int32</type>
        </member>
```

3.8 Properties [adv]

A property is not an object on its own. It is merely an interface into other objects. The way that a property interfaces into an object is defined at start-up, *so it is possible to change the way the interface works without recompiling any applications.*

The interface itself is specified in a dou-file which is slightly different from class definitions.

A simple property (NamedObject)

```
<?xml version="1.0" encoding="utf-8" ?>
<property xmlns="urn:safir-dots-unit"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <name>Safir.NamedObject</name>
    <members>
        <member>
            <name>Name</name>
            <type>String</type>
        </member>
    </members>
</property>
```

The next step is to create a *dom*-file (has .dom as its extension) to define to which object member the property member is mapped.

A NamedObject mapping for Vehicle

```
<?xml version="1.0" encoding="utf-8" ?>
<propertyMapping xmlns="urn:safir-obj"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <property>Safir.NamedObject</property>
    <class>Capabilties.Vehicles.Vehicle</class>
    <memberMapping>
        <member>
            <propertyMember>Name</propertyMembert>
            <classMemberReference>
                <classMember>Callsign</classMember>
            </classMemberReference>
        </member>
    </memberMapping>
</propertyMapping>
```

The mapping above specifies that the member Name in the NamedObject property is mapped to the Callsign member of the Vehicle object.

It is also possible to map property members to values (either to direct values or to references to parameters) or to null.

Property mapping to value or parameter

```
    <member>
        <propertyMember>Int32Member</propertyMember>
        <value>10</value>
    </member>
    <member>
        <propertyMember>Int64Member</propertyMember>
        <valueRef>
            <name>Safir.UnitParameters.SomeParameter</name>
        </valueRef>
    </member>
    <member>
        <propertyMember>NullMember</propertyMember>
    </member>
```

It is even possible to map to members that reside inside an item array inside the mapped class.

Property mapping into an array

```
    <member>
        <propertyMember>Int64Member</propertyMember>
        <classMemberReference>
            <classMember>ItemArrayMember</classMember>
            <index>2</index>
            <classMemberReference>
                <classMember>TheMemberIWanted</classMember>
            </classMemberReference>
        </classMemberReference>
    </member>
```

And so on...

All fields in the property have to be mapped to something (i.e. a member, parameter or to null) to be a complete mapping. Dom-files have to be named "<mapped class name>-<property name>.dom", e.g. "Capabilties.Vehicles.Vehicle-Safir.NamedObject.dom".

It is possible to use properties on any kind of object except Structs. So the same property could be mapped into an item, an entity, a service and a response.

Properties are inherited

Property mappings are inherited, so, for example, any class that inherits from Safir.Vehicles.Vehicle will inherit the property mapping defined above. The derived class can override the inherited property mapping by supplying its own mapping.

3.8.1 Using the property

To use the property you need to have an object of a type that is mapped to it. For example you could have set up a subscription to Capabilities.Vehicles.Vehicle (which is mapped to the NamedObject property).

Using a property

```
void DisplayName(const Safir::Dob::Typesystem::ObjectPtr & obj)
{
    const std::wstring name = Safir::NamedObject::GetName(obj);
    ... display the name ...
}
```

Note that in the code above there is no reference at all to what kind of object `obj` is. It could be any object that has the property mapped to it. It is possible to check if an object has a specific property mapped to it using the `HasProperty` method on the property.

3.9 Generating code from Dou-files

In order to use the classes defined by the Dou-files, interface code has to be generated for the different languages. First, source code has to be generated from the dou-files and then this source code needs to be built into loadable libraries suitable for your language and platform (e.g. dlls, shared libraries, or assemblies). Then libraries and header files have to be installed into a location where they can be accessed by your programs.

The easiest way to perform all these steps is using CMake and *dobmake*, for which you will need to install CMake (version 3.11 or later) and Python (version 3.1 or later).

You need to have your dou files in one or more directories, and in each directory you will need to put a file named `CMakeLists.txt`. This file is a CMake control file that contains the information needed to build the code in that directory (for more information on CMake, visit http://www.cmake.org).

Simple CMakeLists.txt for building a safir_generated module out of two dou files

```
cmake_minimum_required(VERSION 3.16)

find_package(SafirSDKCore REQUIRED)

add_safir_generated_library(
  NAME Test1
  DEPENDENCIES Core
  DOU_FILES Dobmake.Test1.dou
            Dobmake.Test2.dou)
```

Above is a simple CMakeLists.txt file that will build a module out of two dou files, named `Dobmake.Test1.dou` and `Dobmake.Test2.dou`. The names of the resulting binaries will be as defined in Table 3.3, with `<name>` replaced by the value passed as the `NAME` argument to `add_safir_generated_library`. The `DEPENDENCIES` argument specifies that the dou file contains a dependency to the Core dou files, e.g. Safir.Dob.Entity.dou etc.

The full documentation of `add_safir_generated_library` can be found in the SafirSDKCoreConfig.cmake file that is installed as part of the Safir SDK Core installation package. Please read that information, since there are more things that you can do with this function than what has been described here.

Table 3.3: Names of binaries generated by dobmake

Description	Linux file name	Windows file names
C++ shared library	libsafir_generated-<name>-cpp.so	safir_generated-<name>-cpp.dll safir_generated-<name>-cppd.dll
C++ link library	N/A	safir_generated-<name>-cpp.lib safir_generated-<name>-cppd.lib
C# assembly	safir_generated-<name>-dotnet.dll	safir_generated-<name>-dotnet.dll
Java archive	safir_generated-<name>-java.jar	safir_generated-<name>-java.jar

If you are using cmake for your own project, you can incorporate your dou-file cmake files into your "build tree". If you are using another build system, or are not using a build tree in cmake you will need to install the produced binaries and C++ header files somewhere where your code can find them.

CMakeLists.txt commands for installing safir_generated files

```
#relative paths, relative to CMAKE_INSTALL_PREFIX
install_safir_generated_library(
  TARGETS Test1
  CXX_RUNTIME bin
  CXX_LIBRARY lib
  CXX_INCLUDE include
  JAR java
  DOTNET dotnet
  DOU_BASE dou)
```

Above is an example of an installation directive for the binaries and include files generated from the Test1 module. In the example the installation paths are relative, and will be relative to CMAKE_INSTALL_PREFIX (which will be set by dobmake). If you specify absolute installation paths they will be used as specified.

3.9.1 Dependencies between modules

As hinted to above it is possible to have many safir_generated library modules, that have dependencies on each other. Naturally circular dependencies are not allowed.

If you build all your safir_generated library modules in one CMake "build tree", you only have to specify the dependencies using the `DEPENDENCIES` argument as described. But if you are building library modules separately, you will also have to tell the

typesystem where to find other library modules that you depend on. Dobmake uses information in `typesystem.ini` - as described in Section 6.1.3 - to find library module dependencies that it cannot find within the current CMake build tree.

If you download the Safir SDK Core source code you can find an example of a dobmake/CMake build tree under `src/dots/dots_dob`

3.9.2 Running Dobmake GUI

Dobmake is really only a gui for building and installing a CMake project. In fact, it can be used to build any CMakeLists.txt file. Figure 3.1 shows dobmake running under Windows 10.

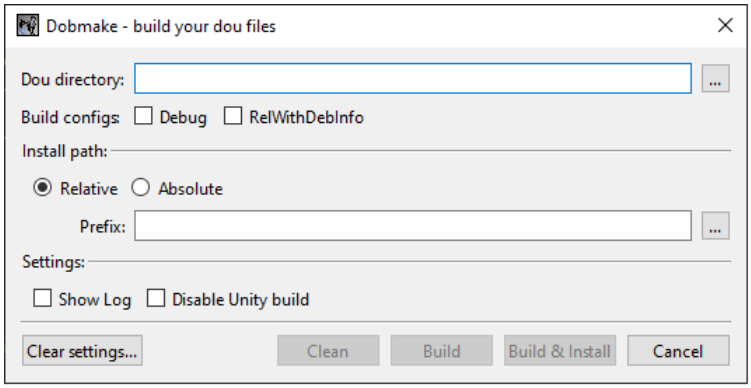

Figure 3.1: Dobmake screen shot

To build a CMake project, enter the path to the CMakeLists.txt file in the *Dou directory* field (or select the CMakeLists.txt file using the browse button). Most likely you should leave the *Build configs* settings unchanged. You should now be able to press the *Build* button to generate code and build it.

If you're using absolute install paths in your CMakeLists.txt file you can select *Absolute* and then press *Build & Install* to also install the results to the directories you specified. If you're using relative install paths you first need to specify a CMAKE_INSTALL_PREF using the *Prefix* field.

Note

Dobmake will build binaries for C++, C# and Java if it can find compilers for each of these languages.

It is possible to explicitly disable Java builds by setting an environment variable `SAFIR_DONT_BUILD_JAVA` to `True`.

3.9.3 Running Dobmake from script or command line

If you want to run dobmake from a script or from the command line, use dobmake-batch (or dobmake-batch.py on Windows) to run without using the GUI.

The dobmake-batch script expects to be run from the directory containing the CMakeLists.txt file, and CMAKE_INSTALL_PREFIX can be set using the `--install` argument.

3.10 Using the generated types

This section contains some pointers on how to use the generated types, and what the different operations and classes "mean".

For the examples in this section we need a couple of Dob-class definitions: B is an item (a Dob-object to use inside other objects) which contains an Int64, called MyInt. A is a Dob-Object (could be an Entity, Service, Message or Response) that contains one B, called MyB and one Float32 called MyFloat. So an A object (myA) could look like in Figure 3.2.

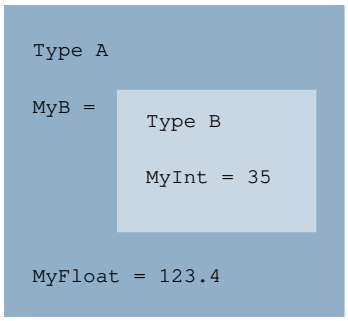

Figure 3.2: Example object

3.10.1 Syntax in the different languages

The design approach is that the generated code should make use of the language's own features in the best possible way. Languages that support operator overloading should use them if it makes the interface "better", and languages that support properties should use them. This makes each language interface "as good as it can possibly be", but it has one obvious side effect, i.e. the interfaces are *not identical*.

The syntax for operating on top-level members in the different languages is shown in Operating on top-level members in C++, Operating on top-level members in C# and Operating on top-level members in Java. The code is identical, except in the C++ example, where some alternative ways of accomplishing the same thing is shown.

(The attentive reader will notice the pointer stuff in the C++ examples. This is because the SDK uses smart pointers to manage memory in languages without a garbage collector. See Section 3.10.3 for some info on smart pointers).

Operating on top-level members in C++

```
// Get MyFloat out of myA
int val = myA->MyFloat();
// or
int val = myA->MyFloat().GetVal();

// Set MyFloat in myA to 3.14
myA->MyFloat() = 3.14;
// or
myA->MyFloat().SetVal(3.14);

// Check if MyFloat is null
if (myA->MyFloat().IsNull())
    ...

// Check if MyFloat is changed
if (myA->MyFloat().IsChanged())
    ...

// Set MyFloat to null
myA->MyFloat().SetNull();
```

Operating on top-level members in C#

```
// Get MyFloat out of myA
int val = myA.MyFloat.Val;

// Set MyFloat in myA to 3.14
myA.MyFloat.Val = 3.14;

// Check if MyFloat is null
if (myA.MyFloat.IsNull())
    ...

// Check if MyFloat is changed
if (myA.MyFloat.IsChanged())
    ...

// Set MyFloat to null
myA.MyFloat.SetNull()
```

Operating on top-level members in Java

```
// Get MyFloat out of myA
int val = myA.myFloat().getVal();

// Set MyFloat in myA to 3.14
myA.myFloat().setVal(3.14);

// Check if MyFloat is null
if (myA.myFloat().isNull())
    ...

// Check if MyFloat is changed
if (myA.myFloat().isChanged())
    ...

// Set MyFloat to null
myA.myFloat().setNull();
```

The "extra level" to get to the actual values, e.g. the `GetVal()` bit in C++ and `Val` bit in C# is needed due to the fact that the value is contained, together with the change and null flags, in a *container*. That there are two different ways of doing

the operations in C++ is because of a clever C++ construction, *proxy objects*, combined with operator overloading. More on containers and proxies later.

C# and Java has only one way of doing the operations, and the C# interface uses properties for accessing the values - not as in Dob properties, but as in the language construct. (Look it up in your favourite C# reference if you don't know what they are.)

Things get a bit more interesting when accessing nested members, see Operating on nested members in C++, Operating on nested members in C# and Operating on nested members in Java, that show operations on nested members in the different languages.

Operating on nested members in C++

```
// Get MyInt
int val = myA->MyB()->MyInt();
// or
int val = myA->MyB()->MyInt().GetVal();

// Set MyInt to 3 (if MyB is not null)
myA->MyB()->MyInt() = 3;
// or
myA->MyB()->MyInt().SetVal(3);

// Create an empty B item for MyB and set B.MyInt to 3
myA->MyB() = B::Create();
myA->MyB()->MyInt() = 3;

// Check if MyInt is null
if (myA->MyB()->MyInt().IsNull())
    ...

// Check if MyInt is changed
if (myA->MyB()->MyInt().IsChanged())
    ...

// Set MyInt to null
myA->MyB()->MyInt().SetNull();
```

Operating on nested members in C#

```
// Get MyInt
int val = myA.MyB.Obj.MyInt.Val;

// Set MyInt to 3 (if MyB is not null)
myA.MyB.Obj.MyInt.Val = 3;

// Create an empty B item for MyB and set B.MyInt to 3
myA.MyB.Obj = new B();
myA.MyB.Obj.MyInt.Val = 3;

// Check if MyInt is null
if (myA.MyB.Obj.MyB.IsNull())
    ...

// Check if MyInt is changed
if (myA.MyB.Obj.MyB.IsChanged())
    ...

// Set MyInt to null
myA.MyB.Obj.MyB.SetNull();
```

Operating on nested members in Java

```
// Get MyInt
int val = myA.myB().getObj().myInt().getVal();
```

```
// Set MyInt to 3 (if MyB is not null)
myA.myB().getObj().myInt().setVal(3);

// Create an empty B item for MyB and set B.MyInt to 3
myA.myB().setObj(new B());
myA.myB().getObj().myInt().setVal(3);

// Check if MyInt is null
if (myA.myB().getObj().myInt().isNull())
    ...

// Check if MyInt is changed
if (myA.myB().getObj().myInt().isChanged())
    ...

// Set MyInt to null
myA.myB().getObj().myInt().setNull();
```

Casting objects up and down in the inheritance hierarchy also differs between the languages, see Section 3.10.4 for more information.

3.10.2 Containers and Container Proxies

As mentioned in other parts of this document each member in a class has two flags associated with it; the *IsNull* flag and the *IsChanged* flag. To associate these flags with the value all three (the value and the two flags) are all put inside a *container*. The container has functions for getting and setting the value (which check/update the flags) and for changing the flags.

There are containers for all of the types in the Dob, e.g. Int32Container, Float64Container EntityIdContainer, and so on. There are also containers for user-generated types (classes and enums), and they are defined in the auto-generated code, so in the above example there would be a definition of AContainer and BContainer.

The members of the class A would be of type BContainer and Float32Container, and B's would be of type Int64Container.

The containers introduce an extra level of indirection, which shows itself in the "GetVal()", "SetVal()" and "Val" parts of the expressions above. This unfortunately has the effect of cluttering up the code a bit, so to reduce this cluttering (in C++, which is the only supported language where it is possible) *container proxies* with *operator overloading* were introduced. They are really just an intermediate object that allows safe use of operator overloading, and they are what allows "myA.MyB()->MyInt().SetVal(3)" to be replaced with "myA.MyB()->MyInt() = 3" (as shown in the examples above). The proxies are meant to be transparent, but they do not have all operations overloaded, so for example if you're manipulating a string, you will probably have to do GetVal() to be able to get to most of the string operations.

Note that these proxies (container proxies) should not be confused with the proxies that the Dob passes to the applications in the distribution callbacks. The Container Proxies are only meant to simplify use of the typesystems containers, whereas the proxies in the distribution callbacks are used to encapsulate a collection of data and metadata into a single object for the callback (basically allowing the callback to take just a few arguments, instead of a whole bunch, where most applications only use a few).

3.10.3 Smart pointers

The generated classes are allocated on the heap and then handled via pointers. Using pointers to objects is necessary for polymorphism to work properly in the Dob interfaces.

For C++, which is not a garbage collected language, we use the smart pointer concept. Smart pointers are *pointer wrapper objects* that store pointers to dynamically allocated objects. They keep track of how many references there are to a certain object, and automatically deletes the referenced object when there are no more references to it.

In C# and Java there is no need for smart pointers, since both these languages have garbage collection.

3.10.3.1 Smart pointers in C++

In C++ to guarantee that all dynamically allocated memory is deallocated, the Dob uses the `std::shared_ptr` implementation of smart pointers. Since C++ is not garbage collected we do not want to use raw pointers. `std::shared_ptr` is a smart pointer that automatically deletes the object pointed to when there are no references to it.

Read more about them in your favourite C++ Reference

Prior to version 7.0 Safir SDK Core used the Boost version of shared_ptr, `boost::shared_ptr`.

See Section 3.10.4.1 for more information on how to cast smart pointers in C++.

3.10.3.2 Hints for Debugging

It is possible to look at the contents of a Dob object in the debugger. Since the containers are all part of a class hierarchy the objects may seem a little difficult to understand at first. But just remember these things:

- Value containers have three values: m_Value, m_bIsNull and m_bIsChanged (which is the member value and the null and change flag respectively).
- Object containers have two values: m_pObject (which can be null) and m_bIsChanged.
- C++ smart pointers have two pointers in them, the reference count (pn) and the pointer to the object they point to (px). It is px that you are interested in.

The naming of the container members is also slightly different in the different languages.

3.10.4 Type casting

Since the Dob types use inheritance any application that uses the Dob will have to do a lot of type casting. It is very important to understand how this is done, otherwise your application is likely to behave in very strange and unexpected ways.

For the discussion in this section the object hierarchy in Figure 3.3 is used.

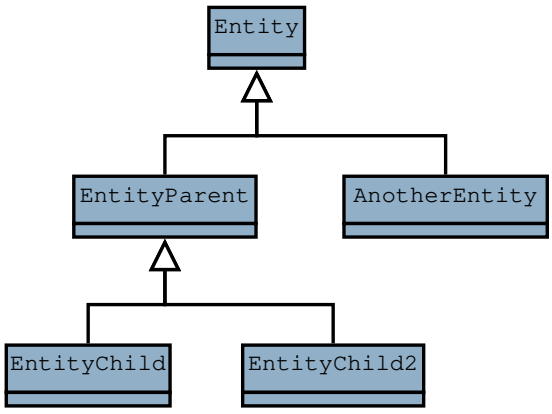

Figure 3.3: Example object hierarchy

3.10.4.1 C++ type casting

The Dob C++ interface uses `std::shared_ptr` (as described above) for passing around objects over the interfaces. For example this means that you get a `std::shared_ptr<Safir.Dob.Entity>` (typedefed to `Safir.Dob.EntityPtr` in the generated code) out of the `EntityProxy` in `OnNewEntity`. If you're subscribing to `EntityParent` you would want to somehow cast this `EntityPtr` to an `EntityParentPtr`.

If it was an ordinary pointer you would either use a `static_cast` or a `dynamic_cast` operation (if you're unfamiliar with them you should look them up in your C++ reference now), but since these are shared_ptrs you have to use `std::static_pointer_c` or `std::dynamic_pointer_cast` to accomplish the same thing.

std::static_pointer_cast

Use std::static_pointer_cast when you know that the cast will succeed (i.e. you know what type the pointer really has). For example if you have an EntitySubscriber that has subscribed to EntityParent *only* you can do:

Using std::static_pointer_cast

```
void MyEntitySubscriber::OnUpdatedEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    EntityParentPtr parent =
        std::static_pointer_cast<EntityParent>(entityProxy.GetEntity());
    ... Your code for handling the entity ...
```

(Note that this code will be used to handle updates to both of the subclasses of EntityParent too, but the assumption here is that you want to handle EntityParent and all its subclasses in the same way.)

Warning: If you add a subscription to AnotherEntity this code will cast AnotherEntity-objects to EntityParentPtrs too, which will result in undefined behaviour.

std::dynamic_pointer_cast

Use std::dynamic_pointer_cast when the pointer could be of several different types that you want to treat differently. For example if you have an EntitySubscriber that has subscribed to EntityChild *and* AnotherEntity you can do:

Using std::dynamic_pointer_cast

```
void MyEntitySubscriber::OnUpdatedEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    EntityPtr entity = entityProxy.GetEntity();

    EntityChildPtr child =
        std::dynamic_pointer_cast<EntityChild>(entity);
    if (child != NULL)
    {
        HandleChild(child);
        return;
    }

    AnotherEntityPtr another =
        std::dynamic_pointer_cast<AnotherEntity>(entity);
    if (another != NULL)
    {
        HandleAnother(another);
        return;
    }
```

This would use a different routine to handle the different entities. Another example would be if you have subscribed to Entity-Parent *only* but want to treat EntityChild2 differently:

Another std::dynamic_pointer_cast example

```
void MyEntitySubscriber::OnUpdatedEntity
        (const Safir::Dob::EntityProxy entityProxy)
{
    EntityPtr entity = entityProxy.GetEntity();

    EntityChild2Ptr child2 =
        std::dynamic_pointer_cast<EntityChild2>(entity);
    if (child2 != NULL)
```

```
    {
        HandleChild2(child2);
        return;
    }
    HandleParent(std::static_pointer_cast<EntityParent>(entity));
```

In this code HandleChild2 gets called for all objects of the EntityChild2 type (remember that it can be a subclass of EntityChild2 too) and all others will be handled by HandleParent.

Remember: Dynamic casts are a lot more expensive than static casts. So use `static_cast` when you can!

Note: You may have noticed that in both the above examples we extract the entity out of the proxy separately (and not inline in the casts). This is because the call to GetEntity in the callback proxy includes a deserialization of a binary blob into a language specific class. This is a non-trivial operation, so don't call it several times if you don't need to.

Direct TypeId comparison

You should **only** use direct TypeId comparison when you want to check that a type is of *one specific type but not a subtype.* For example if you have subscribed to EntityParent and you want to handle EntityParent differently from the subclasses you can do:

Direct TypeId comparison

```
void MyEntitySubscriber::OnUpdatedEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    EntityPtr entity = entityProxy.GetEntity();

    if (entity->GetTypeId() == EntityParent::ClassTypeId)
    {
        HandleParent(entity);
        return;
    }

    HandleChildren(std::static_pointer_cast<EntityParent>(entity));
```

Here the HandleParent routine would be called for EntityParent objects *only* and all other objects would be handled by HandleChildren.

This construction should be used *very* sparingly. It is probably not the behaviour you are looking for, since it makes your application unable to handle derived objects in the way usually expected in object oriented systems.

IsOfType The Dob provides an operation Safir::Dob::Typesystem::Operations::IsOfType that can be used to check the relationship between TypeIds while taking inheritance into account.

IsOfType logic

```
IsOfType(EntityParent::ClassTypeId,
         EntityParent::ClassTypeId)  //True
IsOfType(EntityParent::ClassTypeId,
         AnotherEntity::ClassTypeId) //False
IsOfType(EntityChild::ClassTypeId,
         EntityParent::ClassTypeId)  //True
IsOfType(EntityParent::ClassTypeId,
         EntityChild::ClassTypeId)   //False
IsOfType(EntityChild2::ClassTypeId,
         EntityChild::ClassTypeId)   //False
```

This operation should *only* be used when you need to check relationships between TypeIds. If you have an object you should be using `std::dynamic_pointer_cast`.

3.10.4.2 C# type casting

Since C# is garbage collected the Dob can use normal C# pointers to objects. This means that the normal type casting operations can be used. C# also does not have an equivalent to static_cast, so there is really only the choice between different forms of

the dynamic_cast-like operation to choose between. If you're unsure of how C# casting works you should look it up in your C# reference, since a complete explanation is outside the scope of this document. But here are some pointers:

If you know the type of the object (by only having subscribed to one type), go right ahead and cast it:

Type casting in C#

```
    AnotherEntity ent = (AnotherEntity)entity;
   ... do something ...
```

If for some reason the entity was not of the expected type an exception will be thrown, but remember that in this case that can be only due to a programmer error.

If you need to find out the type there are two ways to do it. One good and one not so good. Either you can use the *is* operation to check that the type is of the right kind and then cast it using c-style casts to get an object of the right type:

Using the is operator

```
if (entity is EntityChild) //not recommended way of checking type
{
    EntityChild child = (EntityChild)entity;
    ... do something ...
}
```

The drawback of using this approach is that you in fact get two type checks, one in the is operation and one in the cast operation. C# provides another operation that means that you only get one type check:

Using the as operator

```
EntityChild child = entity as EntityChild;
if (child != null)
{
    ... do something ...
}
```

Prefer using the *as* operation in C#.

3.10.4.3 Java type casting

Since Java is garbage collected the Dob can use normal Java pointers to objects. This means that the normal type casting operations can be used. Java also does not have an equivalent to static_cast, so there is really only the choice between different forms of the dynamic_cast-like operation to choose between. If you're unsure of how Java casting works you should look it up in your Java reference, since a complete explanation is outside the scope of this document.

3.11 Other details

Details, details, details and even more details...

3.11.1 Change flags

As mentioned before (Section 3.2) each member inside a Dob object (i.e. Entity, Message, Service, and Item, but not Struct, see Section 3.3) has two flags associated with it, an IsNull flag and an IsChanged flag (often called a *null flag* and a *change flag*, respectively).

The distribution mechanism uses the change flags to provide meaningful change information in the different distribution mechanisms, which you can read more about in Section 4.6. This section covers only how the change flags work inside an object inside one single process.

When an object is created all change flags are set to false. When a method to change a members value is called the change flag for that member is set to true (even if the value was the same as before, i.e. it didn't change!). If the value is changed to null (via SetNull()) it is also set to true.

If a member inside an item inside an object is changed only the change flag on the nested member is set.

The change flags' values can be checked using the IsChanged()-methods on the members. In the case of a simple member (i.e. not an item) the flag's value is received, but in the case of an item true is returned if *any one of the item's members are changed*. So IsChanged on an item member is recursive! If you need to find out if a change flag is set on the member itself, rather than on one of its members you can use IsChangedHere().

An example to make this clearer:

Table 3.4: Type definition for MyObject type

Member Name	**Type**
A	Int64
B	String
C	MyItem

Table 3.5: Type definition for MyItem type.

Member Name	**Type**
X	Int32
Y	Float64

We have two types, MyObject and MyItem (Table 3.4 and Table 3.5), where MyObject contains among other things an item of type MyItem.

If we have an object with change flags set like in Table 3.6 (note that member C has a change flag of its own as well as a change flag for each member in it) the result of the calls to IsChanged would be like in Table 3.7.

Table 3.6: An object with change flags

Member	**Change flag**
A	true
B	false
C	false
C::X	true
C::Y	false

Table 3.7: Result of IsChanged() and IsChangedHere()

Member	**IsChanged**	**IsChangedHere**
A	true	N/A
B	false	N/A
C	true	false
C::X	true	N/A

Table 3.7: (continued)

Member	IsChanged	IsChangedHere
C::Y	false	N/A

There are also methods to explicitly set the change flags, SetChanged(bool) and SetChangedHere(bool). As in the case of IsChanged the SetChanged-method on items is recursive, so if you need to change only the flag on the item itself you will have to use SetChangedHere. (In fact the above example could only be produced from a newly created object using some SetChanged methods).

So, what of all this stuff do you need to use? Probably only the IsChanged() method. All the other stuff is mostly used behind the scenes by the Dob to weave its magic. But there may be times that you have to do something like this.

3.11.2 Serialization

All objects of types defined by dou files can be serialized to and from XML and JSON format (JSON support was introduced in Safir SDK Core 5.0). Methods for serialization and deserialization of objects are found in Safir.Dob.Typesystem.Serialization.

Some notes on the XML serialization:

- When serializing objects to XML, all null values are omitted to keep serialization results small and readable. This means that an object with only null values will only consist of a start and end tag, e.g. `<Safir.Dob.Position/>`.
- Serialized strings are trimmed by default and all leading and trailing whitespaces, newlines and tabs are removed. To change this behaviour, use the xml attribute `xml:space="preserve"`, which will cause the spaces in the string to be preserved.
- Arrays in serialized objects can optionally have an index attribute on each array element, which makes it possible to have gaps in arrays. If no index attribute is specified the array elements are assumed to be in order with the first index being 0. Indices have to be specified for all or none of the array elements. When using the Safir.Dob.Typesystem.Serialization API the index attribute will always be generated.
- Serialized objects can have a type attribute that specifies the exact type of the object. This is mostly useful when an object member or parameter array element needs to be of a subtype of the type specified in the dou file. E.g. if the type of the member is specified to be `BaseObject`, but you want to put an instance of `DerivedObject` in that member.

Example of object serialized to XML

```
<Capabilities.TestObj>
    <MyStringArray> <!-- indices are implicit, 0 and 1 -->
        <String>  bla bla </String> <!-- will be trimmed to "bla bla" -->
        <String xml:space="preserve">  bla bla </String> <!-- will not be trimmed -->
    </MyStringArray>
    <MyInt32Array> <!-- indices are explicit, all missing will be null -->
        <Int32 index="1">123</Int32>
        <Int32 index="3">456</Int32>
    </MyInt32Array>
    <MyFirstBaseObject> <!-- type expected to comply with dou file specification -->
        ...
    <MyFirstBaseObject>
    <MySecondBaseObject type="Capabilities.Derived"> <!-- type attr needed since type is a ↩
        subtype -->
        ...
    <MySecondBaseObject>
</Capabilities.TestObj>
```

Some notes on the JSON serialization:

- Null values in non-array attributes are always omitted to keep the JSON serialization small and readable. For arrays, since there is no explicit index attribute in JSON, null values in between two non-null values must be present to ensure correct indices. However consecutive null values at the end of an array can and should be omitted.
- JSON serialized objects must always have an attribute named "_douType" which specifies the type as it is stated in the dou file. For example `"_douType" : "Safir.Dob.NodeInfo"`. The rest of the attributes will be named exactly as they are in the corresponding dou file.

Example of object serialized to JSON

```
{
    "_douType" : "Capabilities.TestObj",
    "MyStringArray" : ["hello", "world"],
    "MyInt32Array" : [null, 123, null, 456],
         //Note: array is padded with null values up to array size
    "MyFirstBaseObject" : {
        "_douType" : "Capabilities.Base",
        ...
    }
    "MySecondBaseObject" : {
        "_douType" : "Capabilities.Derived",
        ...
    }
}
```

3.11.2.1 Porting "old" XML to "new" XML

Safir SDK Core versions prior to 5.0 used a different XML serialization format that did not follow any XML best practices. In 5.0 a new XML serialization was introduced, to make it easier to use third party tools to manipulate the XML, and to make it more like "good" XML.

For backward compatibility reasons and to make the transition to 5.0 as smooth as possible, Safir SDK Core can still read the old format, even though it is considered deprecated.

However, it is quite easy to convert old XML to new XML, since Safir SDK Core provides a script, *xml_convert.py*, that does just that. For example `xml_convert.py -d <path_to_douFiles> -o <converted_result_path> -r` will recursively convert all dou and dom files for you. The script is included in the installation packages, and can be found in the documentation directory.

To convert serialized objects that reside in databases etc, it might be easier to write a small program that deserializes your objects and then serializes them back to XML again. Those two steps will generate equivalent XML but in the new format. Actually such tool already exists, please don't hesitate to contact any Safir SDK Core developers for help and information.

3.11.3 The reflection interface [adv]

The reflection interface can be used to interrogate (and modify) anonymous Objects about their member names, types and values. This is something that not many applications should have to do, and therefore it is not described fully in this document.

But here are some hints:

- The Safir.Dob.Typesystem namespace contains a lot of functions for asking the typesystem about static information about types (e.g. what is the name and type of the third member of this type?).
- Safir.Dob.Typesystem has an ObjectFactory that allows applications to create an object straight from a TypeId.
- All Dob objects have GetMember(...) routines that return a ContainerBase for a specified member and array index. This can be casted to a container of the correct type, and thereafter manipulated.

- Very few of the routines in the reflection interface check for errors. If you call them with incorrect arguments the results are usually undefined. E.g. If you try to get the type of the 4th member of a type that has only 3 members you will probably get an undefined value as the type.

Use with care! or even better: *With great power comes great responsibility!*

Chapter 4

The distribution mechanisms

This chapter will give more details of how to use the distribution mechanisms of the Dob.

4.1 Consumers and Callbacks

Before learning about how to connect to the Dob we need to define a few basic concepts.

The first basic concept is a `Consumer`. A consumer is an interface, or an abstract base class, which represents a *role* or a set of functionality that an application can have. For example, the `MessageSubscriber` consumer is an interface that an application has to implement in order to be able to receive messages through a subscription.

The second basic concept is `Callbacks`. Each consumer has one or more callbacks, which are abstract methods that application has to implement. The MessageSubscriber consumer interface has one callback `OnMessage`, which is called by the Dob each time a message is received.

The third concept is `Dispatching`, which is the name of the strategy that the Dob uses to be able to call the callbacks. We will get to dispatching in a moment (Section 4.2.1), but for the moment all you need to know is that Dispatching is what causes the callbacks to be called.

4.2 Connections

For an application to talk to the Dob it needs a connection. There are in fact two kinds of connections; one that is called Connection and one that is called SecondaryConnection. In this document "connection" will be used when either kind of connection is implied, "primary connection" when Connection is referred to and "secondary connection" when a SecondaryConnection is referred to.

First we'll describe primary connections, and we'll cover secondary connections later in this chapter (Section 4.2.2).

To create a primary connection the application must create a Connection object and then call `Open` on it. The `Open` method takes a number of parameters:

- A connection name and a connection instance. These are used to uniquely identify the connection in the system. No two connections can have the same name and instance.
- A context, which specifies the data "universe" that the connection will operate in. A normal application that is not concerned with Replay should connect with context 0. See Chapter 14 for further details.
- A `StopHandler` consumer, which the Dob uses to tell the application to stop executing. Stop orders are described in Section 4.8.
- A `Dispatcher` consumer, which the Dob uses to tell the application that there is data available to it, which it will receive if it calls the `Dispatch` method. Dispatching is discussed in the next section.

When an application wants to close a connection it calls Close. It is not necessary to call Close before application shutdown, since the Connection destructor will do that automatically.

4.2.1 Dispatching

When a connection is opened a separate thread is started in the application. This thread, known as the *dispatch thread* listens to a shared memory event which signals that something has happened that the application needs to know about, e.g. a message that the application is subscribing to has arrived. When this event is triggered, the dispatch thread calls the `OnDoDispatch` callback of the `Dispatcher` consumer.

When this happens the application **must** notify (through an event or some other mechanism) the thread that the connection was started in that it should call `Dispatch` on the connection. When the Dispatch method is called the Dob will call all "pending" callbacks, e.g. `OnMessage` for received messages.

The reason for all this is to make things easier for the application developer as well as removing the need for the Dob connection to be thread safe (which would inevitably make it less efficient and slower).

The upshot is that all callbacks (except, of course, the `OnDoDispatch` one) are called *from the same thread that opened the connection*, which means that most applications should not have to worry at all about locking and having multiple threads.

Important: Do the thread switch as described above, or things will probably crash in surprising ways.

Figure 4.1 shows the dispatching sequence.

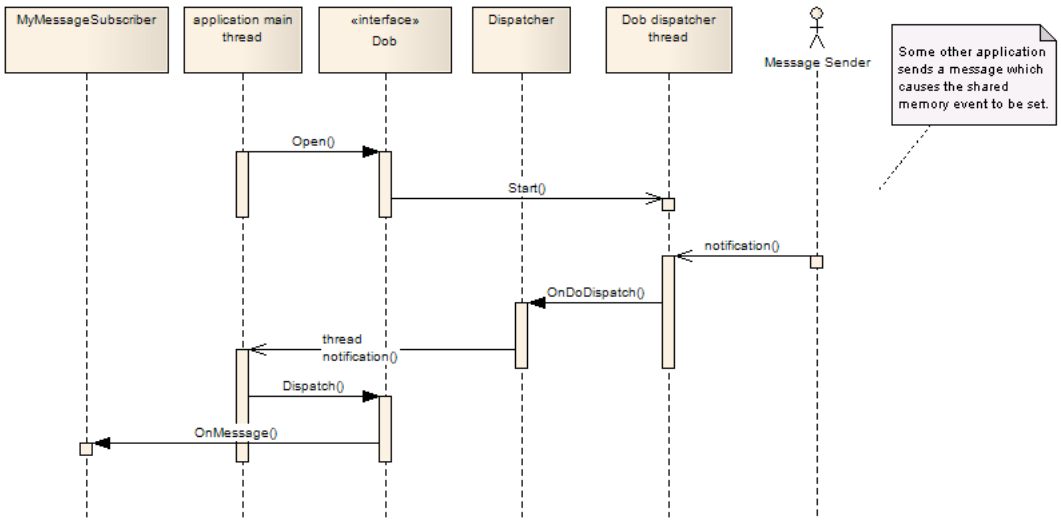

Figure 4.1: Dispatching sequence diagram

If you're using C++ and Boost.Asio or ACE the SDK provides Dispatcher classes that do the thread switch and calls Dispatch, see Section 11.2.

4.2.1.1 Breaking out of Dispatch "early"

As mentioned above, the `Dispatch` call will call *all* pending callbacks before returning. If there is a lot happening in a system this can potientially take a long time, and for example for applications that have requirements to do other tasks periodically this can cause trouble (e.g. missed deadlines).

To resolve this applications can, from within a callback, call `ExitDispatch`, which will cause the `Dispatch` function to return "as soon as possible". ExitDispatch will also cause a new event to be triggered, so that dispatching will be resumed automatically.

Tip
If you're using an ACE_Reactor in your application you will also need to call `max_notify_iterations(1)` on your rector to make the reactor handle your timers in a fair manner. See your ACE documentation for more information.

4.2.2 Secondary Connections

Whereas a primary connection is a real connection to the Dob, a secondary connection is only a "handle" to a primary connection.

Each application needs one primary connection (at least, see Section 12.7). This connection is the one that the Dob knows about and through which data is dispatched. A secondary connection is used by other parts of the application to "attach" to the primary connection.

The reason for providing this functionality is so that for example a C++ library used by a C# application can use the main programs connection. This would otherwise be impossible, since passing the connecton object over the C# to C++ interface is not possible. Instead modules can attach to the main program's connection using a secondary connection.

When creating a secondary connection you can either specify a connection name and instance to get a specific connection instance, or not specify anything at all, in which case you will be given the first connection that was created in the current thread.

4.3 Using the mechanisms

The three different distribution mechanisms of the Dob are used in different ways and require different things from the user application.

But before attacking the mechanisms themselves we need to talk about addressing and something called proxies.

4.3.1 Addressing

The Dob uses logical addressing for its distribution mechanisms, which means that you use a logical name for which applications to send messages to, for example.

There are two concepts that we'll introduce here (they're described in greater detail later), *channels* and *handlers*.

A *channel* is like an FM radio frequency; when you send, you're sending on a particular frequency, and when you're receiving, you're listening to a particular frequency. When you send messages, you send them on a channel, and when you subscribe to messages, you subscribe to a particular channel (or all channels, which is where the analogy isn't quite as good any more).

A *handler* is the concept that is used for addressing requests (both service and entity requests). A handler is more like a telephone number where only one person can have a particular telephone number (and has to register with the phone company to have it), but anyone can phone that person, as long as they know the number. To be able to receive service and entity requests an application has to register as a handler for that entity or service with the Dob. Then any application can send requests to it, as long as it knows the handler id (the "telephone number" to the handler).

Also, to be allowed to *own entity instances* an application has to register an entity handler for that entity type. Entity instances are the *only* thing that it is possible to *own*, and this document tries to only use "own" in that sense. Handlers are registered, not owned, entities are not owned (as opposed to entity instances), but there can be entity handlers registered, and so on.

4.3.2 Proxies

There are quite a few different callbacks for different kinds of data, and most of them take some kind of *proxy* as an argument. The proxies are really only a container for a whole bunch of arguments that the user might need inside the callback, but instead of passing all the arguments separately they are collected into one proxy. This simplifies things for the application, which need only care about the parts of the proxy that it needs, and it makes it easier to add more information in a callback in the future, should there be a need to, without breaking the interfaces.

Not all methods on the proxies are applicable in all situations, for example you cannot call `GetPrevious` on an `EntityProxy` obtained in an `OnNewEntity`, since there is no previous version of an entity that was just created, but in `OnUpdatedEntity`

or `OnDeletedEntity` calling `GetPrevious` will give you the version of the entity that you got in the previous subscription response for that instance.

Internally the proxies contain direct references into the Dob shared memory, which means that there are some constraints on how you can/should use them. Mainly, *you're not allowed to copy a proxy*, and *you're not allowed to keep a proxy*! The rationale for the second is that keeping it would also keep the data in the shared memory, which could cause the Dob to run out of shared memory, and the rationale for the first constraint is to make it more difficult to keep the proxy.

Proxies are (with a couple of exceptions) received as parameters to callbacks. After returning from the callback the shared memory that the proxy refers to is released, and the proxy therefore becomes invalid. If you try to use a proxy obtained through a callback after returning from that callback an exception will be thrown!

Of course it is possible to pass proxies to other methods, as long as the proxy is not copied. Passing it as a const-reference (`const EntityProxy& proxy`) works very well in C++.

4.3.3 Messages

Figure 4.2 shows a sequence diagram of how Dob Messages are handled.

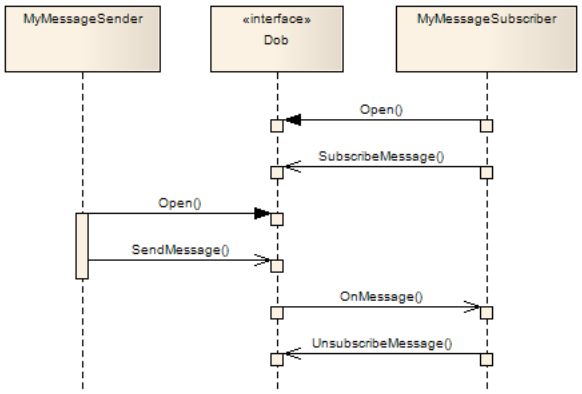

Figure 4.2: Sequence diagram for messages

For messages there are two different roles involved. The Message Sender sends messages and the Message Subscriber receives messages. The subscriber needs to set up the subscription, and subsequently it will receive messages in its `OnMessage` callback function whenever a Message Sender sends a message.

4.3.3.1 Channels

For every message type there can be several message channels. A subscriber can subscribe to all channels, or to only a specific channel. A sender has to send on a specific channel.

Channels are identified by the class *Safir.Dob.Typesystem.ChannelId*, which is a hashed type as described in Section 3.1.2.

Most of the time only one channel is needed, and then the default channel should be used. The default constructor for ChannelId creates a default channel ChannelId.

The constant ChannelId::ALL_CHANNELS is used to identify *all channels*, and can be used to subscribe to all channels for a certain message type.

It is worth noting that the channel is all the addressing there is for messages, i.e. it is completely transparent to the sender and receiver whether they reside on the same or different computers in the system.

4.3.3.2 Subscribing to messages

Two steps are necessary to receive messages. The first is to implement the Safir.Dob.MessageSubscriber interface:

Message subscriber class declaration

```
class VehicleMessageSubscriber : public Safir::Dob::MessageSubscriber
{
public:
    void OnMessage(const Safir::Dob::MessageProxy messageProxy) override;
};
```

The second step is to start the subscription by calling the `SubscribeMessage` method on the Dob connection object. This method takes a TypeId (the message type to subscribe to), a ChannelId (the channel to listen to) and a pointer to a class that implements the MessageSubscriber interface.

Subscribing to a message

```
m_connection.SubscribeMessage
        (Capabilities::Vehicles::VehicleMsg::ClassTypeId,
         Safir::Dob::Typesystem::ChannelId::ALL_CHANNELS,
         this);
```

This subscribes to VehicleMsg (and all messages that inherit from VehicleMsg) on all channels, and the `OnMessage` callback will be called by the Dob for each received VehicleMsg.

If you do not want to include subclasses in your subscription, there is an overload to the `SubscribeMessage` method that allows you to specify whether or not to include subclasses. Default behaviour of most applications should be to include subclasses.

Handling OnMessage callback

```
void VehicleMessageSubscriber::OnMessage
                    (const Safir::Dob::MessageProxy messageProxy)
{
    Capabilities::Vehicles::VehicleMsgPtr vehicle =
        std::static_pointer_cast
        <Capabilities::Vehicles::VehicleMsg>(messageProxy.GetMessage());

    ... do something with vehicle ...
}
```

See Section 3.10.4 for more information on the casting in the example above.

To cancel a subscription you call the `UnsubscribeMessage` method on the Dob connection object. (It is not necessary to do this before closing a connection or before the application shuts down, the Dob will handle that automatically when the connection is destroyed/closed.)

Removing a message subscription (unsubscribing)

```
m_connection.UnsubscribeMessage
    (Capabilities::Vehicles::VehicleMsg::ClassTypeId,
     Safir::Dob::Typesystem::ChannelId::ALL_CHANNELS,
     this);
```

A note on combinations of subscribe and unsubscribe: If you subscribe to ALL_CHANNELS, but unsubscribe the default channel you will receive all VehicleMsg:es except those sent on the default channel. You can also do things like subscribe to VehicleMsg including subclasses, and then do an unsubscribe to VehicleMsg not including subclasses, which would make you only receive subclasses to VehicleMsg, but not VehicleMsg itself. These rules apply *per consumer*, so subscriptions set up with one consumer are not affected by unsubscribes made with another consumer.

4.3.3.3 Sending messages

To send messages it is necessary to implement the Dob interface *MessageSender*. The MessageSender interface is used to manage overflow situations when sending messages.

Message sender class declaration

```
class VehicleMessageSender : public Safir::Dob::MessageSender
{
public:
    void OnNotMessageOverflow() override;
};
```

Messages are sent with the `Send` function in the Safir.Dob.Connection class. This function takes the message as a parameter and a pointer to the MessageSender interface.

When an overflow occurs the `Send` function will throw an exception (Safir.Dob.OverflowException). This means that the applications message out queue is full, and that the application should not try to send any more messages until told otherwise (and the message that was passed to the Send function was not sent). When the message out queue is no longer full the Dob will call the `OnNotMessageOverflow` method in the MessageSender passed in the send call that failed, which means that it is now okay to start sending messages again.

Sending a message

```
Capabilities::Vehicles::VehicleMsgPtr msg =
                            Capabilities::Vehicles::VehicleMsg::Create();
msg->MessageText() = L"Something happened to a vehicle!!!";
try
{
    m_connection.Send(msg, Safir::Dob::Typesystem::ChannelId(), this);
}
catch (const Safir::Dob::OverflowException &)
{
    // Doh! I got an exception so I have to remember the message and
    // send it again after I have received an OnNotMessageOverflow
    // callback
}
```

4.3.4 Services

Figure 4.3 contains a sequence diagram for how services are handled.

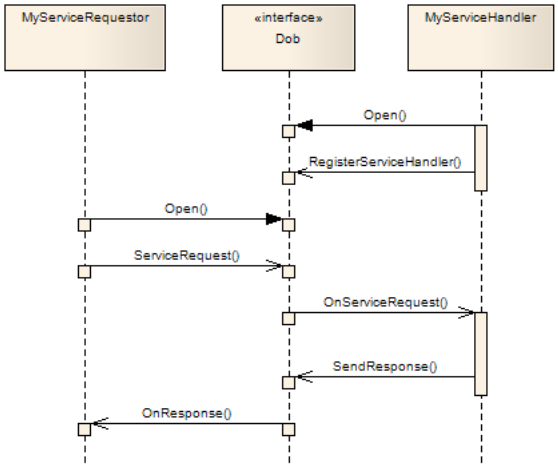

Figure 4.3: Sequence diagram for services

For services there are two different roles involved. The ServiceHandler is the component implementing the service and the Requestor is the component using the service. The Handler needs to register the service and then the Requestor can send requests to it.

4.3.4.1 Handlers

For every service type there can be several service handlers. A service handler has to be registered with a specific handler id, and a requestor has to send service requests to a specific handler, identified by the handler id.

Handler ids are identified by the class `Safir.Dob.Typesystem.HandlerId`, which is a hashed type as described in Section 3.1.2.

For most services there is only one handler, and then the default handler should be used. The default constructor for HandlerId creates a default handler identifier.

There is also a constant `HandlerId::ALL_HANDLERS`, which can be used for subscribing to registered handlers, as described Section 4.3.7. It is considered an error to use `ALL_HANDLERS` for either registration or sending requests.

The HandlerId is all the addressing that is needed. The request will be sent to the registered handler regardless of whether it is located on the same computer in the system or on a different one.

4.3.4.2 Registering a service handler

Two things need to be done to register a service handler. The first is to implement the Dob ServiceHandler interface. The ServiceHandler consumer interface allows the owner to receive service requests and be notified if some other application *overregisters* the handler.

Overregistration occurs when another application registers the same handler for the same service. This will cause the current handler to loose the registration, which is known as an overregistration.

Service handler class declaration

```
class VehicleServiceHandler : public Safir::Dob::ServiceHandler
{
public:
    void OnServiceRequest
        (const Safir::Dob::ServiceRequestProxy serviceRequestProxy,
         Safir::Dob::ResponseSenderPtr         responseSender) override;

    void OnRevokedRegistration
        (const Safir::Dob::Typesystem::TypeId     typeId,
         const Safir::Dob::Typesystem::HandlerId& handlerId) override;
};
```

The second thing that needs to be done is to call the RegisterServiceHandler method on the Dob connection object.

Registering a service handler

```
m_connection.RegisterServiceHandler
    (Safir::BearingDistanceService::ClassTypeId,
     Safir::Dob::Typesystem::HandlerId(),
     this);
```

After this call is complete the service handler is registered, with the default handler id. If another application had previously registered the same handler for the same type it will now get a call to `OnRevokedRegistration`, to tell it that someone has *overregistered* its handler.

The above registration mechanism is "hard" in the way that it overregisters previous registerers. There is an alternative, *pending registration* which is described in Section 4.7.

It is possible to subscribe to handler registrations, so an application might at this point be told that the service handler is registered. See Section 4.3.7.

The `OnServiceRequest` callback will now be called whenever the application receives a service request.

Every service request must be responded to with an object that inherits from the Safir::Dob::Response class. The response is sent using the ResponseSender that is received in the `OnServiceRequest` callback.

Handling the OnServiceRequest callback

```
void VehicleServiceHandler::OnServiceRequest
    (const Safir::Dob::ServiceRequestProxy serviceRequestProxy,
     Safir::Dob::ResponseSenderPtr         responseSender)
{
    Capabilities::Vehicles::BearingDistanceServicePtr bearingService =
        std::static_pointer_cast
            <Capabilities::Vehicles::BearingDistanceService>
            (serviceRequestProxy.GetRequest());

    ... validate request contents ...

    if (bearingService is a correct request)
    {
        ... do something ...

        responseSender->Send(Safir::Dob::SuccessResponse::Create());
    }
    else
    {
        // Send an error response
        Safir::Dob::ErrorResponsePtr response =
           Safir::Dob::ErrorResponse::CreateErrorResponse
               (Safir::Dob::ResponseGeneralErrorCodes::SafirReqErr,
                L"The request didn't contain all expected data");
        responseSender->Send(response);
    }
}
```

It is possible to keep the `responseSender` "for later" if it is not possible to send the response immediately (e.g. if the application has to wait for a database query to complete before sending the response). It is considered a programming error to not send a response to a request (and the ResponseSender destructor will try to alert you to that fact if you fail to use it).

To stop handling a service you call the UnregisterHandler method. (It is not necessary to do this before closing a connection or before the application shuts down, the Dob will handle that automatically when the connection is destroyed/closed.)

Unregistering the handler

```
m_connection.UnregisterHandler(Safir::BearingDistanceService::ClassTypeId,
                               Safir::Dob::Typesystem::HandlerId()));
```

4.3.4.3 Response types

There are two base classes for responses, `Safir.Dob.SuccessResponse` and `Safir.Dob.ErrorResponse` (they in turn inherit from `Safir.Dob.Response`, but no other responses should do that). If a custom response is needed (e.g. responding with the result of a database query, or with error information from a database query), create your own dou-file, inheriting from either of these two.

There is one more predefined error response, `Safir.Dob.ErrorListResponse`, which allows many errors to be specified, typically used to report errors for individual members in the request.

The parameter file `Safir.Dob.ResponseGeneralErrorCodes.dou` contains some predefined error codes that can be used for the `Code` field in error responses.

4.3.4.4 Sending service requests

To send a service request you have to have a class that implements the Requestor consumer interface and create a request and send it using the `ServiceRequest` method in the Dob connection object.

Request sender class declaration

```
class VehicleServiceRequestSender : public Safir::Dob::Requestor
{
public:
    void OnResponse(const Safir::Dob::ResponseProxy responseProxy) override;

    void OnNotRequestOverflow() override;
};
```

The `OnResponse` callback will be called with the response to your request, and the `OnNotRequestOverflow` callback will be called when it is okay to start sending requests again after an overflow has occurred.

Sending a request is done like this:

Sending a service request

```
Capabilities::Vehicles::VehicleServicePtr request =
    Capabilities::Vehicles::VehicleService::Create();

request->ServiceText() = L"Do something!";
try
{
    Safir::Dob::RequestId reqId = m_connection.ServiceRequest
        (request,
         Safir::Dob::Typesystem::HandlerId()
         this);
}
catch (const Safir::Dob::OverflowException &)
{
    //Doh! I got an exception! Better tell the operator to
    //press that button again in a little while.
}
```

The RequestId that is returned by the `ServiceRequest` call is a unique identifier for that request. When the response is received the request id for the request that generated the response can be gotten from the ResponseProxy using the `GetRequestId()` method. This is to allow applications to know which request generated which response.

If, for some reason, no response is sent from the service handler a time-out response is generated by the Dob to ensure that the sender of the request always receives a response.

If the handler is not registered the Dob will send an error response that indicates exactly that.

4.3.5 Entities

Entities are objects that are stored in shared memory and distributed between system nodes by the Dob. It is also possible to send requests on them, to create new entities, to update or delete them. These requests are very similar to service requests, although they are tailored to be used specifically on "things" rather than services. If you haven't read the previous section, on services, do so now, since this section assumes that you are familiar with how services and service handlers work.

Figure 4.4 shows a sequence diagram that describes (a shortened version of) the Entity Handling pattern describing how entities are handled in a Safir system.

When handling entities there are three roles involved. *Entity Handler* is an application registered as a handler of the entity (and is allowed to *own* entity instances), *Entity Subscriber* is an application that subscribes to the entity and *Entity Requestor* is an application sending a request to update the entity. At start-up Entity Subscriber sets up a subscription and Entity Handler registers an entity handler. When Entity Requestor sends a request the Dob sends it to Entity Handler. Entity Handler then validates the

request, creates, updates or deletes the entity and sends a response to Entity Requestor containing the status of the request. The Entity Subscriber gets notified of the new entity.

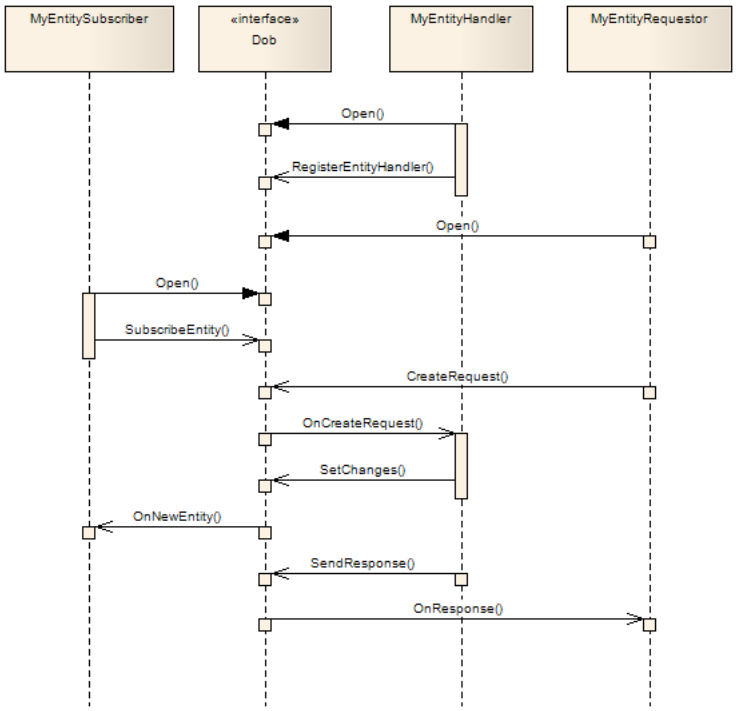

Figure 4.4: Sequence diagram for entities

This section will first describe "simple" entity handling, such as is shown in Figure 4.4, and later go on to describe more advanced concepts, such as *persistence* and *injections*.

First a note on application design: You are not supposed to keep internal copies of entities in your program. It is often better to use Read (see Section 4.3.5.9) when needed. If you often need to locate a particular entity instance depending on its contents you may want to maintain some kind of lookup-table in your program, to be able to find the correct instance quickly. This is better than iterating over all instances to find the correct one.

4.3.5.1 Handlers

For entities there are handlers, just as there are handlers for services. If you haven't already, please read the chapter on Services, and specifically the bit about service handlers.

An entity handler is registered to handle entity create, delete and update requests, and handlers are also allowed to own entity instances.

Use the default handler if there is only going to be one handler for a certain entity type.

When sending Create requests you need to specify which handler should create the instance. When sending an Update or Delete request the request will be automatically sent to the handler of the specified instance, so there is no need to explicitly specify the handler.

4.3.5.2 InstanceId and EntityId

For every entity type there can be many entity instances. Each instance within an entity type is identified by a the class `Safir.Dob.Typesystem.InstanceId`, which is a hashed type as described Section 3.1.2.

The class `Safir.Dob.Typesystem.EntityId` is simply an entity TypeId coupled with an InstanceId, allowing it to be used to uniquely identify an entity instance (remember that there can be instances of different entity types with the same instance id).

There are several ways of generating an instance id:

1. Using a 64 bit integer
2. Using a string (InstanceId is a hashed type, so the string will give a unique 64 bit integer)
3. Randomly (InstanceId::GenerateRandom() will give a random instance id).

The instance id strategy should be chosen very carefully, to ensure that entity instances can be uniquely identified. An example: For Vehicles the instance id could be generated from the "Callsign" string member. This ensures that there can be no two vehicles with the same callsign, and that two applications processing information on the same vehicle (with the same callsign) will be referring to the same entity. Another example is the ProcessInfo entity that the Dob provides (shows processes that have a connection to the Dob) which uses the process PID as instance id.

Using random numbers is another good strategy for generating a unique instance id. Remember that the ids are 64bit integers, so the likelihood of collision when using random numbers is negligible (1 in 10^{19}).

One other aspect of InstanceIds is who gets to decide which instance id to use, the Requestor or the Handler? For some entity types it might make sense to include an InstanceId in create requests, and for others it might make more sense not to include an InstanceId in the create request, but rather be told of which instance was created in the response. For this purpose all entity handlers are registered with a `Safir.Dob.InstanceIdPolicy`, which has two possible values `HandlerDecidesInstanceId` and `RequestorDecidesInstanceId`.

HandlerDecidesInstanceId

As the name states, in this case the handler decides which instance id to use for creating new objects. As a response to the CreateRequest it should send a `Safir.Dob.EntityIdResponse` to indicate success.

A Requestor to this handler **cannot** specify an instance id in its create request. If it tries to an exception will be thrown.

RequestorDecidesInstanceId

The requestor decides which instance it wants, and includes the instance id in the CreateRequest.

The handler **must** then create that instance, or if it cannot it should send an ErrorResponse. A handler that is registered with RequestorDecidesInstanceId is *not allowed to choose another instance than what is specified in the request*!

4.3.5.3 Registering an entity handler

Again there is a consumer interface to implement, `EntityHandler`, which allows the application to receive Create, Update and Delete requests. The interface also allows the applications to be told of overregistrations, just as for ServiceHandler registrations.

Entity handler class declaration

```
class VehicleHandler :
    public Safir::Dob::EntityHandler
{
public:
    void OnCreateRequest
         (const Safir::Dob::EntityRequestProxy entityRequestProxy,
          Safir::Dob::ResponseSenderPtr        responseSender) override;

    void OnUpdateRequest
         (const Safir::Dob::EntityRequestProxy entityRequestProxy,
          Safir::Dob::ResponseSenderPtr        responseSender) override;

    void OnDeleteRequest
         (const Safir::Dob::EntityRequestProxy entityRequestProxy,
          Safir::Dob::ResponseSenderPtr        responseSender) override;
```

```
    void OnRevokedRegistration
         (const Safir::Dob::Typesystem::TypeId     typeId,
          const Safir::Dob::Typesystem::HandlerId& handlerId) override;
};
```

The `OnRevokedRegistration` callback works exactly as for Services, as does the `responseSender`.

Applications register an entity handler by calling the `RegisterEntityHandler` method in the Dob connection object. This method takes an EntityHandler (like the one declared above), a HandlerId, an InstanceIdPolicy and an entity TypeId.

Registering an entity handler

```
m_connection.RegisterEntityHandler
    (Capabilities::Vehicles::Vehicle::ClassTypeId,
     Safir::Dob::Typesystem::HandlerId(),
     Safir::Dob::InstanceIdPolicy::HandlerDecidesInstanceId,
     this);
```

After the call to RegisterEntityHandler the application can immediately use the handler to set entity instances, and other applications can send entity requests to the EntityHandler.

To stop handling an entity you call the UnregisterHandler method. (It is not necessary to do this before closing a connection or before the application shuts down, the Dob will handle that automatically when the connection is destroyed/closed.)

Unregistering an entity handler

```
m_connection.UnregisterHandler(Capabilities::Vehicles::Vehicle::ClassTypeId,
                               Safir::Dob::Typesystem::HandlerId()));
```

4.3.5.4 Handling Create Requests

The method OnCreateRequest is used by the Dob to notify the entity handler of a create request. The arguments to the method is an EntityRequestProxy which contains the request data and metadata, and a ResponseSender which must be used to send the mandatory response to the request.

Handling a create request

```
void VehicleHandler::OnCreateRequest
     (const Safir::Dob::EntityRequestProxy entityRequestProxy,
      Safir::Dob::ResponseSenderPtr        responseSender);
{
    Capabilities::Vehicles::VehiclePtr vehicle =
        std::static_pointer_cast<Capabilities::Vehicles::Vehicle>
            (entityRequestProxy.GetRequest());

    // Callsign and Vehicle ID is required for a create
    if (vehicle->Callsign().IsNull() ||
        vehicle->VehicleKey().IsNull())
    {
        //Set error
        Safir::Dob::ErrorResponsePtr response =
            Safir::Dob::ErrorResponse::CreateErrorResponse
            (Safir::Dob::ResponseGeneralErrorCodes::SafirMissingMember(),
             "Callsign and VehicleKey are mandatory elements");
        responseSender->Send(response);
        return;
    }

    //decide a good instance id (we registered as handler decides)
    const Safir::Dob::Typesystem::InstanceId instance(vehicle.VehicleKey())

    // Create vehicle
```

```
    m_connection.SetChanges(vehicle,
                            instance,
                            Safir::Dob::Typesystem::HandlerId());

    // Create response
    const Safir::Dob::EntityIdResponsePtr response =
        Safir::Dob::EntityIdResponse::CreateResponse
        (Safir::Dob::Typesystem::EntityId(vehicle.GetTypeId(),instance));
    responseSender->Send(response);
}
```

Note the use of the EntityIdResponse class for the response, which must be used as the success response when HandlerDecidesInstanceId is used as the InstanceIdPolicy for the handler. See Section 4.3.5.2.

4.3.5.5 Handling Update Requests

The method `OnUpdateRequest` is used by the Dob to notify the entity handler of an update request on an entity owned by that handler. The arguments to the method is an EntityRequestProxy which contains the request data and metadata, and a ResponseSender which must be used to send the mandatory response to the request.

Note that the Dob guarantees that the update request is on an existing entity. There is no need to check for the existence of the entity inside `OnUpdateRequest`.

Handling an update request

```
void VehicleHandler::OnUpdateRequest
    (const Safir::Dob::EntityRequestProxy entityRequestProxy,
     Safir::Dob::ResponseSenderPtr        responseSender);
{
    Capabilities::Vehicles::VehiclePtr vehicle =
        std::static_pointer_cast
            <Capabilities::Vehicles::Vehicle>
            (entityRequestProxy.GetRequest());

    // Callsign
    if(vehicle->Callsign().IsChanged())
    {
        if(vehicle->Callsign().IsNull() ||
            Callsign has an illegal value)
        {
            //Set error
            Safir::Dob::ErrorResponsePtr response =
                Safir::Dob::ErrorResponse::CreateErrorResponse
                    (Safir::Dob::ResponseGeneralErrorCodes::SafirReqErr(),
                     "Illegal value for callsign");
            responseSender->Send(response);
            return;
        }
    }

    ... check all other members that are changed ...

    // Update vehicle
    m_connection.SetChanges(vehicle,
                            entityRequestProxy.GetInstanceId(),
                            Safir::Dob::Typesystem::HandlerId());

    // send a success response
    responseSender->Send(Safir::Dob::SuccessResponse::Create());
}
```

Note the use of `SetChanges` instead of `SetAll` above. `SetChanges` will take the members that have its change flag set in the passed object and merge them into the existing entity that is stored in the Dob shared memory.

4.3.5.6 Handling Delete Requests

The method `OnDeleteRequest` is used by the Dob to notify the entity handler of a delete request on an entity owned by that handler. The arguments to the method is an EntityRequestProxy which contains the request data and metadata, and a ResponseSender which must be used to send the mandatory response to the request.

Note that the Dob guarantees that the delete request is on an existing entity. There is no need to check for the existence of the entity inside `OnDeletedRequest`.

Handling a delete request

```
void VehicleHandler::OnDeleteRequest
    (const Safir::Dob::EntityRequestProxy entityRequestProxy,
     Safir::Dob::ResponseSenderPtr        responseSender);
{
    m_connection.Delete(entityRequestProxy.GetEntityId(),
                        Safir::Dob::Typesystem::HandlerId());

    // send a response
    responseSender->Send(Safir::Dob::SuccessResponse::Create());
}
```

Note that it is also possible to delete all entity instances owned by a handler by calling `DeleteAllInstances` (but obviously this is not something that should be done as a result of a DeleteRequest, which is something that should operate on only one object).

4.3.5.7 Owning entity instances

Apart from handling requests on entities the entity handler is of course allowed to change an entity whenever it wants to. Generally this should be done using calls to `SetChanges`, rather than `SetAll`. The reason for this is the *injection timestamps* as described in Section 13.2.2

4.3.5.8 Subscribing to entities

Once more there is a consumer interface to be implemented to subscribe to entities, `EntitySubscriber`, which provides callbacks when an entity is created, updated or deleted.

Entity subscriber class declaration

```
class VehicleSubscriber : public Safir::Dob::EntitySubscriber
{
    void OnNewEntity
        (const Safir::Dob::EntityProxy entityProxy) override;

    void OnUpdatedEntity
        (const Safir::Dob::EntityProxy entityProxy) override;

    void OnDeletedEntity
        (const Safir::Dob::EntityProxy entityProxy,
         const bool /*deprecated*/) override;
};
```

The subscription is started by a call to the `SubscribeEntity` method on the Dob connection object.

Subscribing to an entity

```
m_connection.SubscribeEntity
    (Capabilities::Vehicles::Vehicle::ClassTypeId, this);
```

The subscription will be to all instances of that entity, including subclasses. There are a couple of overloads to the `SubscribeEntity` method that the subscriber can use if it wants to exclude subclasses, or just subscribe to a single entity instance.

After the call to `SubscribeEntity` the Dob will start calling the `OnNewEntity` callback for all entities that already existed when the subscription was set up, and thereafter it will call the callbacks whenever an entity is created, updated or deleted.

Handling the OnNewEntity callback

```
void VehicleSubscriber::OnNewEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    Capabilities::Vehicles::VehiclePtr vehicle =
        std::static_pointer_cast<Capabilities::Vehicles::Vehicle>
            (entityProxy.GetEntity());

    if (!vehicle->Callsign().IsNull())
    {
        ... Process Callsign here ...
    }
    else
    {
        ... do something else ...
    }

    ... process other fields ...
}
```

See Section 3.10.4 for more information on what `std::static_pointer_cast` is.

Handling the OnUpdatedEntity callback

```
void VehicleSubscriber::OnUpdatedEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    Capabilities::Vehicles::VehiclePtr vehicle =
        std::static_pointer_cast
            <Capabilities::Vehicles::Vehicle>
            (entityProxy.GetEntityWithChangeInfo());

    if (vehicle->Callsign().IsChanged() &&
        !vehicle->Callsign().IsNull())
    {
        const std::wstring callsign = vehicle->Callsign();
        ... process Callsign ...
    }

    ... process other fields ...
}
```

Note the use of `GetEntityWithChangeInfo()` and the change flag on the callsign member to only look at it if it changes. See Section 4.6 for more information on how to interpret change flags. It is also possible to get hold of the *previous* entity that was seen by a subscription. Use `entityProxy.GetPrevious()` to get information about the previous subscription response.

Handling the OnDeletedEntity callback

```
void VehicleSubscriber::OnDeletedEntity
    (const Safir::Dob::EntityProxy entityProxy,
     const bool /*deprecated*/)
```

```
{
    ... do something ...
}
```

If this class had been used to subscribe to several entities it can use `entityProxy.GetTypeId()` to work out which way to handle the remove.

It is also possible to use `entityProxy.GetPrevious()` to get hold of information about the entity that was deleted, such as actually looking at the entity like it was when it was previously dispatched to the subscriber.

Please ignore the `deprecated` flag, it will be removed in a future version of Safir SDK Core (see Appendix C if you're curious).

To stop subscribing to an entity the application calls the `Unsubscribe` method in the Dob connection object. (It is not necessary to do this before closing a connection or before the application shuts down, the Dob will handle that automatically when the connection is destroyed/closed.)

Removing an entity subscription (unsubscribing)

```
m_connection.UnsubscribeEntity
    (Capabilities::Vehicles::Vehicle::ClassTypeId,
     this);
```

Note: You should read the section on Section 4.6 for more information about the change flags during entity subscriptions. The change flags are a very powerful tool to allow the subscriber to determine what has changed during an entity subscription.

4.3.5.9 Reading an entity

Sometimes the information in an entity is needed directly rather than as a part of a subscription (e.g. a Vehicle might contain an EntityId reference to another object, which can be *read* when an update to the Vehicle occurs). For these occasions the Dob provides a method to synchronously read an entity, `Read`.

The `Read` operation returns an EntityProxy, and as described above (Section 4.3.2) there are some constraints on using proxies, but in this case it is not passed as a parameter into a callback, but as a return value from a method. Since it is illegal to "keep" an EntityProxy this means that a user of `Read` has certain obligations:

Do not keep the proxy (it is possible, and it will appear to work, but remember that it will lock resources in the Dob).

If you're using C#, you *must* dispose of the proxy before letting go of your reference to it. The proxy is an `IDisposable` object, and C# best practice states that `IDisposable` objects shall always be disposed explicitly. The easiest way to do this in an exception safe way is to use the `using` construct:

Using the using construction with EntityProxy

```
using (Safir.Dob.EntityProxy ep = m_connection.Read(anEntityId))
{
    ... do something with the read entity ...
}
```

The advantage of using the above construct is that the `Dispose` method of EntityProxy is guaranteed to be called regardless of whether an exception is thrown or a "return" is done inside the curly brackets.

If you forget to do this the garbage collector will try to dispose the proxy at some later time, at which time the Dob will detect that the proxy was not correctly disposed, and generate an error (as of this writing a dialog will pop up, and the program will terminate).

4.3.5.10 Iterating over entities

The Dob also provides a way of iterating over all instances of a certain entity type (including or excluding subclass instances). The way to do this differs slightly from language to language (the goal is to use the native iteration patters for each language).

Before looking at how this is done, a **word of warning**: Iterating over lots of instances can be quite time-consuming (e.g. imagine a system that can have 10000 vehicles, and every time an update occurs to one of them, a badly written application iterates over

all of them), so in cases where an application wants to do it frequently it is preferable to maintain some kind of lookup table instead, using entity subscriptions. Or maybe using a better algorithm for choosing instance numbers, so that the iteration can be avoided.

Iterating over all Vehicles (including subclasses) in C++:

Iterating over entities in C++

```
for (Safir::Dob::EntityIterator it = m_connection.GetEntityIterator
         (Capabilities::Vehicles::Vehicle::ClassTypeId,true);
     it != Safir::Dob::EntityIterator(); ++it)
{
    ... do something with it ...
}
```

Dereferencing `it` will give an entity proxy.

Note that the default constructor for EntityIterator creates an "end" iterator, pointing "one past" the last instance. This is a pattern that was inspired by the `std::filesystem` directory iterators.

Iterating over all Vehicles (including subclasses) in C#:

Iterating over entities in C#

```
foreach (Safir.Dob.EntityProxy entityProxy in
         m_connection.GetEntityEnumerator
            (Capabilties.Vehicles.Vehicle.ClassTypeId,true))
{
    ... do something with entityProxy ...
}
```

An aside: In this case it is not necessary to explicitly call `Dispose` or somehow use `using`, since that is taken care of by the loop itself.

4.3.5.11 Sending entity requests

To be able to send requests on entities it is necessary to implement the `Requestor` consumer interface, just like when sending service requests (so have a look at that section, because that example will not be repeated here...).

There are four methods in the Dob connection object that are used for sending requests on entities; two variants of `CreateRequest`, one `UpdateRequest` and one `DeleteRequest`. All these methods take (among other things) a Requestor so that the Dob has somewhere to send the response.

All the request methods return a unique RequestId to allow the Requestor to know which response was generated by which request in the `OnResponse` callback.

Sending an update request.

```
Capabilities::Vehicles::VehiclePtr request =
    Capabilities::Vehicles::Vehicle::Create();

 ... set fields in request ...

Safir::Dob::RequestId reqId =
    m_connection.UpdateRequest(request, myVehicleInstance, this);
```

When the handler has processed the request it sends a response back telling the sender the status of the request. The Dob sends this response to the OnResponse callback in the Requestor interface. If the response was successful then the change of the entity is also sent if the requestor has a subscription. There is no predetermined order between the response and subscription messages and no assumptions should be made on what is received first.

If, for some reason, no response is sent from the owner then a time-out reply will be generated to ensure that the sender of the request always receives a response.

Handling the OnResponse callback

```
void VehicleSubscriber::OnResponse
    (const Safir::Dob::ResponseProxy responseProxy)
{
    if (responseProxy.IsSuccess())
    {
        .... handle success response, maybe ...
        return
    }

    ... handle the error ...
}
```

4.3.5.12 Some notes on the CreateRequest methods

There are two variants of the `CreateRequest` method in the Dob connection class:

The two variants of CreateRequest

```
Safir::Dob::RequestId CreateRequest
    (const Safir::Dob::EntityPtr&                request,
     const Safir::Dob::Typesystem::HandlerId&    handlerId,
     Safir::Dob::Requestor* const                requestor) const;

Safir::Dob::RequestId CreateRequest
    (const Safir::Dob::EntityPtr&               request,
     const Safir::Dob::Typesystem::InstanceId& instanceId,
     const Safir::Dob::Typesystem::HandlerId&  handlerId,
     Safir::Dob::Requestor* const              requestor) const;
```

The second one is (obviously, since it contains an InstanceId parameter) only possible to use when the handler uses `RequestorDecide` but contrary to what might be expected the first one can be used for **both** types of handlers. If the handler is `RequestorDecidesInst` an instance id will be randomly generated. This behaviour can be used when the requestor doesn't care what instance id gets generated.

4.3.6 Entity Persistence

The Dob has an area of functionality known as *injections*, that allows entities to be injected from an *external source*. The most common use for this is *persistence*, which is what this section describes. The purpose of persistence is to make entity instances survive an application or system restart.

This functionality is configured using the properties Safir.Dob.InjectionProperty and Safir.Dob.InjectionOverrideProperty. The possible values are:

- `None`
- `SynchronousVolatile`
- `SynchronousPermanent`
- `Injectable`

`None` is the default, where nothing is remembered since the previous registration. The other values deserve their own subsections (although `Injectable` will not be described in this chapter, but in Chapter 13).

But first a word on why they're called *synchronous*; Persistence injections are known as synchronous injections since they will always occur synchronously, at application startup (or rather, when registration occurs), whereas the last kind of injection can occur asynchronously throughout the lifetime of an entity instance.

4.3.6.1 SynchronousVolatile

`SynchronousVolatile` persistence is useful to make entities survive when an application restarts, but when the whole system does not. For example if an application crashes and is restarted automatically, or, in a system with redundant nodes, a node is shut down and the applications restarted on another node.

When an entity handler is unregistered, without the entity instances being explicitly deleted, the instances owned by that handler are kept in shared memory (they stay in the system as *ghosts*). When an application later registers an *injection handler* (see Section 4.3.6.4) with the same handler id, it will be handed all the ghosts of the previous handler (the ghosts will be *injected*), and will be given the choice of whether to revive them or not.

If the whole system (all nodes) is restarted at the same time the ghosts will be lost, hence the "volatile"-part of the name.

Note: The ghosts are distributed to all nodes, including restarting ones, so that the resurrection may take place in any node, "old" or new.

Note: If you explicitly delete instances before unregistering a handler there will be no ghosts. It is only instances that "die from other causes" that become ghosts...

4.3.6.2 SynchronousPermanent

`SynchronousPermanent` persistence works exactly the same way as `SynchronousVolatile` persistence, except that the ghosts are stored on disk, so that they survive whole system restarts. Hence the "permanent" part of the name.

In a system with `SynchronousPermanent` persistence all entities marked as `SynchronousPermanent` are stored to disk (files or a database) by the persistence service (provided by dope_main), and when a system restart occurs the first thing that is done is that Dope recreates all stored entities as ghosts in the system, so that when applications start they will be handed the ghosts to resurrect, just as in the `SynchronousVolatile` case.

As you've probably understood `SynchronousPermanent` persistence is really *just* `SynchronousVolatile` persistence *plus* the permanent storage of ghosts.

4.3.6.3 Waiting for persistence data

When starting a system with persistence the Dob does not allow connections to be opened until the persistent instances have been distributed to all nodes. So when your application is allowed to start you are guaranteed to have persistent data available.

4.3.6.4 Using persistence

If you just want entities to survive a system restart it is very easy to do. This section describes how to do this in the easiest possible way, but if you want more control and understanding over what is going on you probably want to read Section 4.3.6.5.

Instead of inheriting from `Safir::Dob::EntityHandler` you must inherit from `Safir::Dob::EntityHandlerInjectio`
This will give you an additional four callbacks, which *you can leave empty*. If you're using C++ you don't even have to create empty callback bodies, since the base class has empty default implementations.

You need to configure the Dob to persist your entity, and that is done by creating a dom-file for your class that specifies that permanent persistence is to be used:

Permanent persistence dom file

```
<?xml version="1.0" encoding="utf-8" ?>
<propertyMapping xmlns="urn:safir-dots-unit"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance">
    <property>Safir.Dob.InjectionProperty</property>
    <class>Capabilties.Vehicles.Vehicle</class>
    <memberMapping>
        <member>
            <propertyMember>Injection</propertyMember>
            <value>SynchronousPermanent</value>
        </member>
    </memberMapping>
</propertyMapping>
```

The file must be named on the form <class name>-<property name>.dom, e.g. `Capabilties.Vehicles.Vehicle-Safir.Dob.` and placed in the same directory as the dou file for Vehicle.

Assuming that your Dob is configured to support persistence (this is the default behaviour, see also Section 6.3), your created entities should now survive system restarts.

When you have registered your handler, it will become the owner of the entities that were persisted.

If you want to understand what is going on, please read the next section, Section 4.3.6.5.

4.3.6.5 Understanding persistence [adv]

To use persistence (of the volatile or permanent kind) there is a different consumer interface to implement, `Safir.Dob.EntityHandl` This consumer has four additional callbacks, `OnInjectedNewEntity`, `OnInjectedUpdatedEntity`, `OnInjectedDeleted` and `OnInitialInjectionsDone`. The middle two of these are only used for asynchronous injections (i.e. the kind that uses deltas over a radio, as described above), so the only ones you might have to do anything in are `OnInjectedNewEntity` and `OnInitialInjectionsDone`.

In C++ all the four *Injection callbacks* have an empty default implementation. This is not possible in C# or Java since their interfaces cannot have default implementations of any kind. This means that in C++ you don't have to override these callbacks if you don't want to do anything inside of them. For this example we assume that there is something that we need to do in these callbacks (but since this is a C++ example we only need to override the two callbacks that are actually used for persistent entities).

Injection entity handler class declaration

```
class VehicleHandler :
    public Safir::Dob::EntityHandlerInjection
{
public:
    void OnCreateRequest
         (const Safir::Dob::EntityRequestProxy entityRequestProxy,
          Safir::Dob::ResponseSenderPtr        responseSender) override;

    void OnUpdateRequest
         (const Safir::Dob::EntityRequestProxy entityRequestProxy,
          Safir::Dob::ResponseSenderPtr        responseSender) override;

    void OnDeleteRequest
         (const Safir::Dob::EntityRequestProxy entityRequestProxy,
          Safir::Dob::ResponseSenderPtr        responseSender) override;

    void OnRevokedRegistration
         (const Safir::Dob::Typesystem::TypeId     typeId,
          const Safir::Dob::Typesystem::HandlerId& handlerId) override;

    void OnInjectedNewEntity
         (const Safir::Dob::InjectedEntityProxy injectedEntityProxy) override;

    void OnInitialInjectionsDone
         (const Safir::Dob::Typesystem::TypeId     typeId,
          const Safir::Dob::Typesystem::HandlerId& handlerId) override;
};
```

The first four overrides are of course the same as when registering an entity handler without persistence or injection support (see Section 4.3.5.3). I won't mention them again in this section, as they work exactly the same.

The handler is registered by calling `RegisterEntityHandlerInjection`, which has the same signature as `RegisterEntityH` except that it takes an EntityHandlerInjection instead of an EntityHandler as its fourth argument.

Registering an injection entity handler

```
m_connection.RegisterEntityHandlerInjection
    (Capabilities::Vehicles::Vehicle::ClassTypeId,
     Safir::Dob::Typesystem::HandlerId(),
     Safir::Dob::InstanceIdPolicy::HandlerDecidesInstanceId,
     this);
```

When this call is complete the Dob will call `OnInjectedNewEntity` for every entity instance that has been persisted using the specified `HandlerId` (this last bit is important, *persisted objects remember the HandlerId it was created by, and can only be resurrected by the that HandlerId*). Doing nothing in the callback means that the *injection is accepted*, i.e. the application has approved the data and the entity can be created (before this it is not visible to other applications, it is a *ghost*). If the application wishes to reject the instance, it can call `Connection::Delete(...)` which will cause the persisted entity not to become a "real" entity, and to be deleted from persistence storage.

A not so common case is when the application accepts the injected instance but there are some data in the instance that it wants to change before the instance is resurrected. In this case the application can call `Connection::SetAll(...)` in the `OnInjectedNewEntity` callback.

Once all persisted instances have been accepted or rejected by the application the `OnInitialInjectionsDone` will be called. This signals that all persistent objects for a certain handler have been injected. Again, you don't have to do anything in the callback.

(The "you don't have to do anything" bit is the reason why the strategy described in Section 4.3.6.4 works so well.)

Note: In most cases it shouldn't be necessary to check the data coming in through persistence injections. After all, it is data that has already been checked by an earlier incarnation of your application! If it was good then it should be good now.

For our Vehicle example, let's assume that the vehicle handler application has to keep an internal instance lookup table (again, it shouldn't keep all the vehicle data internally, but it might be necessary to have lookup tables to quickly know which instance to `Read`, to get at the data). In this case we will want to add instances to the tables in `OnInjectedNewEntity` and maybe update some status in `OnInitialInjectionsDone`.

Handling the OnInjectedNewEntity callback

```
void VehicleSubscriber::OnInjectedNewEntity
     (const Safir::Dob::InjectedEntityProxy injectedEntityProxy);
{
    Capabilities::Vehicles::VehiclePtr vehicle =
        std::static_pointer_cast<Capabilities::Vehicles::Vehicle>
            (injectedEntityProxy.GetInjection());

    m_myLookupTable.Add(vehicle->Callsign(),
                        injectedEntityProxy.GetEntityId());

    ... do something else ...
}
```

Handling the OnInitialInjectionsDone callback

```
void VehicleSubscriber::OnInitialInjectionsDone
     (const Safir::Dob::Typesystem::TypeId typeId,
      const Safir::Dob::Typesystem::HandlerId& handlerId);
{
    std::wcout
        << "Woohoo! I've got all my persisted entity instances"
        << std::endl;
}
```

Now you need to create a dom file as described in Section 4.3.6.4, and then you are ready to go.

Figure 4.5 shows this mechanism as a sequence diagram (note that the `Open` call from `MyEntityHandlerInjection` blocks until the PersistenceService has signalled that the persistence data is ready, as described in Section 4.3.6.3).

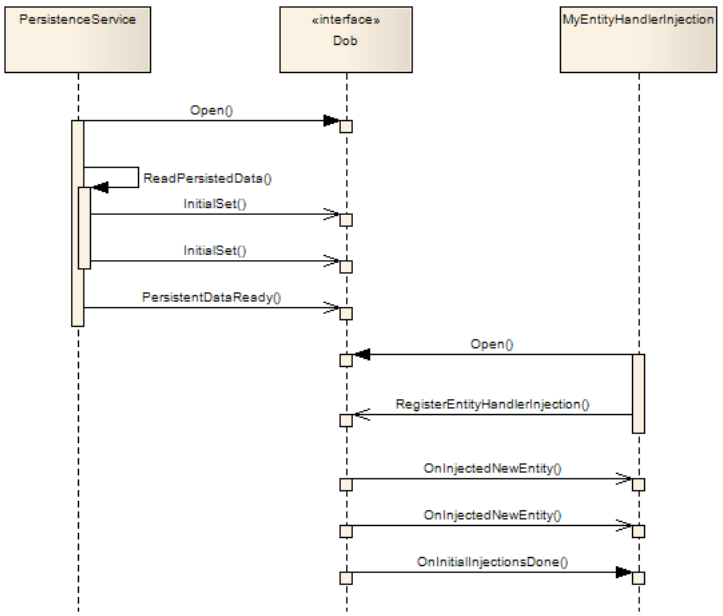

Figure 4.5: Sequence diagram for persistent entity

4.3.7 Handler registration subscriptions

It is possible to subscribe to the registration of entity and service handlers through the `SubscribeRegistration` method on the Dob connection object. This is useful to be able to monitor the status of other applications in the system. E.g. is the application that handles this type of objects started and working?

The application needs a class that implements the `Safir::Dob::RegistrationSubscriber` consumer interface.

Registration subscriber class declaration

```
class VehicleRegistrationSubscriber :
    public Safir::Dob::RegistrationSubscriber
{
    void OnRegistered
        (const Safir::Dob::Typesystem::TypeId      typeId,
         const Safir::Dob::Typesystem::HandlerId&  handlerId) override;

    void OnUnregistered
         (const Safir::Dob::Typesystem::TypeId      typeId,
          const Safir::Dob::Typesystem::HandlerId&  handlerId) override;
}
```

Then start the subscription (remember that you can do this on a service handler too!):

Starting a registration subscription

```
m_connection.SubscribeRegistration
    (Capabilities::Vehicles::Vehicle::ClassTypeId,
     Safir::Dob::Typesystem::HandlerId::ALL_HANDLERS,
     true, //includeSubclasses
     true, //restartSubscription
     this) const;
```

Note the use of `ALL_HANDLERS` above, that means that we're subscribing to the registration of all handlers for the specified type, rather than a specific handler, and since we've also specified `includeSubclasses` we will also get a subscription response if a subclass of Vehicle gets registered.

When someone registers an entity handler or a service handler the Dob will notify the application of this through a call to the `OnRegistered` method. (Note that the function has a TypeId and HandlerId as parameters and if the implementation of the interface is used for several different classes then these can be inspected to ascertain how to handle the information).

Handling the OnRegistered callback

```
void VehicleRegistrationSubscriber::OnRegistered
    (const Safir::Dob::Typesystem::TypeId      typeId,
     const Safir::Dob::Typesystem::HandlerId&  handlerId)
{
    ... do something useful ...
}
```

For entity handler registration subscriptions the `OnRegistered` callback is called immediately when the registration is made (either when the `RegisterEntity...` call is made or when a pending registration is completed), *not* after `OnInitialInjection` is called. Conceptually the handler is registered before the entity instances are injected.

When a service or entity handler is unregistered the Dob will notify the application of this through a call to `OnUnregistered`. (A handler is unregistered when the owning application unregisters the handler or the application dies).

Handling the OnUnregistered callback

```
void VehicleRegistrationSubscriber::OnUnregistered
    (const Safir::Dob::Typesystem::TypeId      typeId,
     const Safir::Dob::Typesystem::HandlerId&  handlerId)
{
    ... do something useful ...
}
```

If a registration goes down and up "very quickly", the subscriber is guaranteed to get first an OnUnregistered and then an OnRegistered callback. I.e. registration subscribers are guaranteed to be notified of the intermediate unregistrations, as opposed entity subscribers, who are not guaranteed to get told if an entity is deleted and then recreated immediately.

4.4 Aspects

Aspects are used to reduce the size of the Connection classes and header files, and to be able to classify peripheral or esoteric functionality as just that.

Currently there are three different aspects:

ConnectionAspectMisc
: Contains miscellaneous functionality, such as GetConnectionName, SimulateOverflow etc.

ConnectionAspectPostpone
: Contains functionality for postponing callbacks.

ConnectionAspectInjector
: Contains functionality needed for injectors, for example the persistence service.

4.5 Postponing callbacks

Sometimes it is not possible to handle a certain callback "right away", either due to an overflow from the Dob, or due to some external circumstance. For just this situation the Dob provides functionality to delay, or *postpone*, a certain callback for a certain type until some criteria is fulfilled.

Calling `Postpone` (can be found in the ConnectionAspectPostpone aspect) in a callback will cause the data that caused the callback to be kept (e.g. a message or request will be kept in the in-queue, or an entity subscription update will be held back) until either an overflow situation is resolved or the application calls `ResumePostponed`.

Note that postponed callbacks are always resumed when an overflowed queue is no longer full, so every time `OnNotMessageOverflo` or `OnNotRequestOverflow` is called an internal call to ResumePostponed is made.

A couple of examples are in place to make this clearer:

An application is required to send a request to some other application for every entity update (`OnUpdatedEntity`) that it receives through its entity subscription. If a lot of entities are updated "quickly" the application would soon get an overflow when sending the requests (remember that there is a limited maximum number of outstanding requests, the default is 10). The correct way of solving this (as opposed to an incorrect way, which would entail creating an infinite queue internally in the application) is to call Postpone when an overflow exception is caught. This will cause the Dob to not call OnUpdatedEntity again for any instance of the subscribed entity (and its subclasses) until the request out queue is no longer full. Once the request out queue is not full all the postponed entity updates will be received.

Another application is required to send some data to an external interface (e.g. a TCP link to some external computer/application) every time a request is received. If the external link overflows or goes down temporarily the application can no longer handle the requests, but it might be the wrong thing to just send an error response and letting the requestor handle the problem. Instead the application can call Postpone, and can then go on to handling other duties, until it detects that the external link is working again when it would call ResumePostponed, which causes the waiting requests to be dispatched (note of course that if it takes too long the request is likely to time out).

4.6 Interpreting change flags

The change flags described in Section 3.11.1 are used by the Dob distribution mechanism to ensure that components can communicate meaning as well as content. This use is slightly different in the different mechanisms. But the aim is for the change flags to mean what you would expect them to mean.

4.6.1 Messages

Change flags are sent unchanged from a sender to all receivers. In a received message a change flag signifies something that the sender has changed, or wants the receiver to do. A member whose change flag is not set is something that the sender does not care about.

4.6.2 Entity subscriptions

Upon receiving an entity subscription callback the application can ask the Dob to provide change information for the entity (using `GetEntityWithChangeInfo()` instead of `GetEntity()`). If `GetEntity` is used all change flags will be set to false, but when `GetEntityWithChangeInfo` is used the Dob will set the change flags to signal what has changed since the last subscription response.

So on the first subscription response (`OnNewEntity`) all change flags are set to true. On subsequent subscription responses for that instance (`OnUpdatedEntity`) only the members that have changed are marked as changed.

Note that this is guaranteed even if the subscriber misses an intermediate state, e.g. if the subscriber misses an entity update (if the owner updates faster than the subscriber can keep up with) all members *since the last update that the subscriber got* will have their change flags set.

4.6.3 Requests on entities

Change flags are sent unchanged from a requestor to the entity handler. In a request a change flag signifies something that the requestor wants to change in the entity.

Any member that has a change flag set is a member that the requestor wants the entity handler to set in the entity.

If a change flag is set on a member that is null that means that the requestor wants that member to be set to null. If the change flag is not set on a null member it means that the requestor does not want that member to be changed.

4.6.4 Requests on services

Change flags are sent unchanged from a requestor to the service provider. In a service request a change flag signifies something that is a part of the service request. Any member that is not changed in the request is something that the requestor does not care about.

4.7 Pending Registrations

A pending registration means "I want to register this handler if there is no handler already. And if there is a handler already registered, I want to become the registerer when that one disappears."

This functionality is to be used for applications that implement hot-standby and/or redundancy. I.e. the application is started in several nodes and if the active instance fails, another should become active instead. See Section 12.8 for more information on how to implement hot-standby and redundancy.

To be able to do pending registration you need to implement the `ServiceHandlerPending` or `EntityHandlerPending` consumer interfaces. These add an additional callback `OnCompletedRegistration` which is used to tell the application that it has become the registerer of a handler.

To perform the pending registration you call `RegisterEntityHandlerPending` or `RegisterServiceHandlerPending`.

The `EntityHandlerPending` consumer has all the Injection callbacks, since it is unlikely to want hot-standby without wanting at least `SynchronousVolatile` persistence.

4.8 Stop orders

Stop orders are used in Safir systems to allow the system to tell applications to shut down gracefully. A stop order is delivered to the application through the `OnStopOrder` callback on the `StopHandler` consumer interface that is supplied when opening a Dob connection.

The Dob will call the `OnStopOrder` callback when it has received a stop order for that application. These are sent by sending a `DeleteRequest` on the relevant instance of the Safir.Dob.ProcessInfo entity. Open Sate (see Section 15.1) and subscribe to Safir.Dob.ProcessInfo, find the instance that describes your application and then send a `DeleteRequest` on it, and your application will receive an `OnStopOrder` callback.

Upon receiving a stop order the application shall shut down gracefully.

When an application has several connections to the Dob, only one of them should supply a StopHandler. It is the handler of this connections responsibility to tell the other parts of the application to shut down gracefully. If several connections have stop handlers there may be race conditions, since there is no guaranteed order between which stop handler gets called first.

Chapter 5

Safir WebSocket/JSON-RPC interface

From version 6.3 of Safir SDK Core, a JSON-RPC interface over WebSockets is provided as an alternative to the traditional language interfaces. The interface is an implementation of the JSON-RPC specification 2.0 (http://www.jsonrpc.org/specification) and uses WebSockets (RFC 6455) as carrier. Most of the common Dob features are available such as subscribing to entities and messages, sending messages, sending and handling requests, registering as a handler of entities, and some typesystem operations. If you are not familiair with JSON-RPC, It is strongly recomended to read the JSON-RPC 2.0 Specification. Its a simple protocol and the specification is short and easy to read, it won't take more than about 10 minutes to read it.

The API is explained in full detail in Appendix B, but basically the API consists of 3 types of messages:

- Requests - sent from client to server.
- Responses - sent from server to client as a response to a request.
- Notifications - sent from server to notify client when something happens.

The pros of using the WebSocket/JSON-RPC interface is that no runtime libraries has to be installed, and the interface is accessible from any language or platform that supports WebSockets. That makes Safir available almost everywhere including web browsers and mobile apps.

The cons are worse performance and that you don't get access to the most advanced features. The programming model may also be a bit awkward in some situations.

The functioality described above is provided by the executable *safir_websocket*. It has a configuration file Safir.Websocket.Parameters.dou where the ip address and port for the server is set.

Chapter 6

Configuration

Safir SDK Core has two different configuration mechanisms, one for basic system settings, using ini-files, and one for service level settings, using dou-parameters.

The reason that we need two mechanisms is that the dou-parameters are not available for the lower level parts of the Safir SDK Core, and we need some way of configuring these. So we use ini-files for such things as where lock files should be kept, how logging should work, and where to actually look for dou-files.

This chapter aims to describe the most important of all the things that can be configured in Safir SDK Core. There are many more things that can be configured, and you're encouraged to have a look in the parameter dou-files, which should contain descriptions of what the parameters do.

6.1 Basic system settings

The basic system settings consist of three ini files: *locations.ini*, *logging.ini*, and *typesystem.ini*:

- locations.ini - locations of low level files and directories, e.g. lock file and crash dump directories.
- logging.ini - settings for system logging and debug logging.
- typesystem.ini - typesystem configuration.

The system will look for these configuration files in several places, which allows them to be kept separate or together with the rest of the Safir SDK Core files. The following locations are checked, and the first configuration found will be used:

Linux

1. /etc/safir-sdk-core/ - system-wide configuration
2. ~/.config/safir-sdk-core/ - user configuration

Windows

1. *CSIDL_COMMON_APPDATA*\safir-sdk-core\config\ - system-wide configuration
2. *CSIDL_LOCAL_APPDATA*\safir-sdk-core\config\ - user configuration

On Windows 10 *CSIDL_COMMON_APPDATA* is `C:\ProgramData` and *CSIDL_LOCAL_APPDATA* is `%USERPROFILE%\Ap` by default. See MSDN documentation for more information about CSIDLs.

The reason for checking for the system-wide configuration before checking for a user configuration is a security measure. It will ensure that an unpriviliged user cannot insert their own configuration files, thereby modifying an installed system. If the load order was reversed an unpriviliged user could just put ini-files in his configuration directory and thereby completely change the behaviour of the system.

The Safir SDK Core installation packages will only provide system-wide configuration files. If you wish to load the configuration from the user configuration instead you will have to remove the system-wide configuration manually after installing Safir SDK Core.

6.1.1 locations.ini

locations.ini contains three parameters:

lock_file_directory
To ensure correct platform independent initialization of some system-wide resources (e.g. shared memory) Safir SDK Core uses lock files. This parameter lets us know where we should keep them.

crash_dump_directory
Controls where the crash reporting library (see Section 11.4) should write the dump files.

ipc_endpoints_directory
Specifies where in the filesystem node internal ipc endpoints should be stored. Applicable for Linux only.

6.1.2 logging.ini

logging.ini has two sections, *SystemLog* and *LowLevelLog*. The LowLevelLog section contains parameters that control the logs for an internal logging and debugging facility, and you should never really need to touch these parameters, unless one of the Safir SDK Core developers have asked you to. The SystemLog section, however, contains parameters of greater interest, since they control the way Safir Logging works:

native_logging
Controls whether or not logs are sent to the native, platform specific, logging mechanism.

send_to_syslog_server
Controls whether or not logs are formatted by the Safir Log mechanism and sent to a syslog server using UDP datagrams.

syslog_server_address
IP address for the syslog server.

syslog_server_port
UDP Port for the syslog server.

replace_newline_with_space
Controls whether or not a newline in the log message is replaced with a space.

truncate_syslog_to_bytes
Truncate syslog messages after this many bytes. RFC 3164 states that 1024 is the limit, but many syslog servers allow for longer messages. Make sure to verify that changing this works with your server. This only applies to syslog messages, not to native logging.

show_safir_instance
Show the current SAFIR_INSTANCE in syslog reports. Useful when running multiple nodes on one computer (see Section 6.2.5)

See Chapter 7 for more information on what "native" and "syslog" means, and how to use and configure the logging functionality.

6.1.3 typesystem.ini

typesystem.ini is a little bit more involved, so let's start with an example:

Example typesystem.ini

```
; Number of megabytes of shared memory that will be allocated by dots kernel.
dots_shared_memory_size=10

; Comma separated list of default directories to look for dou files in.
dou_search_path=/usr/share/safir-sdk-core/dou
```

```
; Comma separated search path for safir_generated-xxx-java.jar files.
java_search_path=/usr/share/java/safir-sdk-core

[Core]
; This section contains the information needed to load the safir_generated-Core-xxx
; libraries and its dou files.
kind=library
dependencies=

[OverrideExample]
; This section provides a parameter override that will only be used on a node with
; SAFIR_INSTANCE set to 1.
; It uses a custom directory for dou/dom files, rather than using the dou_search_path above.
kind=override
dou_directory=/path/to/files
safir_instance=1

[LibraryExample]
; This section contains the information needed to load the
; safir_generated-LibraryExample-xxx libraries and its dou files.
kind=library
dependencies=Core ; it has a dependency on one or more dou files in Core.
; No dou_directory is specified, so the dou files will be looked for under
; dou_search_path/LibraryExample
; No safir_instance is specified, so the section will apply to all nodes on the computer.
```

The file contains a general section, followed by any number of `[sections]`, each of which can be one of two kinds, a *parameter override* or a *library module*.

6.1.3.1 General section

The general section contains parameters that affect the typesystem itself, or all of sections in the rest of the file.

dots_shared_memory_size
: Specifies the size of shared memory allocated for the typesystem. The default value is big enough for most systems, but if the typesystem contains a very large number of types it may be necessary to increase this value. More info on shared memory can be found in Section 6.6

dou_search_path
: Comma separated list of default directories to look for dou files in. The typesystem will look for dou files for each module described in a section by appending the section name to each of the entries in this variable.

java_search_path
: Comma separated list of directores to look for safir_generated jar files in. If a safir_generated-xxx-java.jar file cannot be found in the java class path the typesystem will look in each of the directories specified in this variable.

6.1.3.2 Library module sections

A library section specifies a set of dou files and the safir_generated binaries generated from them that should be loaded by the typesystem at startup. The section name must be the name that was specified when building the module (see <<code-generation).

kind
: Mandatory. Use value *library* to make a library module section.

dependencies
: Mandatory, but can be empty. A comma-separated list of modules that dou files in this module depend on. This information is needed by dobmake to resolve references inside dou files.

dou_directory
Optional. Specifies the directory to load dou files from. If this is not specified the `dou_search_path` in the general section will be used to find the dou files.

cpp_library_location
Optional. Location of the C++ binaries generated by dobmake. If this is not specified the library loader configuration in the operating system will be used instead.

java_jar_location
Optional. Location of the java binaries generated by dobmake. If this is not specified the `java_search_path` in the general section and the java class path will be used instead.

dotnet_assembly_location
Optional. Location of the dotnet binaries generated by dobmake. If this is not specified the dotnet runtime library loader will be used instead.

dotnet_assembly_version
Optional. The version of Safir SDK Core that generated the dotnet binary. If this is not specified the currently installed version of Safir SDK Core will be used, so usually you do not need to specify this. But this can be used if a newer Safir SDK Core was installed and you want to load older assemblies. For example, this can be used to test a new patch release of Core without rebuilding everything.

6.1.3.3 Parameter override sections

A parameter override section specifies a set of dou files that should be loaded instead of the default ones included in a library module. For example, this can be used to override parameters or string or array lengths without having to change the files that are in the Safir SDK Core installation, keeping project-specific changes separate from the Safir SDK Core defaults.

In the above example there is one parameter override section. If, for example, the OverrideExample dou_directory contains a Safir.Dob.NodeParameters.dou file (which is also in the Core dou_directory) the file from the OverrideExample dou_directory will be used.

If a file is found in multiple places, the *last one* found will be the one used.

kind
Mandatory. Use value *override* to make a parameter override section.

dou_directory
Directory containing dou files to load for this parameter override.

safir_instance
Optional integer value. If specified the section will only be applied to the node with that `SAFIR_INSTANCE` number. For more info on running multiple nodes on one computer, see Section 6.2.5.

The name of parameter override sections are not used for anything in the system, so they only need to be unique.

6.1.4 Environment and special variable expansion

Since the configuration files need to be able to refer to system directories they also support environment variable expansion. E.g. `$(HOME)` expands to the value of the `HOME` environment variable.

On Windows platforms we also support the *CSIDL* values mentioned above, since Windows does not guarantee that there are always environment variables that correspond to them. For example the %PROGRAMDATA% environment variable is not guaranteed to be set or to point to the same directory as *CSIDL_COMMON_APPDATA*. So the configuration files support *special variables* as well as environment variables. E.g. `@{SPECIAL_VARIABLE}` will be expanded to the value of that variable (note that the syntax is different from the environment variable syntax).

The supported special variables are listed in Table 6.1. We also provide aliases that use the Windows Known Folder naming.

Table 6.1: Special Variables on Windows

CSIDL name	Windows Known Folder "alias"
CSIDL_APPDATA	FOLDERID_RoamingAppData
CSIDL_LOCAL_APPDATA	FOLDERID_LocalAppData
CSIDL_COMMON_APPDATA	FOLDERID_ProgramData
CSIDL_MYDOCUMENTS	FOLDERID_Documents
CSIDL_COMMON_DOCUMENTS	FOLDERID_PublicDocuments
CSIDL_PROGRAM_FILES	FOLDERID_ProgramFiles
CSIDL_PROGRAM_FILESX86	FOLDERID_ProgramFilesX86

Here are two more links that contain useful information about Windows paths, CSIDLs, and Windows Known Folders.: "Where Should I Write Program Data...?" and "Where Should I Store my Data and Config...?".

There is currently two special variables that are available on both Linux and Windows. They are described in Table 6.2, below.

Table 6.2: Common Special Variables

Special Variable	Meaning
TEMP	/tmp on Linux and TEMP or TMP environment variable on Windows.
SAFIR_INSTANCE	The value of the environment variable SAFIR_INSTANCE, or "0" if that variable is not defined.

6.2 Network config

A Safir SDK Core system can consist of one or more nodes running on one or more computers. This allows division of responsibility between multiple server nodes, or running HMI and business logic on different computers.

To be able to configure a networked system properly a basic understanding of how Safir SDK Core nodes communicate is needed.

6.2.1 Full nodes and Light nodes

There are two basic kinds of nodes in a Safir SDK Core system, *Full nodes* and *Light nodes*. The two kinds of nodes have different features, and will fulfill different roles in a system.

In short Full nodes have access to all distribution mechanism features, but need to have full network connectivity at all times, whereas light nodes only have partial access to the distribution mechanisms, but in exchange can have more flexibility with network connectivity.

For some types of systems it may make sense to use only Full nodes, whereas for others it may make sense to use a mix of full and light nodes. If a system consists of one or more servers and a set of clients, it may be useful to make the clients into light nodes. The clients will then be able to connect and disconnect from the rest of the system in a more flexible way.

6.2.1.1 Details on Light nodes

The properties of a Light node are:

- Can only register handlers for services that are marked as *Local* or *Limited* (see Section 6.2.6)
- Can only register handlers for entities that are marked as Local or Limited.

- Can only own instances of entity types that are marked as Local or Limited.
- Can subscribe to everything.
- Will only receive messages, requests and entity instances from full nodes, not other light nodes.
- Can send messages, that will reach full nodes, but not other light nodes.
- A light node can be disconnected from a system, and then be reconnected at a later time, without having to be restarted.
- A light node can be disconnected from one system and connected to another system, without having to be restarted.

A light node that is disconnected from a system is known as a *detached* light node, and when it is connected it is known as an *attached* node. These statuses are reflected in the `Safir.Dob.NodeInfo` entity.

Light nodes never communicate directly with each other, so there is no need for them to be able to reach each other on the network. Light nodes do however need to be able to reach all full nodes.

A side effect of this is that light nodes may not be aware of each other at all. For example, looking at the `Safir.Dob.NodeInfo` entity on one light node will not show instances for all nodes, but only for the light node itself and all the full nodes. (For this specific information `safir_status` actually provides an entity `Safir.Dob.MirroredNodeInfo`, which lets light nodes see the node status of other light nodes, since that information may be important for an operator to understand system status.)

6.2.1.2 Light node detach/attach behaviour

There are two ways that a detached light node can behave. Either it can wipe all registrations and entities, and essentially become "blank", or it can keep all registrations and entities, but as read-only. This latter mode of course disallows all requests (except to handlers on the same node).

The keep mode is useful for allowing a user on a disconnected light node to still look at the last received state of all entities.

This behaviour is configured using the `KeepStateWhileDetached` parameter, as described in Table 6.3.

When a detached node in keep mode reattaches to the same system that it was attached to previously it will update its state to the state of the system. So instances that no longer exist or handlers that no longer are registered will get removed, and new instances and registrations will be added. Subscribers on the light node will be updated accordingly.

If a detached light node in keep mode attaches to another system (or the same system, but restarted) it will also get resynchronized, so that it reaches a correct state. But of course in this case all instances and registrations from the old system will get deleted first.

Since a light node in non-keep mode will get cleared of all instances and registrations when it detaches, the resynchronization when attaching/reattaching is just a matter of sending the current system state to the light node.

6.2.2 Safir SDK Core networking

Safir SDK Core nodes can communicate using *UDP Unicast* and *UDP Multicast*. Multicast communication means that a single packet sent from one node can be received by multiple nodes, whereas unicast packets are only sent to one node.

For networks that support it, multicast is a lot more efficient if there are many nodes. For example an entity update can be performed using only one packet, whereas in a system using unicast one packet per node is needed. On the other hand, using multicast is not an option on all networks.

Regardless of whether multicast is used or not, all nodes must be reachable using unicast, since things that only need to be sent to one node is sent using unicast. A system can also mix nodes that are reached with multicast with nodes that are reached with unicast.

The nodes communicate with each other over two channels, the *control channel* and the *data channel*. Each of these channels is a separate stream of UDP packets over a separate set of ports.

All these things, the control and data channel ports and multicast/unicast communication, need to be configured. Some things are configured as part of each node's *Node Type*, and some are configured for each individual node.

To clarify how all this relates to *Full* and *Light* nodes, all full nodes need to be able to reach each other as per the above requirements. Light nodes do not need to be able to reach each other, but they need to be able to reach all full nodes.

6.2.3 Node configuration

Each node in a Safir SDK Core system belongs to one and only one *Node Type*. Each node type has a number of parameters that control how nodes belonging to it communicate with other nodes. The node types are configured in the NodeTypes parameter in Safir.Dob.NodeParameters.dou. The parameters for each node type are listed in Table 6.3.

Table 6.3: Node Type Parameters

Parameter Name	Comment
Name	Name of the node type
MulticastAddressControl	Multicast address and port used for the control channel for nodes of this type. An empty string indicates that nodes of this type can't be reached via multicast.
MulticastAddressData	Multicast address and port used for the data channel for nodes of this type. An empty string indicates that nodes of this type can't be reached via multicast.
HeartbeatInterval	How often shall heartbeats/keepalives be sent.
MaxLostHeartbeats	How many lost heartbeats before marking nodes of this type as dead.
SlidingWindowSize	How many messages that can be sent on the network at the same time when communicating with other nodes. This value has a valid range between 1 and 20. Setting this value to 1 means that every message must be acknowledged from all receivers before continuing sending the next message. If this value is set to 20, it means that it is allowed to send at most 20 messages at the same time before waiting for acknowledgements from the receivers. On a stable network with decent speed it is recommended to use the maximum value (20).
AckRequestThreshold	The maximum number of sent messages that are still unacknowledged, before the sender will explicitly request acknowledgements from the receivers. The valid range for this value is between 1 and SlidingWindowSize. For small values on SlidingWindowSize it is recommended to set AckRequestThreshold=SlidingWindowSize. For SlidingWindowSize close to the maximum value (20), it is recommended to set this value to about half the SlidingWindowSize.
RetryTimeout	Time to wait for acknowledgement before sending the same message again to nodes of this type. This parameter is a sequence of time values where the first value is used as time limit before the first retransmission is made, the second value is used as time limit before the second retransmission an so on. The last value in the sequence is used for any forthcoming retransmissions.
RequiredForStart	If true, a node of this type can start a system of its own. A node with a node type where this parameter is set to false will never start a system of its own, it just waits for a system to join. If IsLightNode is set true this value must be false or it can be omitted. I.e a light node can never be required for start.
IsLightNode	If true, nodes of this node type are light nodes. See Section 6.2.1 for more information.
KeepStateWhileDetached	This parameter only has effect if IsLightNode is true. If this value is true, all Entities owned by connections on remote nodes are kept in read-only mode. Any requests will result in an error response. If this value is false, all Entities owned by connection on remote nodes are unregistered and deleted.

Each node has to be configured individually, to know which node type it belongs to, etc. This is done in done in the `Safir.Dob.ThisN` file, which therefore must be different on all nodes (or use environment variables in the parameters, as described in Section 3.5.4).

Table 6.4: This Node Parameters

Parameter Name	Comment
Name	Name of the node. This is only used for presentation purposes, and is not used for any logic. No checks are made to ensure that the node name is unique, since in fact it doesn't need to be unique.
NodeType	Node type for this node.
ControlAddress	Unicast address and port of the control channel.
DataAddress	Unicast address and port of the data channel.
Seeds	List of addresses and ports of the control channel of the seed nodes. Seed nodes are explained in Section 6.2.4.

The addresses are on the format `address:port`, where the address can be an IP address or a DNS-resolvable address.

`ControlAddress` and `DataAddress` are used to identify which local network interface that we should bind to. Wildcards can be used to match parts of an address, e.g. "192.168.1.*" will make Safir SDK Core use the first network interface on that subnet that it can find on the local machine. On Linux it is also possible to specify a network interface explicitly, for example "eth0:20000", meaning use port 20000 on whatever ip address eth0 is using. On Windows there is no such feature, currently.

If no matching interface can be found retries will be made for the interval specified in Safir.Dob.NodeParameters.LocalInterfaceTimeout.

There is a command line utility `safir_resolver` (used to be called communication_resolver) that can be used to check what an address or interface name will resolve to. Use the `-l` flag to resolve local addresses and interfaces.

6.2.4 Node discovery and system start

A newly started node will, as a first step, try to find and contact other nodes. The mechanism used to achieve this is called *node discovery*.

The second step is to either start a new system or, in case the node is in contact with nodes from an already started system, to join an existing system.

6.2.4.1 Node discovery

The node discovery mechanism is used by individual nodes to find and establish connections to other nodes. Once a node finds one other node in a system it will find all other nodes in the system, and all the other nodes will find it.

This process is initiated by setting up *seeds*. Each node can have a list of addresses that it will try to connect to. These addresses are known as seed addresses. A seed address is simply the address of another node's control channel, as configured in `Safir.Dob.ThisNodeParameters.ControlAddress`.

A simple way to use this in a Client/Server system is to have all nodes list all the server nodes' addresses in their seed list. This way, when any node starts, it will attempt to connect to all servers, which will then tell it about all other clients.

6.2.4.2 System start

A system, or more precisely, all nodes that are part of a specific system start, are identified by a unique incarnation id.

The main purpose of the incarnation mechanism is to guarantee that an "old" node cannot join a newly started system. Before a node joins a system it checks that the system incarnation id is not found in a locally kept "blacklist" of old incarnation ids. The typical scenario is when restarting a system consisting of several nodes. Without this mechanism a node that happens to survive the system restart could be included in the restarted system, resulting in a mix of old and new data.

When a node starts it will wait for a certain amount of time to see if there is an existing system with a non-blacklisted incarnation id to join on the network.

If no existing system is discovered during this time the node will act differently depending on the specified value of `RequiredForStar` for the nodes node type. If RequiredForStart is true the node will start a system of its own, possibly together with other newly started nodes. If RequiredForStart is false the node will *not* start a system of its own, it just waits for a system to join.

For example, in a client/server configuration, seeded as described above, setting RequiredForStart to false for nodes of type Client, and to true for nodes of type Server, will prevent clients from starting and forming multiple system on their own without a server. Note that this applies only to the start phase, if clients lose contact with the server in an already started system you can end up with a system that consists only of client nodes.

Note that the "AloneTimeout" is calculated to be the maximum value of 2 * MaxLostHeartbeats * HeartbeatInterval for all node types. One upshot of this is that if you configure one node type to have very large values for these parameters then the system will appear slow to start. But this is all to be expected, since the reason for setting large values is that a node type has bad connectivity, and then we may need a long time to discover it.

6.2.5 Multiple nodes on one computer

As hinted at before it is in fact possible to run multiple Dob nodes on a single computer (this is a new feature in Safir SDK Core 6.0). The way that the nodes are distinguished from each other is by setting the environment variable `SAFIR_INSTANCE`. All processes that have the same value of SAFIR_INSTANCE will form one node.

So, for example, you can start two instances of `safir_control`, one with SAFIR_INSTANCE set to 0, and one with it set to 1. When you then start an application, with SAFIR_INSTANCE set to 1 it will be part of the second safir_control instance. Remember that it is possible to specify parameter overrides that are only loaded for certain values of SAFIR_INSTANCE, as described in Section 6.1.3.

If you want the nodes to talk to each other you will need to set up seeding, as described above, but it is of course possible to run two completely separate and different systems.

6.2.6 Distribution scope

Normal entities, entity and service registrations and messages are sent to all nodes in a system. It is, however, possible to change this behaviour by configuring a type's "Distribution Scope" to be either `Local` or `Limited`.

6.2.6.1 Local types

Local types are object types that are not sent over the network.

For some entities, services and messages it is not desirable to have them sent over the network at all. These are known as local types. Things like settings for an operator on one operator console or other things that only apply to the current node like which object is selected in the map are typically Local objects.

An object is specified as Local by mapping the `Safir.Dob.DistributionScopeProperty` to it, with the value "Local" as DistributionScope, using a dom file.

Specifying that an object is local.

```
<?xml version="1.0" encoding="utf-8" ?>
<propertyMapping xmlns="urn:safir-dots-unit"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance">
    <property>Safir.Dob.DistributionScopeProperty</property>
    <class>Capabilities.Selection</class>
    <memberMapping>
        <member>
            <propertyMember>DistributionScope</propertyMember>
            <value>Local</value>
        </member>
    </memberMapping>
</propertyMapping>
```

6.2.6.2 Limited Types

Light nodes (as described in Section 6.2.1) are only able to be given their extra flexibility by giving up the ability to own "normal" types. Instead they are given the ability to own *Limited types*, that have a number of limitations put upon them. The limitations are there to ensure that a light node can reattach to a system after it has been detached.

- A limited type can have no persistence, i.e. it has neither Volatile, Permanent or Injectable persistence.
- Pending registrations of a limited type is not allowed.
- Overregistration of a limited type is considered a programming or configuration error.

If you have multiple handlers of the same limited type you need to make sure to have unique handler ids for each of them. There is no guarantee that overregistrations will be detected and reported, although some effort is made.

An object is specified as Limited by mapping the `Safir.Dob.DistributionScopeProperty` to it, with the value "Limited" as DistributionScope, using a dom file.

Specifying that an object is limited.

```
<?xml version="1.0" encoding="utf-8" ?>
<propertyMapping xmlns="urn:safir-dots-unit"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance">
    <property>Safir.Dob.DistributionScopeProperty</property>
    <class>Capabilities.ClientStatus</class>
    <memberMapping>
        <member>
            <propertyMember>DistributionScope</propertyMember>
            <value>Limited</value>
        </member>
    </memberMapping>
</propertyMapping>
```

6.3 Persistence config

The persistence service, provided by the `dope_main` executable has a few different features that might be good to know about. It has three *backends*, one that does nothing, one that persists entities to files, and one that persists them to a database.

A few relevant parameters for the persistence service, apart from the ones described in Section 4.3.6.4.

Safir.Dob.PersistenceParameters.Backend
: Which backend should be used, *None*, *Odbc* database or *File*.

Safir.Dob.PersistenceParameters.FileStoragePath
: The path where files are stored if the file backend is chosen. Obviously dope_main needs write permissions here. Remember that you can use environment variables and special variables here, as described in Section 3.5.4.

Safir.Dob.PersistenceParameters.OdbcStorageConnectString
: The connection string to use to connect to the database.

Safir.Dob.PersistenceParameters.StandaloneMode
: This parameter allows enabling deprecated functionality. Please don't do it!

The default behaviour is that an entity is written to external storage as soon as it is updated, which, for frequently updated entities, might generate a lot of writes to external storage.

The "Persistence Throttling" mechanism can be used to limit the number of writes for instances of a certain entity type. The throttling mechanism is applied by mapping the property `Safir.Dob.PersistenceThrottlingProperty` to it. An entity instance will be written no more often than given by the property value `WritePeriod`.

There is more information about the persistence service in Chapter 10.

6.4 Request timeouts config

Timeouts are defined using the properties Safir.Dob.RequestTimeoutProperty and Safir.Dob.RequestTimeoutOverrideProperty. If no response have been received when the timeout occurs, the Dob itself sends a timeout response to the Requestor. If the missing response arrives after this, it is just ignored. I.e., the Dob guarantees that the requestor will receive one, and only one, response. See also Section 12.2.

The default timeout for service requests is 7 seconds and for entity requests 2 seconds, and these can be changed by changing the timeout properties on the `Safir.Dob.Entity` and `Safir.Dob.Service` classes.

Note that the RequestTimeoutProperty is inherited, whereas the RequestTimeoutOverrideProperty is not inherited, so setting a RequestTimeoutProperty on a class will cause all derived classes to get the same timeouts. The Override-property can be used when this is not the desired behaviour, or for making an exception on one level of an inheritance "tree".

6.5 Queue lengths config

By default all the length of the in and out queues of messages and requests is 10. The queue lengths can be customized by changing the parameter `Safir.Dob.QueueParameters.QueueRules`.

The QueueRules parameter is an array of `Safir.Dob.QueueRule` items, each containing one rule to apply to the queue lengths. An example is shown below.

A queue length parameterization

```
<?xml version="1.0" encoding="utf-8" ?>
<class xmlns="urn:safir-dots-unit"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <name>Safir.Dob.QueueParameters</name>
  <baseClass>Safir.Dob.Parametrization</baseClass>
  <parameters>
    <parameter>
        <name>QueueRules</name><type>Safir.Dob.QueueRule</type>
        <array>
            <!-- The first array item is the default queue lengths.
            It matches all connection names. -->
            <Safir.Dob.QueueRule>
                <MessageInQueueCapacity>10</MessageInQueueCapacity>
                <MessageOutQueueCapacity>10</MessageOutQueueCapacity>
                <RequestInQueueCapacity>10</RequestInQueueCapacity>
                <RequestOutQueueCapacity>10</RequestOutQueueCapacity>
            </Safir.Dob.QueueRule>
            <!-- let connections with 'long' in their connection names
            have longer message in queues and request out queues. -->
            <Safir.Dob.QueueRule>
                <ConnectionNameRegex>long</ConnectionNameRegex>
                <MessageInQueueCapacity>20</MessageInQueueCapacity>
                <RequestOutQueueCapacity>50</RequestOutQueueCapacity>
            </Safir.Dob.QueueRule>
            <!-- let connections with at least one capital letter in their
            connection name have longer message out queues. -->
            <Safir.Dob.QueueRule>
                <ConnectionNameRegex>[A-Z]+</ConnectionNameRegex>
                <MessageOutQueueCapacity>300</MessageOutQueueCapacity>
            </Safir.Dob.QueueRule>
        </array>
    </parameter>
  </parameters>
</class>
```

When a new connection is opened the rules are scanned from top to bottom, and the *maximum values* of all rules where the ConnectionName member matches are used. No ConnectionName in a rule means "match all". It is important to have one rule, like the first one in the example above, that provide default values. Finding no match causes undefined behavior.

Here are some example connection names and their resulting queue lengths:

Connection Name	MsgInQ	MsgOutQ	ReqInQ	ReqOutQ
my_connection	10	10	10	10
my_long_connection	20	10	10	50
My_Connection	10	300	10	10
my_long_Connection	20	300	10	50

Warning
Before deciding to change the queue lengths you should have read and understood Section 12.1, Section 12.2, and the two items on overflows in Appendix C. It is important to understand what effects, other than just increasing memory use, this may have on your system.

6.6 Shared Memory config

The Dob has two parameters that control the amount of shared memory used, one that controls the amount of memory available for the typesystem (the internal tables for all types, and all parameters), and one that controls the amount of memory available for the distribution mechanism.

The parameter that controls the typesystem memory can be found in the file `typesystem.ini`, as described in Section 6.1.3. The amount of memory that the typesystem needs is completely static, i.e. it does not change at all during runtime, so if you have been able to start `safir_control` you know that you have enough memory allocated for the typesystem.

The parameter that controls the shared memory used by the distribution mechanism is called `SharedMemorySize` and can be found in Safir.Dob.NodeParameters.dou. Knowing what value to set `SharedMemorySize` to is trickier since the memory usage varies over time, but one approach is to run the system and monitor the memory usage, for example using *dobexplorer*, which is described in Section 15.2.

6.7 Memory levels and graceful degradation

To ensure optimal performance and avoid potential issues related to memory exhaustion, Safir SDK Core implements five different memory levels, namely: *Normal*, *Warning*, *Low*, *VeryLow*, and *ExtremelyLow*. These memory levels are parameterized in the `Safir.Dob.NodeParameters.SharedMemoryLevels` parameter.

The memory levels in the Safir SDK Core's distribution mechanism are designed to enable graceful degradation in resource-constrained environments. When memory consumption gets to the Low level and beyond, the system is able take precautionary measures to prevent critical failures by adapting its behavior and prioritizing essential operations and giving operators a chance to reduce memory usage before actual fatal memory allocation failures occur.

The five memory levels and their descriptions are as follows:

Normal
: Everything works as normal.

Warning
: Everything continues to work as normal, but system operators should be alerted to the fact that memory is running low. Warnings are logged to the syslog. Default threshold is 20%.

Low
: In the Low memory level, the system restricts operations that allocate new object instances. Default threshold is 15%.

VeryLow
In the VeryLow memory level, the system restricts operations that modify existing object instances or potentially lock instances in memory. Default threshold is 8%.

ExtremelyLow
This is the most severe memory level, indicating an extremely constrained environment. At this point, the system attempts to forbid all operations, even those that just cause temporary shared memory allocation. Default threshold is 3%.

If your system is designed in a way that makes it likely that it will reach these memory levels it is recommended to have a component that monitors the memory levels and issues UI alerts to the operators when the warning level and beyond are reached.

6.7.1 Forbidden operations at severe memory levels

At the Warning, Low, and VeryLow memory levels, Safir SDK Core imposes restrictions on certain operations to prevent memory exhaustion and prioritize essential tasks. When one of these operations are performed when they are not allowed a `Safir.Dob.LowMemoryException` will be thrown. The table below outlines the operations that become forbidden at each of the memory levels:

Table 6.5: Operations that *become* forbidden at each memory level

Memory Level	Connections	Entities	Services	Messages
Normal, Warning	-	-	-	-
Low	-	**Create new instance** Inject	-	-
VeryLow	Open	Register Register Pending Subscribe Subscribe Registration Unregister **Read** **Update existing instance** **Delete existing instance** Create iterator GetNumberOfInstances Inject Deletion	Register Register Pending Subscribe Registration Unregister	-
ExtremelyLow	-	CreateRequest UpdateRequest DeleteRequest SendResponse Unsubscribe Registration Unsubscribe Entity	SendRequest SendResponse Unsubscribe Registration	Subscribe Send

The four operations highlighted in bold can be moved to VeryLow or ExtremelyLow for a specific entity type by defining `LowMemoryOperationsAllowedProperty` (which can be overridden using `LowMemoryOperationsAllowedOverrideP` on that type. That property has a member `DisallowAtLevel`, which if it is set to VeryLow will move the *Create new instance* operation down to the VeryLow level, and if set to ExtremelyLow will move all four operations down to the ExtremelyLow level. Any other values for DisallowAtLevel are illegal. Great care should be taken to only use this facility for types that are set rarely and are small, since this introduces a risk of increasing memory usage even higher when it is already high.

6.7.1.1 Some notes on why things are where they are in the table above

GetNumberOfInstances at VeryLow?
GetNumberOfInstances uses entity iterators under the hood, so it gets forbidden along with the iterators.

Entity iteration and Read at VeryLow?
These operations hold references to things in shared memory until the caller releases them.

Unregister at VeryLow?
An unregistration can cause a whole bunch of entity instances to be deleted, which may cause the creation of a lot of ghost entities, which means shared memory allocations.

Read affected by LowMemoryOperationsAllowedProperty?
SetChanges uses Read internally, so the implementation became a lot simpler by moving that too.

6.7.2 API for getting current memory level

`Safir.Dob.ConnectionAspectMisc` contains functions that are useful for monitoring the shared memory:

`GetSharedMemoryLevel`
Gives you the current shared memory level on the current node.

`GetSharedMemoryUsage`
Gives you the current shared memory usage in bytes on the current node.

6.7.3 Testing application memory handling

`safir_memory_allocator` can be used to generate fake memory pressure on a node. For example the commands

```
safir_memory_allocator --allocate Low
safir_memory_allocator --deallocate
```

will allocate shared memory until the system reaches the Low level, and deallocate all memory allocated by this tool, respectively.

Run `safir_memory_allocator --help` for more information.

Chapter 7

Safir Logging

Safir SDK Core has a mechanism, Safir Log, designed to make it simpler for applications to generate and send log messages according to the widely adopted *Syslog* protocol RFC 3164.

Safir Log provides an easy-to-use interface that allows an application to create and send logs by a simple function call, specifying a severity and a message. The framework will do the neccessary formatting and transmission over UDP as is specified in RFC 3164.

It is also possible to configure Safir Log so that the logs are sent to a native, platform specific, logging mechanism. In practice, this could be of use on the Windows platform to send logs to the Windows Event Log and on Linux to send the logs directly to the `syslog(...)` interface instead of using UDP messages.

Example, sending a log message using C++

```
#include <Safir/Logging/Log.h>

Safir::Logging::SendSystemLog(Safir::Logging::Error, L"This is an error log!");
```

Note that Safir SDK Core only provides the functionality related to the sending side for syslog messages. In order to collect and view the log messages a syslog server is needed. Such a server can range from a simple program listening for UDP packets and dumping the result on standard output, to more sophisticated solutions that support filtering, writing logs do disk or database and provide advanced search and report generation functionality. A syslog daemon is included in most Linux distributions and for Windows there are several third-party syslog servers.

7.1 Logging guidelines

This section provides some guidelines that, when applied, will help you make your application behave like a "good citizen" in the syslog ecosystem.

First of all we recommend that you establish project/product specific logging guidelines so that the separate parts that form your system exhibit a consistent logging behaviour.

Syslog is a "one-liner" oriented protocol. This, and the fact that syslog servers are known to handle messages differently regarding new lines, means that we recommend that you try to make your logs one-liners. When put together, the one-liner logs from your application should form a coherent narrative.

7.1.1 Facility and Severity

A syslog message has a Priority value (PRI) that is a representation of both the Facility and the Severity of the message. PRI is used by syslog deamons to filter and redirect logs but is normally not shown in the log output. Therefore, compose the log message text so that it is clear and understandable without the facility and severity context.

The Safir Log mechanism uses facility code 1 (user-level messages) for all syslog messages and it is not a parameter visible in the interface.

The severity level must be set by the application. RFC 3164 has a brief description of the interpretation of the severity level but doesn't really give any detailed advice. To summarize: higher severity levels imply that action from support/tech staff is more urgent, than for lower levels. In practice it isn't always obvious what severity level to use. This is especially true for code in libraries/platforms where there usually is no knowledge of the characteristics of the calling application. One could argue that libraries should refrain from generating logs at all and instead inform the using application that an error has occurred, so that it can take appropriate action. Since Safir SDK Core does not adhere to this rule itself we leave this as something to consider during library design.

The system design in terms of redundancy might also affect the choosen severity level. For example, it might be appropriate for a system built and configured so that there is a standby application/node "taking over" in case of failure, to use severity level Error instead of Emergency for a given event.

Table 7.1: Syslog severity levels

RFC 3164 Severity Level	**RFC 3164 Description**
Emergency	System is unusable.
Alert	Action must be taken immediately.
Critical	Critical conditions.
Error	Error conditions.
Warning	Warning conditions.
Notice	Normal but significant condition.
Informational	Informational messages.
Debug	Debug-level messages.

Here are some recommendations regarding wich severity level to use for some typical classes of errors and events found in Safir systems.

Fatal error
: This type of error includes static conditions that must be fulfilled to be able to start/continue executing the program, for example missing static resources or invalid configuration. It also includes programming errors, that is, errors of assert-type. In these cases send a Safir Log with severity in the range Critical to Emergency and then exit the program with an error code (i.e. a non-zero exit code).

Non-fatal error
: This type of error indicates that an error is detected but the program can continue to execute. An example would be the reception of a message from an external system in an invalid format. In this case send a Safir Log with severity Error. Normally the program continues to execute, possibly in a degraded state.

Dynamic Resource
: A dynamic resource is defined as a resource that is not guaranteed to be available at the time the program starts and/or a resource that can "come and go" during program execution. Send a Safir Log with severity Error to report that the resource is missing. When the resource has been acquired a Safir Log with severity Informational should be sent. Note that since it is "ok" for a dynamic resource to be temporarily missing, an Error log should be sent only after a reasonably number of retries to acquire it.

Warning log
: Use severity Warning for events that indicates that an error will occur if action is not taken, e.g. a disk is almost full.

Informational/Debug log
: For events of type Informational use the most appropriate of the severity Notice or Informational. Although it is possible to send debug logs using the Safir Log mechanism directly have a look Tracer and Backdoor functionality (see Chapter 9) for a more powerful interface for debug logging.

7.2 Safir Logging configuration

The configuration of Safir Logging is located in the file `logging.ini` (see Chapter 6) and contains parameters that determine where the log messages are sent and how newlines in log messages should be handled.

7.3 Safir Logging on Windows

Windows doesn't have built-in support for receiving syslog messages. However, you can find third-party syslog servers that target the Windows platform, both commercial and freely distributed. One example is Kiwi Syslog Server (http://www.kiwisyslog.com), but others are available.

To enable logging using syslog on Windows, the parameter `send_to_syslog_server` shall be set to true and the parameters `syslog_server_address` and `syslog_server_port` shall be set to appropriate values.

7.3.1 Windows Event Log

Safir Log also provides a way of writing syslog messages to the Windows Event Log. To enable this, set the parameter `native_logging` to true.

For native logging the event source will be set to "Safir" and the severity level parameter is mapped to a corresponding Windows Event Log Type according to:

Table 7.2: Mapping severity level to Windows event type

RFC 3164 Severity Level	**Windows Event Log Type**
Emergency, Alert, Critical, Error	Error
Warning	Warning
Notice, Informational, Debug	Information

7.4 Safir Logging on Linux

The recommended configuration on Linux is to have the system configured for native logging which means that Safir Log will use the Linux system call `syslog(...)` for generating logs. If you want the logs sent to a remote syslog server you can configure your local sysloc daemon to forward the logs.

Most Linux distributions come with a built-in syslog deamon capable of receiving and handling syslog messages, and many distributions make it possible to switch syslog daemon, in case you are not happy with the default. Refer to the documentation of your distribution for more information.

7.5 Porting guidelines

The logging interface provided by Safir SDK Core before version 4.5 (aka Software Reporting or SwReports) is now deprecated and will be removed in some future version of the SDK. This section provides some guidelines when porting old code to use the new interface provided by Safir Log.

As already mentioned, syslog has a bias towards one-liner messages which means that it could be a good idea to try to reduce the message text to contain just the most essential information.

The following table shows the recommended mapping between the report types used by Software Reporting and syslog severity level.

Table 7.3: Recommended mapping of Software Reporting types to syslog severity level

Software Reporting	**Recommended Syslog Severity Level**
Fatal Error Report	Emergency, Alert or Critical
Error Report	Error
Resource Report, missing resource	Error
Resource Report, acquried resource	Informational
Program Info Report	Debug
Programming Error Report	Emergency, Alert or Critical

Chapter 8

Node control

A Safir system consists of one ore more Safir nodes that execute on one or more computers. A single computer can host several Safir nodes, see Section 6.2.5.

Safir SDK Core provides a GUI application, Node Control, that presents information about the nodes that constitute the system. The GUI can also be used to execute Safir node stop and computer shutdown/reboot of individual nodes or the complete system, see Section 8.4.

There is also a simple command line tool that can be used to execute different stop orders.

8.1 Node Control GUI

To start the Node Control GUI run `safir_control_gui`.

In order to get system information, the GUI needs another program, `safir_status`. This program act as a gateway to the lower parts and exposes a Dob interface that is used by the GUI to control the nodes. This program also provides an entity `Safir.Dob.MirroredNodeInfo`, which lets light nodes see the node status of other light nodes.

`safir_control_gui` communicates with the local `safir_status` program which means that both programs must execute on the same node. However, it is perfectly ok to start these programs on all (or a subset) of the nodes that constitue the system. The Node Control GUI on any node can be used to view the status of the complete system as well as to stop a specific node or all nodes.

Figure 8.1 shows a system with 3 nodes. In this case all three Safir nodes run on the local computer using the loopback address. The node that is highlighted in green is the node that the gui is connected to (i.e. the local node). The circled-L symbol shows that the node is a light node. The State column shows whether a light node is *Detached* or *Attached* to the system. Full nodes always have state *Normal*.

For an explanation of what full and light nodes are, see Section 6.2.1.

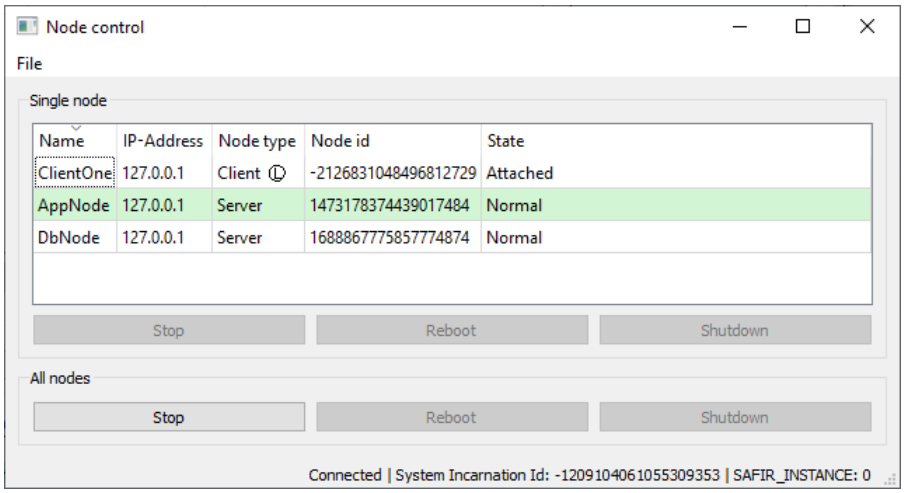

Figure 8.1: Node Control screen shot

The Reboot and Shutdown buttons will only be enabled if the parameters `Safir.Control.Parameters.RebootCommand` and `Safir.Control.Parameters.ShutdownCommand` are set to something useful.

8.2 Command Line Tool

`safir_control_cli` is a simple command line tool that can be used to execute different stop orders. This tool doesn't require the `safir_status` program to be started. Type `safir_control_cli --help` for the available command line options.

8.3 Start

The Dob is started by running the executable `safir_control`. This program will in turn start the executable that is the real framework workhorse, `dose_main`.

How and when `safir_control` is started is outside the control of Safir SDK Core. For example, the program can be started by a script at computer start-up or manually in a terminal window.

8.3.1 Why isn't there any support for start of applications?

You may wonder why there isn't any support for starting applications. The answer is that there are plans for this but as of today this is not implemented.

The application start mechanism will - when it has been implemented - provide functionality along these lines:

- Let you configure which applications to start on different nodes.
- Monitor applications.
- Show application status (fully functional, degraded etc.)
- Start of applications on fallback nodes in case of application failure.

8.4 Stop

Safir SDK Core provides functionality to stop a specific node or to stop the complete Safir system.

After a Safir node is stopped there is an option to perform a computer shutdown or reboot.

"Stop" is used in this description as a generic term for any of these stop mechanisms.

8.4.1 Safir node stop

Stop of a specific Safir node will stop all safir-related processes in a controlled fashion. At stop, `dose_main` will send stop orders to all applications that have a dob connection (see Section 4.8) which means that all application programs will be stopped as well. Note that an application might for some reason (by design or because of a bug) ignore a stop order so there is no absolute guarantee that all applications are stopped.

8.4.2 Safir system stop

A Safir system stop, that is, an order to stop all nodes that constitute a Safir system, triggers a write to the "incarnation blacklist" mechanism, see Section 6.2.4.2.

8.4.3 Shutdown

Safir SDK Core provides a mechanism to execute an optional computer shutdown after the Safir node is stopped.

Since the command to execute a computer shutdown is tightly coupled to the operating system and also in many cases involves setting specific arguments, the actual shutdown command is specified by the parameter ShutdownCommand found in Safir.Dob.NodeParameters.

For example, `/usr/bin/shutdown --poweroff now` for Linux or `C:\\WINDOWS\\System32\\shutdown /s /t 0` for Windows, could be used as shutdown command. The command must be given as an absolute path. Note that it might be necessary to make additional configurations outside Safir, like setting permissions, to be able to run the command.

Note that a shutdown initiated by this mechanism will implicitly execute a Safir node stop before the shutdown command, which means that the Safir programs will be stopped in a controlled fashion before the computer is shutdown.

8.4.4 Reboot

Much like the shutdown command Safir SDK Core also provides a mechanism to reboot the computer where a safir node is running.

The reboot command is specified by the parameter RebootCommand found in Safir.Dob.NodeParameters.

Example of a reboot command for Linux: `/usr/bin/shutdown --reboot now`, or for Windows: `C:\\WINDOWS\\System32\\shutdown /r /t 0`.

Chapter 9

Debugging support

Safir SDK Core provides support for applications to have a "back door" to make it possible to interactively enquire about the status of an application or to turn on and off debug logging. It also provides an interface that makes it easy to add debug trace logging to applications.

9.1 The bd (backdoor) command

The SDK defines a Dob message, Safir.Application.Backdoor to be used to send debugging commands to applications. Backdoor commands can either be sent using Sate (see Section 15.1), or by the command-line program "bd" (use "`bd -h`" to get info on usage).

There are some predefined backdoor commands that applications should handle:

- **Ping:** When receiving this command the application should send a Safir Log (at Debug severity) that indicates that it is alive.
- **Help:** When receiving this command the application should send a Safir Log (at Debug severity) that contains a listing of the supported backdoor commands.

Backdoor commands are Dob messages and are therefore sent to all subscribers. This means that applications have to check whether or not a backdoor message is addressed to them before acting on them. The SDK provides two classes, `Safir.Application` and `Safir.Application.Backdoor`, that simplifies the handling of backdoor commands. The classes provide this functionality:

- Setup of subscriptions for backdoor command messages.
- Checking of whether or not a backdoor command is to be handled by this application.
- Automatic answers to Ping commands.

Output from an application as a response to a backdoor command should be through a system log, preferrably at Debug level (see Chapter 7).

9.2 Tracer

The Tracer is intended to be used for developer/integration logging inside applications. It should never be used as the sole target for logging errors, since the output can be turned on and off, which means that you might not see logged errors. A common way to use trace logging is to log "significant events" or other points of interest, so that the developer can find out what his component is doing when running on a test system or when running in a situation when it is not desirable or possible to halt execution with a debugger.

The interface to the trace logging functionality is a class named `Safir.Application.Tracer` and a class named `Safir.Applic`

Tracer objects are used to send logs to various log outputs, and the TracerBackdoor is used to turn the different Tracers on and off.

9.2.1 Getting started

First a little terminology:

Tracer
: an instance of `Safir.Application.Tracer` that your code writes log lines to.

Prefix
: the name string you pass to the Tracer constructor. It is prepended to every log line from that Tracer and is the key that all enable/disable commands operate on. Several Tracer objects can share the same prefix.

The Tracer functionality is designed to be easy to use and to resemble each language's native input/output syntax.

1. Call `Safir::Application::TracerBackdoor::Start(connection)` just after your program has opened its main Dob connection.
2. Call `Safir::Application::TracerBackdoor::Stop()` when your application is shutting down.
3. Instantiate the Tracer class in each of the classes/packages where you want to use trace logging. Pass a prefix to the constructor.
4. Log to the Tracer instance (in this example the instance is called "debug"):
   - **C++**
     ```
     debug << "Testing logging " << myFloat << ", " << myInt << std::endl;
     ```
   - **C#**
     ```
     debug.WriteLine("Testing logging {0}, {1}", myFloat, myInt);
     ```
   - **Java**
     ```
     debug.println("Testing logging " + myFloat + ", " + myInt);
     ```

Use the **Safir Tracer Viewer** application for convenient manipulation and viewing of tracer logs. Figure 9.1 shows a screen shot of this program. On the left hand side you have two panels. The top panel contains all the applications that have tracers, and there is a check box for each, which allows for enabling or disabling all tracers in that application. The button with three dots opens the bottom panel which allows for turning specific prefixes (i.e. individual Tracers) on and off. It also lets you choose where the application's trace output is sent (stdout, Safir Logging, or Tracer UDP). Destination settings apply to the whole application, not to each individual tracer.

In the middle you have the logs as received by the application, with filtering capabilities at the bottom. Note that it is possible to show/hide columns by right-clicking in the table header. Copy-paste should also work as expected.

On the right hand side you have highlighting functionality, for colouring lines that contain specific regular expressions.

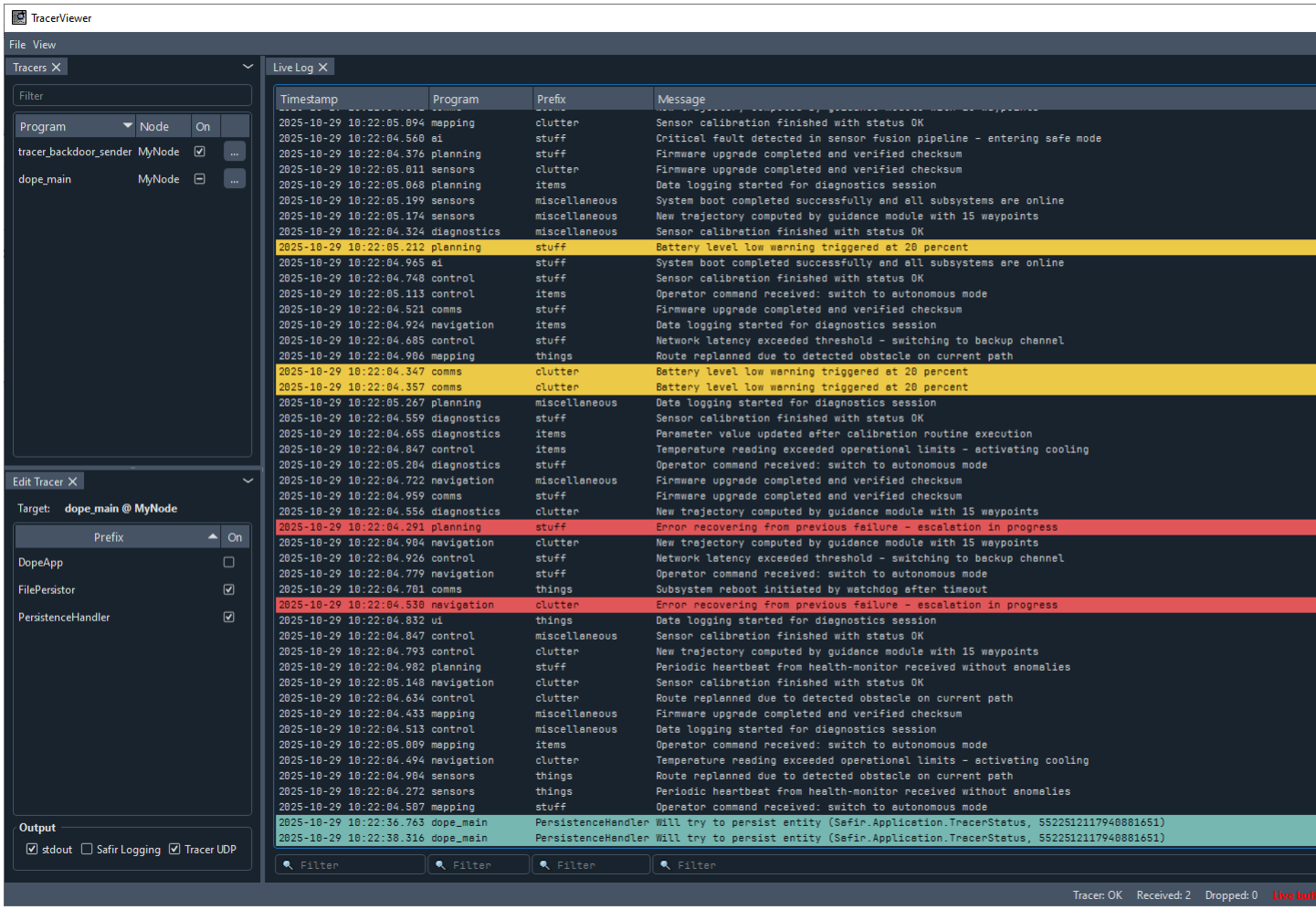

Figure 9.1: Safir Tracer Viewer screen shot

For the controls on the left hand side to work, the application `safir_status` needs to be running.

9.2.2 Details and more functionality

The Tracer has quite a few details to its implementation, that are good to understand to get the full benefit from all the functionality.

9.2.2.1 Output protocols

The Tracer is able to output to three different logging destinations, controllable per application (see the bottom panel in Figure 9.1):

- Standard output - As if logged using iostreams/printf/etc
- Safir Logging - At the Debug level of Safir Logging (Chapter 7)
- Tracer UDP protocol - A UDP Multicast protocol suited for high throughput logging

The first is really just there since it is easy to use when combined with FORCE_LOG (more below) and a terminal or a debugger. The second is there since it was the main tracer logging destination in previous Safir SDK Core releases, and is still useful since it gets combined with the error logs.

The third output option is new from Safir SDK Core 7.4 and is much more suitable for high throughput logging than the Safir Logging facility, and since it is separate from that it also makes it easier to not drown out any error logs when using the trace logging functionality for debugging.

There are two ways of accessing the Tracer UDP data, either using the Safir Tracer Viewer as shown above, or using a command line program `safir_tracer_listener`. The Safir Tracer Viewer will try to listen to whatever ports and multicast groups that are defined in `Safir.Application.TracerParameters.dou`, to make it "just work". The `safir_tracer_listener` does not read those parameters (it has no run time dependencies to either the Dob or the Typesystem), instead you have to specify ports and groups on the command line, if the defaults are changed.

9.2.2.2 Controlling

There are a few ways to control the Tracer logging:

- `bd` command - legacy command-line tool.
- Safir Tracer Viewer - GUI tool.
- `FORCE_LOG` - environment variable.

The first option, the "bd" (for backdoor) tool has been part of Safir SDK Core for a long time. It sends a `Safir.Application.Backd` message with information about what prefix to enable or disable. This message is also used for controlling generic Backdoor functionality, that is not part of the Tracer interface. (see Chapter 9). To use "bd" to turn on a particular prefix, type "bd -c <connection> <prefix> on" in a command line window (if you skip the "-c <connection>" bit all applications in the system will receive the command, which may have undesired consequences). It is possible to turn on/off all prefixes by sending "all on" to your application. Also try "bd help" to see what prefixes are registered and what their current states are).

The second, the Safir Tracer Viewer GUI, was introduced in Safir SDK Core 7.4, and uses a new entity-based facility for enabling and disabling trace logging. This requires that you also run the `safir_status` program, which will own instances of the entity `Safir.Application.TracerStatus`. How it actually works on the inside is that each application that uses Tracers will ask safir_status to maintain an entity instance containing the status of all its prefixes, along with the destinations to log to. The tracer viewer will send requests to safir_status to turn prefixes on and off, and the application will react to this. By default the entity is marked for persistence, meaning that the logging state is persisted through application or system restarts.

The third way to control logging is to use an environment variable `FORCE_LOG`, which allows you to turn logs on by default from the very start of the application. Set the variable to one or several "<prefix>" or "all". See Section 9.2.2.4 for a few hints on how this feature can be used.

9.2.2.3 Expression expansion

The checking of whether a prefix is enabled happens in slightly different ways in the different languages, which means that depending on how you use the logger you may pay for string expansion or you may not.

- In C++ the check is made once for every "<<", but since the check is inlined it can be considered cheap. Floats and suchlike are not expanded into strings until after the check has "been successful", so it is ok to log most stuff using lines like

```
debug << "Testing logging " << myFloat << ", " << myInt << std::endl;
```

- In C# the check is made for every `Write` or `WriteLine` call, which means that if you log using the form

```
debug.WriteLine("Hello " + 123.098);
```

  the whole string expansion will be performed before the check is made. It is better to use the form

```
debug.WriteLine("Hello {0}", 123.098);
```

  since the string expansion will not be performed until the check has passed.

- In Java the check is made for every `print` or `println` call that is made. The tracer supports the `printf(...)` functions that will give the similar functionality as C# described above. Check out the documentation for javas `PrintWriter` to find out how to use this syntax.

Sometimes you might have something that is expensive to calculate in your logs, for example something like

```
debug << "Average: " << ExpensiveAverageCalculation() << std::endl;
```

where the expensive function will be called every time the statement executed whether or not the prefix is enabled. In this situation it is better to check whether logging is enabled using `debug.IsEnabled()` before doing the logging, like this:

```
if (debug.IsEnabled())
{
    debug << "Average: " << ExpensiveAverageCalculation() << std::endl;
}
```

This applies to all languages, even if this example was in C++.

9.2.2.4 Uses for FORCE_LOG

There are a few different ways that the `FORCE_LOG` environment variable can be used. The need for this is less since the introduction of the persisted `Safir.Application.TracerStatus` entity way of controlling the prefixes, but it is still here for backwards compatibility, and might still be useful in some cases. Here are some ways to use it:

The first is, naturally, to set the environment variable in the System Properties → Environment Variables dialog (in Windows), or in your .bashrc file (for Unix bash shell users). This has the sometimes unfortunate side effect of turning on the selected prefixes for all applications, which can be a problem if several applications use the same prefix, or if you set FORCE_LOG to "all" which will mean that all applications will log everything.

The second way is to start your program from the command line: First run `set FORCE_LOG="something somethingelse"` (Windows again, unix bash shell users do `export FORCE_LOG=...`), and then run your application from the command line. Now only your application will be run with those settings. This same way can of course be used in a script.

Lastly, if you want to run your program from the Visual Studio debugger with trace logging on by default, there is support for that too. Under Project Settings → Debugging → Environment it is possible to set environment variables. So set FORCE_LOG="all" there to get your program to start with all logging enabled. Also make sure that "Merge Environment" is set to Yes, or your program (and the libraries it depends on) will not be able to read other environment variables.

9.2.2.5 Using the Tracer without a Dob connection

It is possible to use the Tracer functionality in an application that does not have a Dob connection, or to use it when the connection is not open. This, however, means that it is not possible to turn on and off tracers while the application is running. Instead you would have to use the FORCE_LOG functionality to enable logging. The output will only be sent to standard output and to Safir Logging, and not to the Tracer UDP protocol.

9.2.3 Tracer FAQ

How does flushing work?

Flushing is handled differently in the three output mechanisms, which means that they will behave slightly differently. Standard output is flushed as it is normally done in each language, e.g. on `std::flush` and `std::endl` in C++, and Safir Logging is flushed on each newline. Tracer UDP output does not really have flushing at all, but instead the logging lines are batched and sent "within 50ms" of the log line being flushed (i.e. after std::endl/std::flush/WriteLine/println). Note that in C++ just ending a line with "\n" will not cause a flush. You have to use std::endl or std::flush.

How does the Tracer use Windows Native Logging?

When Native logging is enabled on Windows systems (see Section 7.3.1) the Tracer will not log to Safir Logging. This is by design, since the Windows Event Log is not suited for tracer-style logs.

How does the Tracer connect to the Dob?

Since version 7.4 the Tracer has its own connection to the Dob, and a dedicated background thread to manage it. The connection name will be the name of the current executable file. But for backwards compatibility we also react to bd messages to the connection name of the application.

Do I need to do anything new with this thread thing?

No, the thread is started on calls to `TracerBackdoor::Start` and stopped on `TracerBackdoor::Stop`. If you are stopping in some uncontrolled way there may be issues with stopping the thread, but we make an effort to shut it down properly even in these situations.

What is the difference between a Tracer and a prefix?

A Tracer is the logger object you create in code, whereas the prefix is the label you supply to that Tracer. The prefix is inserted at the start of every log line and is the token that back-door tools and the Tracer Viewer use when enabling or disabling logging. Several Tracer objects can share the same prefix.

I am using Java and the Tracer uses a weird connection name?

The Tracer tries to work out the name of the current executable to use as its connection name, but with Java this is not always easy. Use `TracerBackdoor.setProgramName` to override the auto detected name.

Chapter 10

The persistence service

The persistence service is provided by the executable `dope_main`, which performs the storing of the entities to files on disk or to a database (configuration is described in Section 6.3).

10.1 Converting persisted entities to and from XML

Regardless of whether the database or file backend is used the entities are stored in binary format (blobs). There is a tool `dope_bin2xml` (run with --help to get some brief help) that converts the binary blobs into xml, which is easier to read or manipulate. The xml is placed in the xmldata column of the database, or in a file with .xml as extension (the binaries are removed when this is done, so that there will be only one source of persistence data when starting the persistence service next time).

When the persistence service starts, it first looks for xml data, and only when that is not found does it look for binary data.

This means that it is possible to run dope_bin2xml, edit the xml data, and then start the system to get the edited data presented as persistent data.

10.1.1 Preserving persisted data when objects change

The blob to/from xml mechanism also allows for a solution to a common problem:

When a dou file is changed in certain ways (a member is added or removed, or members are reordered) any old stored binary persistent data becomes unusable. But loading stored data in xml format is much more "compatible" with respect to dou-file changes, so if you run dope_bin2xml, then change the dou-files and rerun dobmake, and then start the system you are much more likely to have usable data in the persistent entities.

This works with adding members, or changing array lengths, but not with removing or renaming members (unless they are *null* in the persisted data, in which case they are not present in the xml). Of course if your persistent data is valuable, you can always edit the xml "by hand" to keep it. Just remember to do dope_bin2xml *before* you change your Dob object definitions.

10.2 Redundancy

The persistence service has support for redundancy.

To enable redundant persistent storage you need to configure your system in the following way:

- Configure the persistence storage to point out a shared network storage like a shared disk/NAS or a database/database cluster.
- Start dope_main in 2 or more nodes.

The dope_main that starts first become the active one, and the other ones starts in a standby mode. If the node running the active dope_main goes down, one of the standby instances is activated and the persistence service will continue to run.

10.3 Configuring database persistence

A complete guide to configuring databases is far outside the scope of this document, but some hints are in place.

Safir SDK Core currently has tested support for *PostgreSQL*, *MySQL/MariaDB*, *Microsoft SQL Server* and *Mimer SQL*. Since dope_main uses ODBC for database communication it is likely that it can be used with other databases as well.

As part of the installation Safir SDK Core provides examples of database installation scripts. These can be found in the in `/usr/share/safir-sdk-core/db` on Linux and in `C:\Program Files\Safir SDK Core\db` on Windows. Also provided is a README.txt file for each database, that gives some information about how to run the script. These scripts are far from complete or fool-proof, so you are expected to read them and adapt them to your needs.

Some common features of the scripts are:

Database name
: The database that the script creates will be called `dope_db`.

User name
: The user name that will be granted access to the persistence database will be `dopeuser`.

Password
: The password of the user will be `dopeuser`

In the case of Microsoft SQL Server no password will be set, as it is assumed that a login for `dopeuser` has already been created in the database.

You will probably want to modify at least the password.

10.3.1 Connection strings

dope_main connects to the database you have configured with a connection string. These differ slightly between different databases, and also we require that you set certain options in them to get correct handling of Unicode code points in the xml data.

Some databases also appear to be sensitive to casing in the connection string, and others are not.

Here are some examples of connection strings for the different databases, taken from our test suite in its Windows configuration. Also listed are any required connection string options, without which something bad will happen (usually what will happen is that Unicode strings and dope_bin2xml will fail on Windows). The connection strings are usually handled as one line of text, but they are broken up into multiple lines here, for readability.

PostgreSQL

```
DRIVER=\{PostgreSQL UNICODE(x64)\};
Server=localhost;
ByteaAsLongVarBinary=1;
LFConversion=0;
Database=dope_db;
Uid=dopeuser;
Pwd=dopeuser
```

Required options: **ByteaAsLongVarBinary=1;LFConversion=0**

MySQL/MariaDB

```
DRIVER=\{MariaDB ODBC 2.0 Driver\};
Server=localhost;
charset=utf8;
Database=dope_db;
Uid=dopeuser;
Pwd=dopeuser
```

Required options: **charset=utf8**

Microsoft SQL Server

```
Driver=\{SQL Server\};
Server=localhost\SQLEXPRESS;
Database=dope_db;
Uid=dopeuser;
Pwd=dopeuser
```

Required options: None

Mimer

```
Driver=\{MIMER\};
Protocol=tcp;
Node=localhost;
Database=dope_db;
Uid=dopeuser;
Pwd=dopeuser
```

Required options: None

Note that the Microsoft SQL Server connection string above implies that we're using mixed mode authentication, which is something you might not want to do. Change accordingly.

Chapter 11

Utilities

Safir SDK Core contains a few utilities to simplify some things.

11.1 Sending many requests

Although the Dob allows you to have several outstanding requests at a time (see also Section 6.5) it is sometimes desirable to be able to send off a bunch of requests and just get a summary of how it all went. Safir SDK Core provides a service for exactly this, Safir.Utilities.ForEach.

Foreach provides three Services:

Safir.Utilities.DeleteAllRequest
: The purpose of this service is to delete all instances of a given entity. This comes handy if you want to delete entities without knowing their instanceId, e.g. for cleaning up.

Safir.Utilities.DeleteRequest
: The purpose of this service is to delete a specified set of entities (fill an array with EntityIds).

Safir.Utilities.UpdateRequest
: The purpose of this service is to update a bunch of entities accordingly to a template object. E.g. if you want to set a flag on a large number of entities you provide a template request and a list of EntityIds that you want the request applied to.

For each service you can choose which type of answer you want. *Immediate*, *Brief* or *Full*. *Immediate* means that you don't care if the operations are successful or not and you get this response immediately before all outstanding requests are finished. Brief waits for all operations to be finished before sending a response back. This information includes a summary of successful, non-successful and total operations. Full response is like Brief but you also get every single response from each operation in an array in the response.

The foreach service is provided by the "foreach" executable, which needs to be running for the foreach requests to be serviced.

11.2 Asio and Ace Dispatchers

For applications written in C++ and using Boost.Asio or ACE the classes `Safir.Utilities.AsioDispatcher` and `Safir.Utilities.AceDispatcher` provide `Dispatcher` classes that performs the thread switch and call to `Dispatch` as described in Section 4.2.1.

Just instantiate the `AsioDispatcher` or `AceDispatcher` passing it your io_service or ACE_Reactor, and pass a pointer to the instance to your connection when calling `Open`, and you're done. The dispatcher will now perform the thread switch through the io_service/reactor main loop using `post`/`notify`.

The ACE dispatcher is deprecated as of Safir SDK Core 6.0 and will be removed in a future release. Since it is a header-only class you can make a copy of it to your own project if you need to keep it around after it has been removed.

11.3 Time

Under the namespace `Safir.Time` there are a few classes that provide functionality for obtaining the current time (both local and UTC) and converting it to and from various formats.

The time library also has support for loading an external library at runtime which could provide a higher accuracy time. The full Safir SDK provides such a library, which provides a much higher accuracy multi-node synchronised clock than NTP can provide on both GNU/Linux or Windows based systems. For more information on this, contact Saab AB (contact information at http://www.saabgroup.com).

11.4 Crash Reporting

Safir SDK Core includes a crash reporting library, google-breakpad (http://code.google.com/p/google-breakpad/). The crash reporting library can be used to generate dumps when applications crash (e.g. access violation or segmentation fault). The dumps contain, among other things, general system information, the call stack for each thread and the list of loaded modules. A dump file, together with symbolic debugging information, can be used to identify where in the software the crash occurred.

11.4.1 Enabling the Crash Reporter

To use the crash reporter you need to initiate it in your application. The initiation should occur as early in your program as possible.

For example, in C++ you would do something like this:

Initializing crash reporting in C++ using RAII object

```
int main(int argc, char* argv[])
{
    Safir::Application::ScopedCrashReporter crashReporter;
    ... run your application ...
}
```

The `CrashReporterStarter` class is an RAII class, which means that it is equivalent to doing the following:

Initializing and stopping crash reporter in C++ manually

```
int main(int argc, char* argv[])
{
    Safir::Application::CrashReporter::Start();
    try
    {
        ... run your application ...
    }
    catch (all exceptions)
    {
        ... handle your exceptions ...
    }
    Safir::Application::CrashReporter::Stop()
}
```

Initializing and stopping crash reporting in C#

```
static void Main()
{
    Safir.Application.CrashReporter.Start();
    try
    {
        ... run your application ...
    }
```

```
    catch (Exception e)
    {
        ... handle your exceptions ...
    }
    finally
    {
        Safir.Application.CrashReporter.Stop();
    }
}
```

Java has its own crash handler, so the Crash Reporter should not be used there, and there is no interface to start it.

Note
In previous versions (older than 4.5) of Safir SDK Core the Crash Reporter was started with a call to `Safir::SwReports::SwReport::EnableCrashReporting()` and stopped with a call to `Safir::SwReports::SwReport::Stop()`. This was changed due to the introduction of Safir Logging and the changes in the Tracer.

11.4.2 Requirements for using crash dumps

Build release builds with debug info enabled
: You need to make sure that you build your application with debug information enabled, even when building release builds. If you are using CMake you can use the RelWithDebInfo configuration, which produces release binaries but with debug information.

Archive the debug information
: When using Visual Studio you need to archive the generated .pdb file for every .dll or .exe you build. For GCC builds you can use the `strip` tool to separate the debug information from the binaries. The debug information shall generally not be shipped with a released product, but must be kept for it to be possible to analyze crash dumps.

11.4.3 Analyzing a crash dump

When an application crashes in a way that is caught by the crash reporter a dump file will be placed in the crash dump directory (see Section 6.1.1) with a rather cryptic name (a lot of digits followed by .dmp). Apart from generating this dump file the crash reporter will attempt to log an error to the Safir Log and to standard output that a crash has happened.

If you're using Visual Studio you can just open the .dmp file and if you've got all the debug information and binaries available you will then see the call stack and be able to inspect the state of the application. On Linux, you can use gdb to analyze the .dmp files, although you will have to convert the dumps to a core file using the minidump-2-core tool that you can find online (for example here as of this writing). For more information about how to use the dump files please see the Google Breakpad website at http://code.google.com/p/google-breakpad/.

Chapter 12

Application Design Considerations

This section discusses some things that you need to be aware of when designing systems and applications.

12.1 Overflow handling

The correct way of managing overflow is to use the designated functions telling the caller when it is ok to perform the action again.

For example, if you get an overflow when sending a message (and that message should then be retried according to your application requirements/design) you can do something like this:

Handling OverflowException

```
try
{
    m_connection.Send(myMessage,this);
}
catch (const Safir::Dob::OverflowException &)
{
    m_unsentMessage = myMessage;
}
```

And then handle resending like this:

Resending messages in OnNotMessageOverflow

```
void MyMessageSender::OnNotMessageOverflow()
{
    try
    {
        m_connection.Send(m_unsentMessage,this);
        m_unsentMessage.reset(); //set the message to NULL
    }
    catch (const Safir::Dob::OverflowException &)
    {
        //don't do anything. We will be called again later
        //to resume sending the messages that have not yet been sent.
    }
}
```

It might be tempting to use a list to keep the unsent messages in, but beware! The Dob design tries very hard to restrict queue sizes to be able to handle graceful degradation, don't ruin it by introducing infinite queues in your application! There might be cases when a queue of a *limited size* may be applicable.

Note that the retry policy of your application *must* be part of the requirements and design of the system and your application.

Another way to handle overflows, which applies if the message or request is sent as a result of a Dob subscription or request, is to use Postpone (see Section 4.5). And in the case of operator requests you probably want a tellback to ask the operator to press the button again a little bit later.

There is a little bit more info on this in Appendix C.

12.2 Request timeouts

As has been mentioned in Section 6.4 and Section 2.3.2 a request can result in a timeout response if the handler does not manage to handle the request in a timely fashion. It is very important to note that a timeout means that a request *may still be processed*!

The reason for this is that the handler may have started to process the request, but while it is being processed the timeout expires, so a timeout response is sent to the request sender. The response that the handler sends is discarded.

This means that a sender may retry sending the request, so handlers should be prepared to receive duplicate requests.

Note that this behaviour applies to both service requests and entity requests.

12.3 Type system freedoms

Since Messages, Entities, Items, and Services all inherits from Object, and it is possible to have members of the type Object, it is also possible to have Services as members in Items. And Entities as Service members etc.

12.4 Handling exceptions

There are two base classes for exceptions. `Safir.Dob.Typesystem.FundamentalException` and `Safir.Dob.Typesyste`
Exceptions should be handled by the caller, but in most situations, FundamentalExceptions should not.

In most cases a FundamentalException is to be treated as an indication that there is a programming error in your application. You should never add code that tries to handle your own programming errors. It is better if the application just dies, instead of limping along, better to detect the crash, fix and restart the application.

In some rare cases catching a FundamentalException could be justified. However, a FundamentalException of type `Safir.Dob.Types` should never be handled by an application. Let it propagate up to your main loop where you report the error and exit the program.

12.4.1 Exceptions in callbacks

Letting an exception propagate out of a callback may cause your connection to the Dob to be corrupted in strange ways. You may loose messages, requests and entity subscription responses if you try to continue execution after this has occurred.

So catch all "expected" exceptions in the callbacks.

12.5 Handling OnRevokedRegistration

For most applications a call to `OnRevokedRegistration` is completely unexpected. Unless your system uses overregistration as a feature getting this callback is a sign of an error, and the following is a recommendation on what to do in this case:

Send a Safir Log (severity Critical) that you have lost the registration. Make the report state clearly that a registration was lost, along with the type and the handler id. Then terminate the application.

The Rationale for this is that this probably happened due to a configuration error, or because someone is playing around with Sate (see Section 15.1), so you want to tell whoever may be interested that the system is probably not working correctly now.

The application should terminate, since it is no longer functioning correctly.

12.6 Set or Delete an entity before OnInitialInjectionsDone is received

The basic rule is that an application that handles persistent entity types should wait for the callback OnIntialInjectonsDone before trying to set or delete any entity. (This callback signals that all ghosts have been injected). To get this to work the application has to distribute the "ok-to-set-entity" status to its different parts which, for some application designs, could be a hassle. An alternative in this case is to actually catch the GhostExistsException. (This is the exception you get when trying to set or delete an entity instance for which there is a ghost that hasn't been delivered yet).

12.7 Multithreading

A connection can only be used from within one thread at a time. The queues and other states associated with a connection are not thread safe, for simplicity and efficiency.

12.8 Hot-standby / Redundancy

Using a combination of persistence and pending registrations (see Section 4.3.6 and Section 4.7, respectively) will let you support hot-standby and redundancy.

For an application to support redundancy the idea is that it is started in (at least) two instances, most likely on different computers in the system. At startup one is chosen as *active*, and the other(s) become *passive*. If the active instance fails (e.g. the application or the computer crashes, or the computer is shut down for maintenance) the Dob will detect that that computer has gone down, or that the application has crashed, and will tell one of the passive instances to become active.

To accomplish this, all the entities that the active instance maintained must be marked as persistent (at least volatile persistence), and one of the handlers, the "main" handler, must be registered using `RegisterEntityHandlerPending`.

When the `OnCompletedRegistration` callback is invoked for the main handler, the rest of the handlers should be registered in the normal way (using overregistration) guaranteeing that one instance of the application has registered all handlers it is supposed to.

The first time the application instances start, one of the instances will be given the registration of the main handler (through a call to `OnCompletedRegistration`, which will lead it to register the other handlers as well. It will then receive any persisted entities through the `OnInjectedNewEntity`. When this instance terminates (due to a crash or something else) the Dob will detect that and call `OnCompletedRegistration` on the other application instance, that will register the other handlers and receive the persisted entities. This will seamlessly cause the entity instances to move from one owner to another.

Figure 12.1 describes this as a sequence diagram, albeit without showing the part about registering the other handlers, since that would clutter up the diagram needlessly.

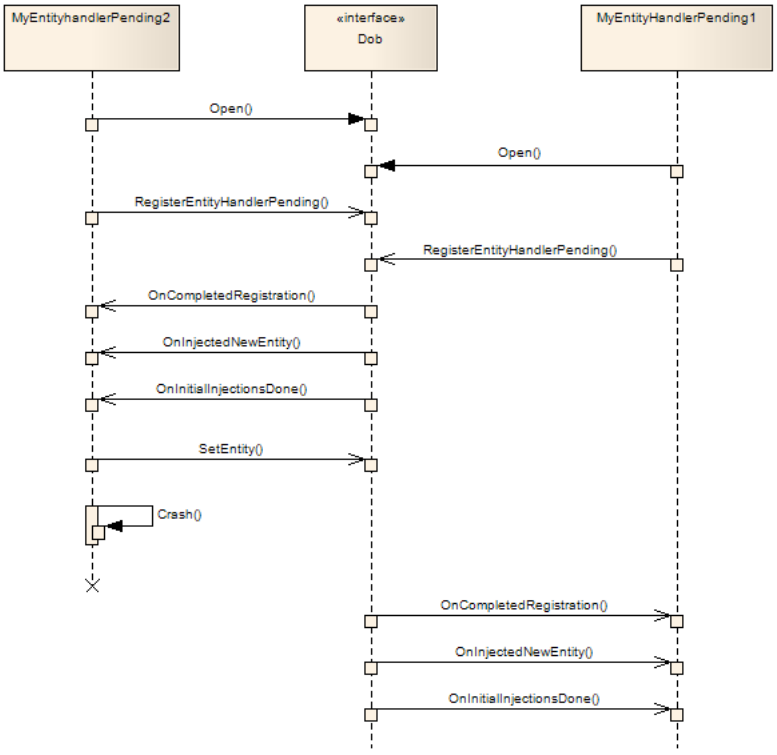

Figure 12.1: Sequence diagram for redundant entity handlers

Note that applications subscribing to the entities that are redundancy-handled *may* see the entity instances as deleted for a short while, since in reality the handler was unregistered for a short while.

12.9 Entity reference gotchas

When using entities that are linked to each other through EntityId members there are a few things that you need to remember/handle in both the application that produces the entities and the applications that use them.

Applications that own the entities must make sure that no *orphans* are left behind, and that incorrect dangling references are avoided. An example of orphaned entites is shown in Figure 12.2, where an entity uses another entity, a Location, to represent its position. If the entity is updated with a new location you must determine whether the old location is to be kept or not, to avoid cluttering the system with orphaned locations.

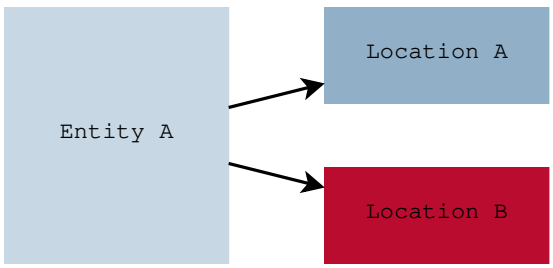

Figure 12.2: Entities that may cause orphans.

For applications that use the entities that contain associations you need to be aware that the Dob does not guarantee the order in which entities are delivered to subscribers, and that while you're handling a subscription an associated entity may be removed, before you've managed to Read it.

There is no surefire way to handle all these cases, it must be decided on a case by case basis, but a first recommendation is to only subscribe to the "main" entities, and do Reads or temporary subscriptions to get the referred entities.

This also applies to applications that handle asynchronous injections, as described in Chapter 13.

Chapter 13

Systems of Systems [adv]

This chapter contains information about how to write applications that support *Systems of Systems*, i.e. a number of Dob-systems communicating over some other media, using *asynchronous injections*.

The Dob itself relies on high reliability and high bandwidth LAN connections for its communication between nodes. Asynchronous injections makes it possible to create an application that links together multiple systems like this using a medium with low connectivity and low bandwidth (e.g. a legacy radio), or a medium with high bandwidth but non-optimal connectivity (e.g. the internet).

A complete description of how to build a system with asynchronous injections, or how to create an *injector*, i.e. an application that performs the injections, is outside the scope if this document, which will concentrate on how to write applications that can handle injections. If you're interested in more information on this than this document provides, please get in touch with us.

But, we need a little bit of background to be able to understand the rest of this chapter:

To save on bandwidth injections are sent as *deltas* over the network. Since some data may reach one system long before it reaches the others (might have been out of radio contact) all data is timestamped, and upon injection the data is merged using these timestamps, to ensure that all the "latest" data is what is used in all systems.

In a system of systems with low connectivity it is also possible that the same entity is being updated while the network is down, and the timestamps are also used to work out which data is the latest.

13.1 Injectable entities

An entity is configured to be *Injectable* using the same configuration mechanism as persistence, as described in Section 4.3.6, but using the `Injectable` persistence type. Entities marked as `Injectable` will have the behaviour of `SynchronousPermanent` (and thus also of `SynchronousVolatile`) entities, with the added functionality that injections can occur throughout the lifetime of an entity, not just at registration-time.

13.2 Asynchronous Injections

Asynchronous Injections are the kind of injection that can occur throughout the lifetime of an entity. The other use of injections is Persistence, as described in Section 4.3.6, can only occur at *registration time*.

The first thing to mention is that if a type is marked as *Injectable* its instances will be persisted, just as if it was marked as `SynchronousPermanent`. Note, however, that it is not the persistence service, dope_main, that performs the persistence in this case, but the *injector*, but this should be completely transparent for the user.

13.2.1 Using asynchronous injections

For an entity to support asynchronous injections, it must have the Safir.Dob.InjectionProperty mapped with a value of "Injectable", and the application must register it using `RegisterEntityHandlerInjection` and an entity handler that implements the `EntityHandlerInjection` consumer interface.

The `EntityHandlerInjection` consumer interface has, as described in Section 4.3.6.5, four callbacks more than the EntityHandler consumer interface: `OnInjectedNewEntity`, `OnInjectedUpdatedEntity`, `OnInjectedDeletedEntity` and `OnInitialInjectionsDone`. These are the circumstances that the callbacks are invoked (the sequence is shown in Figure 13.1):

`OnInjectedNewEntity`

- Immediately after registration for each persisted entity instance, just as for types marked for persistence.
- Any time during the lifetime of an application if a *new instance is injected* by the injector.

`OnInjectedUpdatedEntity`
When the injector injects some changes into an existing entity instance.

`OnInjectedDeletedEntity`
When the injector injects a delete. I.e. the instance is deleted on another system.

`OnInitialInjectionsDone`
After all persisted entity instances have been resurrected or rejected by an `OnInjectedNewEntity` callback.

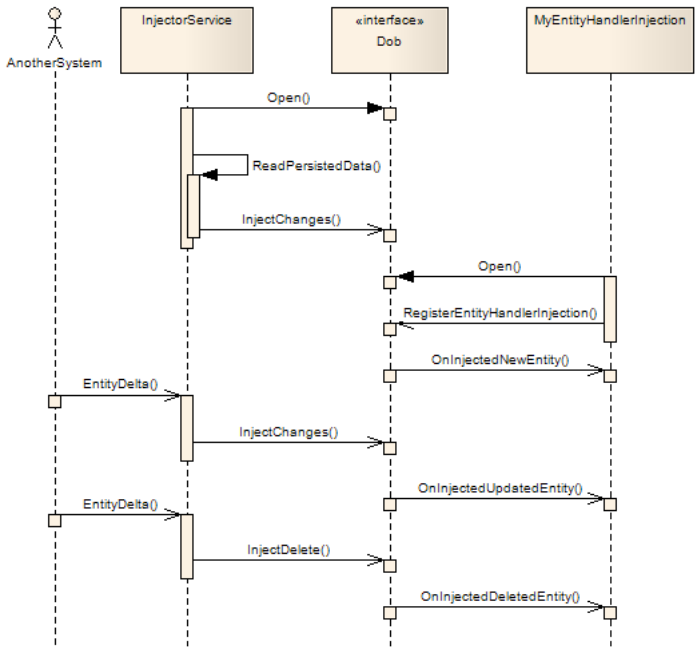

Figure 13.1: Injection sequence diagram

Note that it is not possible to tell the difference between the two kinds of calls to `OnInjectedNewEntity`, and in fact both new and old (i.e. persisted and injected) data can be mixed (i.e. timestamp merged) in a call to `OnInjectedNewEntity`.

When an application receives one of the injection callbacks it can either accept the injection, by just returning, or it can say that the injection is not complete (i.e. all deltas that are needed to make an entity that makes sense have not arrived yet) by calling `IncompleteInjectionState` (see Section 13.2.3) or it can explicitly delete the entity. What is important to note is that if the entity is deleted is that that will cause the delete to be sent to other systems (by the injector), so delete may not be a good idea.

For injections to work correctly it is important that applications use `SetChanges` instead of `SetAll` when modifying entities. This is because `SetChanges` will use the change flags to work out which members should have the new and updated timestamp. In fact *always* prefer `SetChanges` to `SetAll`, it makes more sense w.r.t. change flags even in entities that are not injectable, and it will probably make it easier to add the possibility of handling injections at a later time to your application.

13.2.2 Timestamp merges

Each top-level member (note that it is *only* the top-level members, not members in items or individual array items) have a timestamp that is used to determine which bits of information is "newest".

Note: This fact gives constraints on the design of objects that are to be used with asynchronous injections. They will probably have to be quite "shallow".

The timestamps are based on a normal concept of time, but have an additional feature to make them better adjusted to use when the clocks on different systems (in a system of systems) "drift": When updating an entity instance the time used for timestamps is the maximum of "my own time" and "the highest time I've seen from other systems, plus one". This solves problems in systems with wildly diverging system times. (The timestamps can be thought of as Lamport timestamps, but with the always moving along with the UTC time, not only when there is an event.)

An example of how timestamp merges work is probably the best way to explain the algorithm:

In Table 13.1 an application owns an entity (and has done a single `SetChanges` at time 100). The entity has been reflected to other systems by the Injector.

Table 13.1: Timestamp merge, initial entity

Member	Timestamp	Value
0	100	Foo
1	100	*Null*
2	100	Bar
3	100	*Null*

On another system (at time 110) the application does a `SetChanges` with Member 0 set to "hello" and Member 1 set to "33". This will cause the injector on "our" system to make the injection shown in Table 13.2.

Table 13.2: Timestamp merge, first injection

Member	IsChanged	Timestamp	Value
0	**True**	**110**	**hello**
1	**True**	**110**	**33**
2	False	0	*Null*
3	False	0	*Null*

This will be merged with the current entity (again, on "our" system) into the object in Table 13.3.

Table 13.3: Timestamp merge, merged object

Member	IsChanged	Timestamp	Value
0	**True**	**110**	**hello**
1	**True**	**110**	**33**

Table 13.3: (continued)

Member	IsChanged	Timestamp	Value
2	False	100	Bar
3	False	100	*Null*

This merged object will be presented to the application through a call to `OnInjectedUpdatedEntity` with the change flags set as indicated, and if the application accepts that injection (by just returning) the entity instance shown in Table 13.4 will be set in the Dob (and shown to subscribers or readers).

Table 13.4: Timestamp merge, resultant entity

Member	Timestamp	Value
0	110	hello
1	110	33
2	100	Bar
3	100	*Null*

While all this was happening, a third system (at time 105) the application did a SetChanges with Member 0 set to "Lemon" and Member 3 set to "Curry". On our system the injector will then do the injection shown in Table 13.5 (but note that this delta arrives to our system after the 110-delta).

Table 13.5: Timestamp merge, another injection

Member	IsChanged	Timestamp	Value
0	**True**	**105**	**Lemon**
1	False	0	*Null*
2	False	0	*Null*
3	**True**	**105**	**Curry**

Which will be merged with the current entity to become the object shown in Table 13.6

Table 13.6: Timestamp merge, second merge result

Member	IsChanged	Timestamp	Value
0	False	110	hello
1	False	110	33
2	False	100	bar
3	**True**	**105**	**Curry**

This object will be presented to the application and, if accepted, will be set into the Dob and shown to subscribers and readers.

At this stage all systems will have the exact same entity, since the timestamp merge is completely predictable, regardless of the order that the deltas arrive.

13.2.3 IncompleteInjectionState

The method `IncompleteInjectionState` (can be found in the `ConnectionAspectPostpone` aspect) can be used if an application thinks that an asynchronous injection (in one of the injection callbacks) is missing some information which should be injected "soon". Since entities are updated as deltas it is possible that two deltas arrive in the "wrong order" (due to bad or low connectivity). The application may be able to detect this, and decide that it wants to wait for more entity information. A call to IncompleteInjectionState will cause the injection to be held back until another injection is received for the same instance, when the merged injections will be presented to the application again.

Chapter 14

Contexts [adv]

The Dob makes it possible to have several "data universes" side by side. Such a universe is called a context.

The main principle is that there is no mixing of data (entities, messages etc) between different contexts.

It is possible to override the default no-mixing behaviour by marking a type as *ContextShared*, thereby making it visible in all contexts. (See below for a motivation and explanation of the ContextShared mechanism.)

14.1 What contexts can be used for

This description presumes that the context functionality is used to support different system modes (changing either the whole system mode, or just one operator console). However, note that from a Safir SDK perspective the context functionality is a general mechanism which can be used for any purpose.

The system modes that systems may want to support includes:

- Normal - displaying real world data.
- Replay - replaying data that was previously recorded in a Normal session.
- Simulation - displaying data that is generated by simulators. Typically used for training.

Here we will focus on Normal and Replay. Simulation should be possible to support with this design, but any recommendations how to implement Simulation using Safir are not included for now.

The safety issue is very important when mixing real data sessions and Replay sessions and the intention of the design is that it should be as difficult as possible to do things like "shoot a real gun at a replayed target".

Safir applications are usually classified as either Business applications or Presentation applications (for simplicity we will refer to these as *Apps* and *GUIs* in the rest of this chapter). The first type handles all business logic while the latter takes care of all presentation of data in a GUI (see Appendix A). This section contains some recommendations for how APPs and GUIs should use contexts to support Replay sessions.

14.2 Design principles

A Dob connection is always related to one, and only one, context. The context is given as parameter in the Open call. The contexts are numbered from 0 and upwards where 0 is used to identify the Normal context. Any data (except the ContextShared types, see below), such as entities, services and messages, produced by a connection, can only be seen by connections that is opened in the same context.

In a system that supports Replay there must be at least some data that is control data, i.e. some things that need to be displayed from the Normal context while we're replaying. Typically this is the entities that control the Replay, software error reports, and

a few more things. Fort example, a GUI that is showing a Replay session must be able to pick up changes to the Replay control entities, so that it will know when to switch out of Replay and back to Normal.

Forcing applications to have multiple Dob connections, one in context 0 for the control data, and one in the Replay context, and then dynamically working out which kind of data goes where, is considered a potential security risk. Therefore, Safir SDK Core has a concept of ContextShared types which eliminates the need to have parallel connections to different contexts.

14.3 ContextShared types

A type could be marked as being visible in all contexts by mapping the ContextShared property to it. ContextShared types are typically control types, e.g. types used to control the system mode.

For example, an APP or GUI that is connected to context 0 can "see" all that goes on in context 0 and the ContextShared types. An APP or GUI that is connected to context 1 can "see" all that goes on in context 1 and the ContextShared types.

The following rules applies to ContextShared types:

- A ContextShared Message can only be sent from context 0.
- A ContextShared Service can only be registered in context 0.
- A ContextShared Entity can only be registered and set from context 0.
- Requests on ContextShared Services and Entities can be done from any context.
- ContextShared Messages, Entities and Registration can be subscribed to from any context.
- It is possible to iterate over ContextShared Entities from any context.
- In all other respects all contexts are equal, i.e. all Dob functionality is available in all contexts.
- It is not possible to override the ContextShared property. Thus, types derived from a type with the ContextShared property will also be ContextShared.

These rules prevent a Replay application from accidentally (due to misconfiguration) replaying data that is configured to be ContextShared. If ContextShared types could be produced from any context nothing would stop the Replay application from over-registering a ContextShared entity and producing entity updates that may produce an inconsistent system leading to a number of security issues.

ContextShared types removes the need for GUIs to have connections to several contexts at once. Without ContextShared types GUIs would have to have one connection to context 0 that subscribes to the control entities, and one connection to the context that contains the data that it should currently be displaying. So the GUI would have two connections, and it would probably be easy to use the wrong connection, for example sending a "fire" request accidentally on the context 0 connection instead of on the Replay connection (where it would get ignored).

By introducing ContextShared types this same GUI will only have to have one connection to the Dob. It can subscribe to both the control entities and the data that it displays through the same connection. This makes it impossible for this GUI to send a request in the wrong context, since it only knows of one context at a time.

14.4 Using the context mechanism

This section contains some hints on how APPs and GUIs should be designed to support Replay in different kind of systems.

14.4.1 Terminology

- Replay - the mode that a console is in while showing Replay data.
- ReplayApp - the component that performs the Replay. Typically an application that reads from a database produced by a "recorder" application, and sets entities and sends messages as if they were owned/sent by the real applications.

14.4.2 Each operator console has one mode (Type 1)

This is a multinode system with one or more server nodes (that is running the APPs) and one or (probably) more operator consoles (running the GUIs). Each console can display either the Normal data, or one Replay session. Different consoles can be in different modes, and several consoles can display the same Replay session.

An elaborate example would be: Console 1 and 2 are in Normal mode, console 3 is replaying yesterdays recorded data at three times the speed, and console 4 and 5 is looking at the Replay of data from last week.

ReplayApp runs in one of the server nodes, and there is a GUI in each of the operator consoles that allow the operator to start a Replay session, or to "attach" to a Replay session that another operator is running. This results in changes to some control entities (owned by the ReplayApp) that reflect the mode of each console.

The APPs whose responsibility it is to maintain the Normal context data are not aware of any of the mode changes (the server nodes don't switch modes), and they only connect to context 0.

GUIs that are meant to show Replay data must subscribe to the ReplayApp control entities, and when a mode change occurs they must reconnect to the Dob in the desired Replay context.

There might be some GUIs in the console that does not listen to the mode change, e.g. the GUI that shows alerts.

14.4.3 StandAlone system supporting one mode (Type 2)

Very similar to a Type 1 system. The only difference is that the APPs are running on the same node as the GUIs. A correct implemented APP or GUI will work without any modification in both a Type 1 and a Type 2 system.

14.4.4 Starting extra GUIs showing a Replay session (Type 3)

In this kind of system Replay sessions are started in a new set of windows, enabling the user to look at both Normal and one or more Replay sessions at the same time. When a Replay session is started a new set of GUIs (either new instances of the ones that show Normal data, or special replay-GUIs) are started to show the Replay data.

In this case the GUIs don't listen to the ReplayApp control entities. The GUIs that were started in the Normal context stay in that context. When the operator wants to start a Replay session the system will launch a new set of GUIs (either new instances of the Normal GUIs, or special replay-GUIs), telling them which context to connect to.

GUIs in this type of system must have a way of being told which context to connect to, and this could be either a command line parameter, or by setting an environment variable (the latter is probably better, since it makes it easy for plugins and libraries to get at the value, without having to pass around and parse the command reliably).

14.4.5 Several Replay sessions in one console (Type 4)

In this kind of system the user interface shows replayed and real objects intermingled, allowing operations on all objects.

The GUIs in this type of system must connect to all contexts that are used on the console. The GUIs have many Dob connections, and they must be explicitly be written to be fully aware of which objects belong to which contexts, and what operations are legal on which objects, e.g. knowing that a track from context 3 (Replay) cannot be sent to the Fire service in context 0.

Although supported, this kind of system is usually considered dangerous, since the possibility for mistaking the nature of an object is high.

14.5 APP design

Almost all applications should only be aware of context 0. An obvious exception to this rule is the the ReplayApp itself.

14.6 GUI design

In a Type 1 or Type 2 system a well behaved GUI should be designed according to the following rules:

All GUIs (that want to be able to show Replay data) must subscribe to a ContextShared entity that shows which context the console should "display".

When the GUI gets notified that the console is switching context it:

- Closes its current connection.
- Opens the connection in the "new" context.
- Tells all its "parts" to restart, i.e
  - Clear all internal state information.
  - Attach to the connection.
  - Set up subscriptions again (A GUI normally does not own entities, but if it does, registrations and creation of entities must be done again.)
  - Restart any BackdoorKeepers (these use message subscriptions internally which are lost in connection Close).

The attach and subscription are the same actions that the GUI must take when it starts up the first time. What is very important when changing context (but not when starting for the first time) is the clearing of internal state information. Leaving any internal state information is a serious safety risk!

A ReplayApp will probably send error responses to all requests on entities it owns. Therefore, GUIs in Replay mode needs to disable buttons etc that would generate requests, or be able to ignore the error responses from the ReplayApp.

Chapter 15

Test support and Tools

Safir SDK Core contains some tools that are useful for, among other things, application and system testing.

15.1 Sate

There is an application in Safir SDK Core, Sate (stands for *Safir Application Tester*), which is very useful for testing applications that use the Dob. In short it is a GUI application that allows you to perform most operations that the Dob provides interactively.

Sate has recently (as of version 7.3.3) had a major rewrite. The old version is now called sate_legacy, in case you want to use it instead.

For example, you can set up a subscription to an entity, register entity handlers and set entity instances. You can subscribe to and send messages etc. This can be very useful for interacting with an application, for example before the real HMI is implemented fully, or pretending to be another application that your application communicates with.

Figure 15.1 shows Sate with the Safir.Dob.ProcessInfo entity opened. All the Dob classes are listed to the left, where you can right-click to get operations you can do on the types. The middle tab shows all instances of the entity, and the right tab shows an object editor for a single instance of the entity. The right-hand side of the right tab contains buttons for operations you can do on the opened instance.

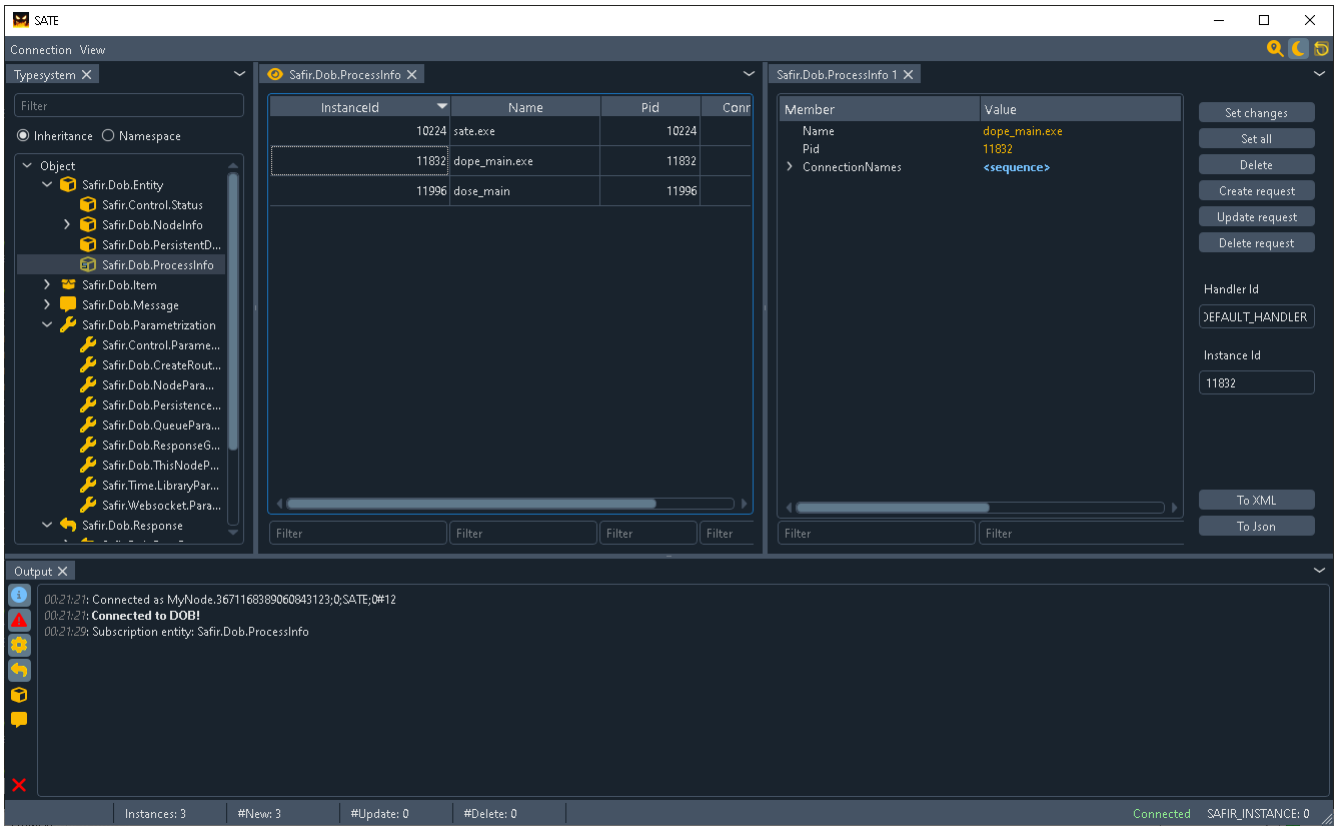

Figure 15.1: Sate screen shot

The lower part shows a history log of what "has happened", e.g. showing if a request has been successfully sent, etc. There are buttons to toggle what type of events are shown in the log. By default the more verbose types of logs are disabled, so as not to spam the log window with uninteresting information.

15.1.1 Scripts in Sate

As of version 7.4 Sate has the ability to automatically run a sequence of commands stored in a script file. This is useful for tests or for quickly performing complex and often repeated tasks.

The script is a JSON array of Dob operations. Each operation is a JSON object with the same syntax as the websocket api (see Appendix B). For readability the fields `jsonrpc` and `id` can be omitted in scripts. In addition to all the operations defined by the websocket api, scripts support one extra object for inserting delays between operations, and a few macros to generate random values. The delay object specifies a delay in milliseconds like this:

```
1 {"method": "delay", "params": {"time": 1000}}
```

The macros that are currently supported are

- `_INSTANCEID_` - Generate a random InstanceId.
- `_NUMBER_<min>_<max>_` - Generate a random number between MIN and MAX. Boundaries included.
- `_STRING_<minLen>_<maxLen>_` - Generate a random string with length between minLen and maxLen.

All macros must be inserted as strings even when the result will generate a number. This is because the script file should always contain valid JSON.

Here is an example of a script that registers an entity handler, starts a subscription, and set two entities with some random values:

An example Sate script.

```
[
  {
    "method": "registerEntityHandler",
    "params": {
      "typeId": "Test.MyEntity"
    }
  },
  {
    "method": "subscribeEntity",
    "params": {
      "typeId": "Test.MyEntity"
    }
  },
  {
    "method": "setEntity",
    "params": {
      "entity": {
        "MyNumber": 1,
        "MyString": "Hello world",
        "_DouType": "Test.MyEntity"
      },
      "instanceId": 123
    }
  },
  {
    "method": "setEntity",
    "params": {
      "entity": {
        "MyNumber": "_NUMBER_-10_10_",
        "MyString": "_STRING_3_10_",
        "_DouType": "Test.MyEntity"
      },
      "instanceId": "_INSTANCEID_"
    }
  }
]
```

It is possible to record all Dob operations as a script from Sate. This is done by opening the Script Recorder from the menu *View → Script Recorder*. In the Settings menu *View→Settings* it is also possible to make Sate automatically run a script at startup.

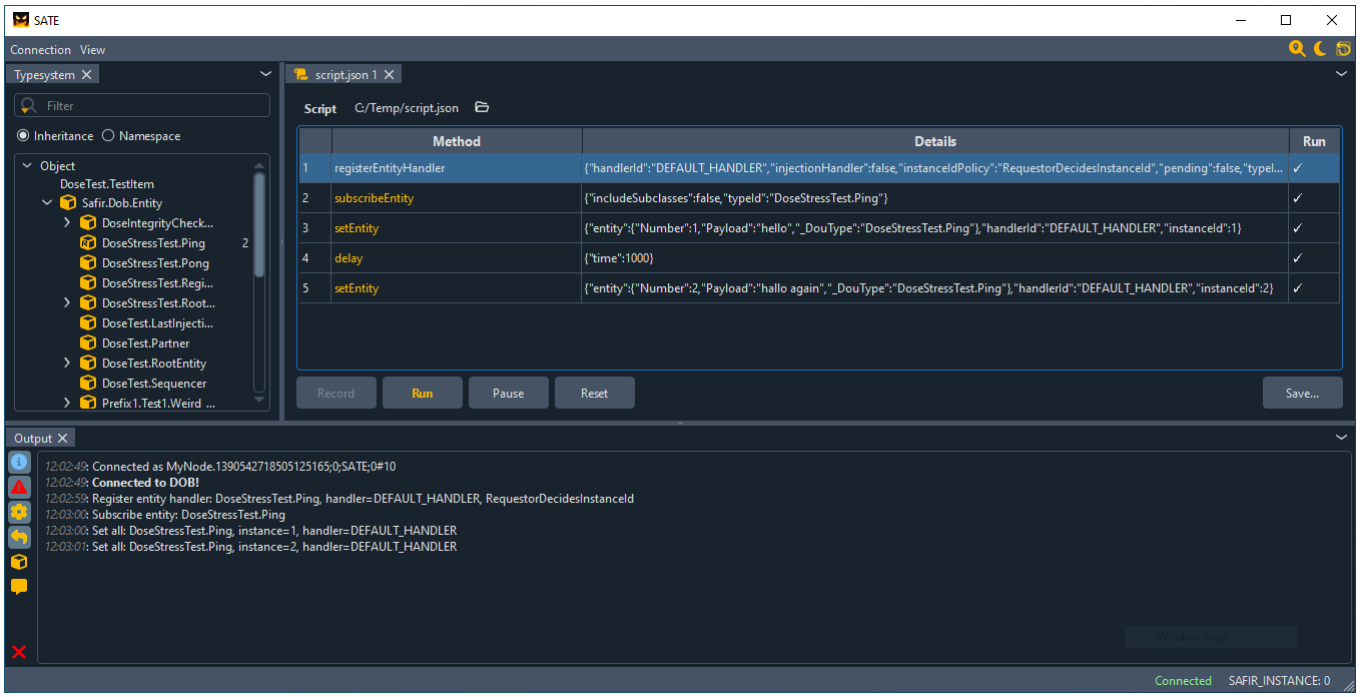

Figure 15.2: Sate script widget is used to execute and record scripts.

Finally, Sate supports script execution from the command line. This feature can be useful in test environments. The syntax to run a script from the command line is `sate --script <script-path>`. For a complete list of the command line options use the `--help` command line option.

15.2 Dobexplorer

Dobexplorer is a tool that is primarily meant for the developers of the Dob, but it has some features that are useful for users of the Safir SDK Core too. New features are added to dobexplorer as they are needed by the Dob developers.

Figure 15.3 shows dobexplorer displaying a graph of the shared memory usage of the Dob. (The amount of memory available is configured in Safir.Dob.NodeParameters, where you also have to look to find out what 100% means.)

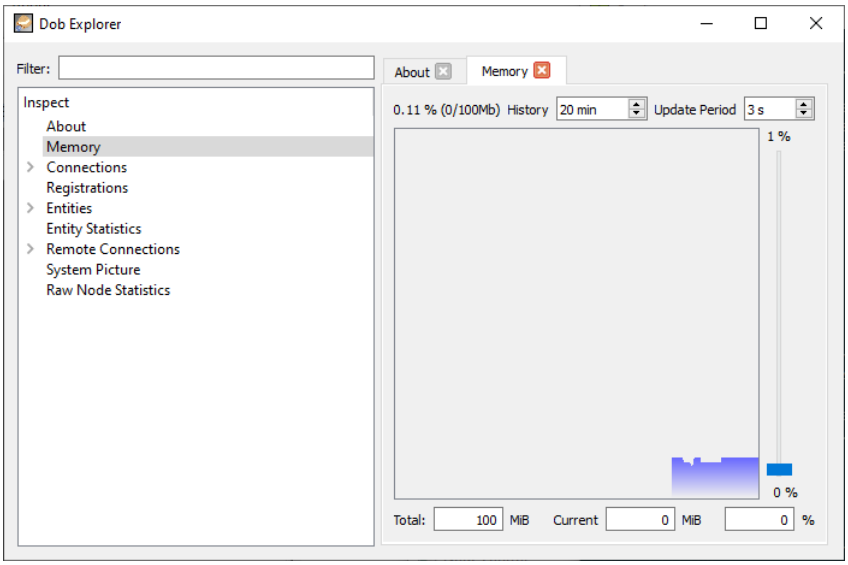

Figure 15.3: Dobexplorer showing a graph of memory usage.

Figure 15.4 shows dobexplorer displaying a table of the node statuses of a multinode system.

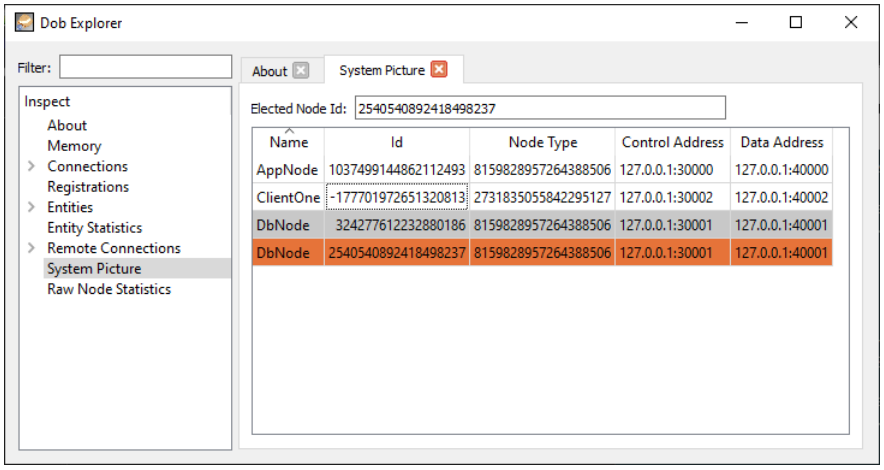

Figure 15.4: Dobexplorer showing node statuses.

15.3 Tool Launcher

The `safir_tool_launcher` tool is useful for launching the different debug tools of Safir SDK Core. Especially useful when there are several nodes running on the same computer (see Section 6.2.5)), as it is possible to specify which SAFIR_INSTANCE the launched tool should connect to.

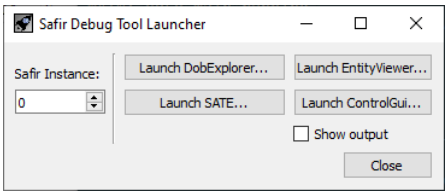

Figure 15.5: Tool Launcher screen shot

15.4 dots_configuration_check

`dots_configuration_check` is a command line tool that can be used to quickly validate any set of dou-files without having to run dobmake. It can also be used to look into the typesystem and retrieve information like:

- Is the typesystem already loaded by another application.
- Get type name that belongs to a specific typeId or vice versa.
- Get parameter values.
- Make a complete textual output of the loaded typesystem.

By typing "`dots_configuration_check --help`" all available options are displayed.

Chapter 16

More help

Apart from this document there are other sources for help.

16.1 Doxygen help

All the language interfaces have comments that can be used to generate documentation (doxygen for C++, javadoc for Java, etc).

The generated doxygen documentation is included in the installation packages, and is located under /usr/share/doc/safir-sdk-core on Linux and under the Start Menu in Windows.

Use this documentation! It is the place where interface details are explained!

16.2 Asking questions and reporting bugs

Questions can be sent directly to us using the contact form or by asking a question in our Google+ Community. Links to both can be found at http://safirsdkcore.com.

Bug reports can either be sent directly to us, or by creating an issue on our GitHub page https://github.com/SafirSDK/safir-sdk-core/issues. When you report a bug, please include as much information as possible, e.g. version information and platform information. If you experience a crash in any part of Safir SDK Core, please include any crash dumps that were created.

Chapter 17

Glossary

This glossary mostly contains Safir-related acronyms and terms. General computer terms are not included, like "UDP", "Multicast" and "ODBC". Wikipedia, for example, is a much better source for explanations of those terms than this glossary could ever be.

Dob

Distributed Objects. Consists of the components Dose and Dots.

Dom

Distributed Objects Mapping. File type for mapping Dob classes to properties (in Xml).

Dope

The Safir SDK Core component that provides the persistent storage of Dob entities.

Dose

Safir component that provides the object distribution.

Dots

Safir component that provides the type system.

Dou

Distributed Objects Unit. File type for defining Dob classes (in Xml).

HMI

Human-Machine Interface.

Safir

Software Architecture For Information And Real-time systems.

Safir SDK

The technical platform of Safir.

Safir SDK Core

The core part of the Safir SDK.

SDK

Software Development Kit.

Swre

A Safir SDK Core component that provides Crash Reporting, Error/Event Logging and Trace logging functionality.

Appendix A

Example applications

Note

This section mentions *VehicleDb* several times. This application has in fact been removed, since Safir SDK Core no longer contains an ODBC wrapper. This section in the documentation has not been reworked, though.

Along with the Safir SDK Core you should have received some example applications that show how to create a small application that will give you a starting point for your own applications. This appendix explains what these example applications intend to do, and why they were designed the way they were. These examples are also used in the Safir SDK training courses, so if you've attended one of them you will be in familiar territory.

The example applications form a very small and simple Safir system. They may execute on one single node or be spread out on different nodes. Since the Dob provides interfaces in several different languages, the example applications are also provided in different languages.

A.1 Some background

The basic design tenets of Safir systems is the separation of business logic from GUI, and of breaking up responsibilities into several applications that each solve a small part of the problem.

So Safir systems usually consist of a number of applications that execute on different computers (nodes), and they are classified as either *Business applications* or *Presentation applications*. The first type handles all business logic – calculation, communication, request processing etc – while the latter takes care of all presentation of data in a GUI. This is shown in Figure A.1.

The applications may execute on the same or different nodes, this is in fact one of the features of Safir SDK, that a developer can run everything on his development-machine, but when the system is deployed to the real or test environment the applications are run on multiple machines, completely transparently to the applications.

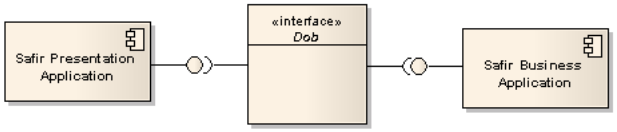

Figure A.1: Separation of logic from GUI

A.2 The (example) problem

The fictive problem is that we need a system that can give us an overview of a number of manually created vehicles. We also want some services associated with the vehicles.

Presentation of information

The vehicle objects shall be distributed to all system nodes with real-time requirements. The critical information about a vehicle (position and speed) shall be presented in a table (applies to all vehicles) and in a detailed window (applies to a selected vehicle). It shall be possible to edit some of the vehicle information. All changes shall immediately be reflected on all nodes.

In addition to the real-time requirements, it shall be possible to store and retrieve some information about a vehicle in a database. This information is not real-time critical and is only to be available upon request (i.e. it is not automatically distributed to all nodes).

Capacity warning

If the number of created vehicles in the system reaches a parameter-specified limit, we want some kind of warning to be sent to all presentation nodes.

Speed difference calculation

It shall be possible to calculate the difference between the speed for a selected vehicle and a given speed. We want to be able to use this calculation algorithm to all objects that have a speed – not only vehicles.

A.3 The solution

The problem is solved by the following applications:

VehicleApp

A Safir business application that is the owner of all vehicle objects. It is designed to execute on one server node and has no GUI.

VehicleMmi

A Safir presentation application that presents all vehicle information in a GUI. It is designed to execute on any number of presentation nodes.

VehicleDb

A Safir database application that on request interacts with a database through the (now deprecated) Safir ODBC database interface. It is designed to execute on one server node that also runs the ODBC database.

A deployment of the applications is shown in Figure A.2 where the applications execute on one standalone node, but they could just as well execute on different nodes.

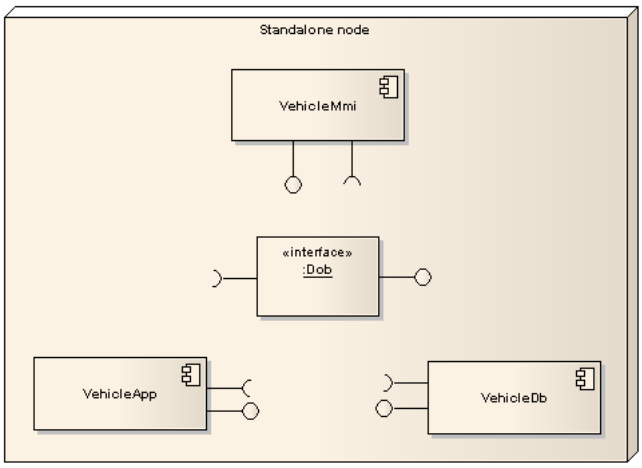

Figure A.2: Deployment of the Vehicle system

A.4 VehicleApp - Business Application

This application exists in the languages C++ and C#.

VehicleApp is the business application in the system. It is the owner of all vehicle entity instances and is therefore the only application with right to create, update and delete vehicle instances. Other applications may send vehicle object requests but it is VehicleApps responsibility to check the requests and perform the changes.

The vehicle information is modelled as global Dob entities, which will ensure that all object updates are distributed on all nodes with real-time requirements.

To be an owner of vehicle objects, VehicleApp uses the Dob interface `EntityHandlerInjection`. This means the following things: - The application will not be a pending owner, i.e. it will override any current owners. - The application will allow injections from other systems and from the persistency service.

VehicleApp handles and responds to create, update and delete requests in the implementation of the interface `EntityHandlerInject`
The registration is also performed here.

To send a warning when the number of vehicle parameters is reached, a `Message` is used. To send a message, no registration is required, but the Dob interface `MessageSender` has to implemented.

A.4.1 Dou-files

The following dou files are provided with VehicleApp. For details, see the corresponding dou file.

`Capabilities.Vehicles.Vehicle`
Definition of a vehicle object. This data will be distributed in the system.

`Capabilities.Vehicles.VehicleCategoryCode`
Enumeration of vehicle category codes.

`Capabilities.Vehicles.Vehicle-Safir.Dob.InjectionProperty`
Mapping that denotes the kind of injection. The vehicle object is SynchronousVolatile, which means that vehicle objects survives an application but not a Dob restart. No injections of vehicle objects from external systems will take place.

`Capabilities.Vehicles.VehicleMsg`
Definition of message that is sent when the number of created vehicle objects reaches the limit specified in VehicleParameters.

`Capabilities.CalculateSpeedDifference`
Definition of a service that calculates the speed difference between a vehicle object speed and a given speed. A property is used to obtain the speed from the vehicle object.

`Capabilities.CalculateSpeedDifferenceResponse`
Definition of the speed difference service response.

`Capabilities.SpeedObjectProperty`
Definition of the speed property.

`Capabilities.Vehicles.Vehicle-Capabilities.SpeedObjectProperty`
Mapping of the speed property onto the speed member of the vehicle class.

`Capabilities.Vehicles.VehicleParameters`
Definition of vehicle parameters.

A.4.2 Internal Design

Figure A.3 shows the classes of the C++ version of VehicleApp and the most important Dob classes and consumer interfaces that they use.

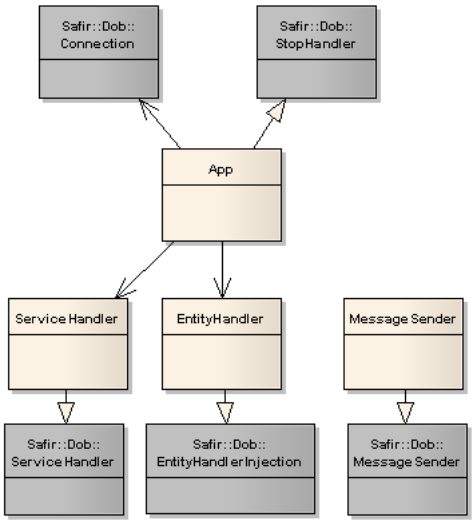

Figure A.3: VehicleApp class diagram

App
Main class. Called by the Dob on application stop. Contains the main Dob connection.

EntityHandler
Registers ownership of the vehicle class and receives all vehicle object requests and injections.

ServerHandler
Registers ownership of the speed difference service and receives all service requests.

MessageSender
Sends message when number of vehicle objects has reached the limit specified through a parameter.

A.5 VehicleMmi - Presentation Application

This application exists in two variants, one in C++ with the Qt widget set and one in C# using WinForms.

VehicleMmi is the presentation application in the system. It subscribes to, and presents all vehicle data in a table in the GUI. It also presens information for a selected vehicle object in a detailed window. In this window, it is possible to enter new data for a vehicle and send a request to change it. It is also possible to create a new vehicle object.

Subscription to the vehicle entities is started through the Dob interface `EntitySubscriber`. As soon as an entity is updated, VehicleMmi will receive a subscription response.

The application also receives the warning message that is sent by VehicleApp. This is done through implementation of the Dob interface `MessageSubscriber`.

To obtain additional database information – that is not received throuh a subscription response – for a selected vehicle, it is possible to request this from the VehicleDb. If the database information does not exist, it is created by VehicleMmi.

It is possible to calculate the difference between a selected vehicle object and an entered speed through the the speed difference calculation service. The calculation itself it vary basic since we want to focus on how to use a Dob service.

A.5.1 Windows and Dialogs

This section is an overview of the windows and dialogs of the VehicleMmi application.

The list view in Figure A.4 contains all published vehicle objects in the system. All created, changed and deleted vehicle objects are received by VehicleMmi as entity subscription responses and presented in the list view. An object may be deleted from the list view, but not modified. It is also possible to delete category information for a category code from the list view.

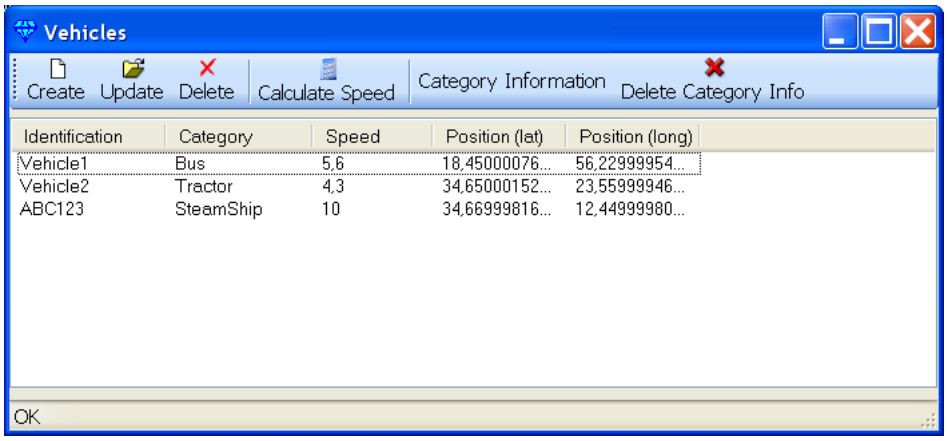

Figure A.4: VehicleMmi list view.

The dialog in Figure A.5 is opened from the list view and is used to send create requests for new vehicle objects. The requests are received by VehicleApp.

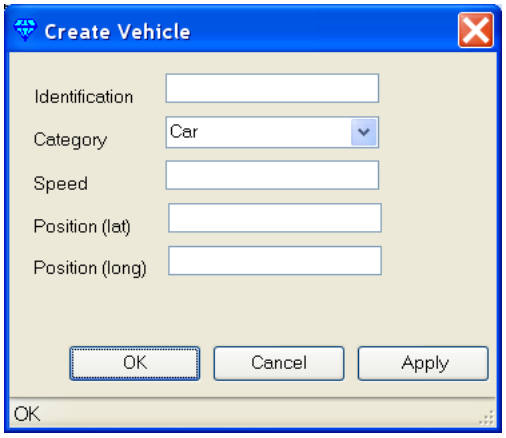

Figure A.5: VehicleMmi Create dialog

The dialog in Figure A.6 is opened from the list view and is used to send update requests for existing vehicle objects. The requests are received by VehicleApp.

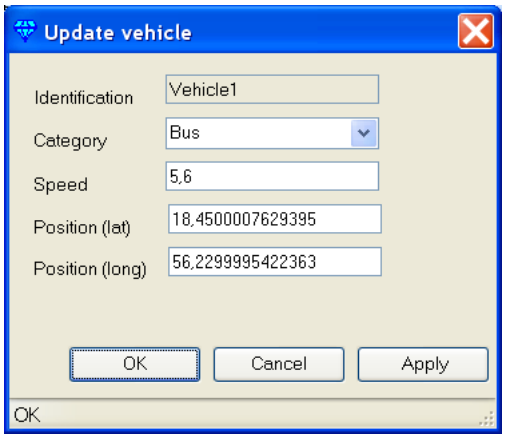

Figure A.6: VehicleMmi Update dialog

The dialog in Figure A.7 is opened from the list view and is used to send to calculate the difference between the speed for a selected vehicle object and an entered speed. The request is sent through a service to VehicleApp and the result is given in the sevice response.

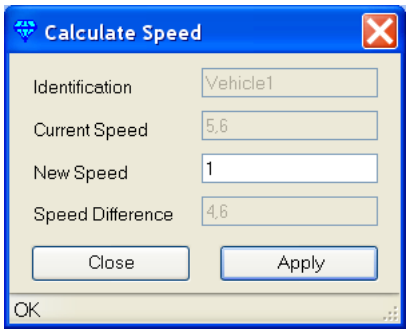

Figure A.7: VehicleMmi Speed calculation dialog

The dialog in Figure A.8 is opened from the list view and is used to obtain category information for the selected vehicle object. If there is no information the category code in the database, the information entered in the dialog is instead stored for the given category code.

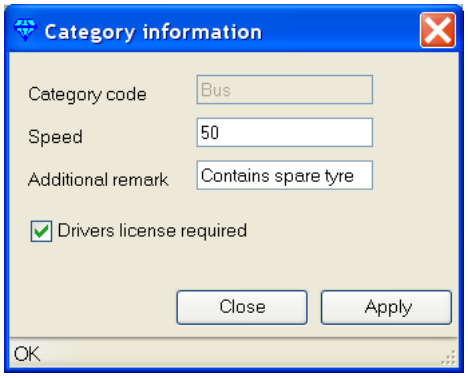

Figure A.8: VehicleMmi Category information dialog

A.5.2 Dou-files

No dou files are provided by VehicleMmi.

A.5.3 Internal Design

Figure A.9 shows the classes of the C# version of VehicleMmi and the most important Dob classes and consumer interfaces that they use.

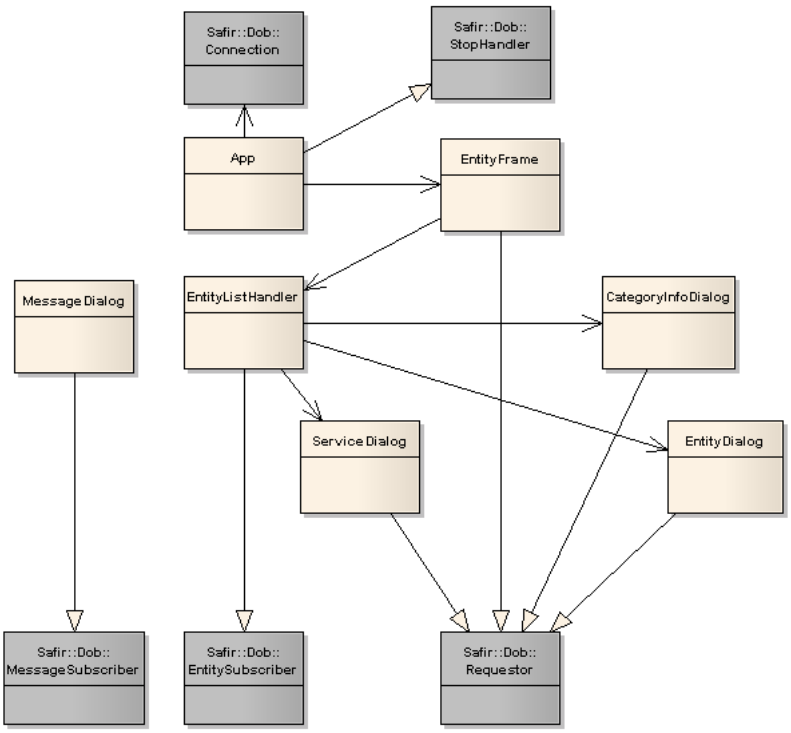

Figure A.9: VehicleMmi class diagram

App
Main class. Called by the Dob on application stop.Contains the main Dob connection.

EntityFrame
Contains the list view and the buttons that open the dialogs and operates on vehicle objects.

EntityListHandler
Subscribes to vehicle objects. Updates the list view according to subscription responses.

EntityDialog
Implements the Create vehicle and Update vehicle dialogs. Sends vehicle object requests that are received by VehicleApp.

ServiceDialog
Implements the Speed difference calculator dialog. Sends a service request that is received by VehicleApp.

MessageDialog
Implements the dialog that is presented when the number of created vehicle objects has reached the limit specified by a parameter. Subscribes to a Dob message.

CategoryInfoDialog
Implements the Category information dialog. Sends service requests to create new category information data code or to obtain existing data for a category information. The requests are received by VehicleDb.

A.6 VehicleDb - Database Application

VehicleDb is the database application in the system. It sets up a connection to an ODBC database and reads and writes data from and to it. These database transactions are triggered by Dob service requests.

The database contains vehicle category information. I.e. for each category code there is additional information that is stored in a database.

All database interaction is made through the Safir ODBC interface (which is deprecated, as of Safir SDK Core 6.0).

A condition for the database application to work properly is that there is an ODBC database setup. A script is provided to set up a Mimer database with the correct tables, columns, stored procedures and user information.

The database connection is setup by a few steps on startup of the application. VehicleDb provides the following services: - Get vehicle category information - Set vehicle category information - Delete vehicle category information

When a service request is received, a corresponding database transaction is performed. The database transactions are performed by calling stored procedures.

When a transaction is performed successfully, a service response is sent. The response depends on the request type.

A.6.1 Dou-files

The following dou files are provided with VehicleDb. For details, see the corresponding dou file.

`Capabilities.Vehicles.DatabaseParameters`
Database connection parameters.

`Capabilities.Vehicles.VehicleCategoryInfo`
Definition of a vehicle category.

`Capabilities.Vehicles.DeleteVehicleCategoryService`
This service is used for deletion of a vehicle category.

`Capabilities.Vehicles.GetVehicleCategoryService`
This service is used to obtain a vehicle category info.

`Capabilities.Vehicles.GetVehicleCategoryResponse`
The GetVehicleCategoryService response.

`Capabilities.Vehicles.SetVehicleCategoryService`
This service is used to create a new vehicle category info.

A.6.2 Internal Design

Figure A.10 shows the VehicleApp classes and the most important interfaces that they use.

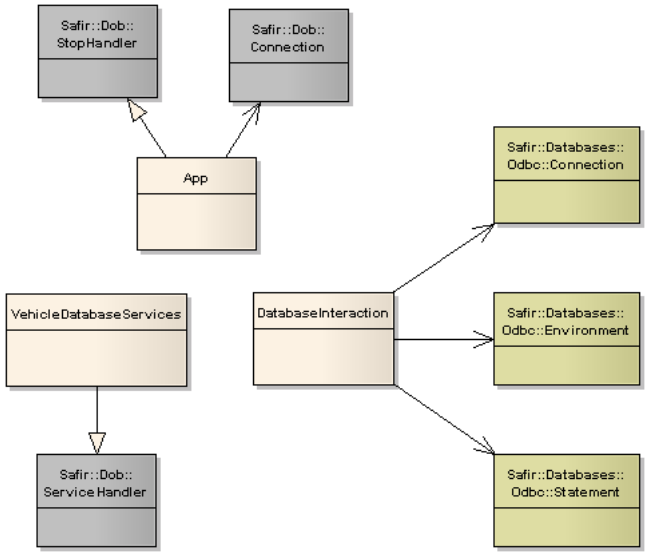

Figure A.10: VehicleDb class diagram

App

Main class. Called by the Dob on application stop. Contains the main Dob connection.

VehicleDatabaseServices

Registers ownership of the Vehicle category information services. Receives all service requests and triggers action in the DatabaseInteraction class.

DatabaseInteraction

Performs all database interaction. Sets up the database connection on startup. Prepares database statements that are executes on request.

Appendix B

Dob WebSocket/JSON-RPC API reference

This section contains an API reference describing the Safir WebSocket JSON-RPC interface.

The API is compliant with the JSON-RPC specification 2.0 (http://www.jsonrpc.org/specification), and the transport mechanism is WebSockets rfc6455. To use the API, connect a WebSocket to the Safir WebSocket server and send the JSON messages defined in this section. Remember that JSON is case sensitive.

B.1 Methods

This section describes all methods exposed by the Dob JSON-RPC interface. Methods can be called by clients and a response with the same id is guaranteed. The id can be either a string or a number. If no id is present the method will still be executed but the caller will not get any response.

Some parameters have a specified default value. Such parameters can be omitted in a method call and will then adopt their default value.

B.1.1 Method: close

Close connection to the Dob.

Returns

- `value` - On success the string *OK* is returned, otherwize an error is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "close", "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.2 Method: createRequest

Send a entity create request to a handler.

Parameters

- `entity` - Entity to create.
- `instanceId` - Instance id of the new entity. Only allowed if the instanceIdPolicy is *RequestorDecidesInstanceId*

- `handlerId` - Id of the handler to send the request to. ***(Default:** DEFAULT_HANDLER)*

Returns

- `isSuccess` - Indicates if the request was successful or not, can be *true* or *false*.
- `response` - The response object that inherits from Safir.Dob.Response.

Example

```
--> {"jsonrpc": "2.0", "method": "createRequest", "params": {"entity": {"_DouType": "Safir. ↩
    Dob.Entity"}, "instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"isSuccess": true, "response": {"_DouType": "Safir.Dob. ↩
    SuccessResponse"}}, "id": 123}
```

B.1.3 Method: deleteAllInstances

Deletes all owned instances.

Parameters

- `typeId` - Type of the entities to delete.
- `handlerId` - Id of the handler. ***(Default:** DEFAULT_HANDLER)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "deleteAllInstances", "params": {"typeId": "Test.MyEntity ↩
    "}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.4 Method: deleteEntity

Deletes an entity instance in the Dob.

Parameters

- `typeId` - Type of the entity to delete.
- `instanceId` - Instance id of the entity to delete.
- `handlerId` - Id of the handler. ***(Default:** DEFAULT_HANDLER)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "deleteEntity", "params": {"typeId": "Test.MyEntity", " ↩
    instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.5 Method: deleteRequest

Send a request to delete an entity instance.

Parameters

- `typeId` - Type of the entity the request is aimed at.
- `instanceId` - Instance id of the entity to delete.

Returns

- `isSuccess` - Indicates if the request was successful or not, can be *true* or *false*.
- `response` - The response object that inherits from Safir.Dob.Response.

Example

```
--> {"jsonrpc": "2.0", "method": "deleteRequest", "params": {"typeId": "Test.MyEntity", " ↩
    instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"isSuccess": true, "response": {"_DouType": "Safir.Dob. ↩
    SuccessResponse"}}, "id": 123}
```

B.1.6 Method: getAllInstanceIds

Get a list of all instance id's that is created of an entity.

Parameters

- `typeId` - Type of the entity.

Returns

- `list` - A list of instance id's.

Example

```
--> {"jsonrpc": "2.0", "method": "getAllInstanceIds", "params": {"typeId": "Safir.Dob. ↩
    Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": [1, 2], "id": 123}
```

B.1.7 Method: getInstanceIdPolicy

Get the instance id policy for a type and handler.

Parameters

- `typeId` - Type of the entity.
- `handlerId` - Id of the handler. ***(Default:** DEFAULT_HANDLER)*

Returns

- `value` - A string that can be either *RequestorDecidesInstanceId* or *HandlerDecidesInstanceId*.

Example

```
--> {"jsonrpc": "2.0", "method": "getInstanceIdPolicy", "params": {"typeId": "Safir.Dob. ↩
    Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "RequestorDecidesInstanceId", "id": 123}
```

B.1.8 Method: getNumberOfInstances

Get the number of existing instances of a type.

Parameters

- `typeId` - Type of the entity.
- `handlerId` - Id of the handler. ***(Default:** ALL_HANDLERS)*
- `includeSubclasses` - Also subscribe for all subclasses to the specified typeId. ***(Default:** true)*

Returns

- `value` - Number of instances.

Example

```
--> {"jsonrpc": "2.0", "method": "getNumberOfInstances", "params": {"typeId": "Safir.Dob. ↩
    Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": 10, "id": 123}
```

B.1.9 Method: getTypeHierarchy

Get the entire DOU type hierarchy. This method is only allowed if *EnableTypesystemCommands* is set to *true* in the WebSocket server configuration.

Returns

- `object` - An instance of Safir.Websocket.Typesystem.TypeHierarchy

Example

```
--> {"jsonrpc": "2.0", "method": "getTypeHierarchy", "id": 123}
<-- {"jsonrpc": "2.0", "result": {"_DouType": "Safir.Websocket.Typesystem.TypeHierarchy", " ↩
    RootClass": {...}}, "id": 123}
```

B.1.10 Method: isCreated

Check if an entity instance exists in the Dob.

Parameters

- `typeId` - Type of the entity.
- `instanceId` - Instance id of the entity.

Returns

- `value` - *true* or *false*.

Example

```
--> {"jsonrpc": "2.0", "method": "isCreated", "params": {"typeId": "Test.MyEntity", " ↩
    instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": true, "id": 123}
```

B.1.11 Method: isOpen

Check if there is an open connection to the Dob.

Returns

- `value` - If connected to the Dob *true* is returned, otherwize *false* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "isOpen", "id": 123}
<-- {"jsonrpc": "2.0", "result": true, "id": 123}
```

B.1.12 Method: open

Opens a connection to the Dob.

Parameters

- `connectionName` - Name of the new connection.
- `context` - Context for connection. ***(Default:** 0)*

Returns

- `value` - On success the string *OK* is returned, otherwize an error is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "open", "params": {"connectionName": "foo"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.13 Method: ping

This method can be used to test a connection. When the Safir WebSocket server receives a ping, it will immediately respond with a "pong". This message can also be used in those rare cases when the client WebSocket implementation doesn't support the ping opcode. That is a low level keep alive message that will prevent connectons with low traffic to be disconnected. In that case manual pings can be sent with a low frequent but periodic timer.

Returns

- `value` - If connected the string "pong" is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "ping", "id": "testId"}
<-- {"jsonrpc": "2.0", "result": "pong", "id": "testId"}
```

B.1.14 Method: readEntity

Read current version of an entity in the Dob.

Parameters

- `typeId` - Type of the entity to read.
- `instanceId` - Instance id of the entity to read.

Returns

- `object` - The read entity.

Example

```
--> {"jsonrpc": "2.0", "method": "readEntity", "params": {"typeId": "Test.MyEntity", " ↩
    instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"_DouType": "Test.MyEntity", "Val1": "foo"}, "id": 123}
```

B.1.15 Method: registerEntityHandler

Register an entity handler.

Parameters

- `typeId` - Entity type to register.
- `handlerId` - Id of this handler. *(**Default:** DEFAULT_HANDLER)*
- `instanceIdPolicy` - Who decides instance ids. A string that can be either *RequestorDecidesInstanceId* or *HandlerDecidesInstanceId*. *(**Default:** RequestorDecidesInstanceId)*
- `injectionHandler` - Does the handler handle injections. *(**Default:** false)*
- `pending` - Is this a pending registration *(**Default:** false)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "registerEntityHandler", "params": {"typeId": "Safir.Dob. ↩
    Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.16 Method: registerServiceHandler

Register a service handler.

Parameters

- `typeId` - Service type to register.
- `handlerId` - Id of this handler. *(**Default:** DEFAULT_HANDLER)*
- `pending` - Is this a pending registration *(**Default:** false)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "registerServiceHandler", "params": {"typeId": "Safir.Dob. ↩
    ServiceRequest"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.17 Method: sendMessage

Send a message.

Parameters

- `message` - Message to send.
- `channelId` - Channel to send on. ***(Default:** DEFAULT_CHANNELS)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "sendMessage", "params": {"message": {"_DouType": "Safir. ↩
    Dob.Message"}}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.18 Method: serviceRequest

Send a service request to a handler.

Parameters

- `request` - Service object.
- `handlerId` - Id of the handler to send the request to. ***(Default:** DEFAULT_HANDLER)*

Returns

- `isSuccess` - Indicates if the request was successful or not, can be *true* or *false*.
- `response` - The response object that inherits from Safir.Dob.Response.

Example

```
--> {"jsonrpc": "2.0", "method": "serviceRequest", "params": {"request": {"_DouType": "Test ↩
    .MyService", "SomeValue": 100}}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"isSuccess": true, "response": {"_DouType": "Safir.Dob. ↩
    SuccessResponse"}}, "id": 123}
```

B.1.19 Method: setEntity

Creates or updates an entity in the Dob. No merge is performed, the entity will look exactly like the one submitted in the entity parameter. Any members not present will be set to null.

Parameters

- `entity` - Entity to set in the Dob.
- `instanceId` - Instance id of the entity to set.
- `handlerId` - Id of the handler. ***(Default:** DEFAULT_HANDLER)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "setEntity", "params": {"entity": {"_DouType": "Test. ↩
    MyEntity", "Val1": "foo", "Val2": "bar"}, "instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.20 Method: setEntityChanges

Merge all members that are present in the entity parameter, with the current version of the entity in the Dob. Only the members that are present in the entity parameter are considered as part of the change. This means that if the intention is to change a specific member to null, that member must explicitly be part of the entity parameter with the value null. Since JSON arrays don't have indices, it limits the possibility to set changes on individual array values. Hence, if an array is to be changed the entire array should be specified in the call.

Parameters

- `entity` - Entity merge into the Dob.
- `instanceId` - Instance id of the entity to change.
- `handlerId` - Id of the handler. ***(Default:** DEFAULT_HANDLER)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "setEntityChanges", "params": {"entity": {"_DouType": " ↩
    Test.MyEntity", "Val1": "foo", "Val2": null}, "instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.21 Method: subscribeEntity

Subscribe for entities.

Parameters

- `typeId` - Entity type to subscribe for.
- `instanceId` - Specific instance. Not allowed if `includeSubclasses` also is present. ***(Default:** ALL_INSTANCES)*
- `includeUpdates` - Get notified on updated entity states. ***(Default:** true)*
- `restartSubscription` - Restart subscription to get onNewEntity for all existing instances. ***(Default:** true)*
- `includeSubclasses` - Include subclasses to specified typeId. Not allowed if `instanceId` is present ***(Default:** true)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "subscribeEntity", "params": {"typeId": "Safir.Dob.Entity ↩
    ", "instanceId": 10}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.22 Method: subscribeMessage

Subscribe for messages.

Parameters

- `typeId` - Name of an existing message type.
- `channelId` - Channel to subscribe on. ***(Default:** ALL_CHANNELS)*
- `includeSubclasses` - Also subscribe for all subclasses to the specified typeId. ***(Default:** true)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "subscribeMessage", "params": {"typeId": "Safir.Dob. ↩
    Message"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.23 Method: subscribeRegistration

Subscribe for handler registrations of entities and services.

Parameters

- `typeId` - Entity or Service type to subscribe for registrations.
- `handlerId` - Id of the handler of interest. ***(Default:** ALL_HANDLERS)*

- `includeSubclasses` - Also subscribe for registrations of subclasses to specified typeId. ***(Default:** true)*
- `restartSubscription` - Restart subscription to get onRegistered for all existing handlers. ***(Default:** true)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "subscribeRegistration", "params": {"typeId": "Safir.Dob. ↩
    Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.24 Method: unregisterHandler

Unregister an entity or service handler.

Parameters

- `typeId` - The type to unregister.
- `handlerId` - Id of the handler to unregister. ***(Default:** ALL_HANDLERS)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "unregisterHandler", "params": {"typeId": "Safir.Dob. ↩
    Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.25 Method: unsubscribeEntity

Subscribe for entities.

Parameters

- `typeId` - Entity type to unsubscribe for.
- `instanceId` - Unsubscribe for specific instance. Not allowed if `includeSubclasses` also is present. ***(Default:** ALL_INSTANCES)*
- `includeSubclasses` - Also unsubscribe for subclasses to specified typeId. Not allowed if `instanceId` is present. ***(Default:** true)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "unsubscribeEntity", "params": {"typeId": "Safir.Dob. ↩
    Entity", "instanceId": 10}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.26 Method: unsubscribeMessage

Unsubscribe for messages.

Parameters

- `typeId` - Name of an existing message type.
- `channelId` - Channel to subscribe on. ***(Default:** ALL_CHANNELS)*
- `includeSubclasses` - Also subscribe for all subclasses to the specified typeId. ***(Default:** true)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "unsubscribeMessage", "params": {"typeId": "Safir.Dob. ↩
    Message"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.27 Method: unsubscribeRegistration

Unsubscribe for handler registrations of entities and services.

Parameters

- `typeId` - Entity or Service type to unsubscribe for registrations.
- `handlerId` - Id of the handler of interest. ***(Default:** ALL_HANDLERS)*
- `includeSubclasses` - Also unsubscribe for registrations of subclasses to specified typeId. ***(Default:** true)*

Returns

- `value` - On success the string *OK* is returned.

Example

```
--> {"jsonrpc": "2.0", "method": "unsubscribeRegistration", "params": {"typeId": "Safir.Dob ↩
    .Entity"}, "id": 123}
<-- {"jsonrpc": "2.0", "result": "OK", "id": 123}
```

B.1.28 Method: updateRequest

Send a entity update request to the registered handler. Only the members that are present in the entity parameter are considered as part of the update request. This means that if the intention is to change a specific member to null, that member must explicitly be part of the entity parameter with the value null. Since JSON arrays don't have indices, it limits the possibility to send change requests on individual array values. Hence, if an array is part of an update request the entire array should be specified in the request.

Parameters

- `entity` - Updated version of the entity.
- `instanceId` - Instance id of the entity to update.

Returns

- `isSuccess` - Indicates if the request was successful or not, can be *true* or *false*.
- `response` - The response object that inherits from Safir.Dob.Response.

Example

```
--> {"jsonrpc": "2.0", "method": "updateRequest", "params": {"entity": {"_DouType": "Test. ↩
    MyEntity", "SomeValue": "changedValue"}, "instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"isSuccess": true, "response": {"_DouType": "Safir.Dob. ↩
    SuccessResponse"}}, "id": 123}
```

B.2 Notifications

This section describes all notifications sent from server to client when something happens that is not an immediate response to a method call.

B.2.1 Notification: onCompletedRegistration

Called when a pending registration has been completed.

Parameters

- `typeId` - Type for which the pending registration has been completed.
- `handlerId` - The handler id.

Example

```
--> {"jsonrpc": "2.0", "method": "onCompletedRegistration", "params": {"typeId": "Test. ↩
    MyService", "handlerId": -6778878277529052275}}
```

B.2.2 Notification: onDeletedEntity

Deleted entity notification. Subscription response to a subscribeEntity. The received entity is what the entity state looked like when it was deleted.

Parameters

- `instanceId` - The instance id of the entity.
- `entity` - The updated entity.

Example

```
--> {"jsonrpc": "2.0", "method": "onDeletedEntity", "params": {"instanceId": 1, "entity":  ↩
    {"_DouType": "Test.MyEntity"}}}
```

B.2.3 Notification: onInitialInjectionsDone

Called when initial injections are done.

Parameters

- `typeId` - Type for which the initial injection is done.
- `handlerId` - The handler id.

Example

```
--> {"jsonrpc": "2.0", "method": "onInitialInjectionsDone", "params": {"typeId": "Test. ↩
    MyEntity", "handlerId": -6778878277529052275}}
```

B.2.4 Notification: onInjectedDeletedEntity

Called when an entity deletetion is injected.

Parameters

- `instanceId` - Instance id of the entity.
- `entity` - The deleted injected entity.

Example

```
--> {"jsonrpc": "2.0", "method": "onInjectedDeletedEntity", "params": {"instanceId": 1, " ↩
    entity": {"_DouType": "Test.MyEntity"}}}
```

B.2.5 Notification: onInjectedNewEntity

Called when a new entity is injected.

Parameters

- `instanceId` - Instance id of the entity.
- `entity` - The injected entity.

Example

```
--> {"jsonrpc": "2.0", "method": "onInjectedNewEntity", "params": {"instanceId": 1, "entity ↩
    ": {"_DouType": "Test.MyEntity"}}}
```

B.2.6 Notification: onInjectedUpdatedEntity

Called when an entity update is injected.

Parameters

- `instanceId` - Instance id of the entity.
- `entity` - The injected entity.

Example

```
--> {"jsonrpc": "2.0", "method": "onInjectedUpdatedEntity", "params": {"instanceId": 1, " ↩
    entity": {"_DouType": "Test.MyEntity"}}}
```

B.2.7 Notification: onMessage

New message notification. Subscription response to a subscribeMessage.

Parameters

- `channelId` - The channel that the message was sent on.
- `message` - Received message.

Example

```
--> {"jsonrpc": "2.0", "method": "onMessage", "params": {"channelId": 3313918482685577033,  ↩
    "message": {"_DouType": "Safir.Dob.Message"}}}
```

B.2.8 Notification: onNewEntity

New entity notification. Subscription response to a subscribeEntity.

Parameters

- `instanceId` - The instance id of the entity.
- `entity` - The new entity.

Example

```
--> {"jsonrpc": "2.0", "method": "onNewEntity", "params": {"instanceId": 1, "entity": {" ↩
    _DouType": "Test.MyEntity"}}}
```

B.2.9 Notification: onNotMessageOverflow

It is meaningful to try to send messages again after an overflow.

Example

```
--> {"jsonrpc": "2.0", "method": "onNotMessageOverflow"}
```

B.2.10 Notification: onNotRequestOverflow

It is meaningful to try to send requests again after an overflow.

Example

```
--> {"jsonrpc": "2.0", "method": "onNotRequestOverflow"}
```

B.2.11 Notification: onRegistered

New registration notification. Subscription response to a subscribeRegistrations.

Parameters

- `typeId` - Type that has been registered.
- `handlerId` - The handler id that has been registered.

Example

```
--> {"jsonrpc": "2.0", "method": "onRegistered", "params": {"typeId": "Test.MyService", " ↩
    handlerId": -6778878277529052275}}
```

B.2.12 Notification: onRevokedRegistration

Called when a handler registration has been revoked.

Parameters

- `typeId` - Type for which the registration has been revoked.
- `handlerId` - The handler id.

Example

```
--> {"jsonrpc": "2.0", "method": "onRevokedRegistration", "params": {"typeId": "Test. ↩
    MyService", "handlerId": -6778878277529052275}}
```

B.2.13 Notification: onUnregistered

Removed registration notification. Subscription response to a subscribeRegistrations.

Parameters

- `typeId` - Type that has been unregistered.
- `handlerId` - The handler id that has been unregistered.

Example

```
--> {"jsonrpc": "2.0", "method": "onUnregistered", "params": {"typeId": "Test.MyService", " ↩
    handlerId": -6778878277529052275}}
```

B.2.14 Notification: onUpdatedEntity

Updated entity notification. Subscription response to a subscribeEntity. The received entity is the complete current state. I.e It does not only contain the changes since last version.

Parameters

- `instanceId` - The instance id of the entity.
- `entity` - The updated entity.

Example

```
--> {"jsonrpc": "2.0", "method": "onUpdatedEntity", "params": {"instanceId": 1, "entity":  ↩
    {"_DouType": "Test.MyEntity"}}}
```

B.3 Receive and respond to requests

This section describes all the requests an entity or service handler can receive. It is also described what a valid response must look like. When a handler (service handler or entity handler) receives a request, that request will always have an *id* that is a *number*. The handler is responsible for sending a valid response as fast as possible with the same id as the request.

B.3.1 Method: onCreateRequest

Entity handler receives a create request.

Parameters

- `handlerId` - The handler that is being requested.
- `instanceId` - Instance id the requestor wants the handler to create. Only present if the instanceIdPolicy is *RequestorDecidesInstanceId*
- `entity` - Entity the requestor wants the handler to create.

Response

- `object` - An instance of Safir.Dob.Response

Example

```
--> {"jsonrpc": "2.0", "method": "onCreateRequest", "params": {"handlerId":  ↩
    -6778878277529052275, "instanceId": 1, "entity": {"_DouType": "Safir.Dob.Entity"}}, "id ↩
    ": 123}
<-- {"jsonrpc": "2.0", "result": {"_DouType": "Safir.Dob.SuccessResponse"}, "id": 123}
```

B.3.2 Method: onDeleteRequest

Entity handler receives a delete request.

Parameters

- `handlerId` - The handler that is being requested.
- `typeId` - Type of the entity the requestor want the handler to delete.
- `instanceId` - Instance id the requestor wants the handler to delete.

Response

- `object` - An instance of Safir.Dob.Response

Example

```
--> {"jsonrpc": "2.0", "method": "onDeleteRequest", "params": {"handlerId":  ↩
    -6778878277529052275, "typeId": "Safir.Dob.Entity", "instanceId": 1}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"_DouType": "Safir.Dob.SuccessResponse"}, "id": 123}
```

B.3.3 Method: onServiceRequest

Service handler receives service request.

Parameters

- `handlerId` - The handler that is being requested.
- `request` - The Safir.Dob.Service object containing the request.

Response

- `object` - An instance of Safir.Dob.Response

Example

```
--> {"jsonrpc": "2.0", "method": "onServiceRequest", "params": {"handlerId":  ↩
    -6778878277529052275, "request": {"_DouType": "Test.MyService"}}, "id": 123}
<-- {"jsonrpc": "2.0", "result": {"_DouType": "Safir.Dob.SuccessResponse"}, "id": 123}
```

B.3.4 Method: onUpdateRequest

Entity handler receives an update request.

Parameters

- `handlerId` - The handler that is being requested.
- `instanceId` - Instance id the requestor wants the handler to update.
- `entity` - Entity containing changes that the requestor wants the handler to merge with the current state.

Response

- `object` - An instance of Safir.Dob.Response

Example

```
--> {"jsonrpc": "2.0", "method": "onUpdateRequest", "params": {"handlerId":  ↩
    -6778878277529052275, "instanceId": 1, "entity": {"_DouType": "Safir.Dob.Entity"}}, "id ↩
    ": 123}
<-- {"jsonrpc": "2.0", "result": {"_DouType": "Safir.Dob.SuccessResponse"}, "id": 123}
```

B.4 Errors

If an error occurs the Safir WebSocket server will send an error to the client. The error is sent in place of a response, and if possible the id of the request will be preserved. However if it is not possible to extract the id, the error will be sent with a null-id. The error messge is specified in the JSON-RPC 2.0 specification.

Besides the predefined error codes, Safir can send the following error codes

Error codes

- `SafirNotOpen (100)` - When fail to open a Dob connection.
- `SafirOverflow (101)` - Sending requests or messges to fast.
- `SafirAccessDenied (102)` - Trying to modify an entity without ownership.
- `SafirGhostExists (103)` - Trying to modify an entity when there is a ghost instance that hasn't been injected.
- `SafirNotFound (104)` - Calling read or getInstanceIdPolicy on non-existing instance.
- `SafirIllegalValue (105)` - A parameter was invalid.
- `SafirSoftwareViolation (106)` - There is a programming error somewhere. For example trying register a global type on a light node.
- `SafirUnexpectedException (108)` - Unexpected exception occurred.
- `SafirLowMemoryException (109)` - Operation could not be completed because Dob shared memory is running low.

Example

```
--> {"jsonrpc": "2.0", "error": {"code": 100, "message": "SafirNotOpen", "data": "Could not ↩
     open connection..."}, "id": 123}
```

Appendix C

FAQ

Why do I get IllegalValueExceptions when sending message/request or setting an entity?

Most likely because you've got a string in your object that is longer than what is specified by the `maxLength` field in your dou-file. For various reasons the string lengths are not checked until the object is "handed" to the Dob. This means that if you put a string which is 100 characters long into a message member that has `maxLength` 10 you will not get the error until you call `Send`.

Remember that you can check the value of maxLength programmatically by calling `<member name>MaxStringLength()` on your class (e.g. `Safir::Dob::ErrorResponse::AdditionalInfoMaxStringLength()` to get the max length of the member AdditionalInfo).

How are the callbacks invoked?

When something occurs that an application may be interested in knowing about the Dob signals an event internally that results in a call to the `OnDoDispatch()` callback. The application now has to *switch threads* to the thread that owns the connection and call `Dispatch()`. The Dispatch call will in turn call all the required callbacks (e.g. OnMessage if a message has been received). Note that a dispatch may be triggered without actually resulting in any callbacks.

Can I trust IsCreated?

IsCreated can really only be trusted by the entity instance owner. All other applications can get true in one statement, and then get an exception (due to timing) on the next, where they to try to read the contents of an instance.

Why does dope_main crash at startup?

Chances are that you have changed you dou-files, and that the persisted data is no longer valid. There is a solution to this, which is outlined in Chapter 10, but of course you can just delete the persistent data by removing the files or clearing the database table contents.

What do I do with the `ResponseSender` when I'm postponing a request?

If you're postponing with the `redispatchCurrent` flag set to true you need to call `Discard` on the ResponseSender, otherwise it will get upset that you're not sending a response and cause your application to report an error and maybe crash.

Why have overflows?

One of the main design focuses of the Dob has been to facilitate creating systems that degrade gracefully, and the overflow mechanism is a result of this. When a Dob-based system comes under heavy load some data will be discarded (due to overflows), which ensures that there is an upper limit to the amount of memory and CPU used. If the Dob had not used the overflow mechanism, but instead had "infinite queues", more and more memory would be consumed, which would lead to more and more CPU load, and eventually to a crash or some other undefined behaviour.

This of course means that your application should not introduce infinite queues of its own, since that would reintroduce the ungraceful degradation problem. There is a little bit more information in Section 12.1.

Why shouldn't I have an infinite queue in my application to avoid overflows?

This is described somewhat in Section 12.1, and the FAQ entry above.

Whatever happened to `deletedByOwner`?

Once upon a time (well, 4.5 and earlier) the deprecated-flag in the `OnDeletedEntity` callback was known as `deletedByOwn`. This was used to signal how an entity was deleted, whether it was due to an unregistration or an explicit delete. Unfortunately this information was useless, since after the introduction of *ghosts* and persistence it was no longer possible for the Dob to guarantee that you got the right value in all circumstances. Unfortunately we did not realize this at the time, so we've had to deprecate this flag now.

The persistence service (*dope*), however, still uses this flag, since it has meaning when using the special *entity injector* subscribe with `wantsLastState` set to true. This will be changed in a future release, but you can safely ignore this bit of information unless you're writing your own persistence service or entity injector.

Have you got any tips for troubleshooting multicast networking?

When using multicast, and especially over routed networks it is important to be sure that bidirectional multicast routing is enabled on the router. To test this we can recommend *iperf* (http://sourceforge.net/projects/iperf) and the gui frontend *jperf* (http://sourceforge.net/projects/jperf). One test strategy could be to run `iperf -u -s -B 224.20.20.20` (listen to UDP packets on a multicast group) on all computers, and then run `iperf -u -c 224.20.20.20` (send UDP multicast packets) in turn on each computer, checking that you get output on all listeners for every run of the sender. If you're not getting output on all listeners for a certain sender you've got some routing problem.

How can I be sure that I'm not using deprecated features?

In C++ you can define a preprocessor symbol `SAFIR_NO_DEPRECATED`. When this symbol is defined any references to deprecated features will fail to compile.

For Java we have tagged deprecated features with the `@Deprecated` attribute, which should cause your compiler to issue warnings.

For C# we have currently *not* used the `Obsolete` attribute, so no warnings are issued on use of deprecated features.

What happened to the `clone()` function in Java and why is `Clone()` deprecated in C#?

In Safir SDK Core 5 and earlier all types generated from dou-files used to have a method for cloning in C# and Java. During the implementation of the new collection types in Safir SDK Core 6.0 - sequence and dictionary - we realized that the Clone methods were actually not needed. Additionally, supporting Clone for the new collection types would be *a lot* of work. So the decision was taken to remove support for Clone.

We asked our customers about this, and they were ok with removing `clone()` in Java, since it wasn't being used, but in C# there is apparently quite a lot of user code that calls Clone(), so in C# we decided to deprecate the API instead. However, the underlying implementation now uses C# reflection, which means that we don't need to write and maintain all the Clone code for the new collection types.

What happened to the `Safir.Dob.PersistenceParameters.TextColumnsAreUtf8` parameter?

This had to do with messy ODBC Unicode support. We have tried to make a solution that should work without parametrization. If you really need the parameter, please let us know!

What happened to Olib and the Safir.Databases namespace?

The Safir ODBC database interface was deprecated and then removed. Please use some other way of interacting with databases.

Appendix D

Building from source

Since Safir SDK Core is released under an Open Source license (as described in the section called "Licences and Copying") you are able to download and build the source code from scratch. Instructions for building is included in the source distribution of Safir SDK Core obtained from http://safirsdkcore.com/.