nehta

Example Technical Implementation of Interoperable Web Services

WCF

Version 2.1 — 30 June 2010

National E-Health Transition Authority

National E-Health Transition Authority Ltd
Level 25
56 Pitt Street
Sydney, NSW, 2000
Australia.
www.nehta.gov.au

Disclaimer

NEHTA makes the information and other material ("Information") in this document available in good faith but without any representation or warranty as to its accuracy or completeness. NEHTA cannot accept any responsibility for the consequences of any use of the Information. As the Information is of a general nature only, it is up to any person using or relying on the Information to ensure that it is accurate, complete and suitable for the circumstances of its use.

Document Control

This document is maintained in electronic form. The current revision of this document is located on the NEHTA Web site and is uncontrolled in printed form. It is the responsibility of the user to verify that this copy is of the latest revision.

Copyright © 2010, NEHTA.

This document contains information which is protected by copyright. All Rights Reserved. No part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without the permission of NEHTA. All copies of this document must include the copyright and other information contained on this page.

Table of contents

1 Introduction ... 1
1.1 Background ... 1
1.1.1 Document history ... 1
1.2 Purpose ... 1
1.3 Scope ... 1
1.4 Intended audience ... 1
1.5 Definitions, acronyms, abbreviations ... 2
1.5.1 Terminology ... 2
1.6 Style Conventions ... 2
1.7 Overview ... 3

2 Example service ... 4
2.1 Organisational ... 4
2.1.1 Testing ... 4
2.1.2 Sending discharge summaries ... 5
2.1.3 Checking discharge summary status ... 5
2.2 Informational ... 5
2.2.1 Discharge summary ... 5
2.2.2 Status ... 6
2.3 Technical ... 6
2.3.1 Informational attributes ... 6
2.3.2 Behavioural attributes ... 6
2.3.3 Non-functional attributes ... 10

3 Overview of WCF ... 16
3.1 General background ... 16
3.2 Technical Overview ... 16
3.3 Requirements ... 16
3.3.1 Client deployment ... 16
3.3.2 Web service deployment ... 16
3.3.3 Client development ... 17
3.3.4 Web service development ... 17
3.4 Platform used ... 17
3.4.1 .NET 3.0 Service Pack 1 ... 17
3.4.2 .NET 3.5 Service Pack 1 ... 17

4 Web service client ... 19
4.1 Modify the WSDL files ... 19
4.1.1 WSDL containing the service interface information ... 19
4.1.2 WSDL containing the service instance information ... 20
4.2 Generate the proxy from the WSDL files ... 20
4.3 Create a client program ... 21
4.4 Edit the client configuration ... 21
4.4.1 Create a custom binding ... 22
4.4.2 Create endpoint behaviour ... 24
4.4.3 Create an endpoint ... 25
4.5 Implement the client application ... 27
4.5.1 Instantiate the proxy object ... 27
4.5.2 Invoke the Web service methods ... 27
4.5.3 Close the proxy object ... 28
4.5.4 Handle exceptions and faults ... 28

5 Web service ... 30
5.1 Modify the WSDL files ... 30
5.2 Generate the service interface from the WSDL files ... 30

5.3 Create a WCF service project ... 30
5.4 Implement the Web service ... 31
5.4.1 Create the service implementation class ... 31
5.4.2 Implement the service methods ... 31
5.4.3 Add the WS-Addressing MessageID header ... 31
5.4.4 Throw faults ... 32
5.5 Edit Service.svc ... 32
5.6 Edit the service configuration ... 32
5.6.1 Create a custom binding ... 33
5.6.2 Create a service behaviour ... 33
5.6.3 Configure service ... 34
5.7 Package and deploy the Web service ... 36
5.7.1 Development Server ... 36
5.7.2 Deploy with IIS ... 36

Appendix A: References ... 38

Appendix B: Installation ... 39
B.1 Visual Studio 2005 ... 39
B.2 .NET Framework 3.0 Redistributable Package ... 39
B.3 Windows SDK for Vista and .NET 3.0 Runtime Components ... 39
B.4 Windows HTTP Services Certificate Configuration Tool ... 39
B.5 Visual Studio 2005 Extensions for .NET 3.0 ... 40
B.6 Internet Information Services (IIS) (optional) ... 40

Appendix C: Key management ... 41
C.1 Setting up the MMC for certificate management ... 41
C.2 Installing certificates for the client application ... 41
C.2.1 Adding the client certificate ... 41
C.2.2 Adding the service certificate ... 42
C.2.3 Adding the CA ... 42
C.3 Setting up the service certificates ... 42
C.3.1 Adding the service certificate ... 43
C.4 Adding the CA ... 43
C.5 Pitfalls ... 43

Appendix D: Debugging ... 44
D.1 Diagnostics configuration ... 44
D.2 Displaying service metadata (WSDL) ... 44
D.3 serviceDebug behaviour element ... 45
D.4 HTTP Debugging Proxy ... 45
D.5 Debugging a service that uses IIS ... 45

Document information

Change history

Version	Date	Comments
1.0	2007-07-03	Release
2.0	2008-12-01	Release
2.1	2010-06-30	Use ATS 5820-2010 Web Services Profiles.

This page is intentionally left blank.

1 Introduction

1.1 Background

The National E-Health Transition Authority (NEHTA) has recommended Web services as the mechanism for communication between organisations in Australia's e-health environment.

NEHTA has published a number of technical documents to support the use of Web services. These include the *Web Services Profile* [ATS 5820—2010].

1.1.1 Document history

Version 1.0 of this document was written to explain how to conform to the *Web Services Standards Profile* [WSSP2006] and *Guidelines for Implementing Interoperable Web Services* [GIIWS2007] using the Microsoft's Windows Communication Foundation (WCF) [WCF].

The *Web Services Profiles* [ATS 5820—2010] supersedes *Web Services Standards Profile* [WSSP2006], *Web Services Profile* [WSP2008] and *Guidelines for Implementing Interoperable Web Services* [GIIWS2007]. This document has been updated primarily to explain conformance to the *Web Services Profiles* [ATS 5820—2010].

1.2 Purpose

This document provides an example of how Web services conforming to the *Web Services Profiles* [ATS 5820—2010] can be implemented using a specific toolkit: Microsoft's Windows Communication Foundation (WCF) [WCF].

The End-to-end security profile in the *Web Services Profiles* [ATS 5820—2010] lists conformance criteria for Web services and clients that are secured from end to end. This document explains a way of building Web services and clients that meet these criteria using the Microsoft's WCF [WCF] toolkit.

The main purpose of this document is to support the understanding and interpretation of the conformance criteria in the *Web Services Profiles* [ATS 5820—2010]. However, it can also assist programmers who are learning how to use WCF.

This document is provided for educational purposes only. The method it describes is only one approach; there might be other, equally valid approaches. The code samples in this document are designed for simplicity and ease of understanding, rather than robustness and reuse. They are not written for use in a production system.

1.3 Scope

This document only covers WCF. Also available are example technical implementation documents covering Java API for XML Web Services (JAX-WS) [JAXWS] and .NET Web Services Enhancements (WSE) 3.0 [WSE]. Those other example technical implementations can interoperate with this implementation, but they will not be discussed in this document.

These examples are not an endorsement of these platforms by NEHTA.

1.4 Intended audience

This document is intended for:

- Software developers; and

- System administrators.

It is expected that the reader is familiar with programming using C#, and an understanding of Web services and Public Key Infrastructure (PKI) security using X.509 certificates.

The reader is also expected to be familiar with the *Web Services Profiles* [ATS 5820—2010]. The criteria from [ATS 5820—2010] are referred to by their criterion number (e.g. “WS 3.1.1.1 (a)”).

1.5 Definitions, acronyms, abbreviations

HTTP	Hypertext Transfer Protocol
HTTPS	Secure HTTP
IDE	Integrated Development Environment
IIS	Internet Information Services
MMC	Microsoft Management Console
SDK	Software Development Kit
WCF	Windows Communication Foundation
WSDL	Web Service Definition Language

1.5.1 Terminology

This document uses the following terms:

Web services	The technology for communicating between computer applications using SOAP, WSDL, and other related standards.
Web service	A computer program that provides services, and uses the Web services technologies to allow access to those services.
Web service client	A computer program that uses the services provided by a Web service. It invokes operations that are provided by the Web service. The abbreviated term “client” can also be used.
Web server	A computer program that makes Web resources (predominantly HTML Web pages) available via Web protocols (predominantly HTTP).
Server	A computer that is hosting a Web server or other programs that provides a service to other programs.

1.6 Style Conventions

This document uses the following style conventions:

Italics	• Document titles • Program names, tool names • File names, directory paths • URLs
`Monospace`	• XML fragments, names of elements and types, namespaces • Code fragments, names of classes, methods, and fields

	• Assemblies, packages • Command-line calls and arguments • Configuration properties
`Monospace + Bold`	• Emphasis for within XML and code fragments
"Double quotes"	• Graphical user interface options

1.7 Overview

Chapter 2 describes the service used as an example for this document.

Chapter 3 provides a brief overview of WCF.

Chapter 4 describes how to create a Web services client program using WCF.

Chapter 5 describes how to create a Web service program using WCF.

Appendix A lists references.

Appendix B provides instructions and notes on software installation.

Appendix C provides information on security key management.

Appendix D provides tips on debugging WCF programs.

2 Example service

This chapter describes the example service that will be implemented.[1]

The specification of a service would normally be produced by an independent organisation, which brings together the requirements of all the stakeholders. This chapter is an abridged version of the service specification that would be produced. Since this document is concerned with programming Web services, it focuses on the WSDL specification.

The example technical implementation described in this document assumes a WSDL-first approach, where the Web service implementation is developed using classes generated from the WSDL. This approach is in contrast to the implementation-first approach, where the WSDL is automatically generated from the implementation code. The WSDL-first approach is more applicable to an interoperable e-health environment, where standard WSDL specifications developed by independent organisations should be used to build Web services.

The structure of this chapter follows the approach described by the NEHTA *Interoperability Framework* [NIF2006] and uses concepts from the *Technical Architecture for Implementing Services* [TAIS2006].

2.1 Organisational

This example scenario is based on the exchange of discharge summaries. It has been simplified for ease of understanding—it is not intended to be a real world discharge summary scenario.

In the community for discharge summary exchange, there are two roles:

- Sending provider: the program that generates the discharge summary and sends it; and
- Receiving provider: the program that receives the discharge summary and tracks whether it has been acknowledged.

The business process of sending a discharge summary involves three activities:

- Testing if the receiving provider's discharge summary receiving service is operating;
- Sending discharge summaries from a sending provider to a receiving provider; and
- Checking the status of a discharge summary to see if the receiving provider has processed and acknowledged it.

2.1.1 Testing

In this activity, one party wishes to determine whether the receiving provider's service is operational or not. It can be used to check if the programs and the network have been correctly configured.

This activity illustrates the use of an operation that requires no parameters. It is implemented as an operation that does nothing, other than to return an empty result.

This operation is called "ping" after the program used to test if an internet protocol host is reachable across an IP network [PING].

---

[1] This chapter is identical to the corresponding chapter in the other *Example Technical Implementation of Interoperable Web Services* documents (i.e. for JAX-WS and WSE 3.0.)

This is a request-response operation at the technical-level compliant with criterion WS 5.1.5.1 (a) from the *Web Services Profiles* [ATS 5820—2010].

2.1.2 Sending discharge summaries

In this activity, a sending provider creates a discharge summary and sends it to the receiving provider.

When the discharge summary has been received, the receiving provider keeps track of which discharge summary it has received and whether it has been acknowledged by a person at the receiving organisation. This behaviour is to support the checking operation described in section 2.1.3.

This is a one-way operation at the business-level, and no response data is returned to the sender. The only way the sender can discover if it was successfully received is to use the check discharge summary status operation.

At the technical-level, this operation is implemented as a request-response operation compliant with criterion WS 5.1.5.1 (a). A response is sent back, but it contains no business-level information.

2.1.3 Checking discharge summary status

In this activity, a sending provider queries the receiving provider about the status of a particular discharge summary. The receiving provider returns the result to the querying provider.

This is a request-response operation: a response containing the status is returned.

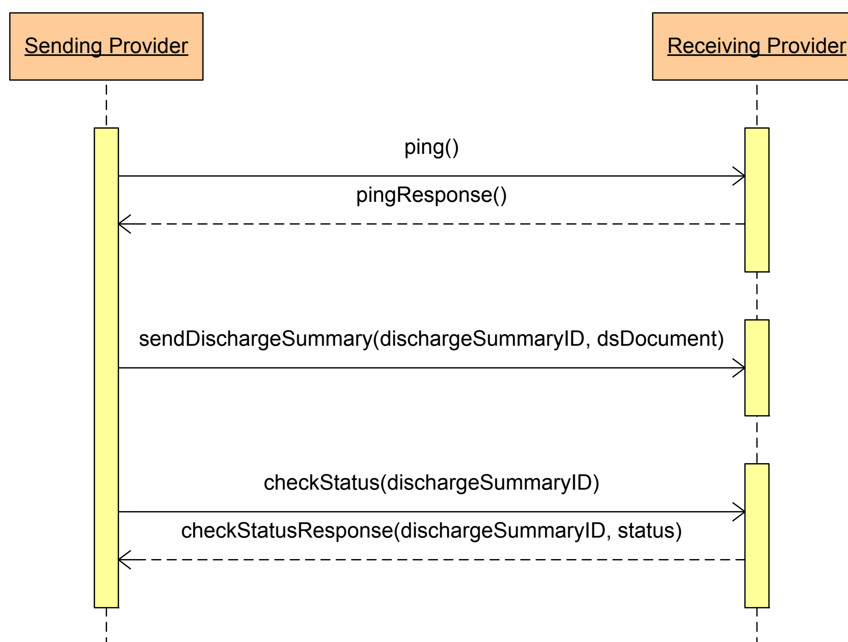

Figure 1: Example discharge summary business workflows

2.2 Informational

2.2.1 Discharge summary

The discharge summary is modelled as a document with an identifier and a notes field. The document identifier should be a globally unique string that is

allocated by the sender of the discharge summary. The notes field contains unstructured text.

The discharge summary data model is simple since the aim of this example is to demonstrate Web services, rather than demonstrate a real discharge summary scenario. NEHTA's *National Discharge Summary Data Content Specification* [NDS2006] contains much more structured data and metadata in the data model of a discharge summary.

2.2.2 Status

The possible status values for a discharge summary are:

- Not received: a discharge summary with the given document identifier has not been received;
- Pending acknowledgement: it has been received, but has not been acknowledged by the receiving party; and
- Acknowledged: it has been received and acknowledged.

The delay between receiving a discharge summary and it being acknowledged is not defined by the service. This is because acknowledgement is a manual process involving a person—it could take minutes or days to perform.

2.3 Technical

This section describes the technical aspects of the service interface specification. It is organised using the three types of attributes, as defined in the Technical Architecture: informational, behavioural and non-functional attributes [TAIS2006].

2.3.1 Informational attributes

The XML Schema used to define a discharge summary document is shown below. It defines a single complex type with 2 child elements: `documentId` and `notes`.

This XSD file will be stored in a file called *DischargeSummary.xsd*.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
  elementFormDefault="qualified">

  <xsd:complexType name="DischargeSummaryType">
    <xsd:sequence>
      <xsd:element name="documentId" type="xsd:string"/>
      <xsd:element name="notes" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

2.3.2 Behavioural attributes

The WSDL file contains a formal specification of the behavioural aspects of the service interface.

A WSDL file is not the complete documentation for a real service, which would require additional documentation to fully describe the service's behavioural attributes.

However, only the WSDL will be provided for this simple test service, because a complete description of the service is not required for the level of interoperability testing that it will be used for.

2.3.2.1 WSDL containing service interface information

Criterion WS 3.1.2.1 (a) recommends separating the service interface information from the service instance information. This section will go through the WSDL containing the service interface information for our sample Web service.

This WSDL file will be stored in a file called *DischargeSummaryReceiverInterface.wsdl*.

2.3.2.1.1 Header

The beginning of the WSDL file contains the start tag of the root element, which contains all the XML namespaces that this document will use.

It is a WSDL 1.1 document as required by criterion WS 3.1.1.1 (a). The `definitions` root element belongs to the WSDL 1.1 XML namespace, namely:

`http://schemas.xmlsoap.org/wsdl/`

Following criterion WS 3.2.3.1 (a), the addressing information in the WSDL is described using *WS-Addressing 1.0 – Metadata* [WSAM2007], whose namespace is:

`http://www.w3.org/2007/05/addressing/metadata`

The service namespace for this service was arbitrarily chosen to be:

`http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0`

It is a URL and uses a slash character as the separator, as recommended by criteria WS 3.1.3.1 (b) and WS 3.1.3.1 (c) respectively. This service namespace is used as the target namespace of the WSDL, following criterion WS 3.1.3.1.4. It is associated with the namespace prefix of `tns` so that it can be referenced in the document. The prefix of the target namespace does not necessarily have to be `tns`; it is just a commonly used convention.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:tns="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
  targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
  name="DischargeSummaryReceiver">
```

2.3.2.1.2 Types

The types section of the WSDL declares the elements and data types of the messages used by the service. It contains a schema defined using the W3C XML Schema language [XSD2004].

The target namespace of the schema in the WSDL's `types` section is the service's namespace. Since the wrapper elements for the service's operations are declared in this schema, the wrapper elements will belong to the service's namespace, as required by criterion WS 3.1.3.1 (h).

The definition of the discharge summary is imported from an external XML Schema file. This file was described in section 2.3.1.

```
  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
      xmlns:ds="http://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
      targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
      elementFormDefault="qualified">

      <xsd:import
namespace="http://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
        schemaLocation="DischargeSummary.xsd"/>
```

The request and response elements for the ping operation are defined below. Since this operation takes no parameters and returns no results, both of these elements have an empty content model and no attributes.

To conform to criterion WS 5.1.4.1 (a), this WSDL follows the wrapped convention. Thus, for all operations in this WSDL, the request element's name matches the operation's name, and the response element's name is the operation's name with a "Response" suffix.

```
      <xsd:element name="ping">
        <xsd:complexType/>
      </xsd:element>

      <xsd:element name="pingResponse">
        <xsd:complexType/>
      </xsd:element>
```

The request and response elements for the send discharge summary operation are defined below.

The request is an element that contains the discharge summary document.

Although the send discharge summary operation requires no business-level response, it has a response element, which has an empty content model and no attributes. This operation is modelled as a request-response operation at the technical level to satisfy criterion WS 5.1.5.1 (a).

```
      <xsd:element name="sendDischargeSummary">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="document" type="ds:DischargeSummaryType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="sendDischargeSummaryResponse">
        <xsd:complexType/>
      </xsd:element>
```

The request and response elements for the check status operation are defined below.

```
      <xsd:element name="checkStatus">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="documentId" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="checkStatusResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="response" type="tns:ReceivedStatusType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

The following simple type defines the enumerated set of status values that could be returned by the check status operation.

```
      <xsd:simpleType name="ReceivedStatusType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="NotReceived"/>
          <xsd:enumeration value="PendingAcknowledgement"/>
          <xsd:enumeration value="Acknowledged"/>
        </xsd:restriction>
      </xsd:simpleType>
```

The send discharge summary and check status operations can return a fault. The structure of this fault element is defined below.

```
      <xsd:element name="invalidIdFault">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="faultDescription" type="xsd:string"/>
            <xsd:element name="documentId" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
```

```
</wsdl:types>
```

2.3.2.1.3 Messages

The messages section of the WSDL identifies the messages used by the three operations of the service.

The messages follow the wrapped convention to conform to criterion WS 5.1.4.1 (a). The messages only have one part, where each part references an XML Schema element that was declared in the types section of the WSDL.

```
<wsdl:message name="pingInMsg">
  <wsdl:part name="body" element="tns:ping"/>
</wsdl:message>

<wsdl:message name="pingOutMsg">
  <wsdl:part name="body" element="tns:pingResponse"/>
</wsdl:message>

<wsdl:message name="sendDischargeSummaryInMsg">
  <wsdl:part name="body" element="tns:sendDischargeSummary"/>
</wsdl:message>

<wsdl:message name="sendDischargeSummaryOutMsg">
  <wsdl:part name="body" element="tns:sendDischargeSummaryResponse"/>
</wsdl:message>

<wsdl:message name="checkStatusInMsg">
  <wsdl:part name="body" element="tns:checkStatus"/>
</wsdl:message>

<wsdl:message name="checkStatusOutMsg">
  <wsdl:part name="body" element="tns:checkStatusResponse"/>
</wsdl:message>

<wsdl:message name="invalidIdFaultMsg">
  <wsdl:part name="fault" element="tns:invalidIdFault"/>
</wsdl:message>
```

2.3.2.1.4 Port Type

The portType section of the WSDL defines the three operations in the service. The operation definitions specify the structure of their input, output and fault messages by referencing the message definitions in the WSDL.

Following criterion WS 6.1.2.1 (a), the input, output and fault messages of all operations are assigned WS-Addressing `Action` values. The WS-Addressing `Action` attributes belong to the namespace of *WS-Addressing 1.0 - Metadata* [WSAM2007]. The values used for the WS-Addressing `Action` conform to the scheme set out in the following criteria: WS 3.1.3.1 (e), WS 3.1.3.1 (f) and WS 3.1.3.1 (g).

```
<wsdl:portType name="DischargeSummaryReceiver">

  <wsdl:operation name="ping">
    <wsdl:input message="tns:pingInMsg"
      wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/pingRequest"/>
    <wsdl:output message="tns:pingOutMsg"
      wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/pingResponse"/>
  </wsdl:operation>

  <wsdl:operation name="sendDischargeSummary">
    <wsdl:input message="tns:sendDischargeSummaryInMsg"
      wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/sendDischargeSummaryRequest"/>
    <wsdl:output message="tns:sendDischargeSummaryOutMsg"
      wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/sendDischargeSummaryResponse"/>
    <wsdl:fault name="invalidIdFault" message="tns:invalidIdFaultMsg"
      wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/sendDischargeSummary/Fault/invalidIdFault"/>
  </wsdl:operation>

  <wsdl:operation name="checkStatus">
```

```
      <wsdl:input message="tns:checkStatusInMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/checkStatusRequest"/>
      <wsdl:output message="tns:checkStatusOutMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/checkStatusResponse"/>
      <wsdl:fault name="invalidIdFault" message="tns:invalidIdFaultMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/checkStatus/Fault/InvalidIdFault"/>
    </wsdl:operation>

  </wsdl:portType>

</wsdl:definitions>
```

2.3.3 Non-functional attributes

The WSDL file can contain the non-functional attributes of a service in a formal specification. A WSDL file is not the complete documentation of a service. Some non-functional attributes of a service cannot be described formally and require additional documentation.

2.3.3.1 WSDL containing service instance information

The non-functional attributes of a service are placed in a separate file as recommended by criterion WS 3.1.2.1 (a). This second WSDL file contains the service instance information. It specifies the concrete aspects of the service interface, such as how data is transported and how it is secured.

This second WSDL will be stored in a file called *DischargeSummaryReceiver.wsdl*.

2.3.3.1.1 Header

In the second WSDL file, the start tag of the root element again contains all the XML namespaces that this document will use.

Criterion WS 5.1.1.1 (a) recommends the use of SOAP 1.2 as the messaging protocol. This is specified in the WSDL by using the XML namespace for the SOAP 1.2 binding, which is namely:

`http://schemas.xmlsoap.org/wsdl/soap12/`

Following criterion WS 3.2.1.1 (a), the WSDL uses the *WS-Policy 1.5 Framework* [WSPL2007] to define the non-functional attributes of the service that can be specified formally. The `wsp` prefix is used to refer to the namespace of *WS-Policy 1.5*, namely:

`http://www.w3.org/ns/ws-policy`

The security policies of the service are specified using *WS-SecurityPolicy 1.2* [WSSPL2007], which follows criterion WS 3.2.2.1 (a). The `sp` prefix is used to refer to the namespace of *WS-SecurityPolicy 1.2*, namely:

`http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702`

The addressing policies of the service are specified using *WS Addressing 1.0 - Metadata* [WSAM2007], which follows criterion WS 3.2.3.1 (a). The `wsam` prefix is used to refer to the namespace of *WS-Addressing 1.0 - Metadata*, namely:

`http://www.w3.org/2007/05/addressing/metadata`

This second WSDL should have a target namespace that matches the service namespace, namely:

`http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0`

The `tns` prefix is again used to refer to the target namespace.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:tns="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
name="DischargeSummaryReceiver">
```

2.3.3.1.2 Addressing Policy

The addressing requirements of the service are declared using *WS-Addressing 1.0 - Metadata* [WSAM2007]. The addressing assertions are specified within a policy, defined using the *WS-Policy 1.5* [WSPL2007] policy language.

The service's addressing policies are specified in a policy called *AddressingPolicy*. The name *AddressingPolicy* is arbitrary - any unique name could have been used.

The `Addressing` assertion element indicates the use of WS-Addressing [WSAM2007], as per criterion WS 6.1.1.1 (a).

```
<wsp:Policy xml:id="AddressingPolicy">
  <wsam:Addressing/>
</wsp:Policy>
```

2.3.3.1.3 Security Policy

The security requirements of the service are specified within a policy, defined using the *WS-Policy 1.5* [WSPL2007] policy language. The *WS-SecurityPolicy 1.2* [WSSPL2007] is used to formally describe the security requirements.

The service's security policies are specified in a policy called *SecurityPolicy*.

```
<wsp:Policy xml:id="SecurityPolicy">
```

The `Wss11` assertion element declares the use of *WS-Security 1.1* as per criterion WS 10.2.1.1 (a).

```
  <sp:Wss11>
    <wsp:Policy>
```

The `MustSupportRefKeyIdentifier` element requires service providers and service clients to support key identifier references. This is needed because some certificates in the request and response messages have to be referenced using their subject key identifiers to conform to criteria: WS 10.2.7.1 (b), WS 10.2.7.2 (a) and WS 10.2.7.2 (b).

```
      <sp:MustSupportRefKeyIdentifier/>
    </wsp:Policy>
  </sp:Wss11>
```

Criterion WS 10.2.3.1 (a) requires that the SOAP body, the SOAP headers and the WS-Security timestamp must be signed. The digital signature requirements of this criterion are met by providing the `SignedParts` assertion element in the policy.

The `SignedParts` element is used to indicate which parts of the SOAP messages are to be signed. When this element has no children, it specifies that the body and all headers targeted to the Ultimate Receiver role are to be signed [WSSPL2007].

An explicit assertion is not needed for signing the timestamp because if the timestamp is included, it implies that it must be signed [WSSPL2007].

```
  <sp:SignedParts/>
```

Criterion WS 10.2.4.2 (a) requires that the entire SOAP body is encrypted, while criterion WS 10.2.4.2 (c) recommends that no SOAP headers are to be encrypted. These encryption requirements are met by providing the `EncryptedParts` assertion element in the policy.

The `EncryptedParts` element indicates which parts of a message should be encrypted. When this element has no children, it specifies that the body is to be encrypted. By default, SOAP headers are excluded from encryption.

```
    <sp:EncryptedParts/>
```

The `AsymmetricBinding` element declares the use of public key cryptography as per criterion WS 10.1.1.1 (a). It permits a child policy within itself to configure additional properties of the public key cryptography.

```
    <sp:AsymmetricBinding>
      <wsp:Policy>
```

Criterion WS 10.2.2.2 (a) requires that SOAP messages provide a Created timestamp in the WS-Security Timestamp element. The closest translation of this criterion is the `IncludeTimestamp` assertion, which requires the WS-Security Timestamp be provided in SOAP messages [WSSPL2007]. The Created element is optional within the WS-Security Timestamp element [WSS2006].

```
        <sp:IncludeTimestamp/>
```

Criterion WS 10.2.3.1 (b) stipulates that the digital signatures must be calculated over the entire element (i.e. including the start and end tags of the element). This behaviour is declared by the `OnlySignEntireHeadersAndBody` assertion element.

```
        <sp:OnlySignEntireHeadersAndBody/>
```

The `EncryptSignature` assertion element declares that the digital signatures in messages must be encrypted, which matches criterion WS 10.2.4.2 (b).

```
        <sp:EncryptSignature/>
```

Criterion WS 10.2.5.2 (b) requires the use of the Lax "Security Header Layout" property of *WS-SecurityPolicy* [WSSPL2007].

```
        <sp:Layout>
          <wsp:Policy>
            <sp:Lax/>
          </wsp:Policy>
        </sp:Layout>
```

An explicit assertion element is not required to declare the behaviour of signing before encrypting as per criterion WS 10.2.5.1 (a). The default value of the "Protection Order" property of *WS-SecurityPolicy* [WSSPL2007] is "SignBeforeEncrypting".

The `Basic256Rsa15` assertion element declares the use of the *Basic256Rsa15* algorithm suite, as required by criterion WS 10.2.6.1 (a).

```
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256Rsa15/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
```

The `InitiatorToken` assertion element provides configuration properties for the service invoker's certificate.

In *WS-SecurityPolicy 1.2* [WSSPL2007], the initiator role belongs to the entity who sends the initial message and the recipient role belongs to the entity that is the target of the initial message. Using the terminology of this document, the service invoker is the initiator, and the service provider is the recipient.

Specifying an `AlwaysToRecipient` inclusion policy in the `IncludeToken` attribute on the `X509Token` element will meet criteria: WS 10.2.7.1 (a) and WS 10.2.7.2 (b). This token inclusion policy means that the service invoker's certificate is included in the messages to the service provider, which are namely the SOAP requests. In the other messages, namely the SOAP

responses, the service invoker's certificate is not in the message itself, and is referenced indirectly.

The `WssX509V3Token10` element indicates that an X.509v3 token should be used for the service invoker's certificate, which follows criterion WS 10.1.2.1 (a).

The `RequireKeyIdentifierReference` element specifies a key identifier reference should be used when the token is indirectly referenced. For an X.509v3 certificate, the key identifier reference is the subject key identifier. This assertion element is needed because criterion WS 10.2.7.2 (b) requires that the subject key identifier be used to reference the service invoker's certificate in responses.

```
        <sp:InitiatorToken>
          <wsp:Policy>
            <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/
ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient">
              <wsp:Policy>
                <sp:RequireKeyIdentifierReference/>
                <sp:WssX509V3Token10/>
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:InitiatorToken>
```

The `RecipientToken` assertion element is similar to the `InitiatorToken`, except that it applies to the security token of the service provider.

In the WSDL, the `RecipientToken` element has almost the same content as that of the `InitiatorToken` element, except that a *Never* inclusion policy is set on the `IncludeToken` attribute on the `X509Token` element. This token inclusion policy will meet criteria: WS 10.2.7.1 (b) and WS 10.2.7.2 (a). The service provider's certificate is never included in requests and responses, and is indirectly referenced.

```
        <sp:RecipientToken>
          <wsp:Policy>
            <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/
ws-securitypolicy/200702/IncludeToken/Never">
              <wsp:Policy>
                <sp:RequireKeyIdentifierReference/>
                <sp:WssX509V3Token10/>
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:RecipientToken>
      </wsp:Policy>
    </sp:AsymmetricBinding>
  </wsp:Policy>
```

2.3.3.1.4 Import

Since the second WSDL file refers to the port type defined in the WSDL containing the service interface information, it must import the first WSDL file.

```
  <wsdl:import location="DischargeSummaryReceiverInterface.wsdl"
    namespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"/>
```

2.3.3.1.5 Binding

The binding section indicates the message format and protocol details for the abstract port types.

```
  <wsdl:binding name="DischargeSummaryReceiverBinding"
    type="tns:DischargeSummaryReceiver">
```

The addressing and security policies that were defined in sections 2.3.3.1.2 and 2.3.3.1.3 have to be applied to the service. This is done using `wsp:PolicyReference` elements. The `wsp:PolicyReference` elements are

applied to the WSDL `binding` element as recommended by criterion WS 3.2.1.1 (b).

```
    <wsp:PolicyReference URI="#AddressingPolicy"/>
    <wsp:PolicyReference URI="#SecurityPolicy"/>
```

Criterion WS 4.1.1.1 (a) recommends the use of HTTP 1.1 as the transport protocol. Although the particular HTTP version cannot be specified in the WSDL, the use of HTTP as the transport protocol can be specified by setting the `transport` attribute of the SOAP `binding` element to:

`http://schemas.xmlsoap.org/soap/http`

To comply with criterion WS 5.1.2.1 (a), the *document/literal* style is used. This is done by setting the `style` attributes of SOAP operation elements to `document` and the `use` attributes of SOAP body and fault elements to `literal`.

To comply with criterion WS 5.1.3.1 (a), `soapAction` values are not assigned to any operation. The `soapActionRequired` attributes are set to false to indicate that the service does not need `soapAction` values in the requests.

```
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="Ping">
      <soap:operation style="document" soapActionRequired="false"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="SendDischargeSummary">
      <soap:operation style="document" soapActionRequired="false"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="invalidIdFault">
        <soap:fault name="invalidIdFault" use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="CheckStatus">
      <soap:operation style="document" soapActionRequired="false"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="invalidIdFault">
        <soap:fault name="invalidIdFault" use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

  </wsdl:binding>
```

2.3.3.1.6 Service

The service part of the WSDL defines a service with concrete ports that are associated with a particular binding.

An address must be provided for the Web service instance. However, it is not necessary to provide an actual hard-coded URL. This address value can be overridden by the toolkit.

```
  <wsdl:service name="DischargeSummaryReceiverService">
    <wsdl:port name="DischargeSummaryReceiver"
      binding="tns:DischargeSummaryReceiverBinding">
      <soap:address location="http://dummy.example.com"/>
    </wsdl:port>
```

```
  </wsdl:service>

</wsdl:definitions>
```

3 Overview of WCF

3.1 General background

Windows Communication Foundation (WCF) [WCF] is a programming framework that is used for building service-oriented applications that inter-communicate.

WCF is a part of .NET 3.0 which is included with the Windows Vista operating system, but is also separately available for Windows XP and Windows Server 2003.

3.2 Technical Overview

One of the main features of WCF is that it has a single service-oriented programming model to unify a number of different distributed communication technologies (such as Web services, .NET Remoting and Microsoft Message Queuing). This allows developers to use a single coherent programming model when developing clients and services that need to communicate. Some of the benefits of the unification include shorter development time, more concise code, and adherence to more modern standards.

In the WCF model, a service exposes one or more endpoints for accessing the service. An endpoint is defined by an address, a binding and a contract. The address specifies the location of the endpoint, e.g. a network address. The binding specifies how a client can communicate with the service, such as what transport protocol to use and how messages should be encoded. The contract defines the operations that a client can invoke.

WCF provides the *svcutil* utility which is used for creating classes from WSDL files. The utility creates classes for the client that can be used for invoking methods on the service. It also can be used to generate a service interface, which can be used as the basis of a service implementation.

WCF supports a range of Web services standards, including WS-Addressing, WS-Security, and SOAP 1.2. This document will explain how to configure WCF Web service and Web service clients to adhere to *Web Services Profiles* [ATS 5820—2010].

3.3 Requirements

The following section details what software is required to build and deploy WCF applications and services.

3.3.1 Client deployment

To run a WCF Web service client, the following software is needed:

- Microsoft Windows Vista, Microsoft Windows XP SP2, Windows Server 2003 R2 or Windows Server 2003 SP1 – Operating system for development and deployment;
- Microsoft .NET Framework 3.0 Service Pack 1 Redistributable Package – Components required to run .NET 3.0 applications.

3.3.2 Web service deployment

To run a WCF Web service, the client run-time requirements (section 3.3.1) plus the following are needed:

- Internet Information Services (IIS) 5.0, 5.1, or 6.0 – Set of Internet based services including a Web server that a Web service can be

deployed on, allowing it to be accessible from a network. However, there are other ways to deploy a WCF Web service that does not require IIS. These are not covered in this document.

- A tool to change the access privileges of the private keys and certificates in the Windows Certificate Store. For example, Windows HTTP Server Certificate Configuration Tool (*WinHTTPCertCfg*).

3.3.3 Client development

To develop WCF Web service client applications, the client deployment requirements (section 3.3.1) plus the following are needed:

- Microsoft Windows Software Development Kit for Windows Vista and the .NET Framework 3.0 Runtime Components – Software development kit for .NET applications that can be used on Vista and Windows XP. The SDK allows applications that use WCF to be created.
- A development environment for .NET programs. For example, Microsoft Visual Studio 2005, Microsoft Visual C# Express – Environment such as an IDE for creating .NET applications.

When using Visual Studio 2005, the Visual Studio 2005 Extensions for .NET 3.0 can be used. This adds a number of features for creating WCF applications such as a graphical configuration editor and the WCF service project template.

3.3.4 Web service development

The requirements for developing a Web service are the same as that for developing a Web service client (section 3.3.3).

3.4 Platform used

The code and configuration fragments in this document were tested using the following software configurations:

3.4.1 .NET 3.0 Service Pack 1

Microsoft .NET 3.0 included the first implementation of the WCF toolkit. This version is suitable for simpler Web services. There are some limitations with this version, specifically with respect to the faults. Faults must comply with the Data Contract specification [MS7331123.0] or the fault members will not be generated.

Development with the WCF toolkit for .NET 3.0 requires the following products:

- Microsoft Windows XP SP2
- Microsoft Visual Studio 2005
- Microsoft Visual Studio 2005 Extensions for .NET 3.0
- Microsoft Windows SDK for Vista and .NET Framework 3.0 Runtime Components
- Microsoft .NET 3.0 Framework Service Pack 1

3.4.2 .NET 3.5 Service Pack 1

This version of the toolkit from Microsoft includes one significant improvement over the previous version. .NET 3.5 now allows the use of non-data contract compatible fault types via SvcUtil's /useSerializerForFaults command line

parameter. In conjunction with the /serializer:XmlSerializer parameter, SvcUtil now allows vastly more complex schemas than would otherwise be possible using the Data Contract specification [MS7331123.5].

These command line parameters are not required for the examples in this document but may be required for future versions of the schema and will be included in release notes and this example implementation document.

It is recommended that consideration be given to future requirements when choosing the platform for implementation as changes to the schema may dictate that .NET 3.0 may not be supported in the future.

Development with the WCF toolkit for .NET 3.5 requires the following products:

- Microsoft Windows XP SP3
- Microsoft Visual Studio 2008 Service Pack 1
- Microsoft SDK for .NET Framework 3.0 Runtime Components
- Microsoft .NET 3.5 Framework Service Pack 1

4 Web service client

This chapter describes how to build a WCF Web service client that conforms to the *Web Services Profiles* [ATS 5820—2010]. The aim of a Web service client is to invoke an operation on a service instance.

A WCF client can be implemented using the following steps:

1. Modify the WSDL files;
2. Generate the proxy from the WSDL files;
3. Create a client program that implements the WCF proxies;
4. Edit the client configuration; and
5. Implement the client application.

4.1 Modify the WSDL files

4.1.1 WSDL containing the service interface information

WCF uses the *WS-Addressing 1.0 - WSDL Binding* [WSAW2006] to specify the addressing information in WSDL files. This W3C specification did not reach the final Recommendation stage, and has been superseded by *WS-Addressing 1.0 - Metadata* [WSAM2007]. Since WCF implements the older specification, it only recognises the XML namespace of the older specification, namely `http://www.w3.org/2006/05/addressing/wsdl`.

In the WSDL file, the XML namespace for the *WS-Addressing 1.0 - WSDL Binding* [WSAW2006] must be declared in the root element as shown below. The XML namespace for *WS-Addressing 1.0 - Metadata* [WSAM2007] is removed.

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  ...>
  <!--
  Removed the following namespace declaration:
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  -->
  ...
```

The convention of the older specification is to use the `wsaw` namespace prefix. The WSDL file has been changed to use this prefix to follow convention. Note that from a technical perspective it is not necessary to change from the `wsam` namespace prefix.

In *WS-Addressing 1.0 - WSDL Binding* [WSAW2006], the WS-Addressing action values are also specified in the WSDL using `Action` attributes. No change is needed apart from using the `wsaw` namespace prefix to associate the `Action` attributes with the XML namespace of the older specification.

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ...>
  ...
  <wsdl:portType name="DischargeSummaryReceiver">
    <wsdl:operation name="ping">
      <wsdl:input message="tns:pingInMsg"
        wsaw:Action="http://..."/>
      <wsdl:output message="tns:pingOutMsg"
        wsaw:Action="http://..."/>
    </wsdl:operation>
    ...
  </wsdl:portType>
</wsdl:definitions>
```

4.1.2 WSDL containing the service instance information

4.1.2.1 WS-Policy

WCF does not support WS-Policy 1.5 [WSPL2007], so all policy details should be removed from the WSDL containing the service instance information.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  …
  <!--
  Removed the following namespace declarations:
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
  -->
  >
  ...
  <!--
  Removed the WS-Policy elements
  <wsp:Policy wsu:Id="AddressingPolicy">
    <wsam:Addressing/>
  </wsp:Policy>
  <wsp:Policy wsu:Id="SecurityPolicy">
    …
  </wsp:Policy>
  -->
  …
</wsdl:definitions>
```

4.2 Generate the proxy from the WSDL files

The proxy is used within a client program to invoke operations on a remote Web service. The proxy takes care of: serialising the operation calls to SOAP requests that are sent to the Web service, and deserialising the SOAP responses from the Web service. This means SOAP messages do not have to be created manually.

To generate a WCF proxy from the WSDL files, use the *ServiceModel Metadata Utility Tool*, *svcutil*, which is included with the *Microsoft Windows SDK for Windows Vista and .NET Framework 3.0 Runtime Components*. The tool can be found in the *<installation directory>\Microsoft SDKs\Windows\v6.0\Bin* directory.

The following call to *svcutil* generates the proxy from the WSDL files:

```
svcutil DischargeSummaryReceiver.wsdl
  DischargeSummaryReceiverInterface.wsdl
  DischargeSummary.xsd
  /serializer:auto
  /namespace:*,Nehta.Example.DS.Client
  /out:DischargeSummaryReceiver.cs
  /config:app.config
```

The first three parameters specify the WSDL and XML Schema files for the Web service. It should be noted that all imported files need to be listed on the command line, not just the root WSDL file containing the service declaration.

The other named parameters are:

- `/serializer`: determines how the types will be serialised. The value of `auto` lets *svcutil* determine which serialisation mechanism to use based on information in the WSDL.
- `/namespace`: specifies what namespace the generated source file will be in. The generated class will be enclosed within the specified namespace.
- `/out`: specifies the name and location of the generated source file. The source file contains the service interface specification and the proxy that

the client will use to send and receive messages from the service. This file should be added to the client project.

- `/config`: specifies the name and location of the generated configuration file.

The *svcutil* tool generates two files:

- A single C# source code file containing the service interface and proxy code. The proxy code includes the data type and fault classes that are specified within the WSDL. The generated source file can be added to the client project without any modifications.
- A configuration file. The SvcUtil tool generates a default configuration file containing many default values for WCF web services. These configuration files by themselves are of limited use and are usually merged into existing application configuration files.

When the generated proxy class is instantiated within the client application to invoke operations on the Web service, it reads the WCF configuration stored within the application configuration file. Although *svcutil* creates a basic configuration file when no WS-Policy information is in the WSDL, that file will not be used. In this example a more sophisticated configuration file will be manually created and edited using the *Microsoft Service Configuration Editor* (as described in section 4.4).

The client can also be configured programmatically but this approach has not been used for this example.

4.3 Create a client program

In the Visual Studio IDE, create a project for the Web service client. There are a number of different ways a client can be implemented (e.g. as a command-line application or a Windows Forms application). It doesn't matter which is used, because this example is only concerned with the Web services part of the application.

In the new project, add references to the following assemblies:

- `System.ServiceModel`: contains the core WCF classes for implementing a Web service and client.
- `System.Runtime.Serialization`: contains classes used for serialising and deserialising types that are used by the Web service and client.

Add the generated source file (that was created in Section 4.1) to the project using the "Add → Existing Item…" function.

Create a new "Application Configuration File" for the project by using the "Add → New Item…" function. This step will create a default configuration file called *App.config* that contains no WCF configuration information. This empty *App.config* file will be set up in section 4.4 using the *Microsoft Service Configuration Editor*.

4.4 Edit the client configuration

The *App.config* file needs to be edited to configure the Web service client to adhere to the *Web Services Profiles* [ATS 5820—2010]. Since it is an XML file, any XML editor or text editor can be used to edit it. However, the easiest way to edit the *App.config* file is using the graphical *Microsoft Service Configuration Editor* tool. This document will describe configuring WCF using this tool. The resulting configuration XML will also be shown.

The following configurations are needed:

1. Create a custom binding;
2. Create an endpoint behaviour; and

3. Create an endpoint.

4.4.1 Create a custom binding

A binding specifies how the client communicates with the Web service. Namely, what transport protocol to use, how messages should be encoded and what protocols and security mechanisms should be used.

WCF provides a number of built-in bindings. However, none of those bindings are conformant to the *Web Services Profiles* [ATS 5820—2010]. The closest applicable binding is `wsHttpBinding`, but it is not suitable. For example, the "SecurityHeaderLayout" property is always "Strict" and it cannot be set to a different value. To conform it needs be set to "Lax". Therefore, a custom binding is needed.

4.4.1.1 Create the binding

To create a custom binding:

1. Right-click on the *App.config* file in Visual Studio and then select the "Edit WCF Configuration…" menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Right-click on the "Bindings" folder in the "Configuration" tree pane and select the "New Binding Configuration…" menu item.
3. In the "Create a New Binding" dialog box that appears, select "customBinding" and click the "OK" button.
4. Enter a name for the new binding in the "Name" field. (In this example, the arbitrary name of "SecureBinding" is used.)

The newly-created binding will automatically have two binding element extensions, which don't need to be altered. The "httpTransport" extension specifies that HTTP is to be used as the transport protocol, which is aligned with criterion WS 4.1.1.1 (a). The "textMessageEncoding" extension specifies how messages are encoded. By default, it's set to use SOAP 1.2 and WS-Addressing 1.0, which is aligned with criteria WS 5.1.1.1 (a) and WS 6.1.1.1 (a).

4.4.1.2 Configure security

A new custom binding doesn't include security by default, this must be configured using the *Microsoft Service Configuration Editor* tool.

To add and configure security to the custom binding:

1. Click on the new custom binding (in this example, "SecureBinding") in the "Configuration" tree pane.
2. In the "Binding element extension position" section, click the "Add…" button.
3. In the "Adding Binding Element Extension Sections" dialog box that appears, select "security" and click the "Add" button.
4. Double-click on the "security" element in the "Binding element extension position" to edit the security values.
5. Set the "AllowSerializedSigningTokenOnReply" property to true. As the name suggests, this property specifies whether the token used for signing response messages can be serialised. This feature should be enabled to improve interoperability with other non-WCF toolkits.
6. Set the "AuthenticationMode" property to "MutualCertificate". This specifies that both the client and service are authenticated using X.509 certificates.

7. Set the "DefaultAlgorithmSuite" property to "Basic256Rsa15". This is in line with criterion WS 10.2.6.1 (a).
8. Leave the "IncludeTimestamp" property set to true. This is in line with criterion WS 10.2.2.2 (a). By default, WCF includes the `Created` and `Expired` elements within the timestamp.
9. Set the "MessageProtectionOrder" property to "SignBeforeEncryptAndEncryptSignature". This follows criterion WS 10.2.5.1 (a), which require SOAP messages to be signed before they are encrypted. It also follows criteria WS 10.2.4.2 (a), WS 10.2.4.2 (a), for encrypting the body and signature, and WS 10.2.3.1 (a) for signing the body and headers.
10. Set the "MessageSecurityVersion" property to "WSSecurity10WSTrustFebruary2005WSSecureConversationFebruary2005WSSecurityPolicy11BasicSecurityProfile10". This is in line with criteria WS 10.2.1.1 (a). Although the *Web Services Profiles* [ATS 5820—2010] recommends the use of WS-Security 1.1, choosing either of the two options that specify WS-Security 1.1 will cause WCF to sign the signature block. Since this feature is not a requirement of WS-Security 1.1 and not part of the criteria, it is not desirable behaviour because it will create interoperability problems with other toolkits. The option specifying WS-Security 1.0, which is given above, will not sign the signature block and should still be compliant with WS-Security 1.1.
11. Set the "RequireDerivedKeys" property to false. This property is related to WS-SecureConversation, which isn't part of the *Web Services Profiles* [ATS 5820—2010]. This property must be set to false; otherwise, there may be interoperability problems. If it is set to true, the client will generate a `DerivedKeyToken` element in the SOAP header, which will be rejected by any Web service that does not support it.
12. Set the "RequireSecurityContextCancellation" property to false. This property is also related to WS-SecureConversation. Although it is unlikely to have an effect when the client and Web service don't use WS-SecureConversation, it should be set to false.
13. Leave the "RequireSignatureConfirmation" property set to false as it is not required.
14. Set the "SecurityHeaderLayout" property to "Lax". The "Lax" policy is compliant with criterion WS 10.2.5.2 (b).
15. Leave the rest of the properties set to their default values.

4.4.1.3 Save changes

Save the changes that were made to *App.config* file by the *Microsoft Service Configuration Editor*.

The following XML fragment will be added to `bindings` element within the `system.serviceModel` element in the *App.config* file:

```
<customBinding>
  <binding name="SecureBinding">
    <textMessageEncoding />
    <security
        defaultAlgorithmSuite="Basic256Rsa15"
        allowSerializedSigningTokenOnReply="true"
        authenticationMode="MutualCertificate"
        requireDerivedKeys="false"
        securityHeaderLayout="Lax"
messageSecurityVersion="WSSecurity10WSTrustFebruary2005WSSecureConversationFebruar
y2005WSSecurityPolicy11BasicSecurityProfile10"
        requireSecurityContextCancellation="false">
    </security>
    <httpTransport/>
  </binding>
</customBinding>
```

4.4.2 Create endpoint behaviour

4.4.2.1 Create the behaviour

To create an endpoint behaviour:

1. Right-click on the *App.config* file in Visual Studio and then select the "Edit WCF Configuration…" menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the "Advanced" folder in the "Configuration" tree pane.
3. Right-click on the "Endpoint Behaviors" folder in the "Configuration" tree pane and select the "New Endpoint Behavior Configuration…" menu item.
4. Enter a name for the new endpoint behaviour in the "Name" field. (In this example, the arbitrary name of "SecureBehavior" is used.)

4.4.2.2 Set the client and service certificates

WCF uses the Windows certificate store to access the keys and certificates used to sign and encrypt the SOAP messages. The required security tokens (i.e. the key and certificates) must be installed into the Windows certificate store before the application can be run. Security token installation is covered in Appendix C.

The Web service client determines which key and certificate to use from the values configured in the endpoint behaviour.

To set the client security token in the endpoint behaviour:

1. Click on the new endpoint behaviour (in this example, it is `SecureBehavior`) in the "Configuration" tree pane.
2. In the "Behavior element extension position" section, click the "Add…" button.
3. In the "Adding Behavior Element Extension Sections" dialog box that appears, select "clientCredentials" and click the "Add" button.
4. Double-click on the "clientCredentials" element in the "Behavior element extension position".
5. Expand the "clientCredentials" element in the "Configuration" tree pane.
6. Select the "clientCertificate" element in the "Configuration" tree pane.
7. Leave the "StoreLocation" property value set to "CurrentUser". This value refers to the current user's certificate store, which is meant to store certificates for client applications.
8. Leave the "StoreName" property value to set to "My". This value refers to the "Personal" folder in the certificate store, which is meant to store certificates with private keys.
9. Leave the "X509FindType" property value set to "FindBySubjectDistinguishedName". The client certificate will then be searched using the distinguished name in the "Subject" field of the X.509 certificate.
10. Set the "FindValue" property to the Subject's distinguished name of the client certificate to be selected (in this example, it is `cn=wcf_client`). There must be a private key associated with this certificate in the certificate store.

The default behaviour of WCF when sending requests from a client is to include the corresponding certificate of the private key that was used for signing within the message. This conforms to criteria WS 10.2.7.1 (a).

To set the service security token in the endpoint behaviour:

1. Expand the "serviceCertificate" element in the "Configuration" tree pane.
2. Select the "defaultCertificate" element in the "Configuration" tree pane.
3. Leave the "StoreLocation" property value set to "CurrentUser".
4. Set the "StoreName" property value to "AddressBook". This value refers to the "Other People" folder in the certificate store, which is meant to store certificates of other entities.
5. Leave the "X509FindType" property value as "FindBySubjectDistinguishedName". The client certificate will then be searched using the distinguished name in the "Subject" field of the X.509 certificate.
6. Set the "FindValue" property to the Subject's distinguished name of the service certificate to be selected (in this example, it is `cn=wcf_server`).
7. Select the "authentication" element under the "serviceCertificate" element in the "Configuration" tree pane.
8. Set the "RevocationMode" property to "NoCheck". When using test certificates like in the example Web service, certificate revocation checking should be turned off since it can be a reason for security failures. This behaviour is disabled only for testing purposes and would not be desirable in production systems.

The default behaviour of WCF when sending requests from a client is to include the subject key identifier of the certificate used to encrypt within the message. This conforms to criterion WS 10.2.7.1 (b).

4.4.2.3 Save changes

Save the changes that were made to the *App.config* file. The following XML fragment will be added to the `behaviors` element within the `service.serviceModel` element in the *App.config* file:

```
<endpointBehaviors>
  <behavior name="SecureBehavior">
    <clientCredentials>
      <clientCertificate findValue="cn=wcf_client" />
      <serviceCertificate>
        <defaultCertificate findValue="cn=wcf_server" />
        <authentication revocationMode="NoCheck" />
      </serviceCertificate>
    </clientCredentials>
  </behavior>
</endpointBehaviors>
```

4.4.3 Create an endpoint

To access a Web service, an endpoint for the Web service must be created in the client configuration.

An endpoint must have a binding, which specifies how to communicate with the Web service. Section 4.4.1 explained how to create a custom binding that conforms to *Web Services Profiles* [ATS 5820—2010]. The created endpoint will have to be associated with this binding.

An endpoint must also have a contract, which specifies what operations the client can call. The created endpoint will have to be associated with the service interface, which is in the source file generated in Section 4.1.

An endpoint must have an address, which specifies the location where the Web service can be accessed. This address can be a logical or physical address. This example uses the physical address of the Web service.

An endpoint can also have an endpoint behaviour, which allows modification of the client run-time behaviour. The created endpoint will have to be associated with the behaviour created in Section 4.4.2. This behaviour

specifies what security tokens are to be used when securing the message with WS-Security.

To create the endpoint:

1. Right-click on the *App.config* file in Visual Studio and then select the "Edit WCF Configuration…" menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the "Clients" folder in the "Configuration" tree pane.
3. Right-click on the "Endpoints" folder in the "Configuration" tree pane and select the "New Client Endpoint" menu item.
4. Enter a name for the new endpoint in the "Name" property. (In this example, it is `ServiceEndpoint`.)
5. Enter the service's physical address (in this example, it is `http://localhost:3916/Server/Service.svc`) in the "Address" property. This value will be used in the WS-Addressing `To` headers of SOAP messages which is in line with criterion WS 6.1.4.1 (a).
6. Select the endpoint behaviour that was created in Section 4.4.2 (in this example, it is `SecureBehavior`) in the "BehaviorConfiguration" property.
7. Select "customBinding" in the "Binding" property and select the binding that was created in Section 4.4.1 (in this example, it is `SecureBinding`) in the "BindingConfiguration" property.
8. Type in the full type name of the service interface for the Web service in the "Contract" property. This can be found by looking in the generated source file that was added to the project. The full type name is found by combining the namespace with the service interface name. The code below shows a fragment of the example service instance. In this case, the full type name of the Web service interface is: `Nehta.Example.DS.Client.DischargeSummaryReceiver` with `Nehta.Example.DS.Client` being the namespace and `DischargeSummaryReceiver` being the type name of the service interface.

```
namespace Nehta.Example.DS.Client {
  …
  public interface DischargeSummaryReceiver
  {
     …
     Nehta.Example.DS.Client.pingResponse1
ping(Nehta.Example.DS.Client.pingRequest request);
     …
  }
}
```

4.4.3.1 Set the identity

The service's security identity needs to be set, because WCF uses this identity to authenticate the service being connected to.

To set the identity:

1. Click on the new endpoint (in this example, it is "ServiceEndpoint") in the "Configuration" tree pane.
2. Click on the "Identity" tab.
3. Enter the common name of the service certificate's "Subject" field in the "Dns" property (in this example, it is `wcf_server`).

4.4.3.2 Save changes

Save the changes that were made to the *App.config* file. The following XML fragment will be added to the `client` element within the `service.serviceModel` element in the *App.config* file:

```
<endpoint address="http://localhost:3916/Server/Service.svc"
     behaviorConfiguration="SecureBehavior"
     binding="customBinding" bindingConfiguration="SecureBinding"
     contract="Nehta.Example.DS.Client.DischargeSummaryReceiver"
     name="ServiceEndpoint">
  <identity>
    <dns value="wcf_server" />
  </identity>
</endpoint>
```

4.5 Implement the client application

To invoke an operation on the service, the client needs to:

1. Instantiate the proxy object;
2. Invoke the Web service methods; and
3. Close the proxy object.

4.5.1 Instantiate the proxy object

The proxy is instantiated using `new`, just like any normal object. This is shown in the code fragment below.

The proxy will load the WCF configuration from the *App.config* file.

```
DischargeSummaryReceiverClient dsrc = new DischargeSummaryReceiverClient();
```

4.5.2 Invoke the Web service methods

The client invokes the Web service operations by calling the corresponding methods on the proxy object.

4.5.2.1 Invoking ping

This operation is the simplest to invoke because it requires no parameters and returns no results. Simply create a new `ping` object (to represent the empty set of parameters) and call the `ping` method on the client proxy object.

```
try
{
  dsrc.ping(new ping());
}
catch (…) {
  …
}
```

4.5.2.2 Send discharge summary

This operation illustrates a Web service operation that expects some parameters.

Invoking the send discharge summary operation involves:

1. Creating a `DischargeSummaryType` object to represent the discharge summary document;
2. Creating a `sendDischargeSummary` object to contain the parameters (i.e. the discharge summary and an identifier for it) for the operation; and
3. Calling the `sendDischargeSummary` method on the client proxy object.

```
// Create the discharge summary document itself
DischargeSummaryType ds = new DischargeSummaryType();
ds.notes = "This is an example discharge summary";
ds.documentId = "someid"; // a unique ID for the discharge summary

// Create and set parameters
sendDischargeSummary sds = new sendDischargeSummary();
sds.document = ds;
```

```
// Invoke the operation using the client stub "dsrc"
try
{
  dsrc.sendDischargeSummary(sds);
}
catch (…) {
  …
}
```

4.5.2.3 Check status

This operation illustrates an operation that returns a response. The value of the response is obtained from the return value of the method.

Invoking the check status operation is similar to the other operations, except that this operation returns a result. This involves:

1. Creating the `checkStatus` object to contain the parameters for the operation;
2. Calling the `checkStatus` method on the client proxy object and storing the response in the `checkStatusResponse` object; and
3. Processing the response object.

```
// Create and set parameters
checkStatus cs = new checkStatus();
cs.documentId = "someid"; // the unique ID of the discharge summary being queried

try {
  checkStatusResponse csr = dsrc.checkStatus(cs);
  Debug.WriteLine("CheckStatus: id=" cs.documentId + ", status=" + csr.response);
} catch (…) {
  …
}
```

4.5.3 Close the proxy object

When the proxy object is no longer needed, it should be closed. This frees any resources being used by the proxy.

```
dsrc.Close();
```

4.5.4 Handle exceptions and faults

Within the WSDL, each operation can specify what SOAP faults can occur when calling that particular operation. When a client invokes a Web service method and a fault occurs, the corresponding exception can be caught using a standard C# try/catch block. Once caught, any normal error handling can take place.

The WCF documentation also recommends catching the `TimeoutException` and `CommunicationException` exceptions when invoking operations or calling `Close`. It also recommends calling `Abort` within any catch block to ensure that proper shutdown of the proxy can take place.

The typical pattern for handling exceptions is given below:

```
// Create the proxy

try {
  // Calls on the proxy methods
  …

  // Close the proxy
  proxy.Close()
}
catch (TimeoutException ex) {
  …
  proxy.Abort();
}
catch(CommunicationException ex) {
  …
  proxy.Abort();
}
catch (CustomServiceFault ex) {
  …
  proxy.Abort();
}
```

5 Web service

This section describes how to build a WCF Web service that conforms to the *Web Services Profiles* [ATS 5820—2010].

A WCF Web service can be implemented with the following steps:

1. Modify the WSDL files;
2. Generate the service interface from the WSDL files;
3. Create a WCF service project;
4. Implement the Web service;
5. Edit *Service.svc*;
6. Edit the service configuration; and
7. Package and deploy the Web service.

5.1 Modify the WSDL files

The same modifications to the standard WSDL files will have to be made for the server-side as described in section 4.1.

5.2 Generate the service interface from the WSDL files

The interface declaration for the service can be generated from the WSDL files using the *svcutil* command line tool supplied with the *Microsoft Windows SDK for Windows Vista and the .NET Framework 3.0 Runtime Components*.

Section 4.1 describes how to use this tool on the client-side to generate the service interface and proxy code. The steps described in Section 4.1 are similar for the server-side. The only difference is that a different namespace is used for the generated code on the server-side. However, it is also possible to use the same generated code for the both the Web service and client with the same namespace.

Like the client-side, the generated configuration file is ignored. Only the generated source file will be used.

5.3 Create a WCF service project

Create a new Web Site project in the Visual Studio IDE. Ensure that the "WCF Service" template is selected.

There are a number of ways to host a WCF service. The descriptions in this document are in terms of an IIS-hosted WCF service, which is created by selecting the default "HTTP" option in the "Location" field. The project's Solution name should be typed in after the "http://localhost/" string in the "HTTP" location value.

A WCF service can be written in different programming languages. The code examples in this document are in C#, which is the default option for the "Language" field.

Once the new project is created, add the source file that was generated in Section 5.1 to it. Unlike the client, there is no need to add the `System.ServiceModel` and `System.Runtime.Serialization` assemblies since this is automatically done by selecting the "WCF Service" template.

5.4 Implement the Web service

When Visual Studio creates the initial WCF service project, it creates a default service that can be modified. The new WCF service project will come with a *Service.cs* file in the `App_Code` folder that contains a sample Web service implementation. Apart from the `using` statements at the top, the contents of this file can be deleted. This section will explain how to then build this service implementation class.

5.4.1 Create the service implementation class

The example source code below shows the class declaration of the service implementation. It should implement the service interface, which is found in the source file generated in Section 5.1. It should have a "ServiceBehavior" attribute, which specifies the WSDL's target namespace.

```
namespace Nehta.Example.DS.Server {

  [ServiceBehavior(
    Namespace = "http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0")]
  public class DSRService : DischargeSummaryReceiver {
    …
  }

}
```

5.4.2 Implement the service methods

Implementations need to be provided for all the methods in the service interface.

```
    public pingResponse1 ping(pingRequest request) {
      UpdateAddressingHeaders();

      pingResponse1 result = new pingResponse1();
      return result;
    }

    public sendDischargeSummaryResponse1 sendDischargeSummary(
      sendDischargeSummaryRequest request)
    {
      UpdateAddressingHeaders();

      string documentId = request.sendDischargeSummary.document.documentId;

      // Check the ID is valid and contains no invalid characters
      if (!IsIdValid(documentId)) {
        ThrowInvalidIdFault(documentId);
      }

      … // Process request and return response
    }

    public checkStatusResponse1 checkStatus(checkStatusRequest request) {
      UpdateAddressingHeaders();

      string documentId = request.checkStatus.documentId;

      // Check the ID is valid and contains no invalid characters
      if (!IsIdValid(documentId)) {
        ThrowInvalidIdFault(documentId);
      }

      … // Process request and return response
    }
```

5.4.3 Add the WS-Addressing MessageID header

Since WCF by default doesn't add the WS-Addressing `MessageID` header to SOAP responses, it needs to be explicitly set in the current operation context, so as to satisfy criterion WS 6.1.3.1 (a). Since this code will have to be called

in each Web service method implementation, in this example it is placed in a separate method with a meaningful name.

```
    private void UpdateAddressingHeaders() {
      OperationContext.Current.OutgoingMessageHeaders.MessageId = new UniqueId();
    }
```

5.4.4 Throw faults

When the *svcutil* tool generates the service interface, it generates classes for any faults that are specified in the WSDL. These classes are only used to specify the details of the fault and are not thrown directly.

To use the fault in the service implementation, create an instance of the generated class and set the details of the fault. Create a `FaultException` object using the created fault object and specify the reason for the fault by creating a `FaultReason` object. The fault exception can then be thrown using the normal `throw` statement.

```
public static void ThrowInvalidIdFault(string id) {
  // Set the data associated with the fault
  InvalidIdFault iif = new InvalidIdFault();
  iif.FaultDescription = InvalidIdFaultMessage;
  iif.DocumentId = id;

  // Create and throw the fault
  FaultException fe =
    new FaultException<InvalidIdFault>(iif,
    new FaultReason("Reason: " + InvalidIdFaultMessage));

  throw fe;
}
```

5.5 Edit Service.svc

With Visual Studio, another file that gets generated by default when creating a "WCFServiceLibrary" project is the *Service.svc* file. This defines the file that contains the service class and the class itself.

Open *Service.svc* and put the full type name in the `Service` attribute. The full type name includes the namespace and the service type name. Ensure that the `CodeBehind` attribute has the correct name of the source file that contains the service implementation class. By default this is set to *Service.cs* and should only need to be changed if the service source filename has been changed.

The contents of the *Service.svc* file for the example Web service is given below.

```
<% @ServiceHost Language=C# Debug="true"
Service="Nehta.Example.DS.Server.DSRService" CodeBehind="~/App_Code/Service.cs" %>
```

5.6 Edit the service configuration

The new project will have a *Web.config* file, which stores the configuration for the Web service. Since it is an XML file, it can be edited by XML editors or plain text editors. However, the *Microsoft Service Configuration Editor* provides the easiest way to edit this file.

To configure the Web service:

1. Create a custom binding;
2. Create a service behaviour; and
3. Configure the service.

5.6.1 Create a custom binding

A WCF binding specifies how the client and Web service should communicate with each other, namely what transport protocol to use, how messages should be encoded and what protocols, e.g. security, should be used. Thus, the bindings in the client and service configurations should match each other.

Section 4.4.1 describes how to create a custom binding that adheres to *Web Services Profiles* [ATS 5820—2010] on the client-side. Follow the steps in Section 4.4.1 for the server-side as well. The only difference is that the file to be edited is *Web.config*, instead of *App.config*.

5.6.2 Create a service behaviour

5.6.2.1 Create the behaviour

To create the service behaviour:

1. Right-click on the *Web.config* file in Visual Studio and select the "Edit WCF Configuration…" menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the "Advanced" folder in the "Configuration" tree pane.
3. Right-click on the "Service Behaviors" folder in the "Configuration" tree pane and select the "New Service Behavior Configuration…" menu item.
4. Enter a name for the new service behaviour in the "Name" field. (In this example, it is `SecureBehavior`).

5.6.2.2 Set the service certificate

WCF uses the Windows certificate store to access the keys and certificates used to sign and encrypt the SOAP messages. The required security tokens (i.e. the key and certificates) must be installed into the Windows certificate store and have the correct permissions before the service can be run. Security token installation is covered in Appendix C.

The certificate that the Web service will use to decrypt requests and sign responses has to be specified.

To set the service certificate:

1. Click on the new service behaviour (in this example, it is `SecureBehaviour`) in the "Configuration" tree pane.
2. In the "Behavior element extension position" section, click the "Add…" button.
3. In the "Adding Behavior Element Extension Sections" dialog box that appears, select "serviceCredentials" and click the "Add" button.
4. Expand the "SecureBehavior" and then the "serviceCredentials" elements in the "Configuration" tree pane.
5. Select the "serviceCertificate" element in the "Configuration" tree pane.
6. Leave the "StoreLocation" property value set to "LocalMachine". This value refers to the local machine certificate store. This certificate store is only visible to administrators using the MMC snap-in tool (it has the "Local Computer" label in this tool), and it is meant for storing certificates for services.
7. Leave the "StoreName" property value set to "My". This value refers to the "Personal" folder in the certificate store, which is meant to store certificates with private keys.
8. Leave the "X509FindType" property value set to "FindBySubjectDistinguishedName". The service certificate will then be

searched using the distinguished name in the "Subject" field of the X.509 certificate.

9. Set the "FindValue" property to the Subject's distinguished name of the service certificate to be selected (in this example, it is `cn=wcf_server`). There must be a private key associated with this certificate in the certificate store.

The client certificate shouldn't be set in the configuration since setting it will tie the service to a single client.

When using test certificates like in the example Web service, certificate revocation checking for the client certificate should be turned off since it can be a reason for security failures. This behaviour is disabled only for testing purposes and would not be desirable in production systems.

To turn off revocation checking for the client certificate:

1. Select the "clientCertificate" element in the "Configuration" tree pane.
2. Set the "RevocationMode" property to "NoCheck".

The default behaviour of WCF when sending responses to clients is to include the subject key identifier of the certificate corresponding to the key used to sign the message, and, include the subject key identifier of the certificate used to encrypt the message. This is compliant to criteria WS 10.2.7.2 (a) and WS 10.2.7.2 (b).

5.6.2.3 Publish service metadata

By default, WCF doesn't make the WSDL publicly available.

To publish the service's WSDL:

1. Click on the new service behavior (in this example, it is `SecureBehavior`) in the "Configuration" tree pane.
2. In the "Behavior element extension position" section, click the "Add…" button.
3. In the "Adding Behavior Element Extension Sections" dialog box that appears, select "serviceMetadata" and click the "Add" button.
4. Expand the "SecureBehavior" and then the "serviceMetadata" elements in the "Configuration" tree pane.
5. Select the "serviceMetadata" element in the "Configuration" tree pane.
6. Set the "HttpGetEnabled" property to true.

5.6.2.4 Save changes

Save the changes that were made to *Web.config* file. The following XML fragment will be added to `behaviors` element within the `service.serviceModel` element in the *Web.config* file:

```
<serviceBehaviors>
  <behavior name="SecureBehavior">
    <serviceCredentials>
      <clientCertificate>
        <authentication revocationMode="NoCheck" />
      </clientCertificate>
      <serviceCertificate findValue="cn=wcf_server" />
    </serviceCredentials>
    <serviceMetadata httpGetEnabled="true" />
  </behavior>
</serviceBehaviors>
```

5.6.3 Configure service

To configure the service:

1. Right-click on the *Web.config* file in Visual Studio and select the “Edit WCF Configuration…” menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the “Services” folder in the “Configuration” tree pane. By default, WCF creates a “MyService” service.
3. Select the “MyService” element in the “Configuration” tree pane.
4. Enter the full type name of the service implementation class in the “Name” property. (In this example, it is `Nehta.Example.DS.Server.DSRService`).
5. Select the service behaviour that was created in Section 4.3.2 (in this example, it is `SecureBehavior`) in the “BehaviorConfiguration” property.

5.6.3.1 Configure the service endpoint

A Web service must declare an endpoint in order for clients to access it. An endpoint is made up of an address, which represents the location to send SOAP messages, a binding, which specifies how to communicate with the service, and a contract, which specifies the available operations.

To configure the service endpoint:

1. Expand the service element (in this example, it is `Nehta.Example.DS.Server.Service`) and then the “Endpoints” element in the “Configuration” tree pane.
2. Select the existing endpoint that was created by default; it has the label “(Empty Name)”.
3. Enter a name for the endpoint in the “Name” property. (In this example, it is `ServiceEndpoint`).
4. Enter the service’s physical address (in this example, it is `http://localhost:3916/Server/Service.svc`) in the “Address” property.
5. Select “customBinding” in the “Binding” property and select the binding that was created in section 5.6.1 (in this example, it is `SecureBinding`) in the “BindingConfiguration” property.
6. Enter the full type name of the service interface in the “Contract” property. This can be found by looking in the generated source file that was added to the project. The full type name is found by combining the namespace with the service interface name. The code below shows a fragment of the example service instance. In this case, the full type name of the Web service interface is `Nehta.Example.DS.Server.DischargeSummaryReceiver` with `Nehta.Example.DS.Server` being the namespace and `DischargeSummaryReceiver` being the service interface.

5.6.3.2 Save changes

Save the changes that were made to *Web.config* file. The following XML fragment will be added to `services` element within the `service.serviceModel` element in the *Web.config* file:

```
<service behaviorConfiguration="SecureBehaviour"
name="Nehta.Example.DS.Server.Service">
  <endpoint address="http://localhost:3916/Server/Service.svc"
    binding="customBinding" bindingConfiguration="SecureBinding"
    name="ServiceEndpoint"
    contract="Nehta.Example.DS.Server.DischargeSummaryReceiver"
  />
</service>
```

5.7 Package and deploy the Web service

Once the Web service has been created and configured, it can be deployed using the in-built Visual Studio 2005 development server or with Internet Information Services (IIS).

5.7.1 Development Server

Visual Studio 2005 has an in-built development server that can be used to test and debug Web applications and Web services.

To deploy a Web service to the development server:

1. Right click the project from within the Solution Explorer and select "Set as StartUp Project".
2. Press F5 to start-up the development server and deploy the service. If a list of files is displayed in the browser, select the *Service.svc* file. This URL will need to be used in step 5. Since the "Address" field of Web service configuration won't match the development server URL, an error page should be displayed.
3. Right-click on the *Web.config* file and select the "Edit WCF Configuration…" menu item. This opens the file in the *Microsoft Service Configuration Editor*.
4. Expand the "Services", "Nehta.Example.DS.Server.DSRService", "Endpoints", "(Empty Name)" nodes.
5. Enter the URL from step 2 into the "Address" field of the endpoint properties.
6. Close the configuration editor and save the configuration.
7. Re-run the Web service by pressing F5.

To stop development server from running:

1. Right click on the system tray icon called "ASP.NET Development Server" and select "Stop".

5.7.2 Deploy with IIS

5.7.2.1 Create a compiled Web service (optional)

When deploying a service under IIS a virtual directory needs to be created. A virtual directory in IIS is a reference to a physical directory that can be on the local machine or a networked machine. The directory contains the Web service files, either as source files or compiled files. Generally source files are used for debugging and testing, and the compiled files for production systems.

To create a Web service binary distribution:

1. Select the Web service project within the Solution Explorer.
2. From the "Build" menu select "Publish Web Site".
3. Select the location to put the compiled Web site. This location can later be used as a virtual directory for IIS.
4. Select "OK" and the compiled Web service will be placed into the directory specified.

5.7.2.2 Create a virtual directory

One way to create a virtual directory is to use the IIS snap-in for the Microsoft Management Console (MMC).

To setup the MMC:

1. Start the MMC by selecting Run from the start menu and type *mmc* in the Open: combo box and press enter, or, by entering "mmc" within a command prompt window, this opens a blank Microsoft Management Console.
2. From the "File" menu, select "Add/Remove Snap-in".
3. Press the "Add…" button and select "Internet Information Services" from the list.
4. Select "Close" and then "OK" on the "Add/Remove Snap-in" dialog.

To create a virtual directory using the IIS snap-in:

1. Open the "Internet Information Services" MMC.
2. Expand the local computer node.
3. Expand the "Web Sites" node.
4. Right click on the "Default Web Site" node and select "New" and then "Virtual Directory…"
5. Select "Next" and then enter the alias for the Web site. The alias is used as part of the URL. For example, if the alias was "SomeWebsite", the URL for the Web service would be `http://<computer name>:<port>/<alias>/Service.svc`. Ensure this matches what was used in the "Address" field. For example, if http://localhost/ExampleDSService/Service.svc was entered in the "Address" field in section 4.3.3, enter the alias as "ExampleDSService".
6. Select the directory where the Web service files are located. This can be compiled Web service that was created in section 5.6.2.1 or the project directory which contains the source code.
7. Make sure the "Read" and "Run scripts" options are checked.
8. Select "Finish" and the virtual directory will be created.

Once the directory has been created, only the contents of the virtual directory need to be updated to redeploy the service. The virtual directory does not need to be re-created each time the code has changed.

5.7.2.3 Remove a virtual directory

When the virtual directory is no longer required, it can be removed from IIS.

To remove the virtual directory from IIS:

1. Open the "Internet Information Services" MMC.
2. Expand the local computer node.
3. Expand the "Web Sites" node.
4. Expand the "Default Web Site" node.
5. Right click on the name of the virtual directory and select "Delete".
6. Select "Yes" and the directory will be deleted.

Appendix A: References

[ATS 5820—2010] Standards Australia, E-Health Web Services Profiles.

[GIIWS2007] NEHTA, *Guidelines for Implementing Interoperable Web Services*, version 1.0, March 2007.

[JAXWS] Sun Microsystems, *Java API for XML Web Services (JAX-WS)*, https://jax-ws.dev.java.net.

[MS7331123.0] Microsoft, .Net 3.0, WCF, Data Contract Schema Reference. http://msdn.microsoft.com/en-us/library/ms733112(VS.85).aspx

[MS7331123.5] Microsoft, .Net 3.5, WCF, Data Contract Schema Reference. http://msdn.microsoft.com/en-us/library/ms733112.aspx.

[NDS2006] NEHTA, *National Discharge Summary: Data Content Specifications*, version 1.0, 21 December 2006.

[NIF2006] NEHTA, *Interoperability Framework*, version 1.0, 1 April 2006.

[PING] Muus, *The Story of the PING Program*, http://ftp.arl.mil/~mike/ping.html.

[TAIS2006] NEHTA, *Technical Architecture for Implementing Services: Concepts and Patterns*, version 1.0, 21 December 2006.

[WCF] Microsoft, *Windows Communication Foundation (WCF)*, http://netfx3.com.

[WSAW2006] W3C, *Web Services Addressing 1.0 –WSDL Binding*, W3C Candidate Recommendation, 29 May 2006, http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529/.

[WSE] Microsoft, Web Services Enhancements (WSE), http://msdn2.microsoft.com/en-us/webservices/aa740663.aspx.

[WSP2008] NEHTA, *Web Services Profile v3.0*, 1 December 2008.

[WSPL2006] XmlSoap, *Web Services Policy Framework*, version 1.2, Draft, March 2006, http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy.pdf.

[WSPL2007] W3C, *Web Services Policy 1.5 – Framework*, W3C Recommendation, 4 September 2007, http://www.w3.org/TR/2007/REC-ws-policy-20070904.

[WSSPL2005] XmlSoap, *Web Services Security Policy Language (WS-SecurityPolicy)*, Draft, July 2005, http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf.

[WSSPL2007] OASIS, *WS-SecurityPolicy 1.2*, OASIS Standard, 1 July 2007, http://docs.oasis-open.org/ws-sx/wssecuritypolicy/200702/ws-securitypolicy-1.2-spec-os.pdf.

[MS7331123.0] Microsoft, .NET 3.0, WCF, Data Contract Schema Reference. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

Appendix B: Installation

The following sections detail how to install the tested environment. All the software was installed on Windows XP SP2.

B.1 Visual Studio 2005

Visual Studio 2005 is an IDE used for developing .NET applications.

1. Run "setup.exe" from the "vs" directory on the installation DVD. This will install the Visual Studio 2005 onto the machine.

B.2 .NET Framework 3.0 Redistributable Package

The .NET 3.0 redistributable package installs the necessary components required to run .NET 3.0 applications. Note this is not required when running Windows Vista as it is pre-installed.

1. Download the .NET Framework 3.0 Service Pack 1 Redistributable Package.

   URL:
   http://www.microsoft.com/downloads/details.aspx?FamilyID=EC2CA85D-B255-4425-9E65-1E88A0BDB72A&displaylang=en

2. Run *dotnetfx3setup.exe* to install. This will use the Web installer to download the installation package. To download the installation package without the Web installer, scroll down to the "Instructions" section and select the appropriate link depending on the architecture.

B.3 Windows SDK for Vista and .NET 3.0 Runtime Components

The Windows SDK for Vista and .NET 3.0 Runtime Components is the SDK used for developing .NET 3.0 applications. Note this can be installed onto Windows XP SP2.

1. Download the Windows SDK for Vista and .NET 3.0 Runtime Components.

   URL:
   http://www.microsoft.com/downloads/details.aspx?familyid=C2B1E300-F358-4523-B479-F53D234CDCCF&displaylang=en

2. Run the *setup.exe* to install.
3. Add the SDK installation *bin* directory to the path so that required utilities can be executed from a command window.

B.4 Windows HTTP Services Certificate Configuration Tool

The "Windows HTTP Services Certificate Configuration Tool" is used for granting access to private keys from computer user accounts. This is necessary for a service when it requires the use of a private key to perform security operations.

1. Download *winhttpcertcfg.msi*

   URL:
   http://www.microsoft.com/downloads/details.aspx?familyid=c42e27ac-3409-40e9-8667-c748e422833f&displaylang=en

2. Run *winhttpcertcfg.msi* to install.
3. Add the installation directory of the tool to the PATH environment variable so it can be called from a command prompt.

B.5 Visual Studio 2005 Extensions for .NET 3.0

The extensions for .NET 3.0 add tools to Visual Studio 2005 that help in the development of WCF applications. More specifically, it contains the service configuration editor tool that can be used to graphically configure WCF services and clients alleviating the need to modify the XML configuration by hand.

1. Download the Visual Studio 2005 Extensions for .NET 3.0

   URL:
   http://www.microsoft.com/downloads/details.aspx?familyid=F54F5537-CC86-4BF5-AE44-F5A1E805680D&displaylang=en

2. Close any running instances of Visual Studio 2005.
3. Run *vsextwfx.msi* to install.

B.6 Internet Information Services (IIS) (optional)

IIS can be used to host WCF Web services. Only install IIS when planning to make Web services accessible from outside the machine the services are being developed on. Use the Visual Studio 2005 development server to do testing within the local development machine.

1. Insert the Windows XP SP 2 CD/DVD into the drive.
2. Open the "Control Panel" and select "Add/Remove Programs".
3. From the left hand set of icons select "Add/Remove Windows Components".
4. On the "Windows Components Wizard" dialog, check the "Internet Information Services (IIS)" option and select "Next". IIS will then be installed.

ASP.NET needs to be installed into IIS before .NET applications can be run under IIS. When IIS is installed after Visual Studio, the ASP.NET components must be installed into IIS manually. To install ASP.NET use the ASP.NET registration command-line tool *aspnet_regiis.exe*. This is located in the *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727* directory.

From a command window run the following to install ASP.NET into IIS:

```
aspnet_regiis -i
```

Appendix C: Key management

Windows stores certificates in the Windows certificate store and provides access to the security tokens from applications and services through an API. The store can also be accessed and managed using the Microsoft certificate MMC. The store is split into two locations, *CurrentUser* and *LocalComputer*. Each location is then split into a number of logical stores that hold security tokens with each store having a specific purpose.

User certificates with private keys are stored in the *CurrentUser* location within the *My* store, with *CA* certificates being placed in the *Trusted Root Certificate Authorities* store and other issued certificates in the *Other People* store. Service certificates are normally stored in the *LocalComputer* location within the *My* `store` with *CA* certificates being placed in the *Trusted Root Certificate Authorities* store.

C.1 Setting up the MMC for certificate management

The certificate MMC is used for managing certificates installed within the Microsoft certificate store.

1. Start the MMC by selecting Run from the start menu and type *mmc* in the Open: combo box and press enter, or, by entering "mmc" within a command prompt window, this opens a blank Microsoft Management Console.
2. From the "File" menu, select "Add/Remove Snap-in".
3. Press the "Add…" button and select "Certificates" from the list.
4. Select "My user account" and select "Finish", this will add a management snap-in that'll allow personal certificates to be managed.
5. Press the "Add…" button again and select "Certificates" from the list.
6. Select "Computer Account" and then select "Local computer:" on the select computer dialog.
7. Select "Close" and then "OK" on the "Add/Remove Snap-in" dialog.

C.2 Installing certificates for the client application

The client application requires the client's private key to sign the message, a service certificate for encrypting the message, and a CA (certificate authority) certificate for verification. The follow sections describe how to add the client certificates that are required by the client in order to secure and validate the message.

The following steps are required to setup the client application certificates:

1. Add the client certificate (that has a private key) which is used for signing.
2. Add the service certificate which is used for encrypting.
3. Add the CA.

C.2.1 Adding the client certificate

1. Open the certificate MMC.
2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Personal" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.

4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the PFX file that contains the client certificate and private key and select "Next".
6. Enter the password (if any) of the private key and select "Next".
7. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Personal".

C.2.2 Adding the service certificate

1. Open the certificate MMC.
2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Personal" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the service certificate and select "Next".
6. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Other People".

C.2.3 Adding the CA

1. Open the certificate MMC.
2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Trusted Root Certification Authorities" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the certificate file that is the CA and select "Next".
6. Select "Next" again and finally select "Finished", the certificate will then appear within the "Certificates" folder under "Trusted Root Certification Authorities".

C.3 Setting up the service certificates

The Web service requires a certificate with a private key to sign the message. A client certificate which would be used for encryption does not need to be specified since this can be extracted from the client message. The following sections describe how to add the certificates that are required by the service in order to secure and validate messages.

The following steps are required to setup the service certificates:

1. Add the service certificate (that has a private key) which is used for signing the response to the client.
   a. Set the service certificate permissions.
2. Add the CA.

C.3.1 Adding the service certificate

1. Open the certificate MMC.
2. Expand the “Certificates (Local Computer)” tree by left clicking on the plus sign.
3. Right click on the “Personal” folder and select “All Tasks” and then “Import”, the “Certificate Import Wizard” dialog should appear.
4. Select “Next” to go to the “File to Import”.
5. On the “File to Import” dialog, select “Browse” then choose the service certificate and select “Next”.
6. Select “Next” again the finally select “Finished”, the certificate will then appear within the “Certificates” folder under “Personal”.

C.3.1.1 Set the service certificate permissions

For ASP.NET to be able to access the private key of the service certificate it must have the correct security permissions. The "Windows HTTP Service Certificate Configuration Tool" can be used to set the permissions on the certificate so that the “ASPNET” account has access to the private key.

To set the permissions on the service certificate:

1. Open a command prompt.
2. Execute the following command:

```
winhttpcertcfg -g -c LOCAL_MACHINE\My
  -s <certificate name> -a ASPNET
```

C.4 Adding the CA

1. Open the certificate MMC.
2. Expand the “Certificates (Local Computer)” tree by left clicking on the plus sign.
3. Right click on the “Trusted Root Certification Authorities” folder and select “All Tasks” and then “Import”, the “Certificate Import Wizard” dialog should appear.
4. Select “Next” to go to the “File to Import”
5. On the “File to Import” dialog, select “Browse” then choose the certificate file that is the CA and select “Next”.
6. Select “Next” again and finally select “Finished”, the certificate will then appear within the “Certificates” folder under “Trusted Root Certification Authorities”.

C.5 Pitfalls

When installing the security tokens into the Windows certificate store always use the certificate MMC import function. The certificate MMC allows the security tokens to be copied between locations and logical stores but copying a private key can cause problems.

Appendix D: Debugging

There are a number of ways to debug WCF applications. This appendix describes some possible debugging techniques which can be used:

1. Diagnostics configuration;
2. Displaying service metadata (WSDL);
3. `serviceDebug` behaviour element;
4. HTTP debugging proxy; and
5. Debugging a service that uses IIS.

D.1 Diagnostics configuration

When implementing and testing a client or Web service, message and trace logging can be enabled and configured. The configuration for the diagnostics can be added using the service configuration tool or added directly to the *web.config* file.

To add diagnostics configuration using the service configuration tool:

1. Open the service configuration tool on the service applications configuration file, `web.config`.
2. Left click on the "Diagnostics" folder. In the left panel a number of diagnostics options will be displayed.
3. Left click on the "Enable Log Auto Flush" link that is next to the "Log Auto Flush" text to enable log flushing.
4. Left click on the "Enable Message Logging" option next to the "Message Logging" text to enable message level logging. The "Log file:" text specifies where the file will be created. Use the *svctraceviewer.exe* utility to view the log file.
5. Left click on the "Enable Tracing" option next to the "Tracing" text to enable tracing. The "Log file:" text specifies where the file will be created. Use the *svctraceviewer.exe* utility to view the log file.

D.2 Displaying service metadata (WSDL)

When a service is initially created, by default the WSDL is not accessible. To allow the WSDL for a service to be accessible, the `serviceMetadata` behaviour element extension needs to be added to the behaviour used by the service.

To add the `serviceMetadata` behaviour element extension:

1. Open the service configuration tool on the service applications configuration file, `web.config`.
2. Expand the "Advanced" folder.
3. Expand the "Service Behaviors" folder.
4. Right click on the behaviour that's being used for the service and select "Add Service Behavior Element Extension".
5. From the list select `serviceMetadata` to add it to the behaviour.
6. Left click on the `serviceMetadata` element
7. Set the "HttpGetEnabled" to true. Set the "HttpsGetEnabled" to true if using HTTPS. This allows the WSDL to be accessed via HTTP.

D.3 serviceDebug behaviour element

On the service side, the `serviceDebug` element can be added to the service behaviour to allow a HTML help page to be displayed when the service URL is contacted with a browser and to allow fault details be included when SOAP faults are thrown from the service.

To add the `serviceDebug` behaviour element:

1. Open the service configuration tool on the *web.config*.
2. Expand the "Advanced" folder.
3. Expand the "Service Behavior" folder.
4. Right click on the behaviour that is being used for the service and select "Add Service Behavior Element Extension".
5. From the list select the `serviceDebug` behaviour element extension so it is added to the behaviour.
6. Left click on the `serviceDebug` element that has been added.
7. Set the "IncludeExceptionDetailInFaults" to true and leave the rest as their default values.

D.4 HTTP Debugging Proxy

A HTTP debugging proxy can be used to view all messages that are sent and received to and from a client. This can be useful to see the raw SOAP messages in XML, and any HTTP headers that are sent with the message.

Once the HTTP debugging proxy has been installed and configured, the client must be configured to route messages through the debugging proxy. This can be done using the service configuration tool.

To configure the client to use the HTTP debugging proxy:

1. Open the service configuration tool on the *web.config*.
2. Expand the "Binding" folder.
3. Expand the binding that is being used for the client.
4. Left click on the "httpTransport" element to view the associated configuration.
5. Set the "UseDefaultWebProxy" to false. When this is set to true the client tries to use the proxy that has been configured within Internet Explorer.
6. Within the "ProxyAddress" field, enter the address of the proxy that's being used. Specify the computer name (even if using the local machine) and the port of the debugging proxy.

D.5 Debugging a service that uses IIS

To debug a service that's running under IIS:

1. Deploy the service to IIS.
2. From within Visual Studio 2005, select "Debug" and then "Attach to Process…".
3. From the list of processes, select `aspnet_wp.exe` to connect to the ASP.NET process that runs the service. Now normal debugging operations can take place.