Enterprise Service Bus and Web Services Security

Daniel Huss

University of Stuttgart, IAAS
hussdl@studi

Abstract. In most organizations, technological heterogenity is more the rule than the exception. While new software applications are often built with integration in mind and using promising approaches such as service-oriented architecture (SOA), valuable business data remains locked up within existing applications. The Enterprise Service Bus (ESB) is an integration middleware that enables an SOA by coordinating the communication between (web) services and those existing applications. In this paper we'll take a look at how the DecidR application can benefit from the Apache Synapse ESB despite being developed in a homogenous SOA environment. In addition we'll look into the value that Web Services Securtiy (WSS) can add to the project.

1 Enterprise Service Bus

As an organization or business grows, so does the number of software applications that need to communicate with each other in order to exchange business data. Unless integration is considered an important matter early, it is very likely that an "accidental architecture" will emerge:

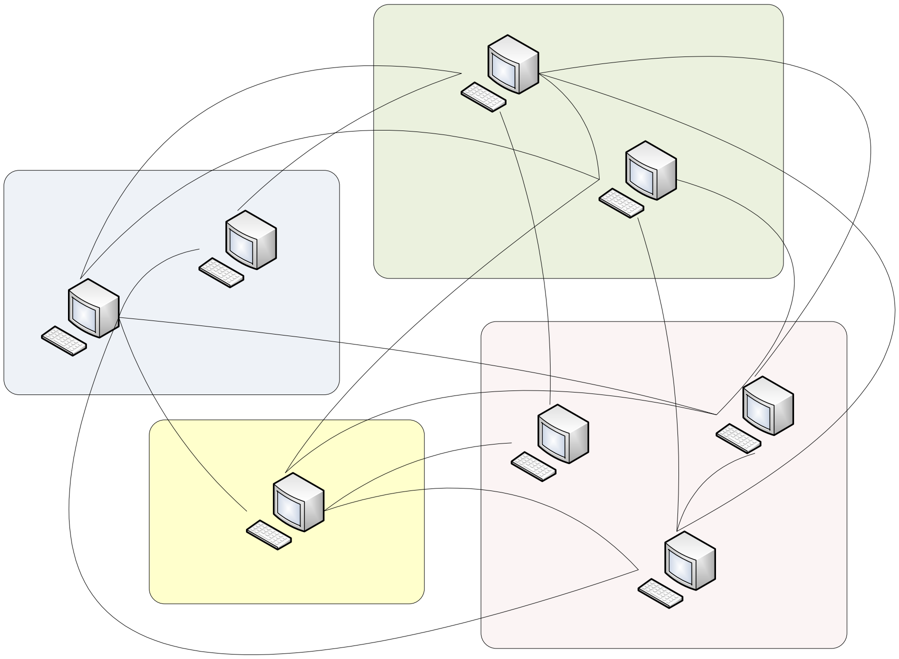


Fig. 1. The problem: point to point connectivity between applications requires definition, implementation and maintenance of $\mathcal{O}(n^2)$ interfaces.

Enterprise Application Integration (EAI) is an attempt to reduce integration complexity and costs by introducing a mediator. Applications connect to a central integration broker via an adapter, which significantly reduces the number of interfaces that need to be developed and maintained.

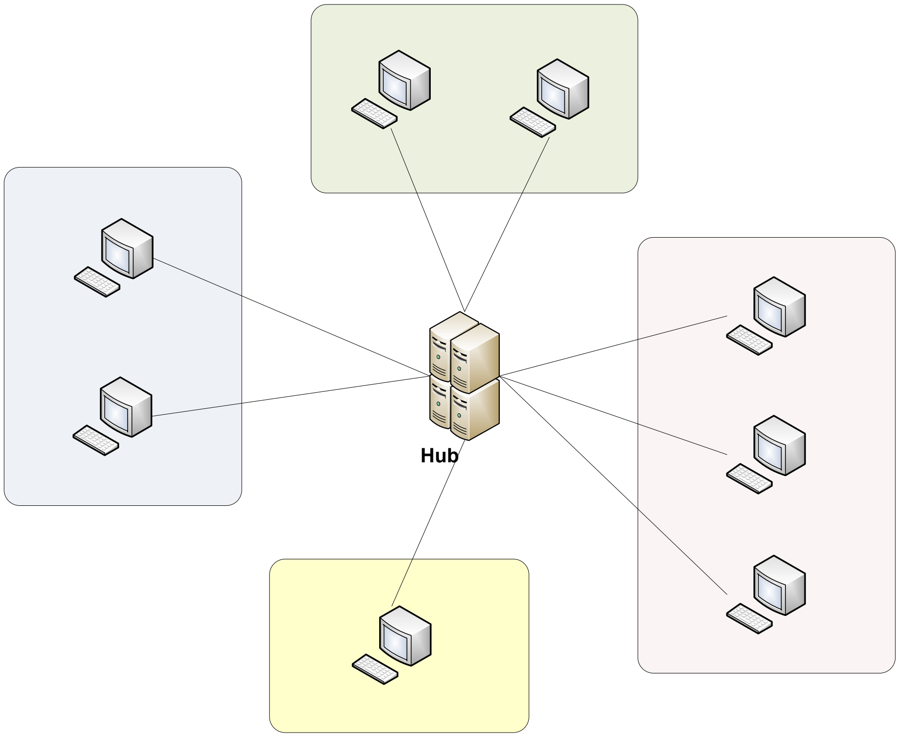


Fig. 2. The hub-and-spoke integration network reduces complexity to $\Theta(n)$ interfaces.

A major disadvantage of previous EAI implementations is that most rely on proprietary, competing standards, making business-to-business communication difficult. Also, all information must traverse the central hub, which prohibits a highly distributed infrastructure on the side of the integration broker. In contrast, a distinguishing key feature of an ESB is its distributable infrastructure:

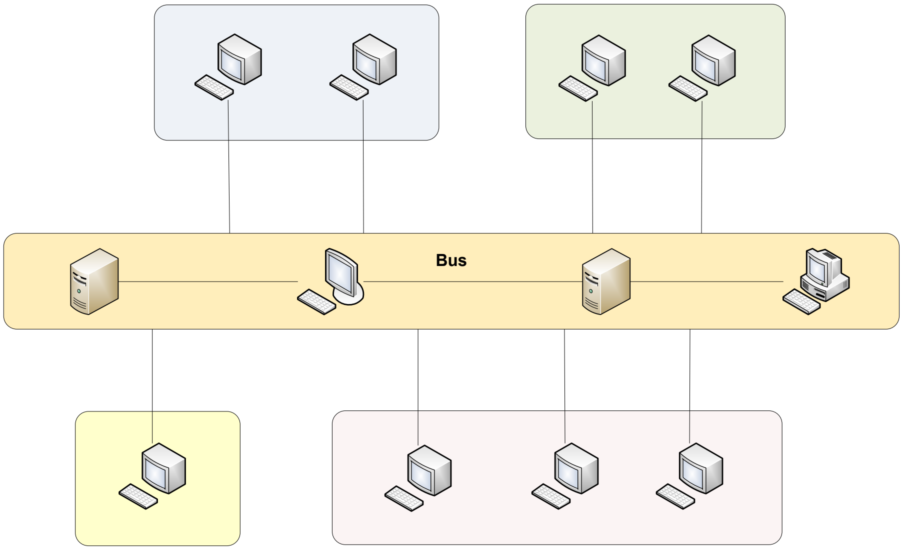


Fig. 3. The bus topology allows for highly distributed deployment.

Unfortunately, there is no standardized definition of what comprises an ESB and what doesn't. Some comfort can be taken from the description of a set of capabilities that make the basis of an ESB, given by David Chappell, a major contributor to the emergence of the term:

> [chpll] " An ESB is a standards-based integration platform that combines messaging, web services, data transformation and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications . . . with transactional integrity. "

1.1 XML

Connecting diverse applications requires a loosely coupled, flexible and extensible data model such as the one provided by XML / XML Schema. It is widely used in available ESB implementations [chpll] . With the Extensible Stylesheet Language Transformations (XSLT) an xml-to-any data transformation language is also available. (see section 1.3)

1.2 Endpoints

From a conceptual view, the ESB functionality is hidden behind abstract endpoints. Endpoints are just logical abstractions of services that are connected into the ESB. Endpoints also abstract from the underlying protocol or message queueing implementation. An endpoint may represent many things, such as a single operation of an exam registration service. The actual implementation of the endpoint may use an application that is deployed on a local server, or may redirect to an external web service. An endpoint may just as well represent a legacy application that handles matriculation management, an Enterprise Resource Management (ERP) system or even an entire department of a University, such as the IAAS. Each endpoint is an equal SOA participant regardless of what it represents.

It is the task of an integration architect to connect these endpoints by defining routing rules, data transformations and business process logic. (Service orchestration)

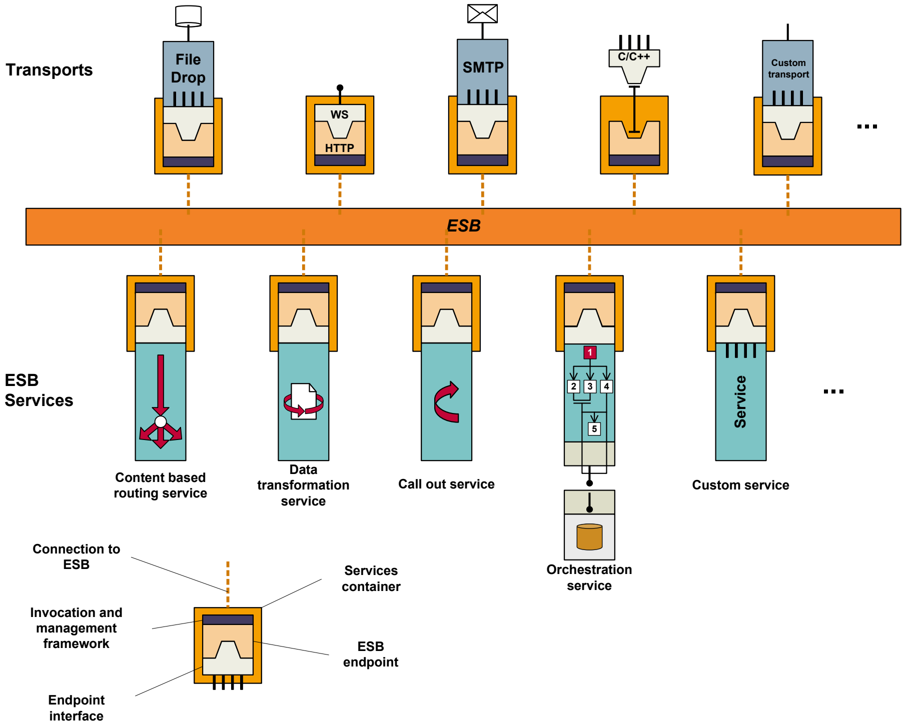


Fig. 4. Everything that is connected into the bus is viewed as an abstract endpoint. Drawn using Sonic Software ESB Visio Stencils [snc-icns]

1.3 ESB Tasks

Messaging Message-oriented Middleware (MOM) is a key part of the ESB. All communication operations are performed on self-contained units of information called messages. This allows each communication operation to be a self-contained, standalone unit of work. [chpll] At the core of the ESB sits a powerful messaging system such as the Java Messaging Service (JMS) that provides asynchrous and event-driven messaging capabilities.

Mediation and Invocation The ESB must support many transport protocols and be able to convert between them in order to enable communication between many different applications. The ESB should support synchrous as well as asynchrous transports, and event-driven messaging via publish / subscribe. For legacy applications the ESB must support the creation of adapters that build a service frame around the application.

Content-based Routing A content-based routing (CBR) service decides which endpoint(s) receive messages that pass through the ESB. The routing decisions are based on a set of rules that may be defined by configuration files, scripts or programs. The routing rules may be based on the content of each message such as specific XML elements / attributes or message metadata.

Data transformation is a capability that applies mostly to XML documents. The ESB can use transformation rules such as XSLT documents to covert between XML dialects or enrich messages with additional information.

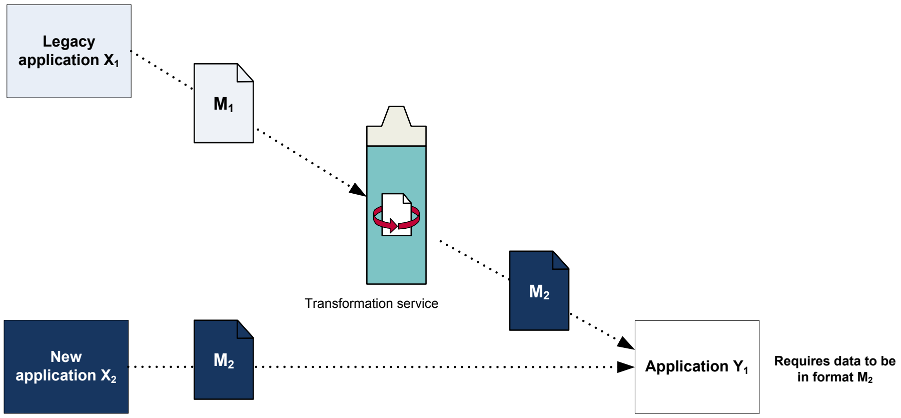


Fig. 5. In this example data transformation is used to provice compatibility of a legacy application without having to modify its code.

Quality of Service The loose coupling that can be achieved in an SOA gives the ESB an opportunity for optimization. Since services may be deployed accross several physical nodes, the ESB can include server load in its routing decisions and provide a failover option.

Furthermore, a good ESB provides reliable communication channels using standards such as WS-ReliableMessaging and WS-Reliability. It also supports encryption and signatures, which provides end-to-end security that doesn't depend on the underlying protocol for its services.

Management Since the entire data flow in an SOA passes through the ESB, it is fit for monitoring, logging or audits.

1.4 ESB Usage in DecidR

From what we can tell at this point, the DecidR application is being developed in a pure SOA environment with no requirements pointing to integration with existing applications. However, we also can't declare impossible that such a requirement arises at a later project phase. Therefore it is reassuring to have an integration platform ready in such a case. The DecidR system components are implemented using web service technology, which is natively supported by the ESB; the effort for "getting on the bus" is minimal, but comes with certain benefits:

Service Discovery / Invocation The service bus adds a layer of indirection that allows service instances to be moved to other physical locations without having to modify the adressing configuration of the clients.

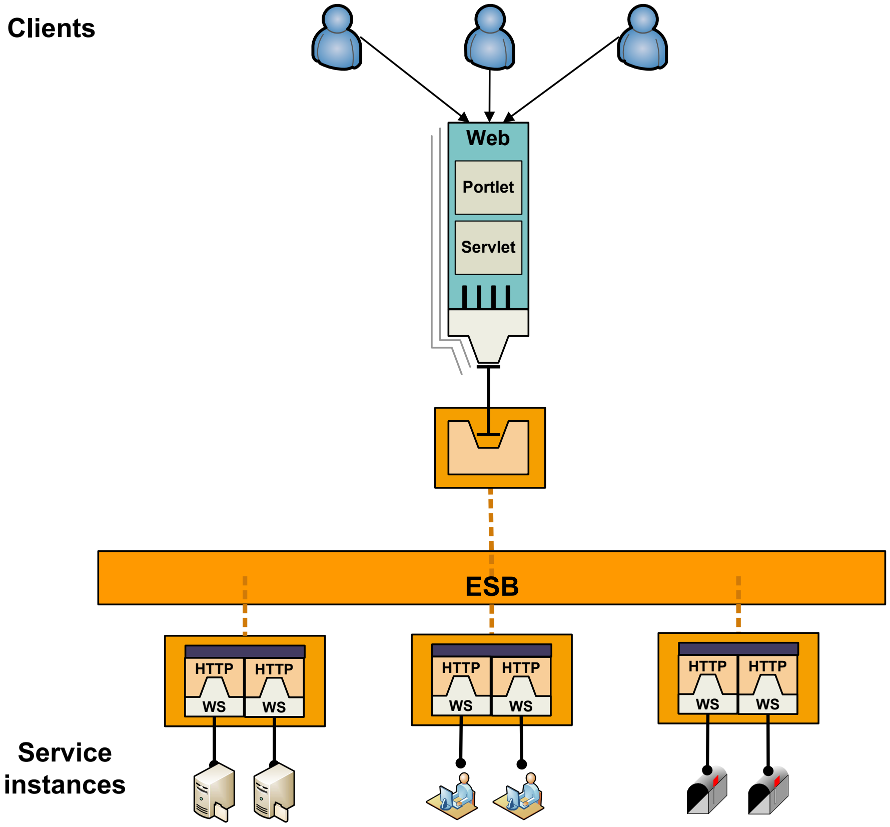


Fig. 6. DecidR subsystems invoke web services through the ESB, never knowing where the actual service instance is located.

Load Balancing and Service Availability If necessary due to high server load, multiple instances of a service can be run on different physical nodes. The ESB will then take care of automatically balancing the load across multiple service instances.

The same mechanism can be used to provide high availability by temporarily disabling a defunct service endpoint and switching automatically to another service endpoint that provides the same functionality. See chapter 2 for samples.

Logging The logging capabilities of the ESB can be used for debugging and administration of the DecidR application. Intense logging during the development phase is highly recommended by the client.

1.5 Apache Synapse

Due to the limited experience of the project team with SOA, web services, etc. a service bus that doesn't unnecessarily complicate things is desirable. In the following section we're going to take a look at Apache Synapse, which claims to be a "lightweight" ESB and is built upon the Axis2 web services engine. Its main features as advertised by the project website are:

> [synps] " Apache Synapse is an ESB that has been designed to be simple to configure, very fast, and effective at solving many integration and gatewaying problems. Synapse has support for HTTP, SOAP, SMTP, JMS, FTP and file system transports, Financial Information eXchange (FIX) and Hessian protocols for message exchange as well as first class support for standards such as WS-Addressing, Web Services Security (WSS), Web Services Reliable Messaging (WSRM), efficient binary attachments (MTOM/XOP). Synapse can transform messages using key standards such as XSLT, XPath and XQuery, or simply using Java. Synapse supports a number of useful functions out-of-the-box without programming, but it also can be extended using popular programming languages such as Java, JavaScript, Ruby, Groovy, etc..."

Synapse is configured using an XML file that contains the mediation rules for routing incoming and outgoing messages, data transformations, etc. For a complete description of the XML configuration language please refer to [syncf] . The second releavant configuration file is axis2.xml, where the underlying Axis2 engine that provides the transport protocols is configured.

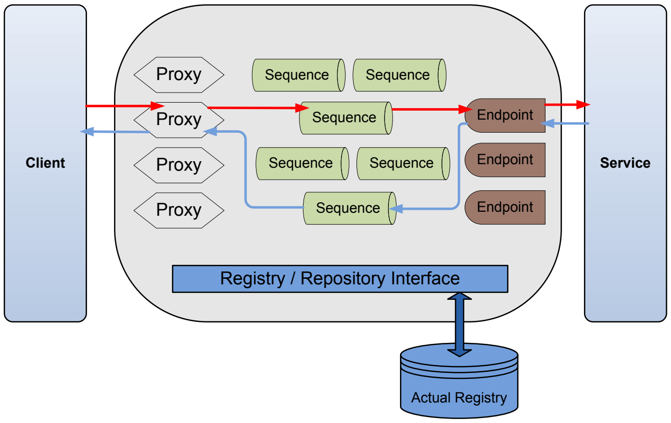


Fig. 7. Synapse service mediation, based on [syncf]

Synapse supports two modes of operation: in *message mediation* mode the clients must be configured to use the Synapse node as a HTTP proxy. In *service* *mediation* mode, Synapse explicitly exposes existing services or applications as services using the underlying Axis2 engine. Client applications communicate directly with the bus.

Deployment To install Synapse, simply unpack the standard binary distribution from `http://synapse.apache.org/download/1.2/download.cgi` to a directory of your choice. Before running Synapse, you musti nstall the latest 32 bits JDK on your system.

Alternatively you may use the package that is bundeled with this document. It contains the samples that are covered in the next chapter.

2 Synapse sample configurations

The standard distribution of Synapse v1.2 comes with 56 sample configuations that are very helpful for understanding its functionality. In this section, we will explore some sample configurations that are loosely based on the standard samples. Of course, the following samples only show the tip of the iceberg, many features remain unmentioned. However, the topics that are actually relevant to the DecidR application (load balancing, logging. . .) are addressed.

2.1 Sample 1: Open Proxy

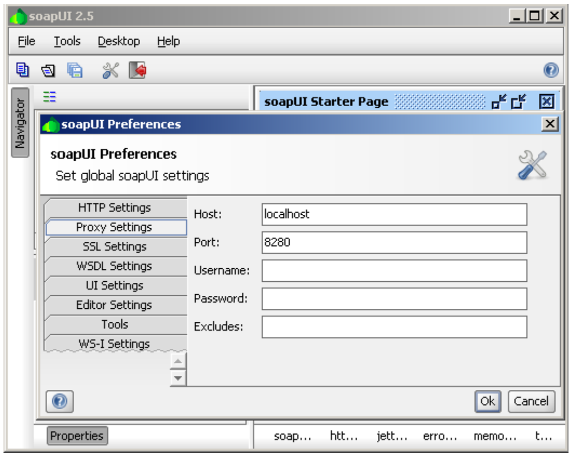
We start with a very simple configuration that basically does nothing except logging any incoming messages, then sends the messages to their implicit destination. Implicit means that Synapse must be able to automatically determine the final destination of the message based on the message contents, URL, SOAP header properties, or some other bits of information that are attached to the message.

- Start Axis2: `synapse-1.2/samples/axis2Server/axis2server -http 9000 -https 9002`
- Run Synapse: `synapse-1.2/bin/synapse -sample 1`

```
    <!-- Sample Synapse configuration that creates an open proxy -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <!-- Every Synapse config has at least main sequence and a fault squence.
         If either is not defined in the config file (or via the registry),
         synapse generates default sequences instead. -->
    <sequence name="main">
        <!-- The in-mediator accepts only incoming messages. -->
        <in>
            <!-- There are also pre-defined log levels such as "simple" or
                 "full."  -->
            <log level="custom">
            <property name="Text" value="Incoming message!"/>
            </log>
            <!-- This sends the message to its implicit destination, if
                 known. -->
            <send/>
        </in>
        <!-- The out-mediator accepts only outgoing messages. -->
        <out>
            <log level="custom">
            <property name="Text" value="Outgoing message!"/>
            </log>
            <send/>
        </out>
    </sequence>

    <!-- The sequence named ”fault” is executed if a fault occurs that isn't
         handled by another sequence. Faults are very similar to exception
         handling in programming languages like Java. The ”fault” sequence
         basically represents the default handler for otherwise uncaught
         expections.
        -->
    <sequence name=”fault”>
        <log level=”custom”>
            <property name=”text” value=”An unexpected error occured”/>
            <property name=”message” expression=”get-property('ERROR_MESSAGE')”/>
        </log>
        <!-- Discards the message. -->
        <drop/>
    </sequence>
</definitions>
```

To test this configuration, start up SoapUI and create a new project. Enter a project name and use `http://localhost:9000/soap/TimeService?wsdl` as the initial WSDL. Create a test suite for the imported WSDL. Now go to the preferences and make SoapUI connect through the Synapse proxy at localhost:8280.



Host: localhost
Port: 8280
Username:
Password:
Excludes:

If the test suite is run, the SOAP response should look similar to this:

```
<soapenv:Envelope xmlns:soapenv=”http://www.w3.org/2003/05/soap-envelope”>
    <soapenv:Body>
        <ns:getTimeResponse xmlns:ns=”http://esb.seminar.decidr.de”>
            <ns:return>2009-01-05T06:42:24.832Z</ns:return>
        </ns:getTimeResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

2.2 Sample 2: Simple Service Gateway

In this sample we turn Synapse into a gateway for an existing web service.

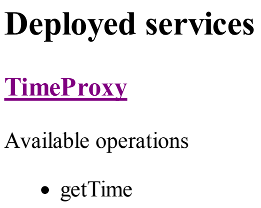
- Start Axis2: `synapse-1.2/samples/axis2Server/axis2server -http 9000 -https 9002`
- Run Synapse: `synapse-1.2/bin/synapse -sample 1`

```
<!-- Sample Synapse configuration that exposes a single web service using the
     underlying Axis2 engine -->
<definitions xmlns=”http://ws.apache.org/ns/synapse”>
    <!-- Sequences can be named and later reused in multiple places -->
    <sequence name=”timeFault”>
        <!-- Logs and discards the message. -->
        <log level=”full”/>
        <drop/>
    </sequence>

    <!-- A simple time service that runs on the included Axis2 server is exposed
         and will be reachable directly through the ESB via http/s -->
    <proxy name=”TimeProxy” transports=”http,https” faultSequence=”timeFault”>
        <target>
            <!-- The endpoint defines where messages for this proxy will be
                 delivered to-->
            <endpoint>
                <address uri=”http://localhost:9000/soap/TimeService”/>
            </endpoint>
            <outSequence>
                <send/>
            </outSequence>
        </target>
        <!-- Synapse uses the given WSDL file to build its representation of
             the above endpoint -->
        <publishWSDL uri=”http://localhost:9000/soap/TimeService?wsdl”/>
    </proxy>

    <!-- For securtiy reasons we reject all unexpected messages -->
    <sequence name=”main”>
        <drop/>
    </sequence>
</definitions>
```

Navigating to `http://localhost:8280/soap` with any web browser reveals the underlying Axis2 server. The web service it hosts simply forwards all messages to `http://localhost:9000/soap/TimeService`.



Deployed services

TimeProxy

Available operations

- getTime

2.3 Sample 3: Service Gateway with Load Balancing

Now that we know how to set up a service proxy, we can add load balancing to its endpoint. Currently Synapse supports spreading all incoming requests over a set of endpoints in a round-robin fashion.

- Start 3 Axis2 servers:
  - `synapse-1.2/samples/axis2Server/axis2server -http 9000 -https 9002`
  - `synapse-1.2/samples/axis2Server/axis2server -http 8000 -https 8002`
  - `synapse-1.2/samples/axis2Server/axis2server -http 7000 -https 7002`
- Run Synapse: `synapse-1.2/bin/synapse -sample 3`

```
<!-- A proxy service that performs load balancing -->
<definitions xmlns=”http://ws.apache.org/ns/synapse”>
  <sequence name=”timeFault”>
      <!-- Logs and discards the message. -->
      <log level=”full”/>
      <drop/>
  </sequence>

  <proxy name=”LoadBalancedTimeProxy” transports=”http,https”>
    <target faultSequence=”timeFault”>
      <inSequence>
        <send>
          <endpoint>
            <!-- The roundRobin is the only available algorithm at
                 the moment -->
            <loadbalance algorithm=”roundRobin”>
              <endpoint>
                <address uri=”http://localhost:9000/soap/TimeService”>
                  <!-- Activates WS-Addressing for this endpoint -->
                  <enableAddressing/>
                  <!-- If we do not specify a suspend duration, the
                       endpoint will be permanently disabled upon
                       failure -->
                  <suspendDurationOnFailure>30</suspendDurationOnFailure>
                </address>
              </endpoint>
              <endpoint>
                <address uri=”http://localhost:8000/soap/TimeService”>
                  <enableAddressing/>
                  <suspendDurationOnFailure>30</suspendDurationOnFailure>
                </address>
              </endpoint>
              <endpoint>
                <address uri=”http://localhost:7000/soap/TimeService”>
                  <enableAddressing/>
                  <suspendDurationOnFailure>30</suspendDurationOnFailure>
                </address>
              </endpoint>
            </loadbalance>
          </endpoint>
        </send>
        <drop/>
      </inSequence>
      <outSequence>
        <send/>
      </outSequence>
    </target>
    <publishWSDL uri=”file:repository/conf/sample/resources/TimeService.wsdl”/>
  </proxy>

  <sequence name=”main”>
    <drop/>
  </sequence>
</definitions>
```

This configuration exposes a single web service to the client, but actually forwards incoming messages to one of three service endpoints:

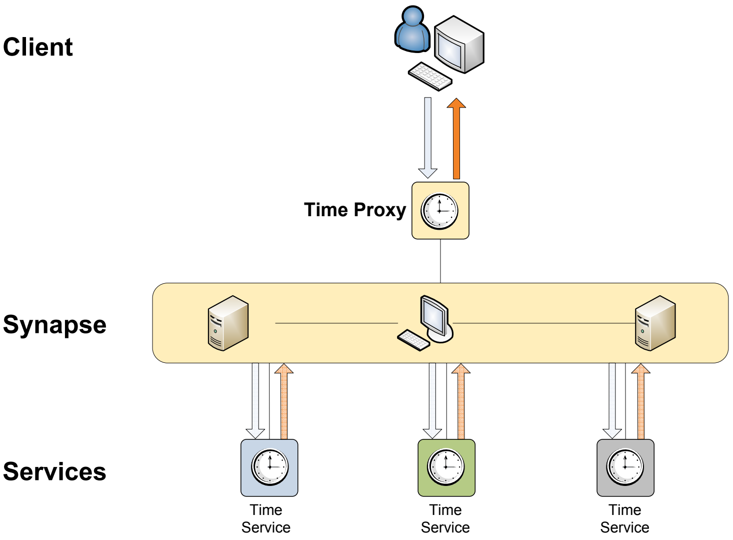


Fig. 8. Using a virtual endpoint to implement load balancing.

2.4 Sample 4: Retrieval of an External Configuration via the Registry

Multiple instances of Synapse can share the same configuration by using the registry to retrieve the mediation definitions from a remote source.

- Start Axis2: `synapse-1.2/samples/axis2Server/axis2server -http 9000 -https 9002`
- Run Synapse: `synapse-1.2/bin/synapse -sample 4`

```
<!-- A registry based configuration. Synapse will look for the key ”synapse.xml”
      to retrieve the actual definitions -->
<definitions xmlns=”http://ws.apache.org/ns/synapse”>
        <registry provider=”org.apache.synapse.registry.url.SimpleURLRegistry”>
        <parameter name=”root”>file:./repository/conf/sample/resources/</parameter>
        <parameter name=”cachableDuration”>100000</parameter>
    </registry>
</definitions>
```

The resulting configuration is the same as in example #3.

3 Web Services Security

In a nutshell, the goal of WS-Security is to enable applications to conduct secure SOAP message exchanges [ws-sec] independently from the underlying transport protocol. WS-Security itself does not guarantee a secure architecture, it only provides the means to encrypt or sign the headers and contents of SOAP messages. The standard was initially developed by IBM, Microsoft, VerySign and Sun. The 1.0 version was released by the Organization for the Advancement of Structured Information Standards (OASIS) in 2004. The current version 1.1 was released in 2006.

3.1 Comparison to Transport Layer Security

With WS-Security we have end-to-end security, which means that sensitive data stays encrypted *all the way* to its final destination. Transport layer security such as HTTPS cannot guarantee end-to-end security due to the fact that a message may travel through more than one logical network node. Not only would every network node between initial sender and ultimate reveiver know the secret conents of that message, but there is also no guarantee that the underlying transport protocol doesn't change in between.

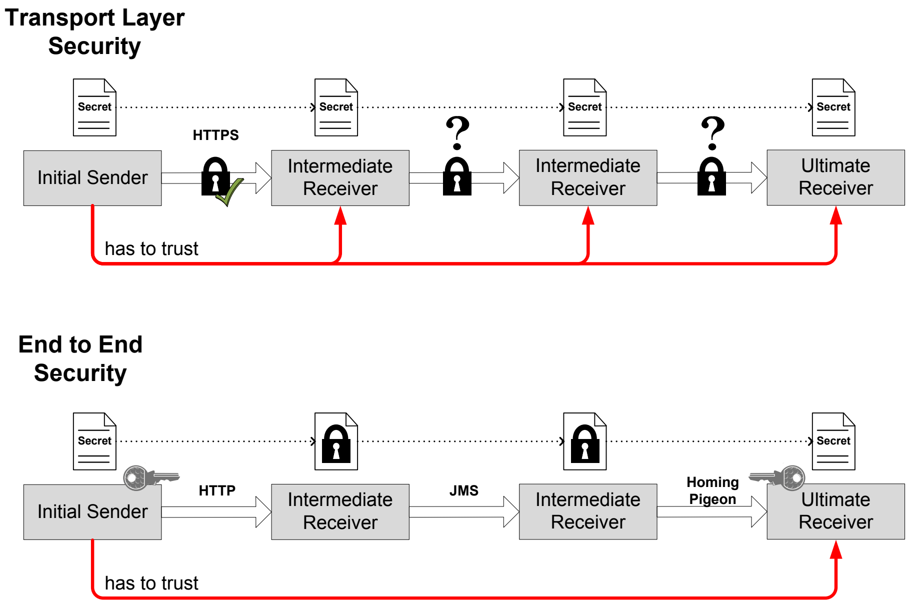


Fig. 9. In an SOA transport layer security is not sufficient because the underlying transport protocol may change between sender and receiver of a message.

3.2 WS-Security in DecidR

Security is an important requirement for DecidR, especially concerning privacy protection obligations and resilience to attacks on sensitive data [dspec] . Therefore all communication between client applications, DecidR web services and *external* web services should be encrypted. WS-Security supports X.509 certificate based authentication as described by [x509] , which provides the DecidR project team with a professional approach for secure message exchanges between the system components:

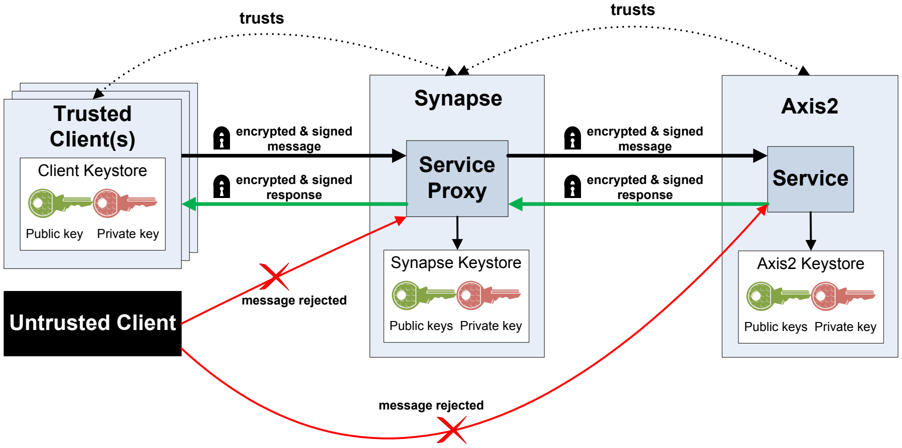


Fig. 10. The Synapse ESB can act as a trusted service mediator with enabled security features.

3.3 Using WS-Security with Synapse

Adding WS-Security support in Synapse is done simply by adding the `<enableSec>` element to a `<proxy>` or `<address>` definition, along with a `<policy>` element that specifies the attached security policy.

As a result, it's quite easy to create a service proxy that comes with WS-Security features, even though the actual service endpoint does not support WS-Security. The following listing shows a sample Synpase configuration file that achieves this. Note that this approach only makes sense if there's a need to provide faked security to a client, since the communication between Synapse and the actual service endpoint remains possibly unencrypted.

```
<!-- Sample Synapse configuration that creates a virtual endpoint with enabled
     security, even though the virtalized endpoint does not support WSS -->
<definitions xmlns=”http://ws.apache.org/ns/synapse”>
    <!-- Defines the security policy used for the virtual endpoint -->
    <localEntry key=”sec_policy” src=”file:repository/policies/my_policy.xml”/>

    <!-- Virtual endpoint with enabled security -->
    <proxy name=”SecuredServiceProxy” transports=”http”>
        <target>
            <endpoint>
                <address uri=”https://localhost:9000/soap/UnsecuredService”/>
            </endpoint>
            <outSequence>
                <send/>
            </outSequence>
        </target>

        <publishWSDL uri=”http://localhost:9000/soap/SecuredService?wsdl”/>
        <!-- The security policy to use for the virtual endpoint -->
        <policy key=”sec_policy” type=”in out”/>
        <!-- Enables WS-Security for the virtual endpoint -->
        <enableSec/>
    </proxy>

    <sequence name=”main”>
        <drop/>
    </sequence>
</definitions>
```

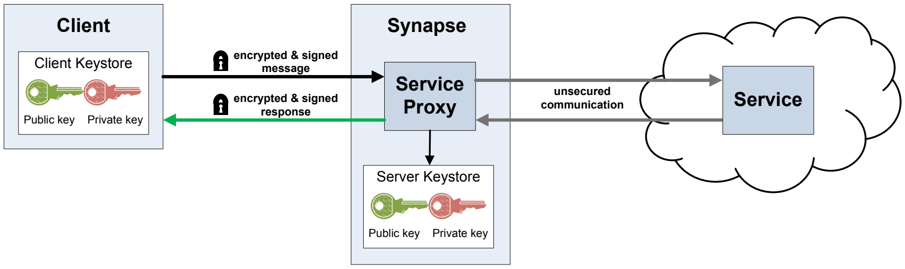


Fig. 11. The resulting configuration provides secure communication between the client and Synapse, although this might give the client a false impression of the overall security of the message exchange.

References

[chpll] David A. Chappell, "Enterprise Service Bus" first Edition. O'Reilly, Beijing, Cambridge, Farnhman (2004)

[dspec] Modood Alvi, Markus Fischer, Daniel Huss, Jonas Schlaak, "Software Requirements Specification for DecidR" `https://decidr.googlecode.com/svn/trunk/docs/specification/Software%20Requirements%20Specification.doc` (accessed February 2, 2009)

[snc-icns] Sonic Software, "The Sonic Software ESB Icon and Diagram Library", O'Reilly Media, Inc. `http://oreilly.com/catalog/9780596006754/esb_icons.csp` (accessed January 3, 2009) used with permission.

[synps] Apache Software Foundation, "Apache Synapse" web site, `http://synapse.apache.org/index.html` (accessed January 4, 2009)

[syncf] Apache Software Foundation, "Apache Synapse Configuration Language", `http://synapse.apache.org/Synapse_Configuration_Language.html` (accessed February 2, 2009)

[ws-sec] Anthony Nadalin, Chris Kaler, Phillip Hallam-Baker, Ronald Monzill, "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)" sepcification, `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf` (accessed January 5, 2009)

[x509] Anthony Nadalin, Chris Kaler, Ronald Monzillo, Phillip Hallam-Baker, "Web Services Security X.509 Certificate Token Profile 1.1", `http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-x509TokenProfile.pdf` (accessed February 2, 2009)