

Safir SDK Core Users Guide

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
	30 Aug 2010		A

Contents

1	What is Safir SDK Core?	1
1.1	Provided services	1
1.2	Why “Core”?	2
1.3	Prerequisites / Dependencies	2
1.4	Supported Platforms	3
2	What is the Dob?	4
2.1	What does the Dob do?	4
2.1.1	A bit about the Dob architecture	4
2.2	Which problem does the Dob solves?	5
2.3	Quick intro	5
2.3.1	Messages	5
2.3.2	Services	5
2.3.3	Entities	5
2.3.4	Defining the information model	6
2.3.4.1	Defining objects	6
3	The type system	8
3.1	The simple types	8
3.1.1	SI Types	8
3.1.2	Hashed types	9
3.2	Flags on members	9
3.3	Items and structs	9
3.4	Arrays	10
3.5	Parameters	10
3.5.1	Parameter arrays	11
3.5.2	Objects in parameters	11
3.5.3	Environment variables in parameters	12
3.6	Create routines	12
3.7	The syntax - putting it all together	13
3.8	Properties ^{adv}	15

3.8.1	Using the property	16
3.9	Generating code from Dou-files	17
3.9.1	Running <code>dobmake.py</code>	17
3.10	Using the generated types	17
3.10.1	Syntax in the different languages	18
3.10.2	Containers and Container Proxies	21
3.10.3	Smart pointers	22
3.10.3.1	Smart pointers in C++	22
3.10.3.2	Smart pointers in Ada	22
3.10.3.3	Hints for Debugging	23
3.10.4	Type casting	24
3.10.4.1	C++ type casting	24
3.10.4.2	C# type casting	26
3.10.4.3	Java type casting	27
3.10.4.4	Ada type casting	27
3.11	Other details	28
3.11.1	Change flags	28
3.11.2	The reflection interface ^{adv}	29
4	The distribution mechanisms	31
4.1	Consumers and Callbacks	31
4.2	Connections	31
4.2.1	Dispatching	32
4.2.2	Secondary Connections	32
4.3	Using the mechanisms	33
4.3.1	Addressing	33
4.3.2	Proxies	33
4.3.3	Messages	34
4.3.3.1	Channels	34
4.3.3.2	Subscribing to messages	34
4.3.3.3	Sending messages	35
4.3.4	Services	36
4.3.4.1	Handlers	37
4.3.4.2	Registering a service handler	37
4.3.4.3	Response types	38
4.3.4.4	Sending service requests	39
4.3.5	Entities	39
4.3.5.1	Handlers	40
4.3.5.2	InstanceId and EntityId	41

4.3.5.3	Registering an entity handler	41
4.3.5.4	Handling Create Requests	42
4.3.5.5	Handling Update Requests	43
4.3.5.6	Handling Delete Requests	44
4.3.5.7	Owning entity instances	44
4.3.5.8	Subscribing to entities	44
4.3.5.9	Reading an entity	46
4.3.5.10	Iterating over entities	47
4.3.5.11	Sending entity requests	48
4.3.5.12	Some notes on the CreateRequest methods	48
4.3.6	Entity Persistence	49
4.3.6.1	SynchronousVolatile	49
4.3.6.2	SynchronousPermanent	49
4.3.6.3	Waiting for persistence data	50
4.3.6.4	Using persistence	50
4.3.6.5	Understanding persistence ^{adv}	50
4.3.7	Handler registration subscriptions	53
4.4	Aspects	54
4.5	Postponing callbacks	54
4.6	Interpreting change flags	54
4.6.1	Messages	55
4.6.2	Entity subscriptions	55
4.6.3	Requests on entities	55
4.6.4	Requests on services	55
4.7	Pending Registrations	55
4.8	Stop orders	56
5	Configuration	57
5.1	Network config	57
5.1.1	Standalone	57
5.1.2	Multinode	58
5.1.2.1	Routed networks	58
5.1.2.2	Local objects	58
5.1.2.3	Distribution Channels	59
5.1.2.4	Priorities	59
5.1.2.5	Ports used	60
5.2	Persistence config	60
5.3	Request timeouts config	60
5.4	Queue lengths config	61
5.5	Shared Memory config	62

6	Software Error Reporting	63
6.1	Report types	63
6.2	Send interface	63
6.3	Logger application	64
7	Software program information	65
7.1	The backdoor command	65
7.2	Tracer	65
7.2.1	How to use	66
7.2.2	Notes	66
7.2.2.1	Expression expansion	67
7.2.3	The FORCE_LOG environment variable	67
7.2.4	Troubleshooting the tracer	68
8	The persistence service	69
9	Utilities	70
9.1	Sending many requests	70
9.2	Ace Dispatcher	70
9.3	Time	71
10	Tips and Tricks	72
10.1	Overflow handling	72
10.2	Type system freedoms	73
10.3	Handling exceptions	73
10.3.1	Exceptions in callbacks	73
10.4	Handling OnRevokedRegistration	73
10.5	Multithreading	73
10.6	Hot-standby / Redundancy	74
10.7	Entity reference gotchas	75
11	Systems of Systems ^{adv}	76
11.1	Injectable entities	76
11.2	Asynchronous Injections	76
11.2.1	Using asynchronous injections	77
11.2.2	Timestamp merges	78
11.2.3	IncompleteInjectionState	80

12 Contexts ^{adv}	81
12.1 What contexts can be used for	81
12.2 Design principles	81
12.3 ContextShared types	82
12.4 Using the context mechanism	82
12.4.1 Terminology	82
12.4.2 Each operator console has one mode (Type 1)	83
12.4.3 StandAlone system supporting one mode (Type 2)	83
12.4.4 Starting extra PRSes showing a Replay session (Type 3)	83
12.4.5 Several Replay sessions in one console (Type 4)	83
12.5 APP design	83
12.6 PRS design	84
13 Test support and Tools	85
13.1 Sate	85
13.2 Dobexplorer	86
14 C++ ODBC database wrapper	88
14.1 Connecting	88
14.2 Exceptions	89
14.3 Making a query	89
14.4 Transactions	90
15 More help	92
15.1 Doxygen help	92
15.2 Forum	92
16 Glossary	93
A Example applications	94
A.1 Some background	94
A.2 The (example) problem	94
A.3 The solution	95
A.4 VehicleApp - Business Application	96
A.4.1 Dou-files	96
A.4.2 Internal Design	97
A.5 VehicleMmi - Presentation Application	97
A.5.1 Windows and Dialogs	98
A.5.2 Dou-files	100
A.5.3 Internal Design	100
A.6 VehicleDb - Database Application	101
A.6.1 Dou-files	102
A.6.2 Internal Design	102

B	FAQ	104
C	Building from source	105

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.

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 "beginners".

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 database section is written by Jörgen Johansson, the software error reporting section was originally written by Anders Widén, 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 Users Guide written by Jörgen Johansson. The context section, and the bits on how to use the Ada interfaces, are written by Anders Widén.

This document has also been reviewed by a number of people: Henrik Sundberg, Mattias Gålnander and Anders Widén. If I've left someone out, then I apologise and I'll happily add you.

Versions of this document

To obtain the latest revision, or the latest approved and reviewed revision, please visit http://www.safirsdk.com/downloads/users_guide.

Licences and Copying

This document is licensed under the Creative Commons "Attribution - Share Alike" license, a copy of which you should have received along with the "source-code" of this document. The license can also be viewed at <http://creativecommons.org/licenses/by-sa/3.0/legalcode> with a "friendlier" (i.e. non-legal) version available at <http://creativecommons.org/licenses/by-sa/3.0/>.

For more information about the Creative Commons licenses see <http://creativecommons.org/>

Safir SDK Core itself is available under the GPL v3 (GNU General Public License version 3) license from <http://www.safirsdk.com/> or a commercial license from Saab AB.

The GPL license means that you are free to try out or modify the software to your hearts content and create your own applications on top of it. But you are not allowed to distribute the modified software or your applications (which will classify as derivative 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.safirsdk.com>).

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

Chapter 1

What is Safir SDK Core?

This section is meant to contain a high-level overview of what the Safir SDK Core is, what services it provides and what kind of systems it is suitable for, without going too much into the technical details.

But, we're not there yet. Due to time constraints we haven't been able to get that information in here in a form that we're satisfied with. If you want that kind of information please contact Saab AB (contact information at <http://www.safirsdk.com>).

So for the time being this chapter just contains a collection of background information.

1.1 Provided services

Safir SDK Core is mainly aimed at providing a 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*, that 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

Acronym	Full name	Provided Service(s)
Lluf	Low Level Utility Functions	Low level functions that are not provided by ACE or Boost
Dots	Distributed Objects Type system	The type system used for the distributed objects
Dose	Distributed Objects Service	Inter-process and inter-computer distribution
Douf	Dob Utility Functions	Utility functions for applications using the Dob
Swre	Software Reports	Software error reporting and debug trace logging
Olib	ODBC Library	C++ wrapper classes for Odbc database access.
Dope	Dob Persistence	Persistent storage of Dob entities

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 are illustrated in Figure 1.1.

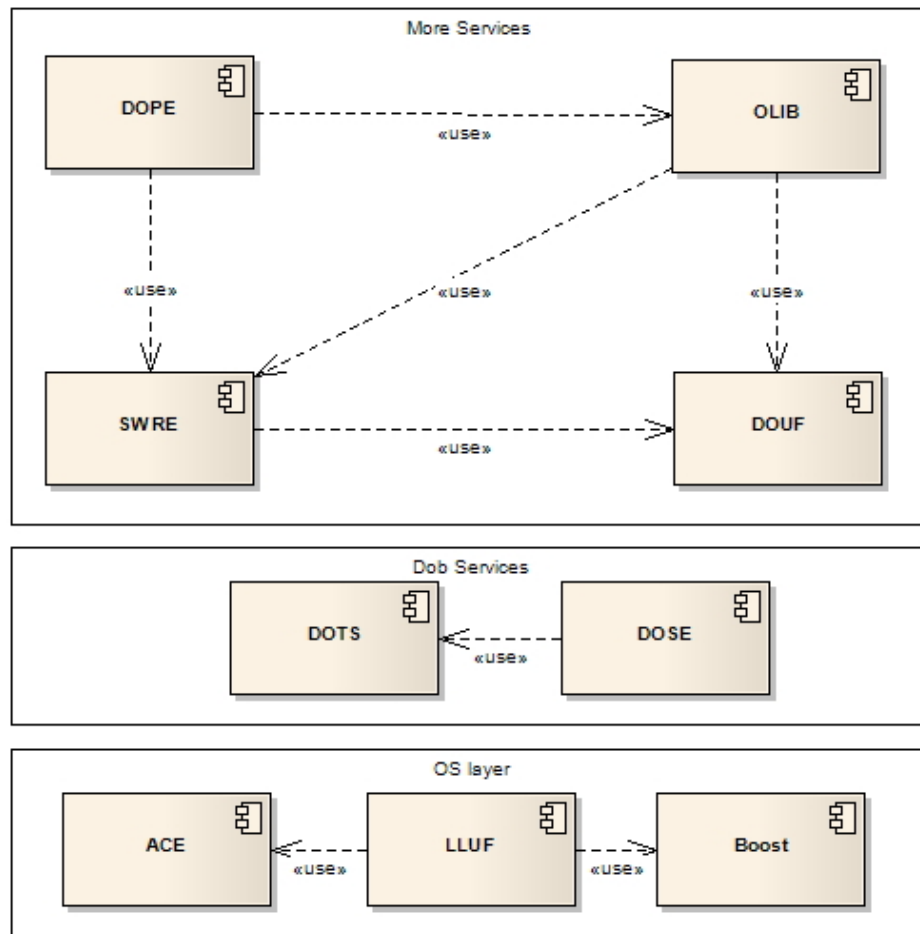


Figure 1.1: Safir SDK Core component dependencies

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 civil and military market.

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.safirsdk.com>).

1.3 Prerequisites / Dependencies

Safir SDK Core uses two open-source products to provide low level functionality and operating system independence wrappers.

ACE, The ADAPTIVE Communication Environment, a freely available, open-source, C++ network programming toolkit. See <http://www.cs.wustl.edu/~schmidt/ACE.html> for more information. At Saab we are using 5.6.3 and later, but Safir SDK Core is quite likely to work with earlier versions.

Boost, a collection of portable high-quality C++ libraries. See <http://www.boost.org/> for more information. Safir SDK Core requires boost version 1.36 or later (it is possible to use 1.35 by patching a few bugs, or 1.34 by installing the Boost.Interprocess library separately).

1.4 Supported Platforms

Safir SDK Core supports both the Windows and Linux platforms on the x86 platform. It also supports Linux on 64 bit x86_64 platforms, but this has not been extensively tested. It should work well on 64 bit Windows platforms as well.

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 ACE and Boost and Safir SDK, be portable to all the supported platforms with no or very little effort.

The compilers that we know can be used for the C++ code are GNU gcc (version 4.0 or later), Microsoft Visual Studio (2005 SP1 or later). For C# code we use Visual Studio (2005 SP1 or later) or Mono (version 1.9.1 or later). Any Java 5 compiler should be able to compile the Java code (we've used Sun's), and for Ada we use Adacore's GNAT Pro (6.1.0w) and GNAT GPL (2008 and 2009).

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#, Ada, 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.
- Redundancy and hot-standby.
- Language independent inter-process/inter-computer communication.
- Allows modular information model.

2.1.1 A bit about the Dob architecture

The Dob 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 distribution 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, "dose_main", that must be running. This executable is responsible for managing 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).
- Has a 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 5.3), 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.

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. 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.

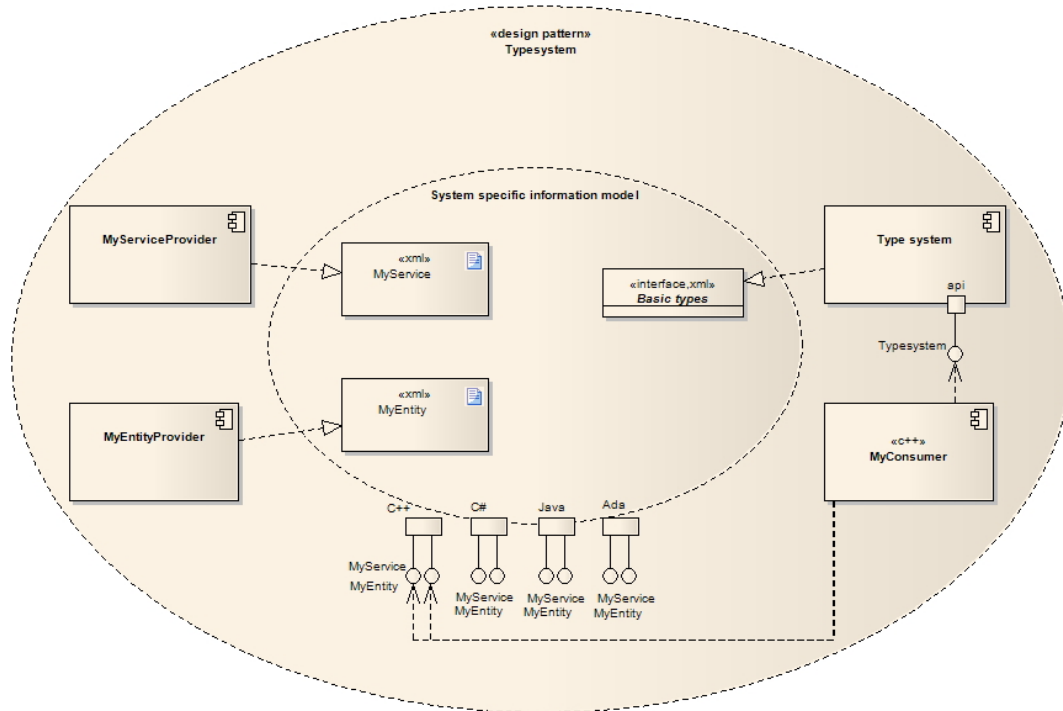


Figure 2.1: Defining the information model

The figure Figure 2.1 attempts to show how applications contribute to, or partake of, the information model.

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 arrays 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 as arrays (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.

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.

3.2 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, classes can also contain other classes. These "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 struct members. 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 Arrays

Members can also be declared as arrays of simple types or complex types (e.g. entities, items or arrays).

There are two types of arrays - static and dynamic arrays.

Static arrays must be declared with an array size ([Static array member declaration](#)). The generated code for these arrays will create a static array where all elements in the array exists at start, compare with C++ `myArray[myArraySize]`. Array length can be changed without regenerating the code. The iterator will iterate over all elements in the array, i.e. from 0 to `arraySize`.

Dynamic arrays are declared without an array size ([Dynamic array member declaration](#)). The generated code for these arrays will create a dynamic array where the array is empty at start. It works as a sparse array, i.e. you can directly access any index in the array and it will always return a valid array element. If no value is set for the given index the array will return a null element of the type specified for the array. The iterator will iterate over all existing elements in the array, e.g. if the array has elements at index 1, 2 and 598, the iterator iterates over these 3 elements.

Static and dynamic arrays generate a slightly different interface, therefore they are treated as different types. If you switch the definition between static and dynamic array you must regenerate the definitions and rebuild your code.

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 (under `$(SAFIR_RUNTIME)/data/text/dots/classes/`) 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).

A parameter definition

```
<?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>MyParameter</name>
      <type>Int32</type>
      <value>25</value>
    </parameter>
  </parameters>
</class>
```

This defines a parameter `MyParameter` 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.

3.5.1 Parameter arrays

A parameter can also be an array of values, as shown below, where an array of string parameters are defined with two values.

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>
      <arrayElements>
        <arrayElement>
          <value>Safir (R)</value>
        </arrayElement>
        <arrayElement>
          <value>(R) rifaS</value>
        </arrayElement>
      </arrayElements>
    </parameter>
  </parameters>
</class>
```

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

3.5.2 Objects in parameters

Parameters can also contain whole Dou-defined objects, not just the basic types. Below is an example from `Safir.Dob.NodeParameters`, 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>
      <arrayElements>
        <arrayElement>
          <object>
            <name>Safir.Dob.NodeDefinition</name>
            <members> <!-- Node 0 -->
              <member>
                <name>NodeName</name>
                <value>My Server</value>
              </member>
            </members>
          </object>
        </arrayElement>
      </arrayElements>
    </parameter>
  </parameters>
```

```

        </object>
      </arrayElement>
    </arrayElements>
  </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` at the top and the `name` in the array item).

3.5.3 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>Safir SDK Core runtime is located
    in $(SAFIR_RUNTIME), and the sdk in $(SAFIR_SDK)</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.

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>Safir.Geodesy.Position.DummyAltitude</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.

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.

Static array member declaration

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

Dynamic array member declaration

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

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

String member declaration

```
<member>
  <name>Callsign</name>
  <type>String</type>
  <maxLength>10</maxLength>
</member>
```

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

Note that it is not possible to change array size from a numerical value to dynamic since it would be like changing the member type.

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. (For C++/Java they will be made into *doxygen/javadoc* comments.)

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>Capabilities.Vehicles.Vehicle</class>
  <memberMapping>
    <member>
      <propertyMember>Name</propertyMember>
    </member>
  </memberMapping>
</propertyMapping>

```

```

        <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. "Capabilities.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 will be generated from the dou-files and then this source code is built into loadable libraries suitable for your specific platform (e.g. dlls, shared libraries, or assemblies). All these steps are performed by running a single script, `dobmake.py`.

The code generation tools expect the dou-files to be located under `$(SAFIR_SDK)/dots/dots_generated`, so this is where you put your own defined dou-files. It is possible to place the files in sub-directories under this directory which can be a good idea for projects with a large number of dou-files. For instance, all dou-files containing parameter values can be placed under a separate sub-directory.

The generated source code gets put in subdirectories under `$(SAFIR_SDK)/dots/dots_generated` (e.g. `cpp/` `dotnet/` etc). The C++ and Ada header files are also copied to the `$(SAFIR_SDK)/include` and `$(SAFIR_SDK)/ada` directories so that they are available to include into your code.

The generated loadable libraries (e.g. `dots_generated-cpp.dll`) are copied to the `$(SAFIR_RUNTIME)/bin` or `lib` directory.

As part of the code generation procedure the original dou-files are copied from `$(SAFIR_SDK)/dots/dots_generated` to `$(SAFIR_RUNTIME)` (the sub-directory structure is preserved). This is the directory used by the Dob at runtime to parse dou-files, including parameters.

Note that any changes to parameter files under `$(SAFIR_RUNTIME)/data/text/dots/classes` will be overwritten when `dobmake.py` is executed. If you want the parameter change to "survive" make the change under `$(SAFIR_SDK)/dots/dots_generated`.

3.9.1 Running `dobmake.py`

To execute all code generations steps just run `dobmake.py` (located under `$(SAFIR_RUNTIME)/bin`) and you're done!

The script requires that Python (at least version 2.4, from <http://www.python.org>) and CMake (at least version 2.6 from <http://www.cmake.org>) are installed.

A GUI will open, and you will be able to choose between some different actions and to disable build of some languages. To get the fastest possible build time, select *Rebuild* (radio button) if you are building for the first time or if you have changed a lot of dou-files, otherwise select *Build*.

If you experience strangeness, it may be that the underlying build system has gotten confused. A rebuild might solve the problem in this case.

If you want to run `dobmake.py` from a script, use the "-b" flag (for batch mode) to run without using the GUI. Use the `--help` flag to find out what other command line options there are.

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 3.1.

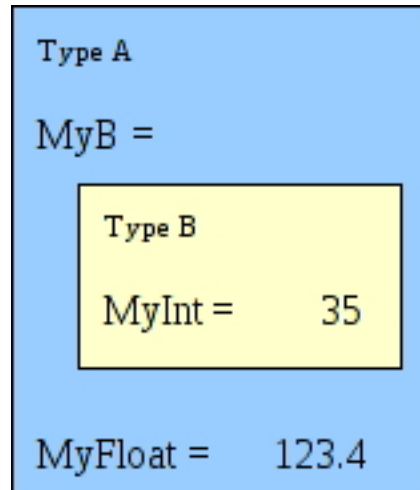


Figure 3.1: Example object

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#](#), [Operating on top-level members in Java](#) and [Operating on top-level members in Ada](#). 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 and the `Ref` calls in the Ada 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();
```

Operating on top-level members in Ada

```
declare
    My_A_Ptr : A.Test_Item_Class_Access := My_A.Ref;
begin

    -- Get My_Float out of My_A
    Val := My_A_Ptr.My_Float.Get_Val;

    -- Set My_Float in My_A to 3.14
    My_A_Ptr.My_Float.Set_Val(3.14);

    -- Check if My_Float is null
    if My_A_Ptr.My_Float.Is_Null then
        ...
    end if;

    -- Check if My_Float is changed
    if My_A_Ptr.My_Float.Is_Changed then
        ...
    end if;

    -- Set My_Float to null
    My_A_Ptr.My_Float.Set_Null;
```

```
end;
```

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#, Ada 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#](#), [Operating on nested members in Java](#) and [Operating on nested members in Ada](#), 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();
```

Operating on nested members in Ada

```
declare
  My_A_Ptr : A.Test_Item_Class_Access := My_A.Ref;
begin

  -- Get My_Int
  Val := My_A_Ptr.My_B.Ref.My_Int.Get_Val;

  -- Set My_Int to 3 (if My_B is not null)
  My_A_Ptr.My_B.Ref.My_Int.Set_Val(3);

  -- Create an empty B item for My_B and set B.My_Int to 3
  My_A_Ptr.My_B.Set_Ptr(B.Create);
  My_A_Ptr.My_B.Ref.My_Int.Set_Val(3);

  -- Check if My_Int is null
  if My_A_Ptr.My_B.Ref.My_Int.Is_Null then
    ...
  end if;

  -- Check if My_Int is changed
  if My_A_Ptr.My_B.Ref.My_Int.Is_Changed then
    ...
  end if;

  -- Set My_Int to null
  My_A_Ptr.My_B.Ref.My_Int.Set_Null;
end;
```

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).

Ada uses container proxies to avoid exposing pointers into the internals of objects, but since Ada does not have the same support for operator overloading as C++ it is not possible to use them to reduce code clutter.

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++ and Ada, that are not garbage collected languages, 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 `boost::shared_ptr` implementation of smart pointers. You can find more information on how to use them here: http://www.boost.org/libs/smart_ptr/smart_ptr.htm.

Since C++ is not garbage collected we do not want to use raw pointers. `shared_ptr` is a smart pointer that automatically deletes the object pointed to when there are no references to it.

Here is part of the introduction from that page:

Smart pointers are objects which store pointers to dynamically allocated (heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners.

Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed.

The `boost::shared_ptr` type is going to be included in the next C++ standard (TR1).

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

3.10.3.2 Smart pointers in Ada

Ada doesn't have a ready-to-use smart pointer abstraction (as C++), instead, each type defines a corresponding smart pointer type in the generated code.

An Ada smart pointer exposes an operation called `Ref` which could be thought of as corresponding to the "arrow" operator (`->`) in C++.

Ada smart pointers work like you expect (Well, almost. See below), they are reference counted and the pointed-to object will be deleted when there are no more references to it.

The user of an Ada smart pointer must be aware of some important differences when compared to a "normal" smart pointer. (The differences are due to Ada language rules). When accessing members via an Ada smart pointer using prefix notation you must start with a local variable holding the underlying "raw" pointer (Ada access type). Assuming that we have a smart pointer `My_A`:

Smart pointer usage in Ada

```
-- Do like this
declare
  My_A_Ptr : A.Test_Item_Class_Access := My_A.Ref;
begin
  My_A_Ptr.My_B.Ref.My_Int.Set_Val (3);
end;

-- This doesn't work. Ambiguous.
My_A.Ref.My_B.Ref.My_Int.Set_Val (3);
```

`My_A_Ptr` is a "raw" pointer and therefore not reference counted. This means that you should only use this variable as "starting-point". Do **not** store it in some data structure or use it as a subprogram parameter. In these situations the smart pointer should be used.

Note that a "raw" pointer is needed only at top-level, nested levels can use the normal `Smart_Ptr.Ref` syntax.

Code style recommendation: Access members using the prefix notation since the "normal" Ada call style gives very hard-to-read code.

Ada compiler bug

For Adacore's GNAT Pro (6.1.0w) and GNAT GPL (2008 and 2009) we have found a compiler bug. In some cases a member reference not starting with a "raw" pointer actually compiles but the method for the base type is called instead of the overridden one. The problem is particularly nasty since the code will both compile and run without any error indication.

So the bottom line is: *Always use a local variable holding the underlying "raw" pointer.*

See Section [3.10.4.4](#) for more information on how to cast smart pointers in Ada.

3.10.3.3 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.

Value containers and object containers in Ada are identical to the corresponding C++ containers (the value names are slightly different). Ada smart pointers also have two pointers in them, the reference count (`Counter_Ptr`) and the pointer to the object (`Data_Ptr`). So, in the Ada case, it's the `Data_Ptr` that is of interest.

For example, if you want to see the contents of a received smart pointer named `Entity` (that really points to a derived entity) using the GPS (GNAT Programming Studio) debugger, just type "print Entity.Data_Ptr.all" in the debugger console window. (It seems that the debugger Data Window can't display `Dob` objects correctly, so the recommendation is to use the debugger console window.)

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 3.2 is used.

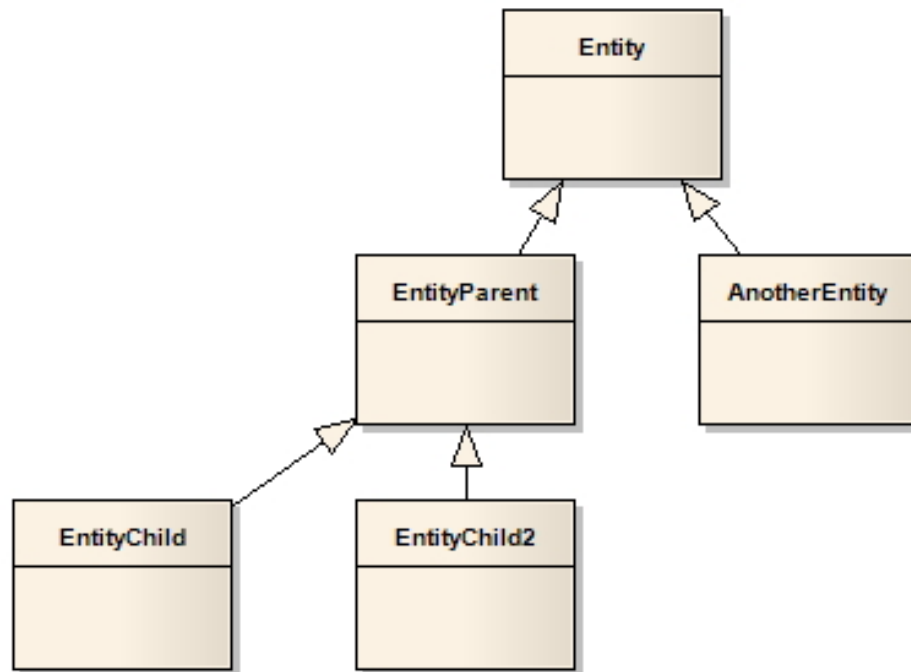


Figure 3.2: Example object hierarchy

3.10.4.1 C++ type casting

The Dob C++ interface uses `boost::shared_ptr` (as described above) for passing around objects over the interfaces. For example this means that you get a `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_ptr`s you have to use `boost::static_pointer_cast` or `boost::dynamic_pointer_cast` to accomplish the same thing.

`boost::static_pointer_cast`

Use `boost::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 `boost::static_pointer_cast`

```

void MyEntitySubscriber::OnUpdatedEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    EntityParentPtr parent =
        boost::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.

boost::dynamic_pointer_cast

Use `boost::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 boost::dynamic_pointer_cast

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

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

    AnotherEntityPtr another =
        boost::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 `EntityParent` *only* but want to treat `EntityChild2` differently:

Another boost::dynamic_pointer_cast example

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

    EntityChild2Ptr child2 =
        boost::dynamic_pointer_cast<EntityChild2>(entity);
    if (child2 != NULL)
    {
        HandleChild2(child2);
        return;
    }
    HandleParent(boost::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(boost::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 `boost::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.

3.10.4.4 Ada type casting

Ada is not garbage collected so smart pointers (as described above) are used for passing around objects over the interfaces. For example, this means that you get a `Safir.Dob.Entity.Smart_Pointer'Class` out of the `Entity_Proxy` in `On_New_Entity`. If you're subscribing to `Entity_Parent` you would want to somehow cast this `Safir.Dob.Entity.Smart_Pointer'Class` to an `Safir.Dob.Entity_Parent.Smart_Pointer`.

If you know the type of the object (by only having subscribed to one type), this is the way to do it in Ada:

Type casting in Ada when handling a single type

```
overriding
procedure On_Updated_Entity
(Self          : in out Consumer;
 Entity_Proxy : in Safir.Dob.Entity_Proxies.Entity_Proxy) is

    Another_Ent : Another_Entity.Smart_Pointer :=
        Another_Entity.Smart_Pointer (Entity_Proxy.Get_Entity);
begin
    ... do something with Another_Ent ...
end;
```

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

When the pointer can be of several different types that you want to treat differently, for example if you have an `Entity_Subscriber` that has subscribed to `Entity_Child` and `Another_Entity`, you can do like this:

Type casting in Ada when handling multiple types

```
overriding
procedure On_Updated_Entity
(Self          : in out Consumer;
 Entity_Proxy : in Safir.Dob.Entity_Proxies.Entity_Proxy) is
```

```

    Entity : Safir.Dob.Entity.Smart_Pointer'Class := Entity_Proxy.Get_Entity;
begin

    if Entity in Entity_Child.Smart_Pointer then
        declare
            Child : Entity_Child.Smart_Pointer :=
                Entity_Child.Smart_Pointer (Entity);
        begin
            Handle_Child (Child);
            return;
        end;
    end if;

    if Entity in Another_Entity.Smart_Pointer then
        declare
            Another : Another_Entity.Smart_Pointer :=
                Another_Entity.Smart_Pointer (Entity);
        begin
            Handle_Another (Another);
            return;
        end;
    end if;
end;

```

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 3.3: Type definition for MyObject type

Member Name	Type
A	Int64
B	String
C	MyItem

Table 3.4: Type definition for MyItem type.

Member Name	Type
X	Int32
Y	Float64

We have two types, MyObject and MyItem (Table 3.3 and Table 3.4), where MyObject contains among other things an item of type MyItem.

If we have an object with change flags set like in Table 3.5 (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 3.6.

Table 3.5: An object with change flags

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

Table 3.6: Result of IsChanged() and IsChangedHere()

Member	IsChanged	IsChangedHere
A	true	N/A
B	false	N/A
C	true	false
C::X	true	N/A
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 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 specify 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 12 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.

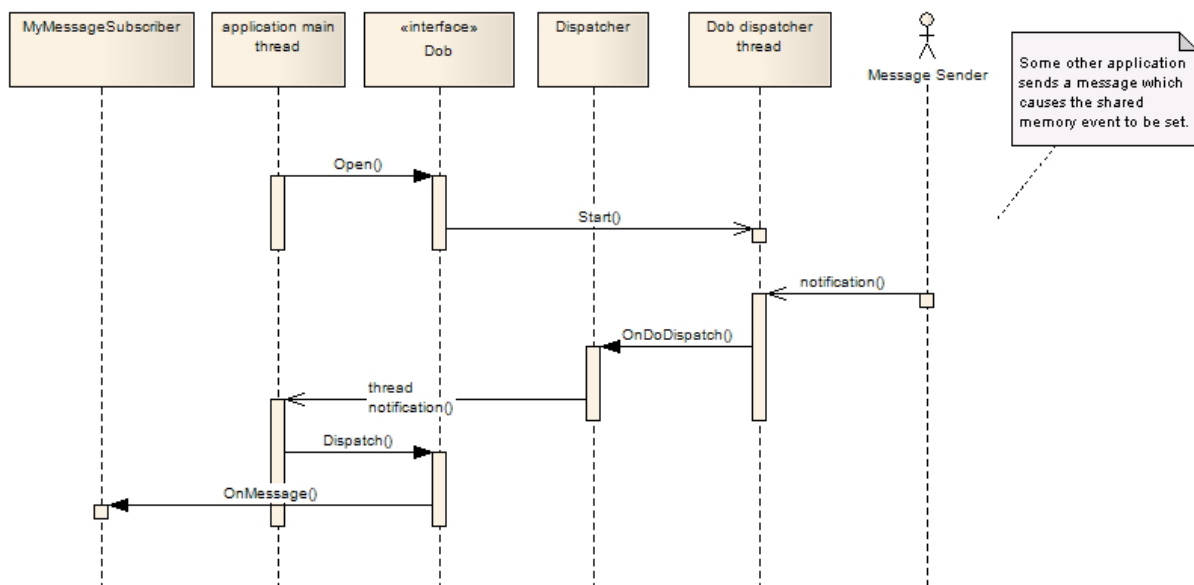


Figure 4.1: Dispatching sequence diagram

If you’re using C++ and ACE the SDK provides a `Dispatcher` class that does the thread switch and calls `Dispatch`, see Section 9.2.

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 10.5). 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 different modules of an application can use the same connection without passing around the primary connection object, something that is not even possible if the parts are written in different languages

(like a C# plugin to a C++ application). Instead the modules 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.

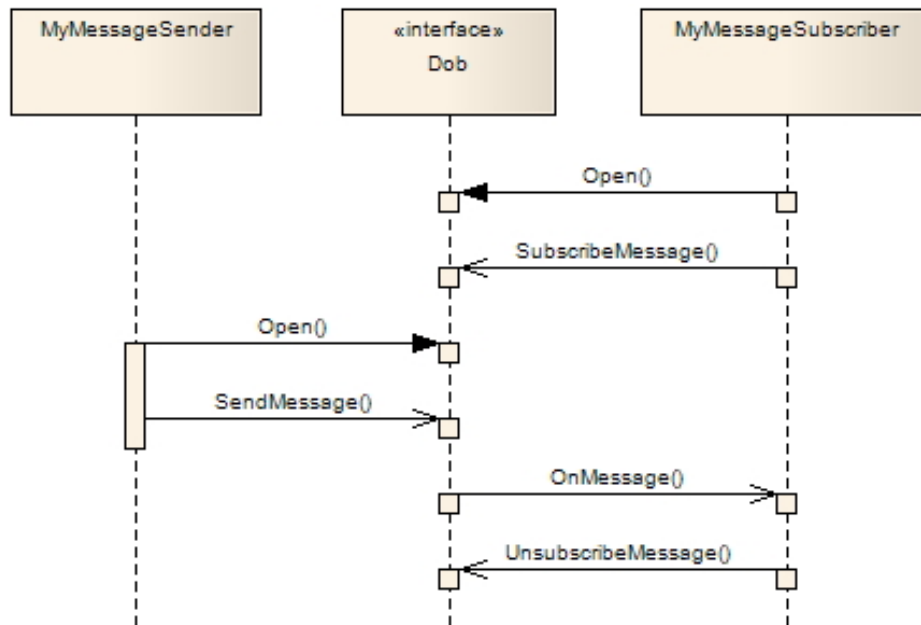


Figure 4.2: Sequence diagram for messages

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:
    virtual void OnMessage(const Safir::Dob::MessageProxy messageProxy);
};
  
```

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 =
        boost::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`s 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:
    // Overrides MessageSender
    virtual void OnNotMessageOverflow();
};
```

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.

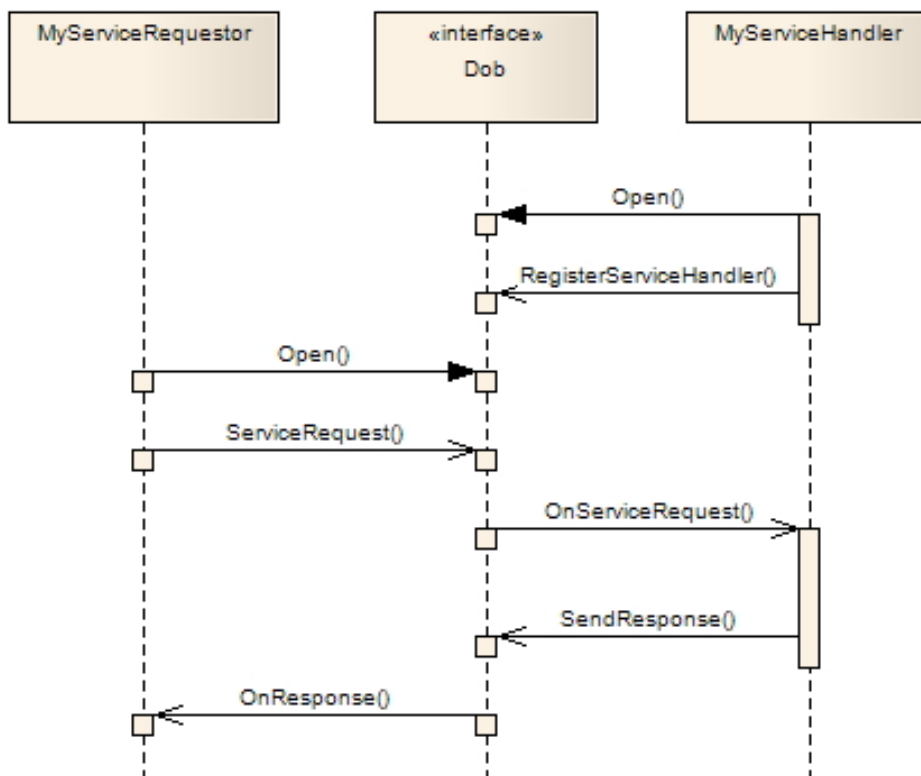


Figure 4.3: Sequence diagram for services

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:

    //Overrides ServiceHandler
    virtual void OnServiceRequest
        (const Safir::Dob::ServiceRequestProxy serviceRequestProxy,
         Safir::Dob::ResponseSenderPtr responseSender);

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

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 =
        boost::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:
    // Overrides Requestor
    virtual void OnResponse(const Safir::Dob::ResponseProxy responseProxy);

    virtual void OnNotRequestOverflow();
};
```

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.

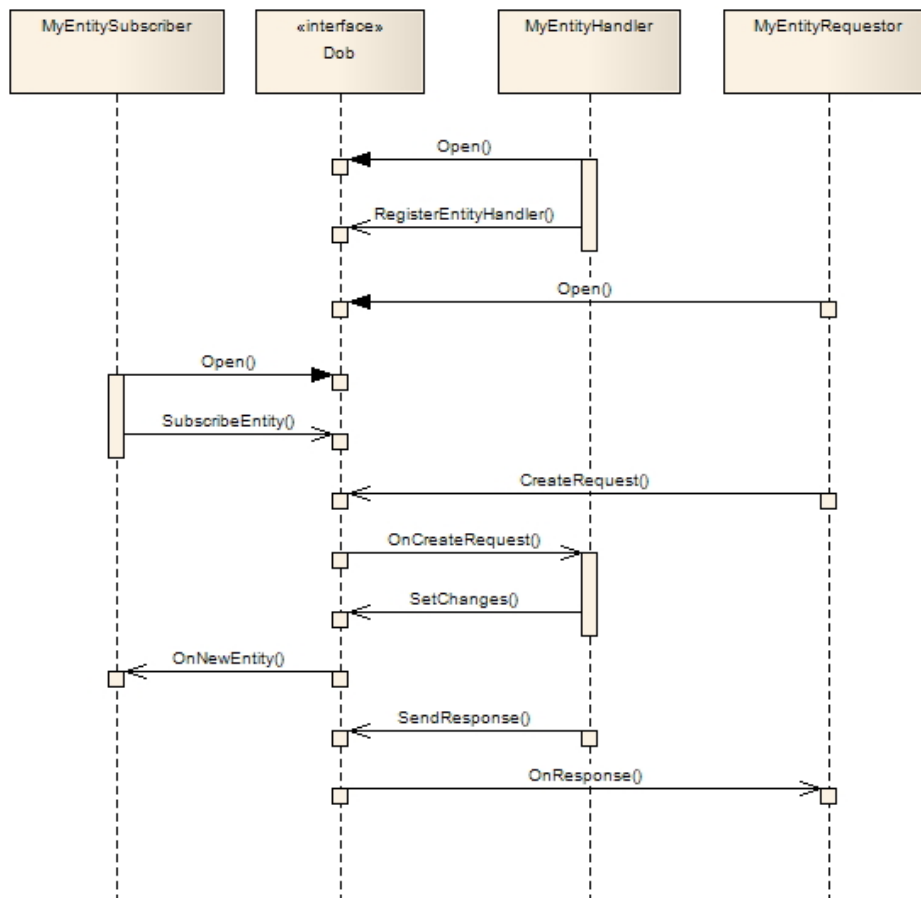


Figure 4.4: Sequence diagram for entities

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!*

It is also not legal for the handler to send an `EntityIdResponse` in this case (exception will be raised).

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:
    // Overrides EntityHandler
    virtual void OnCreateRequest
        (const Safir::Dob::EntityRequestProxy entityRequestProxy,
         Safir::Dob::ResponseSenderPtr responseSender);

    virtual void OnUpdateRequest
```

```

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

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

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

```

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 =
        boost::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 =
        boost::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 [11.2.2](#)

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
{
    // Override EntitySubscriber
    virtual void OnNewEntity
        (const Safir::Dob::EntityProxy entityProxy);

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

    virtual void OnDeletedEntity

```

```
(const Safir::Dob::EntityProxy entityProxy,
    const bool                deletedByOwner);
};
```

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 =
        boost::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 `boost::static_pointer_cast` is.

Handling the `OnUpdatedEntity` callback

```
void VehicleSubscriber::OnUpdatedEntity
    (const Safir::Dob::EntityProxy entityProxy)
{
    Capabilities::Vehicles::VehiclePtr vehicle =
        boost::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 deletedByOwner)
{
    ... 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.

The `deletedByOwner` flag will be set to true if the owner deleted the entity with a call to `Delete`. It will be false if the owning handler was merely unregistered. This is to allow applications to work out whether the entity was removed due to it being explicitly deleted or because the application that owned the entity was stopped for some reason.

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 patterns 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 `boost::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
             (Capabilities.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.

Iterating over all Vehicles (including subclasses) in Ada:

Iterating over entities in Ada

```
declare
  use Safir.Dob.Entity_Iterators;
  It : Safir.Dob.Entity_Iterators.Entity_Iterator :=
    Self.Connection.Get_Entity_Iterator
      (Type_Id          => Capabilities.Vehicles.Vehicle.Class_Type_Id,
       Include_Subclasses => True);
  Entity_Proxy : Safir.Dob.Entity_Proxies.Entity_Proxy;
begin
  while not Safir.Dob.Entity_Iterators.Done (It) loop
    Entity_Proxy := Safir.Dob.Entity_Iterators.Get_Entity_Proxy (It);
    ... do something with Entity_Proxy ...
    Safir.Dob.Entity_Iterators.Next (It);
  end loop;
end;
```

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 `RequestorDecidesInstanceId`, but contrary to what might be expected the first one can be used for **both** types of handlers. If the handler is `RequestorDecidesInstanceId` 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*. A special case of this is *persistence*, i.e. making entities survive an application or system restart. Persistence injections are known as *synchronous* injections, since they will always occur synchronously, at application startup (or rather, when registration occurs), whereas the other kind of injection can occur asynchronously throughout the lifetime of a system.

Note that it is possible (indeed, it is easy) to use the persistence injections without having to use the other kind of injections.

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 11).

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::EntityHandlerInjection`. 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>Capabilities.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. `Capabilities.Vehicles.Vehicle-Safir.Dob.InjectionProperty.dom`, 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 5.2), 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.EntityHandlerInjection`. This consumer has four additional callbacks, `OnInjectedNewEntity`, `OnInjectedUpdatedEntity`, `OnInjectedDeletedEntity` 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++ and Ada 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++ and Ada 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:
```

```

virtual void OnCreateRequest
    (const Safir::Dob::EntityRequestProxy entityRequestProxy,
     Safir::Dob::ResponseSenderPtr      responseSender);

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

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

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

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

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

```

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 `RegisterEntityHandler` 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 =
        boost::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).

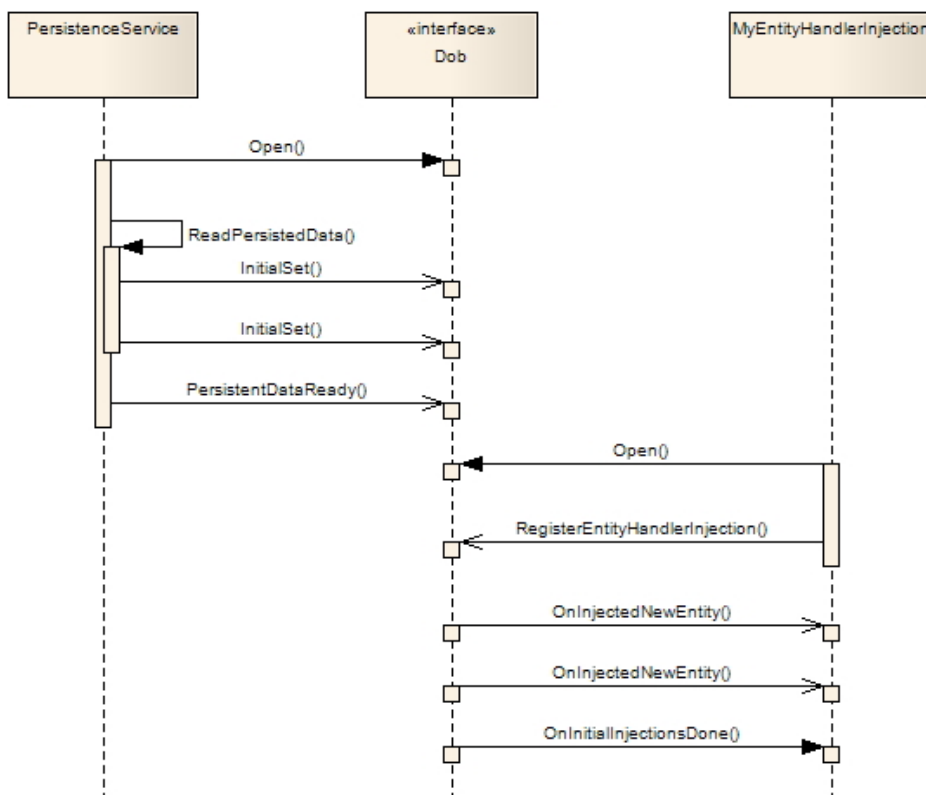


Figure 4.5: Sequence diagram for persistent entity

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
{
    virtual void OnRegistered
        (const Safir::Dob::Typesystem::TypeId      typeId,
         const Safir::Dob::Typesystem::HandlerId& handlerId);

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

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 `OnInitialInjectionsDone` 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 `OnNotMessageOverflow` 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 10.6 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 13.1) 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

Configuration

There are many things in Safir SDK Core that can be configured. This chapter aims to describe the most important ones, but the adventurous can always delve into the parameters in the dou-files of Safir SDK Core.

5.1 Network config

A Safir SDK Core system can have two basic configurations, with respect to networking. *Standalone* or *Multinode*. Standalone means that the system consists of only one node or computer. This is typically used for developer machines or systems that have only one node. Multinode is typically used when a system consists of several nodes, for example a couple of servers (where applications and databases run) and a few operator consoles (where only the HSI runs).

5.1.1 Standalone

To configure a system to be Standalone (which is the default configuration anyway) you need only change one parameter in `Safir.Dob.DistributionChannelParameters.dou` (and you can do that without recompilation, as described in Section 3.5). Set the `MulticastAddress` of the first `DistributionChannel` in the array (the one named *System*) to be "127.0.0.1" (i.e. localhost), and your system is now standalone, and *no data is sent over the network by the Dob*.

Standalone configuration

```
<arrayElements>
  <arrayElement>
    <object>
      <name>Safir.Dob.DistributionChannel</name>
      <members>
        <member>
          <name>Name</name>
          <value>System</value>
        </member>
        ... IncludedNodes parameter that doesn't fit here ...
        <member>
          <name>MulticastAddress</name>
          <value>127.0.0.1</value>
        </member>
      </members>
    </object>
  </arrayElement>
</arrayElements>
```

Once the `MulticastAddress` is set to 127.0.0.1 all the rest of the network configuration files are ignored.

You should also set up the parameters `Safir.Dob.NodeParameters.NumberOfNodes` and `Safir.Dob.NodeParameters.Nodes` to only contain the one node. This is not mandatory, but it may cause confusion to have several nodes defined in a one-node system.

5.1.2 Multinode

This section could have a lot of examples, but I think it would become unwieldy with huge amounts of xml files, so I'm keeping it brief and descriptive instead.

First of all you should specify the number of nodes that are part of your system. Do this using the two parameters `Safir.Dob.NodeParameters.NumberOfNodes` and `Safir.Dob.NodeParameters.Nodes`. If you're not using persistence, set the parameter `Safir.Dob.PersistenceParameters.SystemHasPersistence` to false, otherwise configure the persistence as described in Section 5.2).

Each node has to be configured to know which node number it is, which is done in the parameter `Safir.Dob.ThisNodeParameters.NodeNumber`. Change this to contain the node number of each node (yes, this means having different contents in that file on each node in the system, unless you use an environment variable in the parameter, as described in Section 3.5.3).

If your nodes have multiple network cards the Dob needs to be told which network card it should use, and this is done by specifying a network address in `Safir.Dob.NodeParameters.NetworkAddress` (you only have to specify enough to uniquely identify the subnet that the card has, so if all your nodes are on the 192.168.123.* network you only need to specify 192.168.123.0).

The last step to get a working multinode configuration is to change the `Safir.Dob.DistributionChannelParameters.dou` file. What this file actually does is described below, but for now you just need to know that it specifies which multicast addresses the Dob uses. You need two `DistributionChannels` defined, the `System` and `PoolDistribution` ones, which must be the first two distribution channels in the array. Set the `MulticastAddress` members to be valid multicast addresses, that are not used for something else on your network (e.g. 224.10.10.127 and 224.10.10.128).

Now you should be able to fire up `dose_main` on the nodes of your system and start communicating. The easiest way to tell if it is working is to have a look at what `dose_main` prints to its stdout. If there is stuff there about "NodeUp" you're on the right track. Then start up Sate (see Section 13.1) and send and subscribe to some stuff across the network.

5.1.2.1 Routed networks

In the default configuration the Dob assumes that it runs on a non-routed network. To run on a routed network there are some further considerations that need to be made.

By default the *time-to-live* (TTL) of the multicast packets of the Dob is set to 1. This means that packets will not cross routers and will stay within the current subnet. By changing the parameter `Safir.Dob.NodeParameters.RoutingHops` you can modify this behaviour. For example, if it is set to 2 the packets will survive one routing hop.

When using routed networks it is important to be sure that bidirectional multicast routing is enabled on the router. To test this I 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.

5.1.2.2 Local objects

One last thing before going into the deeper technicalities of network configuration: Objects that are not sent over the network, even in a multinode system.

For some entities, services and messages it is not desirable to have them sent over the network at all. These are known as *Local* objects. 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.DistributionChannelProperty` to it, with the value "Local" as `DistributionChannel`, 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.DistributionChannelProperty</property>
  <class>Capabilities.Selection</class>
  <memberMapping>
    <member>
      <propertyMember>DistributionChannel</propertyMember>
      <value>Local</value>
    </member>
  </memberMapping>
</propertyMapping>
```

5.1.2.3 Distribution Channels

In a system with many nodes with different tasks it is not always necessary or desirable (e.g. for shared memory usage reasons) to send all data to all nodes. For example say there is one server that handles database stuff, and one that handles real-time tracking of all vessel in a harbour (using a radar or transponders or something like that). The applications on the database server have no need of the vessel data, and having them there would only use memory and generate CPU load (the Dob would still have to process them).

`DistributionChannels` are the solution to this problem. They allow you to specify which data is sent to which nodes.

For the above example you could set up a distribution channel called `VesselData`, which only contains the nodes that need the vessel data (i.e. the tracking server and the operator consoles). You have to specify a new `MulticastAddress` for this distribution channel (this allows the filtering of the data to occur on the network card, generating no load on the CPU).

For each type that is associated with the vessels (e.g. the vessel entities, and maybe some messages) you then specify that they use that distribution channel, using properties.

A DistributionChannel for VesselEntity

```
<?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.DistributionChannelProperty</property>
  <class>Capabilities.Vessels.VesselEntity</class>
  <memberMapping>
    <member>
      <propertyMember>DistributionChannel</propertyMember>
      <value>VesselData</value>
    </member>
  </memberMapping>
</propertyMapping>
```

There is no need to do any modifications to the applications involved.

It is possible to have up to 32 distribution channels defined for a system. But having more than 8 (or maybe 16 depending on your hardware) causes the filtering may move from the network card into the CPU. If you have other uses of multicast (IGMP) in your system one or more of these 8 hardware accelerated "slots" may already be occupied.

5.1.2.4 Priorities

In a system with many Dob objects competing for the network bandwidth it is possible to specify different priorities for the different objects.

The Dob supports up to six priority levels, which are defined in the `Safir.Dob.PriorityParameters` parameter file.

It is also possible to specify whether a certain priority level uses *acks* (acknowledgements) on the LAN. For most stuff acks are needed, since otherwise it is not certain that an entity update or a message is received on the network or not, but for cyclic data

(stuff that is updated regularly, and where it doesn't matter that one update is missed every now and then) it is useful to avoid the extra network load generated by the acks.

An acked entity update, for example, will generate one packet for the entity update itself (sent by multicast to all nodes in the system), and then one ack packet from each of the receiving nodes. Disabling the acking will potentially reduce the collisions on the network quite drastically, allowing for higher throughput or lower LAN load (which might be wanted for other things).

5.1.2.5 Ports used

The Dob uses the following ports:

Port(s)	Use
6970	KeepAlive messages, to know which nodes are running.
6971	Acknowledgements
6972-6977	Data (one port for every priority level)
31221	Low level error logs (from components that can't use Software Reports)

If the system is configured to be StandAlone (as described in Section 5.1.1) only the 31221 port is used, the others are not opened or used.

5.2 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 two *backends*, 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.SystemHasPersistence

Specifies if persistence service is expected to run in the system. (if set to false `dope_main` should not be started.)

Safir.Dob.PersistenceParameters.Backend

Which backend should be used, Odbc database or files.

Safir.Dob.PersistenceParameters.FileStoragePath

The path where files are stored if the file backend is chosen. Obviously `dope_main` needs write permissions here.

Safir.Dob.PersistenceParameters.OdbcStorageConnectString

The connection string to use to connect to the database.

Safir.Dob.PersistenceParameters.StandaloneMode

StandaloneMode means that each `dope_main` that is started is saving persistent data in its own storage. Only valid if several `dope_main` runs on different nodes in a redundant system. **This functionality should be used with caution.** If the `dope_main` starting as active is missing persistent data, all other persistent data storages will be cleared and persistent data will be lost! I.e. the active `dope_main`'s persistent data will override the other storages.

There is more information about the persistence service in Chapter 8.

5.3 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.

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".

5.4 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>
    <arrayElements>
<!-- The first array item is the default queue lengths.
  It matches all connection names. -->
      <arrayElement><object><name>Safir.Dob.QueueRule</name>
        <members>
          <member><name>MessageInQueueCapacity</name>
            <value>10</value></member>
          <member><name>MessageOutQueueCapacity</name>
            <value>10</value></member>
          <member><name>RequestInQueueCapacity</name>
            <value>10</value></member>
          <member><name>RequestOutQueueCapacity</name>
            <value>10</value></member>
        </members>
      </object></arrayElement>
<!-- let connections with 'long' in their connection names
  have longer message in queues and request out queues. -->
      <arrayElement><object><name>Safir.Dob.QueueRule</name>
        <members>
          <member><name>ConnectionNameRegex</name>
            <value>long</value></member>
          <member><name>MessageInQueueCapacity</name>
            <value>20</value></member>
          <member><name>RequestOutQueueCapacity</name>
            <value>50</value></member>
        </members>
      </object></arrayElement>
<!-- let connections with at least one capital letter in
  their connection name have longer message out queues. -->
      <arrayElement><object><name>Safir.Dob.QueueRule</name>
        <members>
          <member><name>ConnectionNameRegex</name>
            <value>[A-Z]+</value></member>
          <member><name>MessageOutQueueCapacity</name>
            <value>300</value></member>
        </members>
      </object></arrayElement>
    </arrayElements>
  </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 [10.1](#), and the two items on overflows in Appendix [B](#). It is important to understand what effects, other than just increasing memory use, this may have on your system.

5.5 Shared Memory config

The `Dob` has two parameters that control the amount of shared memory used, `TypesystemSharedMemorySize` and `SharedMemorySize` (both can be found in `Safir.Dob.NodeParameters.dou`). The first parameter controls the amount of memory available for the typesystem (the internal tables for all types, and all parameters), and the second one controls the amount of memory available for the distribution mechanism.

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 `dose_main` you know that you have enough memory allocated for the typesystem.

Knowing what value to set `SharedMemorySize` to is trickier, but one approach is to run the system and monitor the memory usage, for example using `dobexplorer`, which is described in Section [13.2](#).

Chapter 6

Software Error Reporting

Software Reporting provides functionality for sending software reports from applications using the Dob as communication mechanism. Software Reporting introduces a number of Dob messages that correspond to the different types of software reports that can be sent.

To make the sending of a software report as simple as possible, Software Reporting provides interface libraries to be used by applications. The interfaces add common information, like timestamp, node name and sequence number, to the report.

Software Reporting provides a logger application (`swre_logger`) that subscribes for software reports and writes them to standard output or a log file. It is possible to set various filter conditions for the logger application. Type "`swre_logger -?`" to get a complete listing of the command line options.

6.1 Report types

There are five predefined report types with the following intended usage:

- **Fatal Error Report:** Use it to report static conditions that must be fulfilled to be able to start/continue executing the program, for example missing static resources or invalid configuration. After a fatal error has been reported the program should terminate.
- **Error Report:** Use it to report detected runtime errors, for example a message from an external system in an invalid format. Normally the program continues to execute, possibly in a degraded state.
- **Resource Report:** Use it to report a missing/acquired dynamic resource. Note that it is "ok" for dynamic resource to be temporary missing which means that a Resource Report should be sent only after a reasonably number of retries to acquire it.
- **Program Info Report:** Use it to report internal program information for debugging purposes. Normally the sending of this report type is controlled by internal status variables that are set by sending program info commands to the program. This report type is used by the tracer functionality described in Chapter 7.
- **Programming Error Report:** Use it to report programming errors, i.e. errors of assert-type. After reporting the application should terminate.

6.2 Send interface

To make the sending of a software report as simple as possible, Software Reporting provides interfaces (currently available for C++, C#, Java and Ada) to be used by applications. The interface adds common information to the report before it is sent. The following information is added:

- Timestamp
-

- **Sequence Numbers:** A software report has two sequence numbers. The Connection Sequence Number is incremented for each software report sent from a particular Dob connection. The Type Sequence Number is related to the report type and each type (Fatal Error, Error etc) has its own numbering series. This makes it easy to detect missing reports even when a certain report type isn't logged.
- Connection name
- Node name

All `Send` methods are thread safe.

6.3 Logger application

Software Reporting provides a separate logger application (`swre_logger`) that subscribes to software reports and writes them to standard output or file.

When logging to file, the max size of the log file is supervised by `swre_logger`. If the max size is reached the file is renamed to `old_<original name>` (an already existing "old" file is deleted) and a new log file is opened. This mechanism makes it possible to have logging to file without the risk of filling up the available disk space.

It is possible to set various filter conditions for `swre_logger`. Type "`swre_logger -?`" to get a complete listing of the command line options.

`Swre_logger` can be started in any number of instances in one or several nodes, each with its own filter conditions. For instance, it is possible to start an instance that logs only program information (PI) reports from a particular application.

Chapter 7

Software program information

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.

7.1 The backdoor command

Software Reporting defines a Dob message (`Safir.Application.Backdoor`) to be used to send program information commands to applications. Backdoor commands can either be sent using Sate (see Section 13.1), or by the command-line program "backdoor" (use `"backdoor -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 Program Info Report as an indication that it is alive.
- **Help:** When receiving this command the application should send a Program Info Report that contains a listing of the supported backdoor commands.

Backdoor commands are defined as Dob messages and therefore sent to all subscribers. This means that applications have to check whether or not a backdoor message is addressed to the own application. Software Reporting provides classes (`Safir.Application.Backdoor` and `Safir.Application.Backdoor`) that simplifies the reception of backdoor commands. The classes provide this functionality:

- Setup of subscriptions for PI command messages.
- Checking of whether or not a PI command is to be handled by own application.
- Automatic answers to Ping commands.

Output from an application as a response to a backdoor command should be through Program Information Reports (see Chapter 6).

7.2 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 only be viewed by developers or integration. 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 in a reference site or when running in a situation when it is not desirable or possible to halt execution with a debugger.

The functionality is supported in C++, C#, Ada and Java. The interface to the trace logging functionality is a class named *Tracer* (which is how it will be referred to in the rest of this chapter).

7.2.1 How to use

The main goal of the Tracer class is for it to be as easy to use as possible, and that its usage should be as close to the languages own text input/output-syntax as possible.

1. Instantiate the Tracer class in each of the classes/packages where you want to use trace logging. Each instantiation requires a "prefix" which is used to enable and disable the logging from each tracer instantiation (described below). Several Tracer instances can use the same prefix, and will then all be enabled/disabled by the same command.
2. 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);
```

- **Ada**

```
debug.Put_Line("Testing logging " & Float'Wide_Image(myFloat) &
              ", " & Float'Wide_Image(myInt));
```

- **Java**

```
debug.println("Testing logging " + myFloat + ", " + myInt);
```

3. Enable and disable the logging by using the "backdoor" command. Send "<prefix> on" to turn on the logging of a prefix (also try "help" to see what prefixes are registered and what their current states are). To do this, type "backdoor -c myconnection myprefix on" in a command line window (if you skip the "-c myconnection" bit you will send the command to all applications in the system, which may have undesired consequences).
 - It is possible to turn on/off all prefixes by sending "all on" to your application.
 - It is possible to turn on/off prefixes immediately from the start of an application by setting the environment variable FORCE_LOG to one or several "<prefix>" or "all". This is useful to be able to have logging on by default or for logging the startup behaviour of an application. See Section 7.2.3 for a few hints on how this feature can be used.
 - Log output will be sent to the swre_logger application. Remember that it is possible to filter output in the swre_logger. Do a swre_logger -? to find out how to do it.
 - Log output is also sent to the applications standard output (its console).

7.2.2 Notes

Some notes on how the Tracer works and is implemented.

Buffering: Logging is buffered. The buffer is flushed every 0.5 seconds into one ProgramInformationReport. This means that a lot of output will be sent as one report.

Prefixes: The prefix is not "registered" in the internal data structures until it is actually used (the reason for this is rather complicated). This means that if you send a backdoor help command to your application before all Tracers (with unique prefixes, of course) have been used, you will not see all your prefixes in the help text.

Extra threads and connections: When logging is on an extra thread will be started in the application. This thread has its own Dob connection from which the output will be sent. The Connection name is set to the name of the process and the PID of the process. One upshot from this is that if all applications logs are turned on at the same time the number of connections in the system may double. The easiest way to avoid this is to only turn on logging in the application(s) that you are interested in (Use the -c <connectionName> syntax when using "backdoor").

When to start logging: A rule for using the Tracer is to not send any output to it before you have opened your own connection (instantiation is ok though). Since the connection is usually opened in the constructor of the application this means that all that should be avoided is logging in constructors that are called by the application constructor.

7.2.2.1 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 Ada the check is made for every call to `Put` or `Put_Line`. As far as the author is aware there is no equivalent of the C# functionality in Ada, so it is not possible to avoid the check without encapsulating the debugging inside if-statements.
- 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 `java's 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++.

7.2.3 The FORCE_LOG environment variable

There are a few different ways that the `FORCE_LOG` environment variable can be used.

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.

7.2.4 Troubleshooting the tracer

Some solutions to common problems.

Why do I have to call `flush()` to get the logging to work?

You don't. You need to send a `std::endl` directly to the tracer to end lines. Do not use `"\n"` in your strings to generate newlines, and don't expect `std::endl`'s that you've put into a `std::wostream` to generate a flush in the tracer.

The reason is that `std::endl` is in fact a function object that, when passed into an output stream (such as `std::ofstream`, `std::wcout` and the swre tracer), generate one newline and one flush call. The flush call is interpreted by the swre tracer as a request to flush the buffers "soon", so the tracer starts a timer which will flush the buffers in half a second, hoping to collect more trace statements that can be sent along as well.

An explicit call to `flush()` will generate an immediate flushing of the buffers (as opposed to the timed kind), and should only be used when you explicitly need to get the buffers completely flushed immediately. One use I can think of is to add a call to `flush()` when debugging, to get all trace statements flushed just before entering some code that crashes.

In short: If you have to add `flush()` statements to your code to get the logging to work, you're not using the tracer right.

Can I get timestamps on every trace line?

No, sorry. That is not implemented (yet).

Can I get logging to only go to <my favourite logging output>?

Nope, but I've toyed with the idea of making it possible to explicitly control the logging output to

- a UDP port (with or without broadcast)
- Program Information
- `stdout/stderr`
- etc.

But this is not implemented yet...

Chapter 8

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 5.2).

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.

This also allows for a solution to a common problem: When a `dou` file is changed (a member is added or removed or an array length is changed) any old stored binary persistent data is unusable (this is because the binaries are interpreted directly using the `dou`-files, e.g. by jumping straight in and getting some specific bytes out for a certain member). 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 recompile `dots_generated`, 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.

Redundancy The persistence service has support for hot-standby/redundancy. To enable hot-standby persistence configure the persistence storage to point out a shared network storage like a shared disk/NAS or a database/database cluster. Then you can 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 hot-standby mode. If the node running the active `dope_main` goes down, one of the hot-standby `dope_main` is activated and the persistence service will continue to run.

Simple redundancy There is also a possibility to have a simple redundancy where each started `dope_main` is saving persistent data in its own storage. The storage can be as in normal case files or database and can be different on each node (i.e. one node can have file storage and another can have database storage). The `dope_main` that becomes the active one during system startup will read its persistent data and inject it to the system, the other backup `dope_mains` will synchronize their storage with the data in the `dob`. All `dope_mains` will then start saving all data in the system that is tagged for persistence. This functionality is enabled with the parameter `StandaloneMode` (see Section 5.2).

This functionality should be used vid caution. If the `dope_main` starting as active is missing persistent data, all other persistent data storages will be cleared and persistent data will be lost! I.e. the active `dope_mains` persistent data will override the other storages.

Chapter 9

Utilities

Safir SDK Core contains a few utilities to simplify some things.

9.1 Sending many requests

Although the `Dob` allows you to have several outstanding requests at a time (see also Section 5.4) 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.

9.2 Ace Dispatcher

For applications written in C++ and using ACE the class `Safir.Utilities.AceDispatcher` provides a `Dispatcher` class that performs the thread switch and call to `Dispatch` as described in Section 4.2.1.

Just instantiate the `AceDispatcher` passing it your `ACE_Reactor`, and pass a pointer to the instance to your connection when calling `Open`, and you're done. The `AceDispatcher` will now perform the thread switch through the reactor main loop using `notify`.

9.3 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.safirsdk.com>).

Chapter 10

Tips and Tricks

This section discusses some good design strategies for applications

10.1 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 B.

10.2 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.

10.3 Handling exceptions

There are two base classes for exceptions. `Safir.Dob.Typesystem.FundamentalException` and `Safir.Dob.Typesystem.Exception`. Exceptions should be handled by the caller, but not `FundamentalExceptions`.

`FundamentalExceptions` are programming errors that can be avoided. 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 and restart the application (maybe after fixing it...)

10.3.1 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 lose messages, requests and entity subscription responses if you try to continue execution after this has occurred.

So catch all "expected" exceptions in the callbacks, but let all programming-error type exceptions propagate up to your main loop where you report the error and exit the program.

10.4 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 *FatalErrorReport* (see Chapter 6) 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 13.1), 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.

10.5 Multithreading

A connection should only be used from within one thread. The queues and other states associated with a connection are not thread safe, for simplicity and efficiency. One thread can handle many connections, though.

10.6 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 10.1 describes this as a sequence diagram, albeit without showing the part about registering the other handlers, since that would clutter up the diagram needlessly.

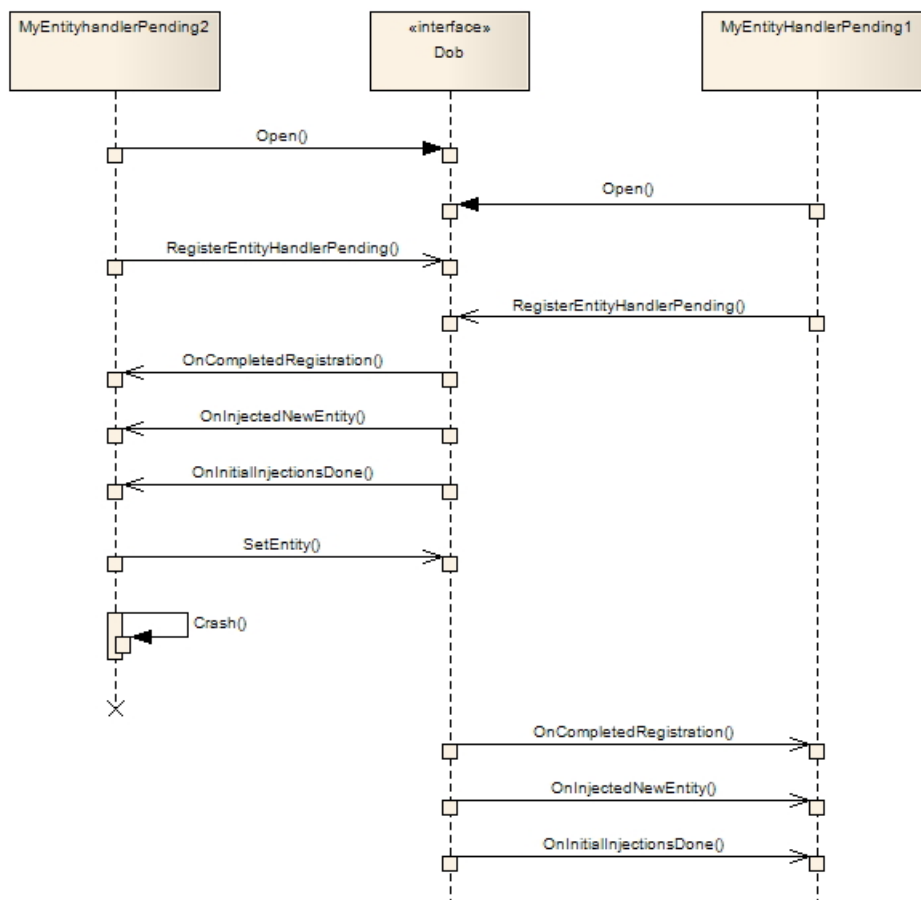


Figure 10.1: Sequence diagram for redundant entity handlers

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.

10.7 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 10.2, 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.

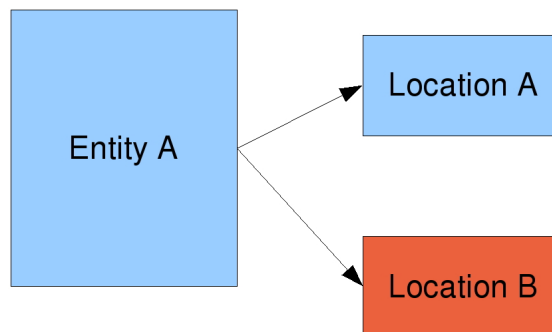


Figure 10.2: Entities that may cause orphans.

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 11.

Chapter 11

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 of 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 contact Saab AB at <http://www.safirsdk.com/>.

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.

11.1 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.

11.2 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.

11.2.1 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 11.1):

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.

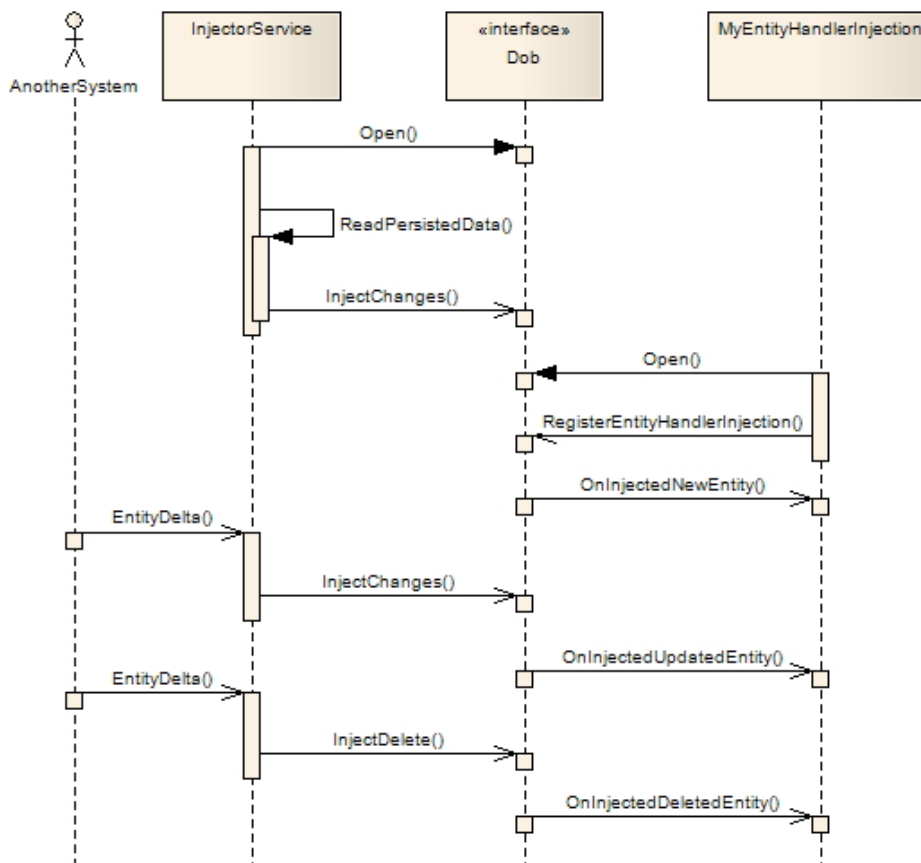


Figure 11.1: Injection sequence diagram

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 11.2.3) 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.

11.2.2 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 11.1 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 11.1: Timestamp merge, initial entity

Member	Timestamp	Value
0	100	Foo
1	100	<i>Null</i>
2	100	Bar
3	100	<i>Null</i>

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 11.2.

Table 11.2: Timestamp merge, first injection

Member	IsChanged	Timestamp	Value
0	True	110	hello
1	True	110	33
2	False	0	<i>Null</i>
3	False	0	<i>Null</i>

This will be merged with the current entity (again, on "our" system) into the object in Table 11.3.

Table 11.3: Timestamp merge, merged object

Member	IsChanged	Timestamp	Value
0	True	110	hello
1	True	110	33
2	False	100	Bar
3	False	100	<i>Null</i>

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 11.4 will be set in the Dob (and shown to subscribers or readers).

Table 11.4: Timestamp merge, resultant entity

Member	Timestamp	Value
0	110	hello
1	110	33
2	100	Bar
3	100	<i>Null</i>

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 11.5 (but note that this delta arrives to our system after the 110-delta).

Table 11.5: Timestamp merge, another injection

Member	IsChanged	Timestamp	Value
0	True	105	Lemon
1	False	0	<i>Null</i>
2	False	0	<i>Null</i>
3	True	105	Curry

Which will be merged with the current entity to become the object shown in Table 11.6

Table 11.6: 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.

11.2.3 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 12

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.)

12.1 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 (APP) or Presentation applications (PRSEs). 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 PRSEs should use contexts to support Replay sessions.

12.2 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. For example, a PRS 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.

12.3 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, or types used for logging purposes, e.g. Swre reports.

For example, an APP or PRS that is connected to context 0 can "see" all that goes on in context 0 and the ContextShared types. An APP or PRS that is connected to context 1 can "see" all that goes on in context 1 and the ContextShared types.

The following rules apply 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 PRSes to have connections to several contexts at once. Without ContextShared types PRSes 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 PRS 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 PRS 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 PRS to send a request in the wrong context, since it only knows of one context at a time.

12.4 Using the context mechanism

This section contains some hints on how APPs and PRSes should be designed to support Replay in different kind of systems.

12.4.1 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.
-

12.4.2 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 PRSes). 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 PRS 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.

PRSes 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 PRSes in the console that does not listen to the mode change, e.g. the PRS that shows alerts.

12.4.3 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 PRSes. A correct implemented APP or PRS will work without any modification in both a Type 1 and a Type 2 system.

12.4.4 Starting extra PRSes 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 PRSes (either new instances of the ones that show Normal data, or special replay-PRSes) are started to show the Replay data.

In this case the PRSes don't listen to the ReplayApp control entities. The PRSes 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 PRSes (either new instances of the Normal PRSes, or special replay-PRSes), telling them which context to connect to.

PRSes 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).

12.4.5 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 PRSes in this type of system must connect to all contexts that are used on the console. The PRSes 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.

12.5 APP design

Almost all applications should only be aware of context 0. An obvious exception to this rule is the the ReplayApp itself.

12.6 PRS design

In a Type 1 or Type 2 system a well behaved PRS should be designed according to the following rules:

All PRSes (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 PRS 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 PRS normally does not own entities, but if it does, registrations and creation of entities must be done again.)

The attach and subscription are the same actions that the PRS 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, PRSes in Replay mode needs to disable buttons etc that would generate requests, or be able to ignore the error responses from the ReplayApp.

Chapter 13

Test support and Tools

Safir SDK Core contains some tools that are useful for, among other things, application and system testing.

13.1 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.

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 HSI is implemented fully, or pretending to be another application that your application communicates with.

Figure 13.1 shows Sate with the `Safir.Application.BackdoorCommandMessage` opened. All the Dob classes are listed in the left sidebar, where you can right-click to get operations you can do on the types. The right sidebar contains operations you can do on the object opened.

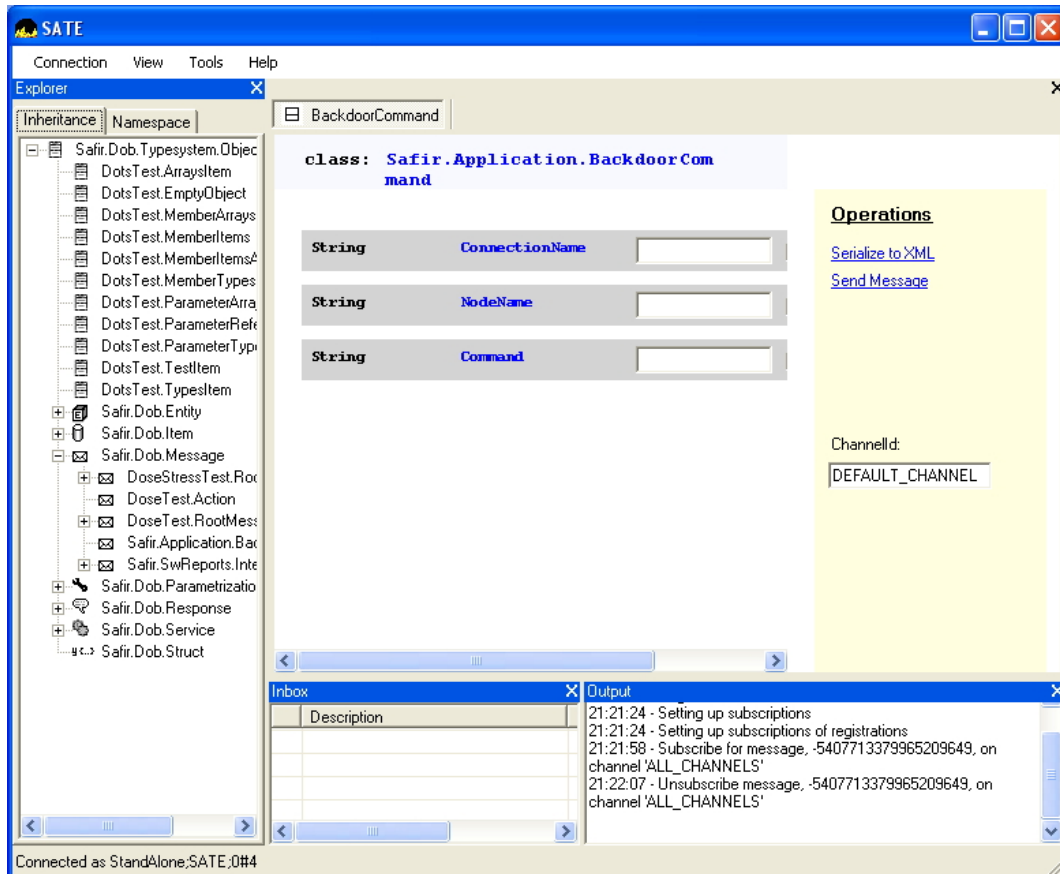


Figure 13.1: Sate screen shot

The lower left part is an "inbox", containing the most recent entity updates and requests, service requests and messages received. The lower right hand part shows a history of what "has happened", e.g. showing if a request has been successfully sent, etc.

13.2 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 13.2 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.)

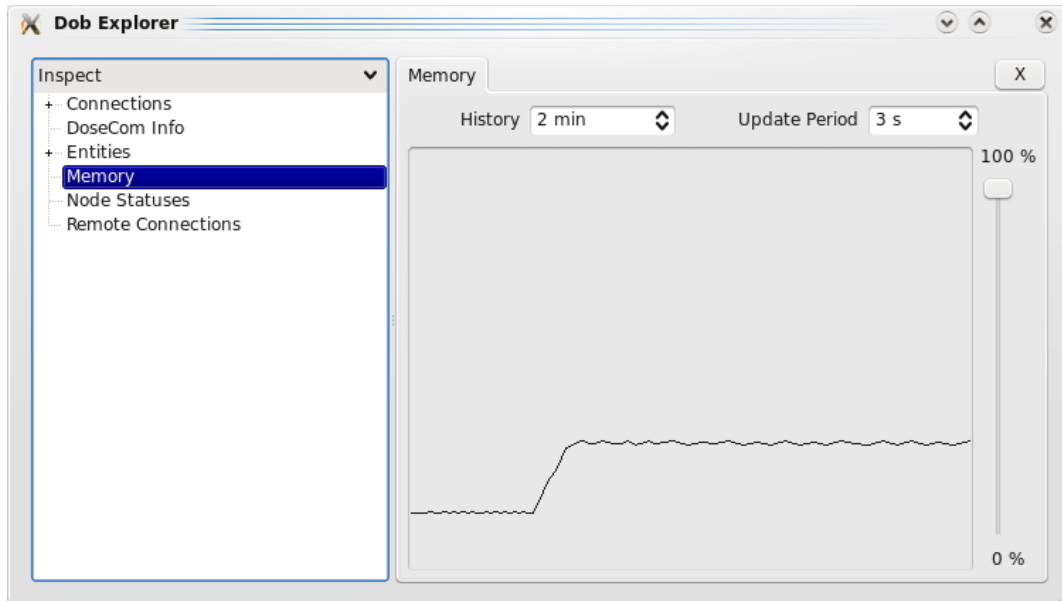


Figure 13.2: Dobexplorer showing a graph of memory usage.

Figure 13.3 shows dobexplorer displaying a table of the node statuses of a multinode system.

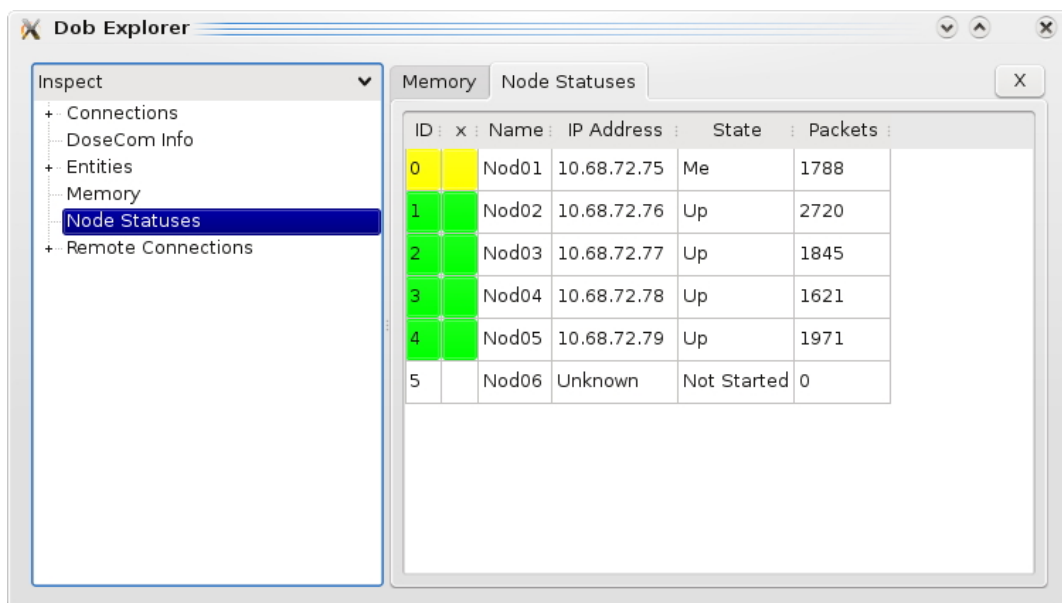


Figure 13.3: Dobexplorer showing node statuses.

Chapter 14

C++ ODBC database wrapper

Safir SDK Core contains a C++ *ODBC* wrapper, which is located in the `Safir.Databases.Odbc` namespace. The wrapper library supports all odbc capable databases, but Saab are using the Mimer SQL database (<http://www.mimer.com>).

This library contains three different classes which groups different parts of the odbc interface. The first class is `Environment` and this class contains the environment specific part of the odbc library. The environment is allocated once in an application and is used by odbc to store application global data. The next class is `Connection` and this class contains all connection specific information. An application can have several connections made through a single environment if several queries needs to be made in parallel. The last class is `Statement` which contains all information about individual statements made through a connection.

The odbc wrapper also supports stored procedures, a small example of which can be seen in Section 14.3.

14.1 Connecting

A connection to a database is made through an `Connection` object and a `Connection` object requires a `Environment` object. The first step to connect is to declare these two objects in the class that will use them.

Declaring Connection and Environment

```
class EntryService
{
private:
    Safir::Databases::Odbc::Connection    m_connection;
    Safir::Databases::Odbc::Environment  m_environment;

public:
    bool Allocate();

    void Deallocate();
};
```

The second step is to allocate the environment and connection objects. This is done in the `Allocate()` method. The environment should be considered a static resource when allocated but the connection is a dynamic resource and may be lost during run-time. (The `Allocate` method is written so that it can be called again to re-connect to the database should the connection fail.)

Allocating Environment and Connection

```
bool EntryService::Allocate()
{
    try
    {
        if (!m_environment.IsValid())
            m_environment.Alloc();
    }
```

```

    if (!m_connection.IsValid())
        m_connection.Alloc(m_environment);

    if (!m_connection.IsConnected())
    {
        m_connection.Connect
            (Capabilities::DiaryEntries::DiaryEntryDb::
             Parameters::ConnectionString());
        m_connection.SetConnectAttr
            (SQL_ATTR_CONNECTION_TIMEOUT, 5L);
    }
}
catch(const Safir::Databases::Odbc::ReconnectException & ex)
{
    ... error handling ...
}

```

When both the environment and connection objects are allocated the connection to the RDBMS can be made. Before the connection is made a timeout period should be set. If no timeout period is set then ODBC will use an indefinite timeout period.

14.2 Exceptions

There are two exceptions that you need to handle when communicating with the database `Safir.Databases.Odbc.ReconnectException` and `Safir.Databases.Odbc.RetryException`. When the first happens you have lost your connection to the database and you need to reconnect. The second happens if the database is “too busy” to handle your operation. In this case you need to retry the operation in a short while.

14.3 Making a query

Before a query to the database can be made the query has to be prepared properly. Preparing a query involves sending the actual sql statement to the database to be precompiled before execution. In the sql statement parameters can be set as a question mark (`'?'`). This tells the Database that a value will be sent for this parameter at execution time. By using parameters an application can prepare the queries it needs at start-up and then use them several times during run-time by setting the parameter values. The application needs to bind these parameters to specific variables that can be used to set the value of the parameters before execution. The columns in the recordset returned by the query is managed in a similar way to parameters. Each column is bound to variable that the application can read to process each row of the recordset. When setting up a query for later execution a timeout period should be set and if no timeout period is set then odbc will use an indefinite timeout period.

All queries are made through a connection and if that connection is lost all prepared statements made through that connection is also lost and must be remade. ODBC cannot manage multiple queries in a single connection so a query needs to be closed before setting up another.

Declaration of variables used as parameters/columns in the query:

Declarations

Statement	<code>m_ReadStmt;</code>	// The query object.
Int32Parameter	<code>m_paramId;</code>	// The parameter to the query.
Int32Column	<code>m_columnAmount;</code>	// The first field returned.
WideStringColumn	<code>m_columnName;</code>	// The second field returned.

Preparation of the statement and binding of parameters/columns:

Preparing a statement

```

try
{
    if (!m_ReadStmt.IsValid())

```

```

    {
        m_ReadStmt.Alloc( &m_connection );
        m_ReadStmt.Prepare("call spReadEquipment(?)");
        m_ReadStmt.BindParameter(1, m_paramUnitId);
        m_ReadStmt.BindColumn(1, m_columnAmount);
        m_ReadStmt.BindColumn(2, m_columnName);
        m_ReadStmt.SetStmtAttr(SQL_ATTR_QUERY_TIMEOUT, 5L);
    }
}
catch(const Safir::Databases::Odbc::RetryException & ex)
{
    ... error handling ...
}

```

When executing a query all parameters to the query needs to be set and then `Execute` is called. When `Execute` is finished the query has been made but no information has yet been retrieved to the columns. This is made by the method `Fetch`. Each call to `Fetch` retrieves a new row from the database and `Fetch` can be called until it returns false to signal that all rows has been returned. If no rows was returned by the query the `Fetch` will return false on the first call. After all rows has been processed then the query should be closed to release the rows returned by the query. After the query has been closed the statement is ready to be used again.

Executing and fetching the result

```

try
{
    m_paramId.SetValue(1);           // Set parameter value.
    m_ReadStmt.Execute();           // Execute the query.
    bContinue = m_ReadStmt.Fetch(); // Read first row.
    while (bContinue)
    {
        if (!m_columnName.IsNull())
        {
            Do(m_columnName.GetValue()); // Process Name.
        }

        if (!m_columnAmount.IsNull())
        {
            Do(m_columnAmount.GetValue()); // Process Amount
        }

        bContinue = m_ReadStmt.Fetch(); // Read next row.
    }

    m_ReadStmt.CloseCursor(); // Close this query.
}
catch(const Safir::Databases::Odbc::RetryException & ex)
{
    ... error handling ...
}

```

14.4 Transactions

When setting up the connection to the database we also specify whether transactions are manual or automatic. If nothing else is specified then the connection will be using automatic transaction and every update will be committed when completed. If something more complex is required such as creating a new vehicle table row and a new equipment table row within one transaction then manual transactions should be used. Manual transactions require that `UseManualTransactions()` is called on the connection and that calls `commit` or `rollback` are made when the transaction is finished.

Calling `UseManualTransactions` in `Allocate`

```
if (!m_connection.IsConnected())
{
    m_connection.Connect
        (Capabilities::DiaryEntries::DiaryEntryDb::
            Parameters::ConnectionString());
    m_connection.UseManualTransactions();
    m_connection.SetConnectAttr(SQL_ATTR_CONNECTION_TIMEOUT, 5L);
}
```

The `Commit` function is to be called when all operations are completed successfully and all updates shall be written to the database persistently for use by other operations. If an error has occurred then the function `Rollback` shall be called to remove all changes this transaction has made.

Transaction example

```
try
{
    m_paramId.SetValue(1);
    m_DeleteStmt.Execute();
    m_pConnection->Commit();
}
catch(const Safir::Databases::Odbc::RetryException & ex)
{
    m_pConnection->Rollback();

    ... more error handling ...
}
```

Chapter 15

More help

Apart from this document there are other sources for help.

15.1 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 downloads, and should be located under `$(SAFIR_SDK)/docs/`.

Use this documentation! It is the place where interface details are explained!

15.2 Forum

Questions can be posted in the Safir SDK Core forum at <http://www.safirsdk.com>.

Chapter 16

Glossary

This glossary mostly contains Safir-related acronyms and terms. General computer terms are not included, like "UDP", "Multi-cast" 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).

HSI

Human - System 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 Software Reporting and Trace logging functionality.

Appendix A

Example applications

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 (also referred to as an MMI, Man-Machine Interface, or HMI, Human-Machine Interface). 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.

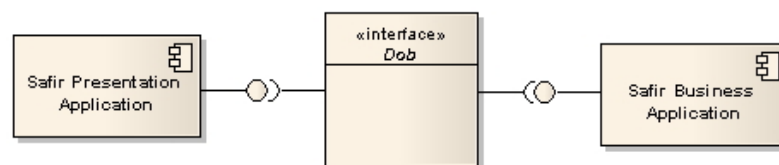


Figure A.1: Separation of logic from GUI

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 the Safir ODBC database interface (see Chapter 14). 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.

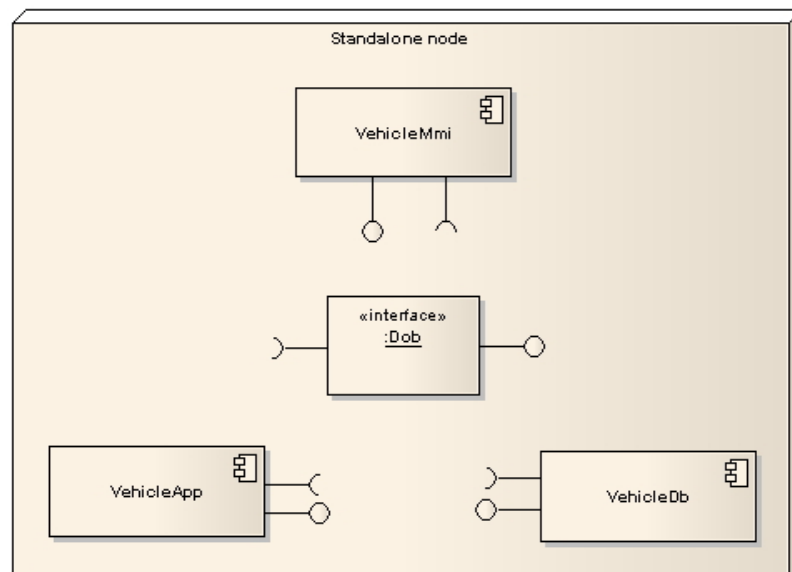


Figure A.2: Deployment of the Vehicle system

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 `EntityHandlerInjection`. 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.

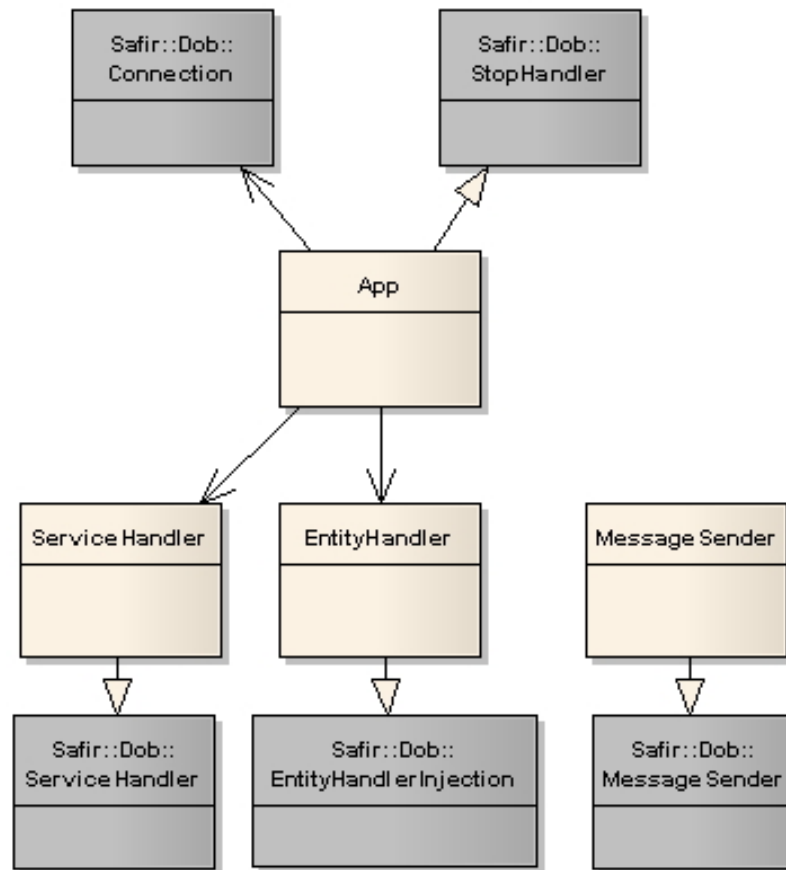


Figure A.3: VehicleApp class diagram

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 presents 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`.

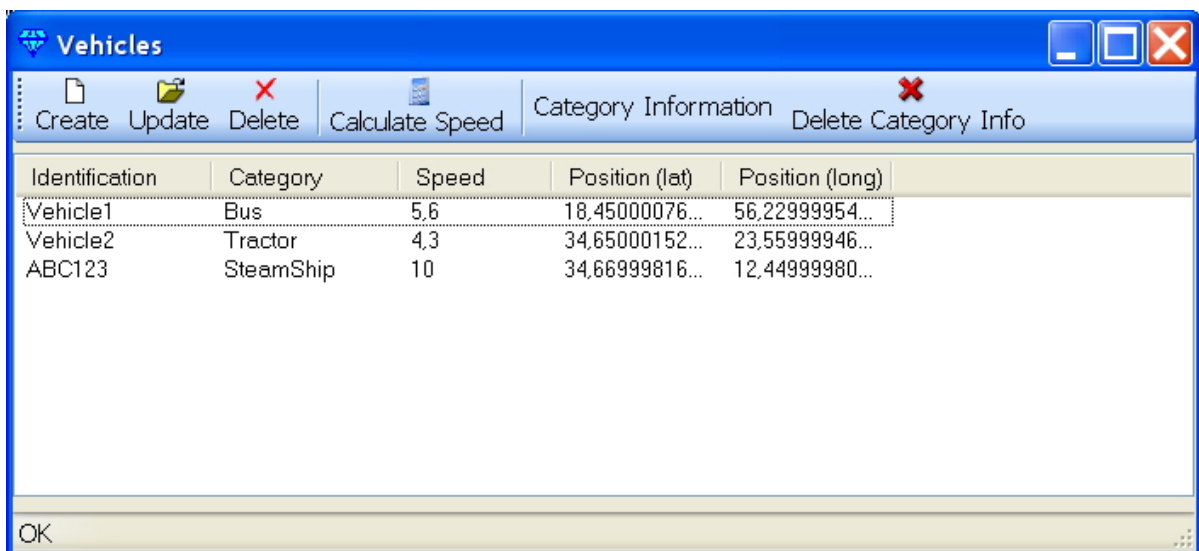
To obtain additional database information – that is not received through a subscription response – for a selected vehicle, it is possible to request this from the `VehicleDb`. If the database information does not exist, it is created by `VehicleMmi`.

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.



Identification	Category	Speed	Position (lat)	Position (long)
Vehicle1	Bus	5,6	18,45000076...	56,22999954...
Vehicle2	Tractor	4,3	34,65000152...	23,55999946...
ABC123	SteamShip	10	34,66999816...	12,44999980...

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`.

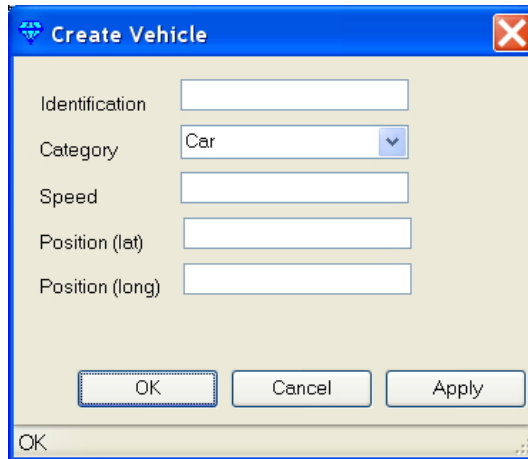


Figure A.5: VehicleMmi Create dialog

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.

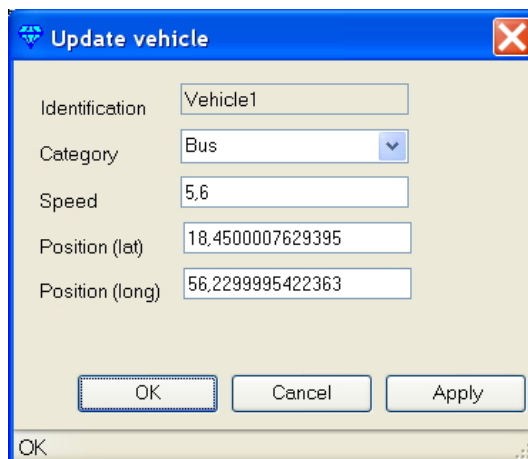


Figure A.6: VehicleMmi Update dialog

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 service response.

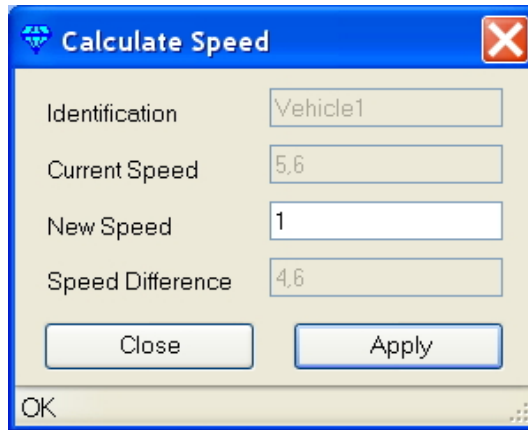


Figure A.7: VehicleMmi Speed calculation dialog

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.

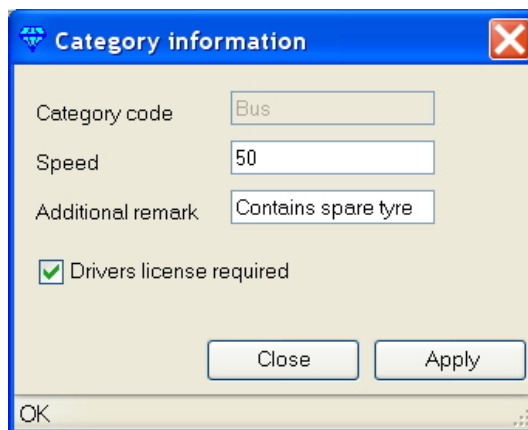


Figure A.8: VehicleMmi Category information dialog

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.

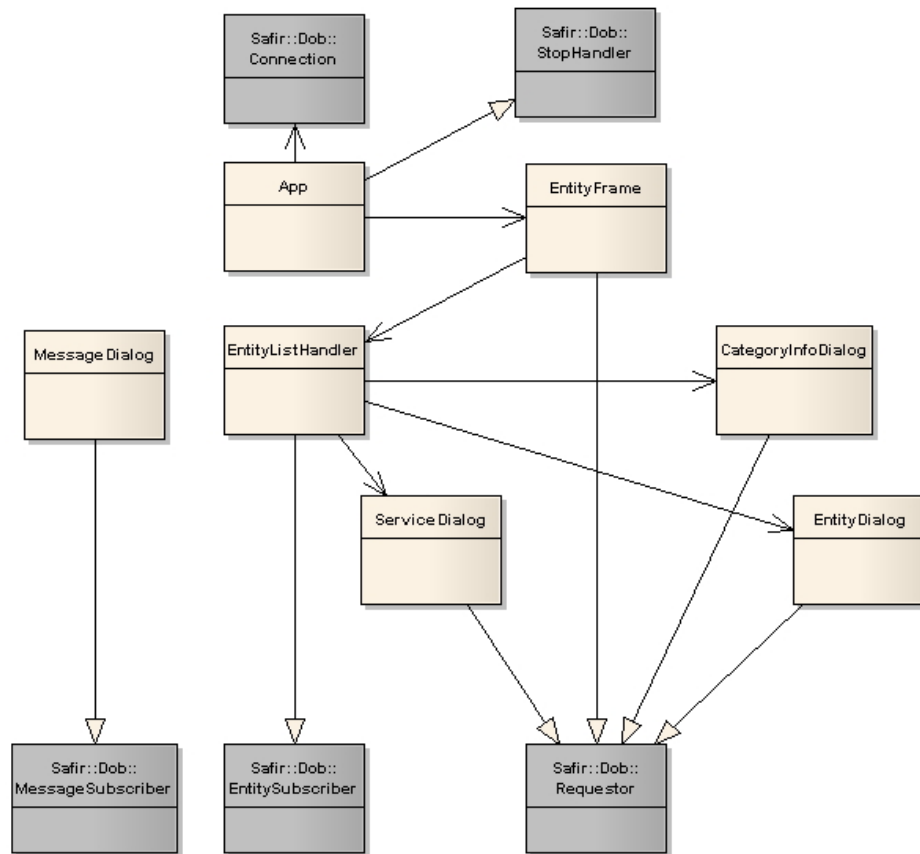


Figure A.9: VehicleMmi class diagram

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 (see Chapter 14).

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.

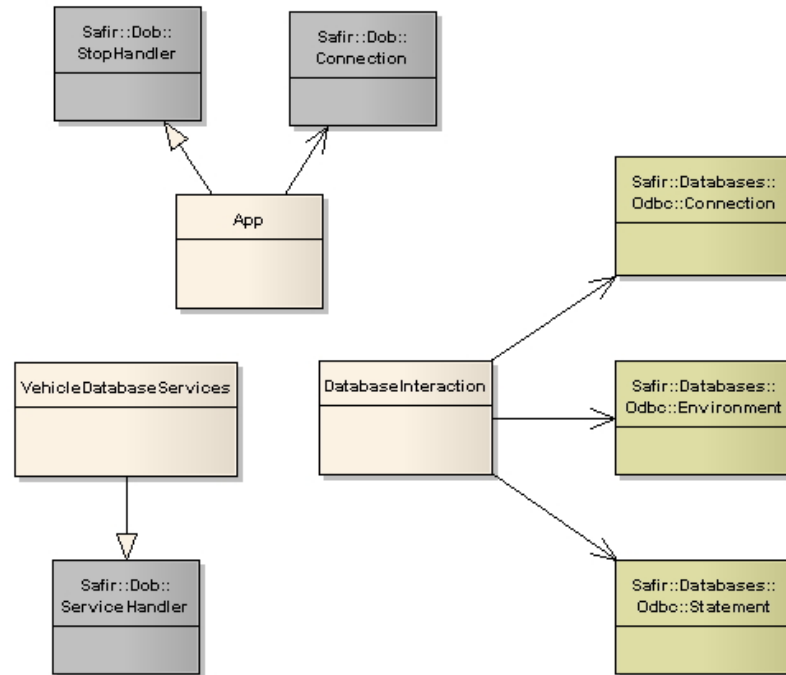


Figure A.10: VehicleDb class diagram

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 executed on request.

Appendix B

FAQ

Why do I get `IllegalArgumentException` 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 try to read the contents of an instance.

Why does `dope_main` crash at startup?

Chances are that you have changed your dou-files, and that the persisted data is no longer valid. There is a solution to this, which is outlined in Chapter 8, 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 10.1.

Why shouldn't I have an infinite queue in my application to avoid overflows?

This is described somewhat in Section 10.1, and the FAQ entry above.

Appendix C

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://www.safirsdk.com/>.
