[3] Web Services

1. Lecture notes
2. Book notes

Web Services and Java Concurrency - 29 September 2020

Threads & Synchronization (7.4)

When your program needs to do more than one thing at the same time. You often want to do things faster.

Threads

- One process can spawn many threads (may correspond to cores)

A process can create threads.

- Threads execute in parallel
- Threads can access memory and resources
- Problems? Race Condition and Starvation

Since they are accessing the same memory - sometimes things can go wrong...

- Without Synchronization - random results

Synchronization issue - Race Conditions

- Problems? Race Condition
- Solution: **Use locks**

While i am working on it - no other thread can use it.

Dead locks can happen tho..

- Thread unsafe: shared resource **without** lock

The example with two people k

- Thread safe: shared resource is synchronised - only thread with lock can enter
- In Java: special data types - atomic variables and **synchronized** collections.

Synchronization issue - Starvation

- Problem? Starvation
- How do we make sure have a fairness condition?
- How do we make sure all threads get access to resources?
- Solution: randomisation, priorities, timeouts

Indirect communication (6.1, 6.3 - 6.4)

Holy Grail of System Design

- **Goal:** High Cohesion and Low Coupling

It means that basically you have entities in Distributed System - and it is always hard to makes things communicate and there is always space and time issues,

because they don't always communicate in the same system.

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Time decoupling : you don't know if the reciever exists

Space decoupling : You don't know the recievers' identity

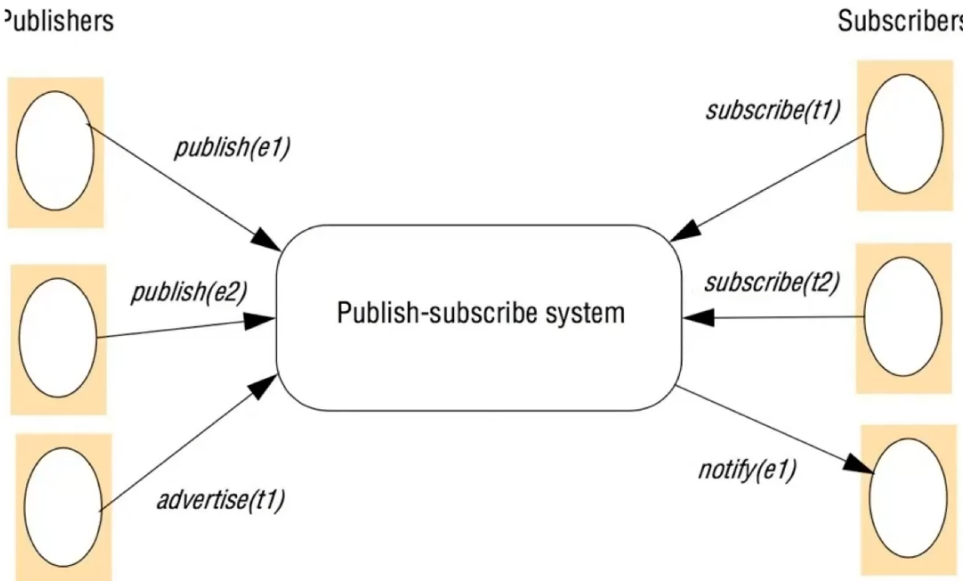
Publish/subscribe

Pub/sub programming model:

- **Publisher**: publish(e)
- **Subscriber**: subscribe(f) / unsubscribe(f) (I want to listen...)
- **Subscriber:** notify(e)
- **Publisher:** advertise(f) / unadvertise(f)

Think of Twitter...

People publish their tweet, people subscribe to the tweet - people who is subscribed get notified of new tweets.



The system will take care of the notification.

Implementations:

- Centralised - client/server.

This is always the easiet goto when working with distributed system.

The problem is: If the server crashes, you are doomed. It doesn't scale.

Like twitter, you want to scale.

- Multicast
- Overlay networks

Summary

- Time-coupled

Sender and receiver must exist at the same time.

- Space-coupled

Sender and receiver must know each other

- Publish-subscribe

API, Central server-multicast-overlay, Twitter

Remote invocation (5.1-5.2)

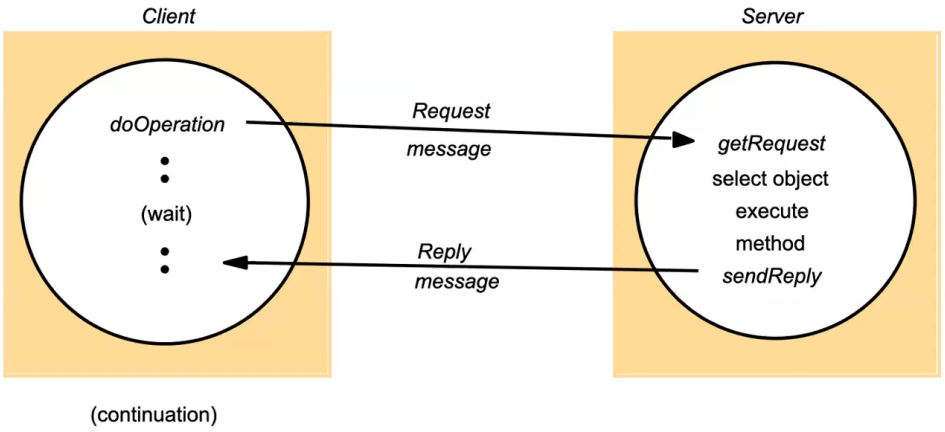
Request-reply

You use it every day. You have a client or is the client and ask the server - and the server replies back.

```
public byte[] doOperation (RemoteRef s int operationId, byte[] arguments)
/* sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked
and the arguments of that operation. */

public byte[] getRequest();
// acquires a client request via the server port.

public void sendReply(byte[] reply, InetAddress clientHost, int clientPort);
// sends the reply message reply to the client at its Internet address and port.
```



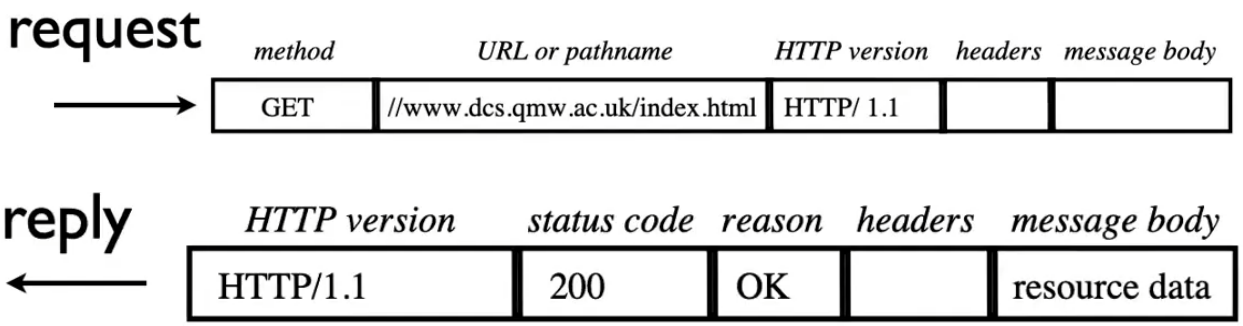
Why is RR not trivial?

Omission failures (req & rep), out-of-order messages:

Remember at the exam:

💡 **Distributed Systems are great but the most important thing we always worry about is FAILURES.**

HTTP Principle:



Different HTTP Method

Application Task	HTTP Method
Update	POST: sending data to a specified URI
Read	GET: To retrieve a resource from specified URI
Create	PUT: To store a resource at specified URI
Delete	DELETE: To delete a resource

- **GET** (and **HEAD**) supposed to be safe, i.e. no side-effects.
- **GET, PUT, DELETE (, HEAD** and **OPTIONS)** supposed to be idempotent, i.e. no additional effect if repeated.

Remote Procedure Call (RPC)

- Transparently call remote procedures.

Method is basically making a procedure call - but maybe this procedure is not available on the local and only on the server-side.

- What are the problems?

The problems you inherit from the Request-reply, things can go wrong.

- Interface

The procedures are implemented somewhere else - this is how we can use procedures from other places than local.

RPC Goals

- Transparency through failure masking

If there is a failure, what do you do?

- Can we have this? Can it be completly transparent?

No, we can't be completly transparent.

RPC Semantics & failures

- Maybe semantics

Easiest to implement. I made a procedure call - maybe I get a result or not.

- At-least-once semantics

You are making a request and it is been recieved by the server. You know for a fact that the server has recieved AT LEAST once.

- At-most-once semantics

Here we want to make sure, that the request have been recieved by the server at MOST ONCE.

RPC implementation

- Request-reply

Remote method invocation (RMI)

- Object references
- Interfaces
- Actions
- Exceptions
- Garbage collection

You may send information to the server - but the server has to know the type, the object. Hard to implement. Object references, you'll have to deal with cloning.

Summary

- Request-reply
- HTTP
- RPC

Webservices (WS and REST) (9.1-9.4 + IBM article)

SOAP Web Services

APIs for the Web? HTTP, but HTTP is not really sufficent for doing everything we want to do

What & why

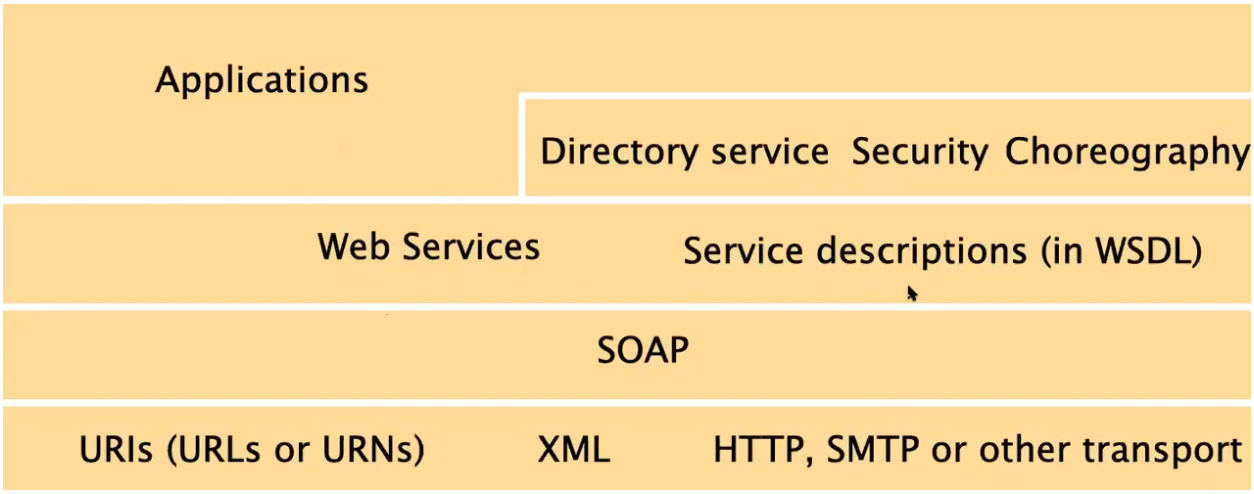
- HTTP as request-reply mechanism
- URI = URL + URN

(Identifier, Locator, Name)

- Operation descriptors

- Textual representations

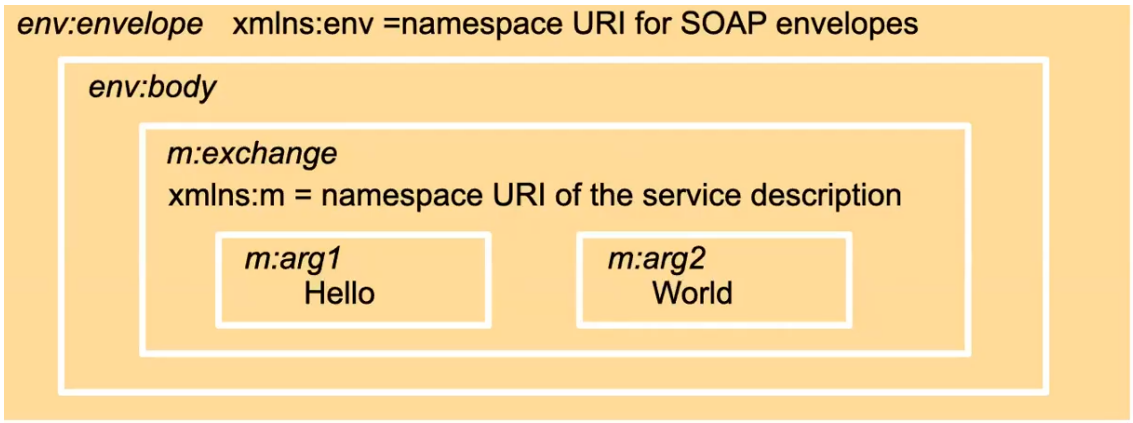
How do I represent data? Basically, like XML: Fix conventions for the obvious.



Middleware: SOAP which provides operations and ways for these above web services, to exists.

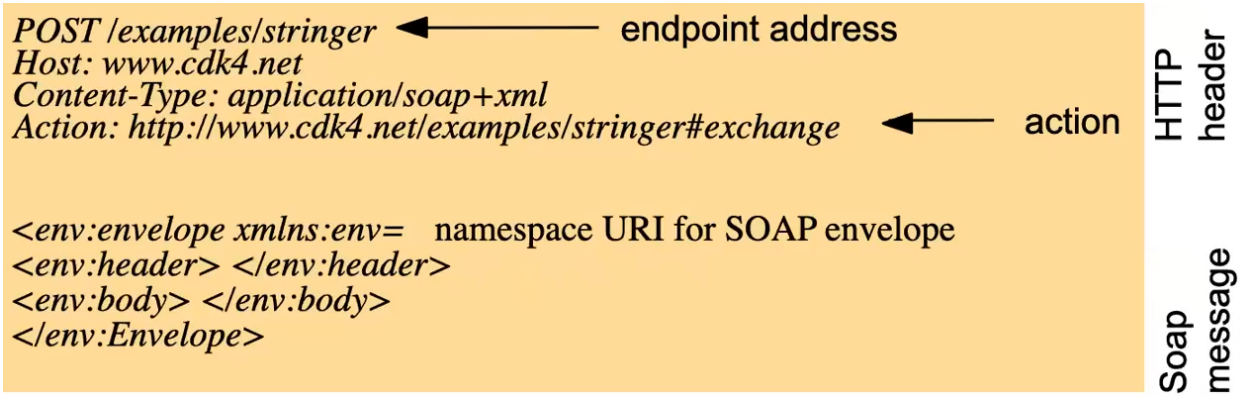
using XML to represent data and URIs for basically locating these services.

It is nice that it uses XML to represent data but also the bad things. XML became popular in the end of zeros'. Really there was a need for easy way to represent data.

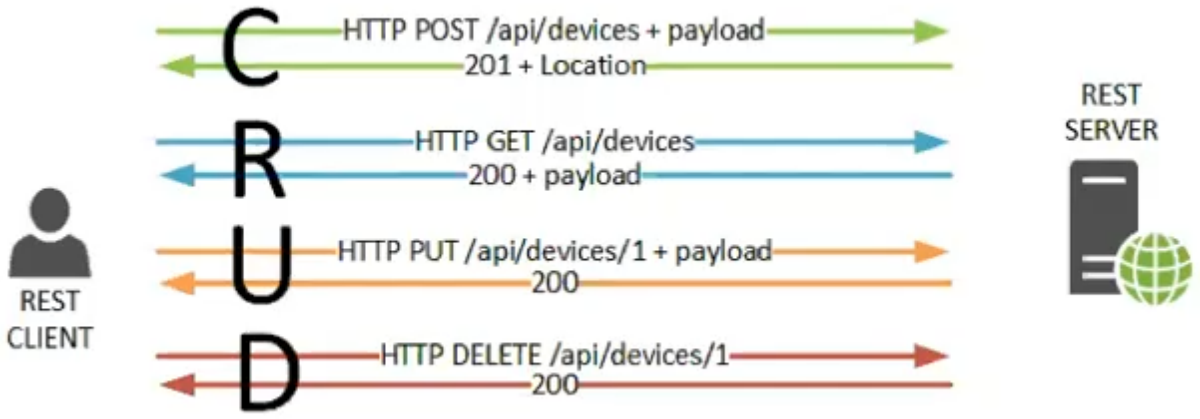


In this figure and the next, each XML element is represented by a shaded box with its name in italic followed by any attributes and its content

You have an env:envelope, that consist of a child body (env:body). Something that contains other things. The body has a message that you want to exchange and the message has two more children, two arguments you could say.



REST (REpresentational State Transfer)



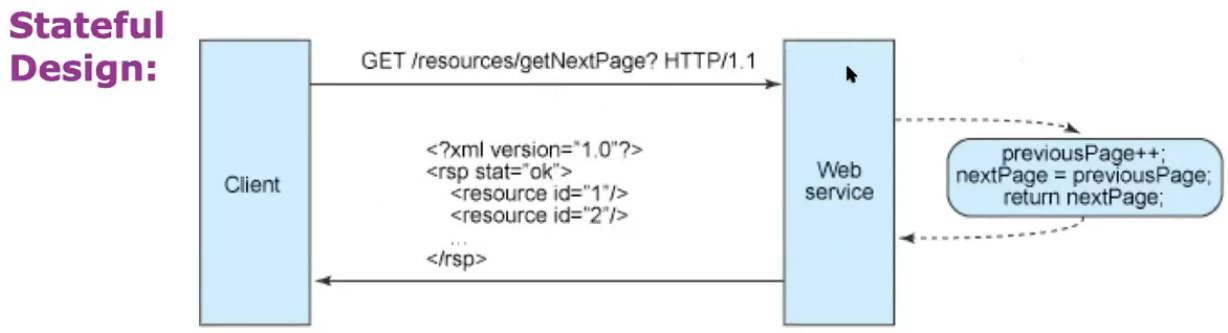
This Web Service idea of having a middleware - was a great idea - but not good for scalability.

- SOAP and WS-* turned to be somewhat a nightmare because of **scalability and evolution.**
- WWW runs fine on HTTP and evolves and scales **OK**.
- Lets use HTTP protocol as it is designed
- Use XML or JSON for representation
- See Open API example: http://www.petstore.swagger.io

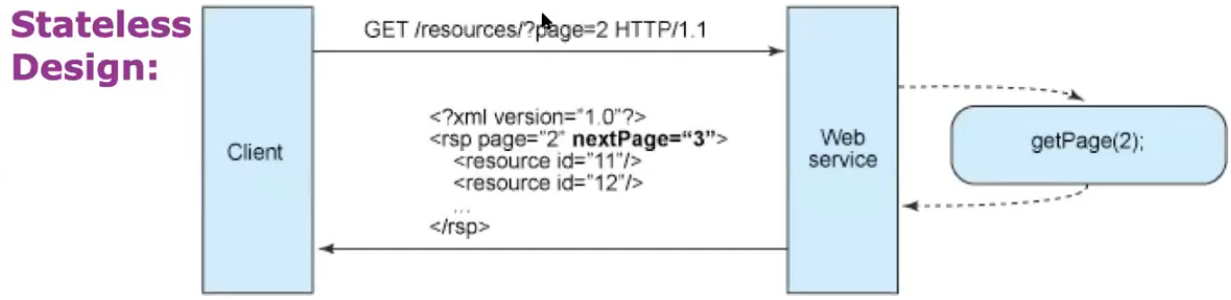
The REST idea is that you provide Open API

What about state?

REST is **not** stateful.



This is what ruins scalability ^ can be heavy on the server side. WebService keeps track of what page you are on.



The solution is stateless design. You declare in your request what page you wish to get and WebService simply get that page - so the server side does not have to remember.

9.1 Introduction

> 💡 **Web Services** provide an infrastructure for maintaing a *richer* and more *structured* form of interoperability* between clients and servers.

*__interoperability__: the ability of computer systems or software to exchange and make use of information.

💡 **Web Services** provide a *basis* whereby a client program in one organization may interact with a server in another organization without human supervision.

💡 **Web Services** allow complex applications to be developed by providing services that integrate several other services.

💡 **Web Services** generality of interactions results in that Web Services cannot be accesed directly by browsers.

💡 **Web Server** provides a basic HTTP service

💡 **Web Service** provides a service based on the operations defined in its interface.

💡 **Web Services** are an extension of the Web and can be provided by Web Servers.

💡 **Web Services** generally provides a **Service Description**, which includes an interface definition and other information, such as the server's **URL**. This is used as the basis for a common **understanding between client** and **server** as to the service on offer.

9.2 Web services

💡 What is a **Web Service** ? An interface that generally consists of a **collection of operations** that can be used by a client over the Internet.

💡 You can **combine Web Services** - providing an interface for a web service allows its operations to be combined with those of other services to provide new functionality.

💡 In general, **web services either** use a synchronous request-reply pattern of
communication with their clients **or** communicate by means of asynchronous messages.

💡 **Loose coupling** is a considerable interest in distributed systems - paticulary in the Web Services Community. Loose coupling refers to minimizing the dependencies between services in order to have a flexible underlying architecture (reducing the risk that a change in one service will have a knock-on effect on other services). This is partially supported by the intended independence of web services with the subsequent intention to produce combinations of web services as discussed above.

Loose coupling is further enhanced by a number of additional features:

- Programming with interfaces provides one level of loose coupling by serparating the interface from its implementation (and also supports

important areas of heterogeneity, - for example in the choice of programming language and platform used).

- There is a trend towards simple, generic interfaces in distributed systems and this is exemplified by the minimal interface offered by the World Wide Web and the REST approach in Web Services.
- As mentioned above, web services can be used with a variety of communication paradigms, including request-reply communication, asynchronous messaging or indeed indirect communication paradigms. The level of coupling is **directly affected by this choice**.

For example: In request-reply communication, the two parties are intrisically *(in an essential or natural way.)* coupled

Asyncrhonous messaging offers a degree of cecoupling, whereas indirect communication also offerst time and space uncoupling.

💡 In **Conclusion:** there are a number of dimensions to loose coupling, and it is important to
bear this in mind when using the term. Web services intrinsically support a level of loose
coupling due to the design philosophy adopted and the programming with interfaces
approach used. This can be further enhanced by additional design choices, including the
adoption of the REST approach and the use of indirect communication.

💡 **Representation of message:** Both SOAP and the data it carries are **represented in XML**, a textual self-dscribing format introduced in Section 4.3.3. Textual representations take up more space than binary ones and the parsing that they require takes more time to process. In document-style interactions speed is not an issue, but it is important in reuqest-reply interactions.

REST (Representational State Transfer)

REST is an approach with a **very constrained style of operation**, in which clients use **URLs** and the **HTTP** operations **GET**, **PUT**, **DELETE** and **POST** to manipulate resources that are represented in XML. The emphasis is on the manipulation of data resources rather than on interfaces. When a new resource is created, it has a new URL by which it can be accessed or updated. Clients are supplied with the entire state of a resource instead of calling an operation to get some part of it. Fielding argues that in the context of the Internet, the proliferation of different service interfaces will not be as useful as a simple minimum uniform set of operations. 80% uses REST 20 % uses SOAP - of requests in Amazon.com

Service references - In general, each web service has a URI, which clients use to refer to it. The URL is the most frequently used form of URI. Because a URL contains the domain name of a computer, the service to which it refers will always be accessed at that computer. However, the access point of a web service with a URN can depend on context and can change from time to time -its current URL can be obtained from a URN lookup service. This service reference is known as an **endpoint** in webservices.

Activation of services - A web service will be accessed via the computer whose domain name is included in its current URL. That computer may run the web service itself or it may run it on another server computer. For example, a service with tens of thousands of clients may need to be deployed on hundreds of computers. A web service may run continuously, or it may be activated on demand. The URL is a persistent reference, meaning that it will continue to refer to the service for as long as the server the URL points to exists.

💡 **Transparency -** A major task of many middleware platforms is to protect the programmer from the details of data representation and marshalling; another is to make remote invocations look like local ones. None of these things are provided as a part of an infrastructure or middleware platform for web services. At the simplest level, clients and servers may read and write their messages directly in SOAP, using XML. But for convenience, the details of SOAP and XML are generally hidden by a local API in a programming language such as Java, Perl, Python or C++. In this case, the service description may be used as a basis for automatically generating the necessary marshalling and unmarshalling procedures.

Proxies

One way to hide the difference between local and remote calls is by providing a client proxy or a set of stub procdures. Client proxies or stubs provide a static form of invocation in which the framework for each call and the marshaling procedures are generated before any invocations are made.

Dynmaic invocation

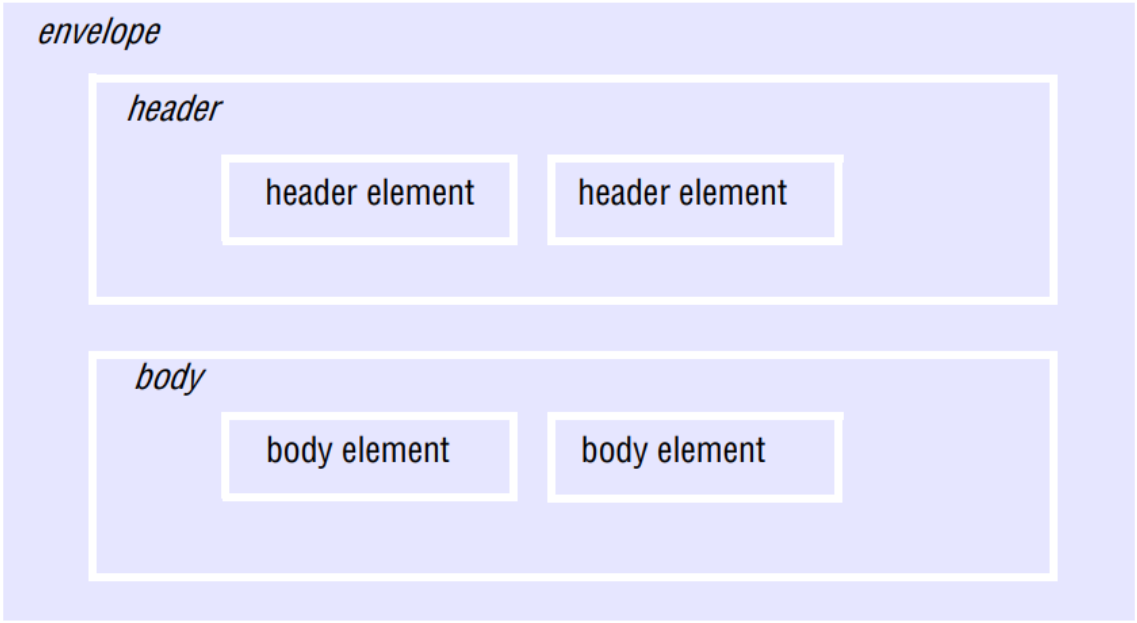
An alternative to proxies ^ is to provide clients with a generic operation to be used irrespective of the remote procedure to be called. In this case, the client specifies the name of an operation and its arguments and they are converted to SOAP and XML on the fly. The asynchronous communication of a single messages can be achieved in a similar way by providing clients with generic operations for sending and receiving messages.

SOAP

SOAP is designed to enable both client-server and asynchronous interaction over the Internet. It defines a scheme for using XML to represent the contents of

request and reply messages as well as a scheme for the communication of documents. **Originally** SOAP was based only on HTTP, but the current version is designed to use a variety of transport protocols including SMTP, TCP or UDP.

SOAP message in an envelope



The **SOAP** specification states:

- how XML is to be used to represent the contents of individual messages;
- how a pair of single messages can be combined to produce a request-reply pattern;
- the rules as to how the recipients of messages should process the XML elements that they contain;
- how HTTP and SMTP should be used to communicate SOAP messages. It is expected that future versions of the specification will define how to use other transport protocols, for example, TCP.

💡 **SOAP messages -** A SOAP message is carried in an 'envelope'. Inside the envelope there is an optional header and a body, as shown in the Figure above.

💡 **Message header** can be used for establishing the necessary context for a service or for keeping a log or audit of operations. An intermediary may interpret and act on the information in the message headers, for example by adding, altering or removing information.

💡 **Message body** carries an XML document for a particular web services.

The XML elements

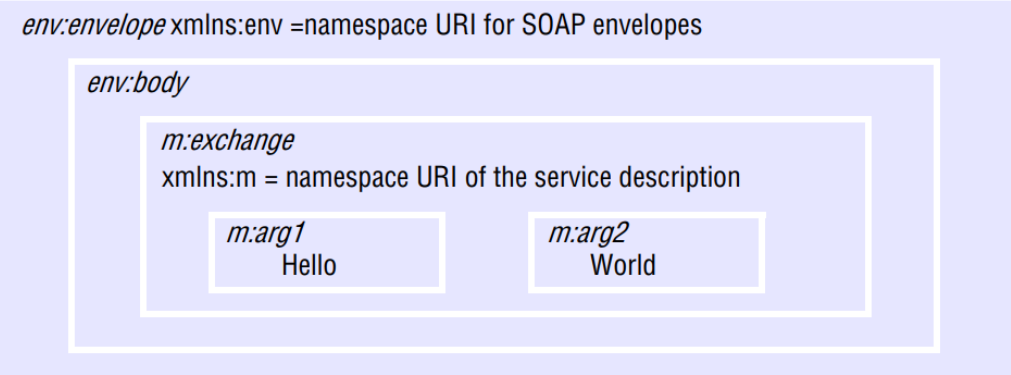
- envelope
- header
- body
- ... and some other attributes

A SOAP message may ...

A SOAP message may be used either to convey a document or to support clientserver communication:

- A document to be communicated is placed directly inside the *body* element together with a reference to an XML schema containing the service description - which defines the names and types used in the document. This sort of SOAP message may be sent either synchronously or asynchronously.
- For client-server communication, the *body* element contains either a *Request* or a *Reply*. These two cases are illustrated in Figure 9.4 and Figure 9.5

Figure 9.4 Example of a simple request without headers



In this figure and the next, each XML element is represented by a shaded box with its name in italics,at the top left corner,followed by any attributes and its content

Figure 9.4 (above) shows an example of a simple request message without a header. The *body* encloses an element containing the name of the procedure to be called and the URI of the namespace (the file containing the XML schema) for the relevant service description, which is denoted by *m*. The inner elements of a request message contain the arguments of the procedure. This request message provides two strings to be returned in the opposite order by the procudere at the server. The XML namespace denoted by *env* contains the SOAP definitions for an *envelope*.

Figure 9.5 Example of a reply corresponding to the request in Figure 9.4

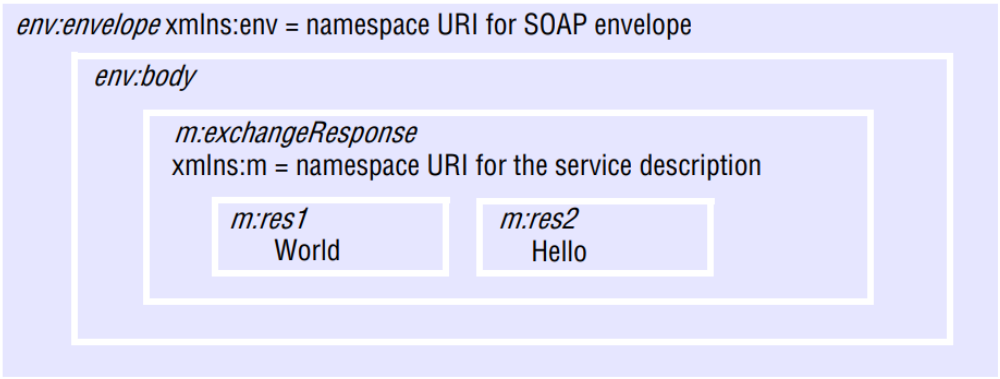


Figure 9.5 (above) shows the corresponding successful reply message, which contains the two output arguments. Note that the name of the **procedure has 'Response'** added to it. If a procedure has a return value, then it may be denoted as an elemnt called *rpc:result*. The reply message uses the same two XML

schemas as the request message, the first defining the SOAP envelope and the second the application-specific procedure and argument names.

SOAP faults: If a request fails in some way, the fault descriptions are conveyed in the body of a reply message in a *fault* element. This element contains information about the fault, including a code and an associated string, together with application-specific details.

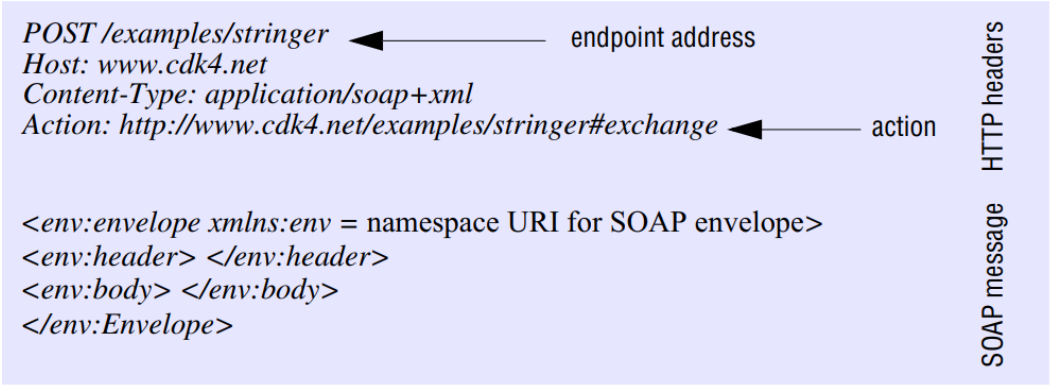
SOAP headers

Message header are intended to be used by intermediaries to add to the service that deals with message carried in the corresponding body. However, two aspects of this usage are left unclear in the SOAP specification

1. How the headers will be used by any particular higher middleware service. For example, a header might contain:
   - a transaction identifier for use with a transaction service;
   - a message identifier for relating messages to one another, for example, for implementing reliable delivery;
   - a username, a digital signature or a public key.
2. How the messages will be routed via a set of intermediaries to the ultimate recipient. For example, a message transported by HTTP could be routed via a chain of proxy servers, some of which might assume a SOAP role.

The transportation of SOAP messages

Figure 9.6 Use of HTTP POST Request in SOAP client-server communication



A transport protocol is required to send a SOAP message to its destination. SOAP messages are **independt** of a the type of transport used - their envelopes contain no reference to the destination address.

- The HTTP headers specify the endpoint address (the URI of the ultimate receiver) and the action to be carried out. The *Action* header is inteded to optimize dispatching by revealing the name of the operation without the need to analyze the SOAP message in the body of the HTTP message.
- The HTTP body carries the SOAP message.

As **HTTP is a synchronous protocol**, it is used to return a reply containing the SOAP reply, like the one shown in Figure 9.5. Section 5.2 details the status codes and reasons returned by HTTP for successful and failing requests

If a SOAP Request **is just a request for information to be returned**, has no
arguments and does not alter data in the server, then the HTTP GET method can be used
to carry it out.

SOAP message to be delievered *at-least-once, at-most-once* or *exactly-onc*, with the following semantics:

- *At-least-once:* The message is delievered at least once, but an error is reported if it cannot be delivered.
- *At-most-once*: The message is delivered at most once, but without any error report if it cannot be delivered.
- *Exactly-once*: The message is delivered exactly once, but an error is reported if it cannot be delivered.

Ordering of messages is also provided in combination with any of the above:

- *In-order*: Messages will be delivered to the destination in the order in which they were send by a particular sender.

Something short about Firewalls:

"However, firewalls do normally allow both HTTP and SMTP messages to pass through them. Therefore it is convenient to use one of these protocols for transporting SOAP messages."

A comparison of web services with CORBA

The main difference between web services and CORBA or other similar middleware is the inteded usage context. CORBA was designed for use within a signle organization or between a small number of collaborating organizations. This resulted in certain aspects of the design being too centralized for collaborative use by independent organizations or for **ad hoc** use without prior arrangements. Read 9.2.4 to see the differences.