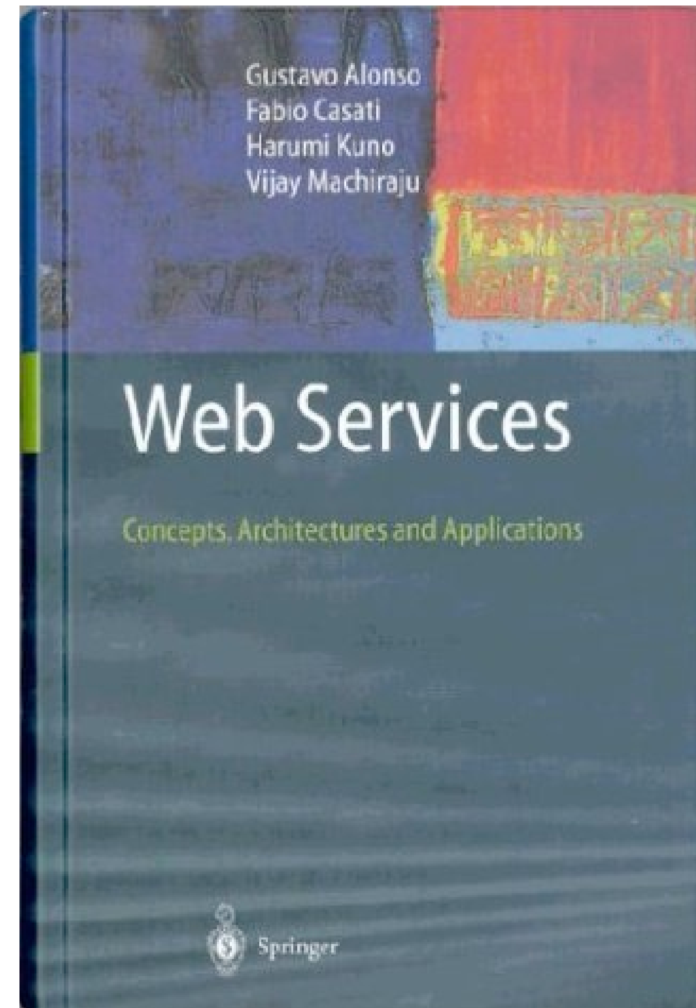
Standard Software for Enterprise Resource Planning

Lecturer: Prof. Dr. Ralf Möller
Lab classes: Rainer Marrone, Michael Wessel

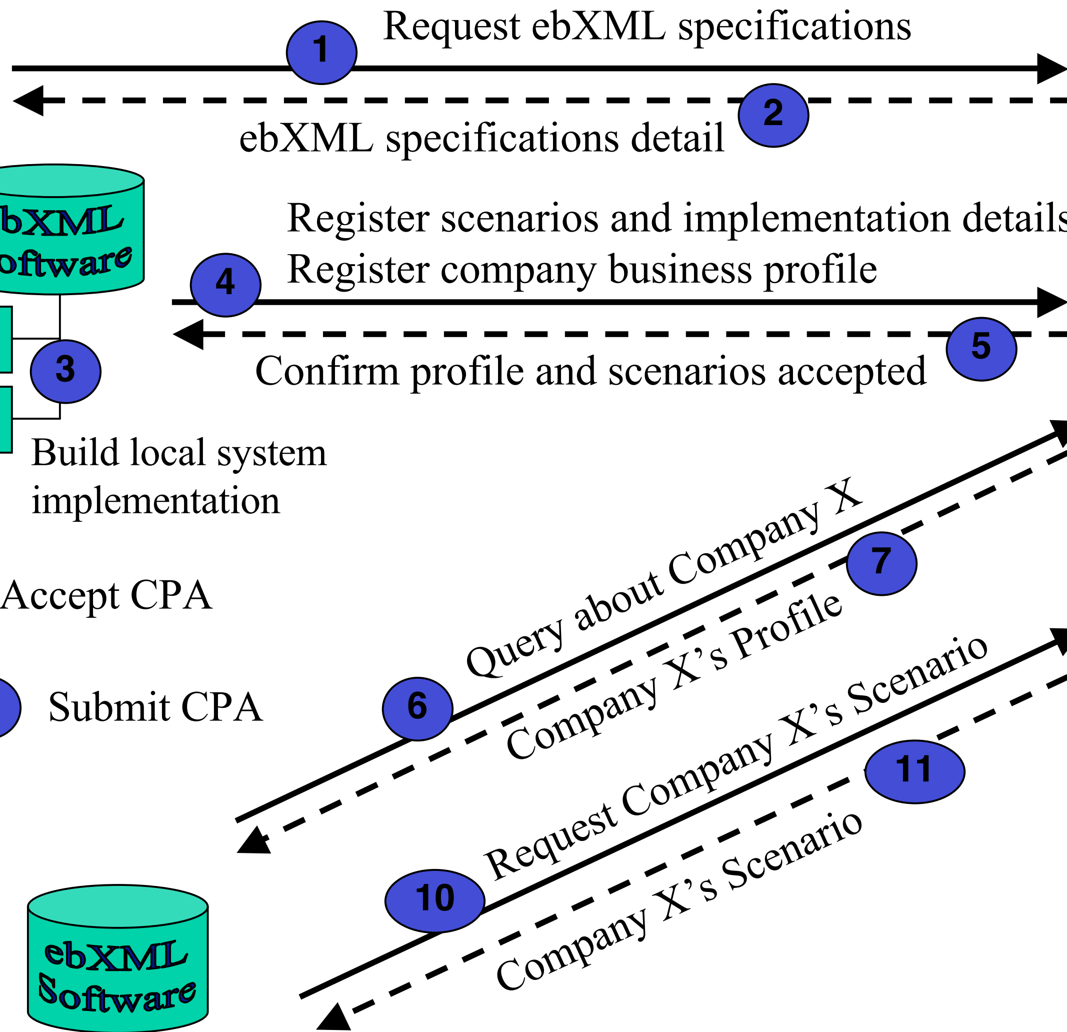
Lecture: Thursdays (90 minutes)
Lab classes: Fridays (60 minutes)

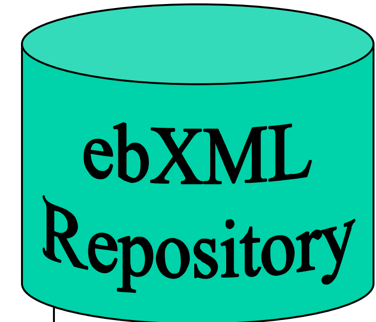
Prerequisite:
Lecture on ECommerce

This lecture is based on:



Recap: ebXML





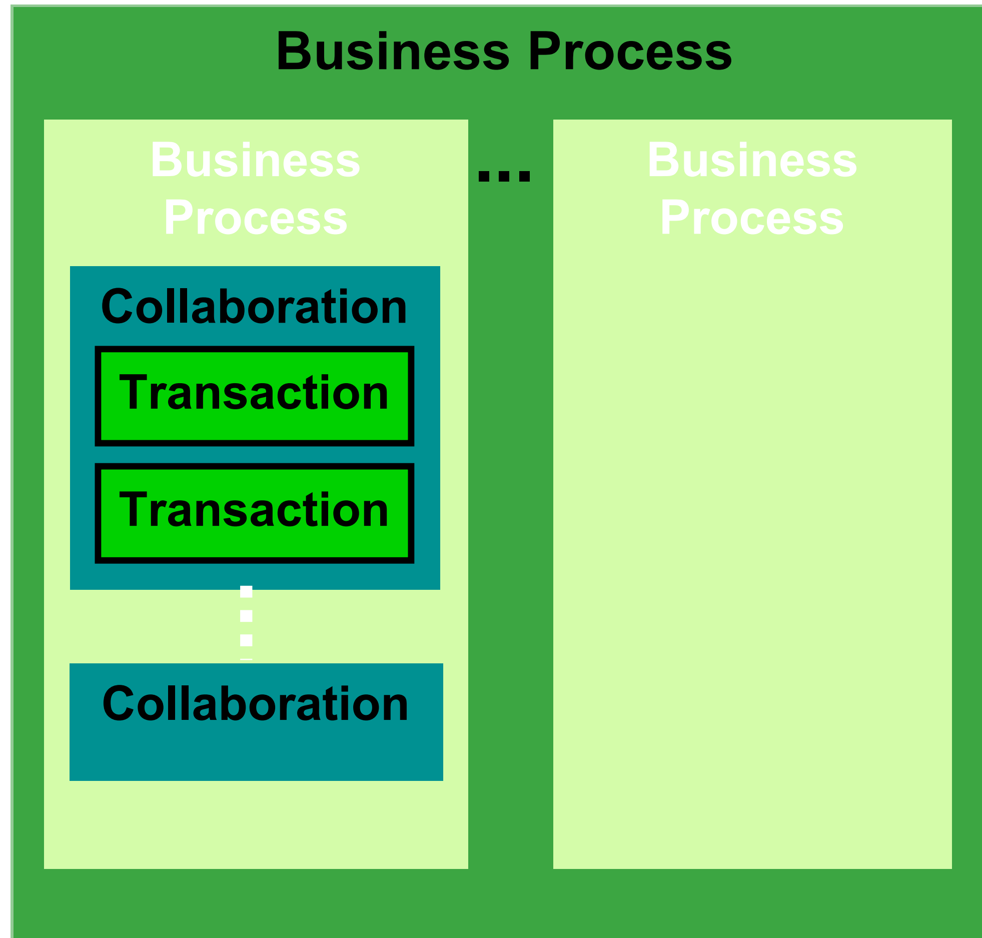
Company Profile

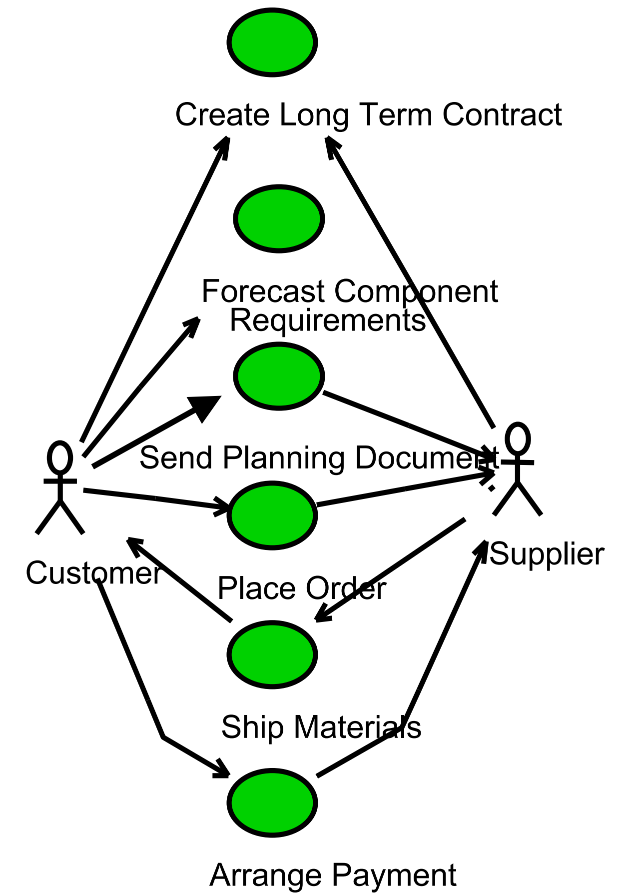
- Collaboration Protocol Profile
  - Defined using ebXML Specification Schema
  - Concrete specification of your ebusiness offerings
    - Business scenarios you support
    - Service interfaces you implement
    - Document formats exchanged
    - Technical requirements/options (protocols, security, reliability)
- Composed of
  - Business process models
  - Information models
  - Context rules

Business Scenarios

- Often defined by Industry Groups
  - ➡ Standard business scenarios remove the need for prior agreements among trading partners
- Business Process Model
  - ➡ Interactions between parties
  - ➡ Sequencing of interactions
  - ➡ Documents exchanged in each interaction
- Information Model
  - ➡ Document definition
  - ➡ Context definition
  - ➡ Context rules

Business Process





Processing an XML Document

```
<?xml version="1.0"?>
<People>
  <Person>
    <Name>
      <First>Patrick</First>
      <Last>Joe</Last>
    </Name>
  </Person>
</People>
```

XML Content

```
<!ELMENT People (Person)* >
<!ELMENT Person (Name) >
<!ELMENT Name (First, Last) >
<!ELMENT First (#PCDATA) >
<!ELMENT Last (#PCDATA) >
```

DTD Structure

An application that wishes to use XML data should parse the data using a DTD.

There are two approaches:

- **DOM** – read the entire document using the DTD and build a tree of elements
- **SAX** – parse on demand (interactively) as each new element is encountered

DOM = Document Object Model

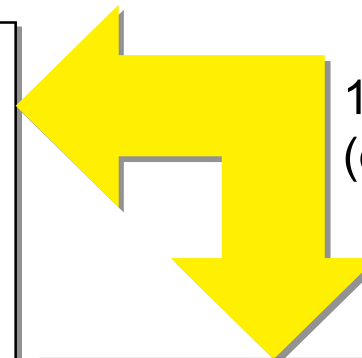
SAX = Simple API for XML

XML-Schema

- More expressive than DTDs
- Uses XML as the syntactic basis

```
<schema>
 <element name="bib">
  <complexType>
   <element name="paper" minOccurs="0" maxOccurs="unbounded">
    <complexType>
     <attribute name="id" type="ID" use="required"/>
     <sequence>
      <element name="author" type="authorType"
       maxOccurs="unbounded"/>
      <element name="year" type="string"/>
      <element name="publisher" type="string" minOccurs="0"/>
     </sequence>
    </complexType>
   </element>
  </complexType>
 </element>
</schema>
```

XML-Schema



1:1-Mapping
(except for author)

```
<!DOCTYPE bib [
  <!ELEMENT bib (paper*)>
  <!ELEMENT paper (author+, year, publisher?)>
  <!ATTLIST paper id ID #REQUIRED>
  <!ELEMENT author (firstname*, lastname)>
  <!ATTLIST author age CDATA #IMPLIED>
  <!ELEMENT firstname (#PCDATA)>
  <!ELEMENT lastname (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
  <!ELEMENT publisher (#PCDATA)>
  ...
]>
```

DTD

XML-Schema: Elements

- Syntax:
  <element name=“Name“/>
- Optional attributes:
  - Type
    - type = “Type“
  - Cardinalities (Default [1,1]):
    - minOccurs = “x“ x ∈ { 0, 1, n }
    - maxOccurs = “y“ y ∈ { 1, n, unbounded }
  - Defaults:
    - default = “v“ can be overwritten
    - fixed = “u“ cannot be overwritten
- Examples:
  - <element name=“bib”/>
  - <element name=“paper” minOccurs=“0” maxOccurs=“unbounded”/>
  - <element name=“publisher” type=“string” minOccurs=“0”/>

XML-Schema: Attributes

- Syntax:
  <attribute name="Name"/>
- Optional attributes:
  - Typ:
    - type = "Type"
  - Existence:
    - use = "optional" Cardinality [0,1]
    - use = "required" Cardinality [1,1]
  - Defaults:
    - use = "default" value = "v"
    - use = "fixed" value = "u"
  - Examples:
    - <attribute name="id" type="ID" use="required"/>
    - <attribute name="age" type="string" use="optional"/>
    - <attribute name="language" type="string" use="default" value="de"/>

XML-Schema: Types

- XML-Schema distinguishes between atomic, simple and complexen types
- Atomic types:
  - Built-in types such as int or string
- Simple types:
  - Have neither embedded elements nor attributes
  - Usually derived from atomic types (”subranges“)
- Complex types:
  - May have elements and attributes
- The following distinctions are possible:
  - Type definitions describe (reusable) type structures
  - Document definitions describe which elements can occur where in a document instance

XML-Schema: Atomic Types

- XML-Schema a large number of atomic types (>40):
  - Numerical: byte, short, int, long, float, double, decimal, binary, …
  - Time-related: time, date, month, year, timeDuration, timePeriod, …
  - Additional: string, boolean, uriReference, ID, …
- Examples:

```
<element name=“year“ type=“year“/>
<element name=“pages“ type=“positiveInteger“/>
<attribute name=“age“ type=“unsignedShort“/>
```

XML-Schema: Simple Types

- Type derivation:
  - ➡ Type definition:
    <simpleType name="humanAge" base="unsignedShort">
    <maxInclusive value="200"/> </simpleType>
  - ➡ Document definition:
    <attribute name="age" type="humanAge"/>
- Simple type do not have nested elements!
- Lists can be defined as follows:
  - ➡ Type definition:
    <simpleType name="authorType" base="string"
    derivedBy="list"/>
    (Name of an author represented as a list of strings separated by blanks)
  - ➡ Document definiton:
    <element name="author" type="authorType"/>

XML-Schema: Complex Types

- Complex Types can have attributes as well as embedded elements
- Example:
  - Type definition:

```
<complexType name="authorType">
    <sequence>
        <element name="firstname" type="string" minOccurs="0"
            maxOccurs="unbounded"/>
        <element name="lastname" type="string"/>
    </sequence>
    <attribute name="age" type="string" use="optional"/>
</complexType>
```

- Group name:
  - <sequence> … </sequence> Fixed order (a,b)
  - <all> … </all> Arbitrary order (a,b oder b,a)
  - <choice> … </choice> Alternatives (entweder a oder b)

XML-Schema: Complex Types

```
<complexType name="authorType">
   <sequence>
      <element name="firstname" type="string"
               minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="lastname" type="string"/>
   </sequence>
   <attribute name="age" type="string" use="optional"/>
</complexType>
```

Type hierarchies

- Type definition by
  - Extension or
  - Restriction of an existing type definition
- All types in XML Schema are either
  - Atomic types (e.g., string) or
  - Extensions and restrictions, respectively
- All Types form type hierarchy
  - Tree with root: Type string
  - No multiple inheritance
- Types are upward-compatible along the type hierarchy:
  - Principle of Substitution
  - Element of a certain type accept data of an extension or retriction of that type

Type hierarchies: Extension of types

- Types can be extended to include new elements or attributes
- Example:

```
<complexType name=“extendedAuthorType“>
    <extension base=“authorType“>
        <sequence>
            <element name=“email“ type=“string“ minOccurs=“0“
                maxOccurs=“1“/>
        </sequence>
        <attribute name=“homepage” type=“string” use=“optional”/>
    </extension>
</complexType>
```

- Extends the previously defined type authorType by
  - ➡ adding an optional element email
  - ➡ adding an optional attribute homepage

Typhierarchies: Extensions of types (2)

- The extensions are added to the previous definitions:
- ```
  <complexType name="extendedAuthorType">
  <sequence>
      <element name="firstname" type="string" minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="lastname" type="string"/>
      <element name="email" type="string" minOccurs="0"
               maxOccurs="1"/>
  </sequence>
  <attribute name="age" type="string" use="optional"/>
  <attribute name="homepage" type="string" use="optional"/>
  </complexType>
  ```

```
<complexType name="authorType">
   <sequence>
       <element name="firstname" type="string" minOccurs="0"
                maxOccurs="unbounded"/>
       <element name="lastname" type="string"/>
   </sequence>
   <attribute name="age" type="string" use="optional"/>
</complexType>
```
```

Type hierarchies: Restriction of Types

- Types are restricted by increasing the constraints given in the definition of a previously defined type (i.e., the set of possible values is reduced)
- Examples:
  - Adding type, default, or fixed attributes
  - Adding more specific cardinalities minOccurs, maxOccurs
- Principle of Substitution
  - The set of instances of the restricted types must be a subset of the set of instances of the referenced type (supertype).
- Restriction of complex types
  - Structure remains identical, it is not possible to leave away (required) elements and attribures
- Restriction of simple types
  - Restriction is supported also for simple types (in contrast to extension)

Typhierarchies: Restriction of Types (2)

Previously: maxOccurs=“unbounded“

- Example (complex type):

```
<complexType name=“restrictedAuthorType“>
    <restriction base=“authorType“>
        <sequence>
            <element name=“firstname“ type=“string“ minOccurs=“0“
                maxOccurs=“2“/>
            <element name=“lastname“ type=“string“/>
        </sequence>
        <attribute name=“age” type=“string” use=“required”/>
    </restriction>
</complexType>
```

Previously : use=“optional“

- Compared to the original type the number of first names is restricted to 2 and the age attributes is required

Schema vs. DTDs

- Both are XML document definition languages
- Schemata are written using XML
- Unlike DTD’s, XML Schema are Extensible – like XML!
- More verbose than DTD’s but easier to read & write

XML : XML Namespaces

- The XML namespaces recommendation defines a way to distinguish between duplicate element type and attribute names.
- An XML namespace is a collection of element type and attribute names. The namespace is identified by a unique name, which is a URI.
- XML namespaces are declared with an xmlns attribute, which can associate a prefix with the namespace.

XML-Schema : A Simple XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<marketing
    xmlns:xsi = "http://www.w3c.org/2001/XMLSchema-Instance"
    xsi:noNameSpaceSchemaLocation = "http://www.Dot.com/mySchema.xsd">

  <employee>Gustav Sielmann</employee>
  <employee>Arnold Rummer</employee>
  <employee>Johann Neumeier</employee>

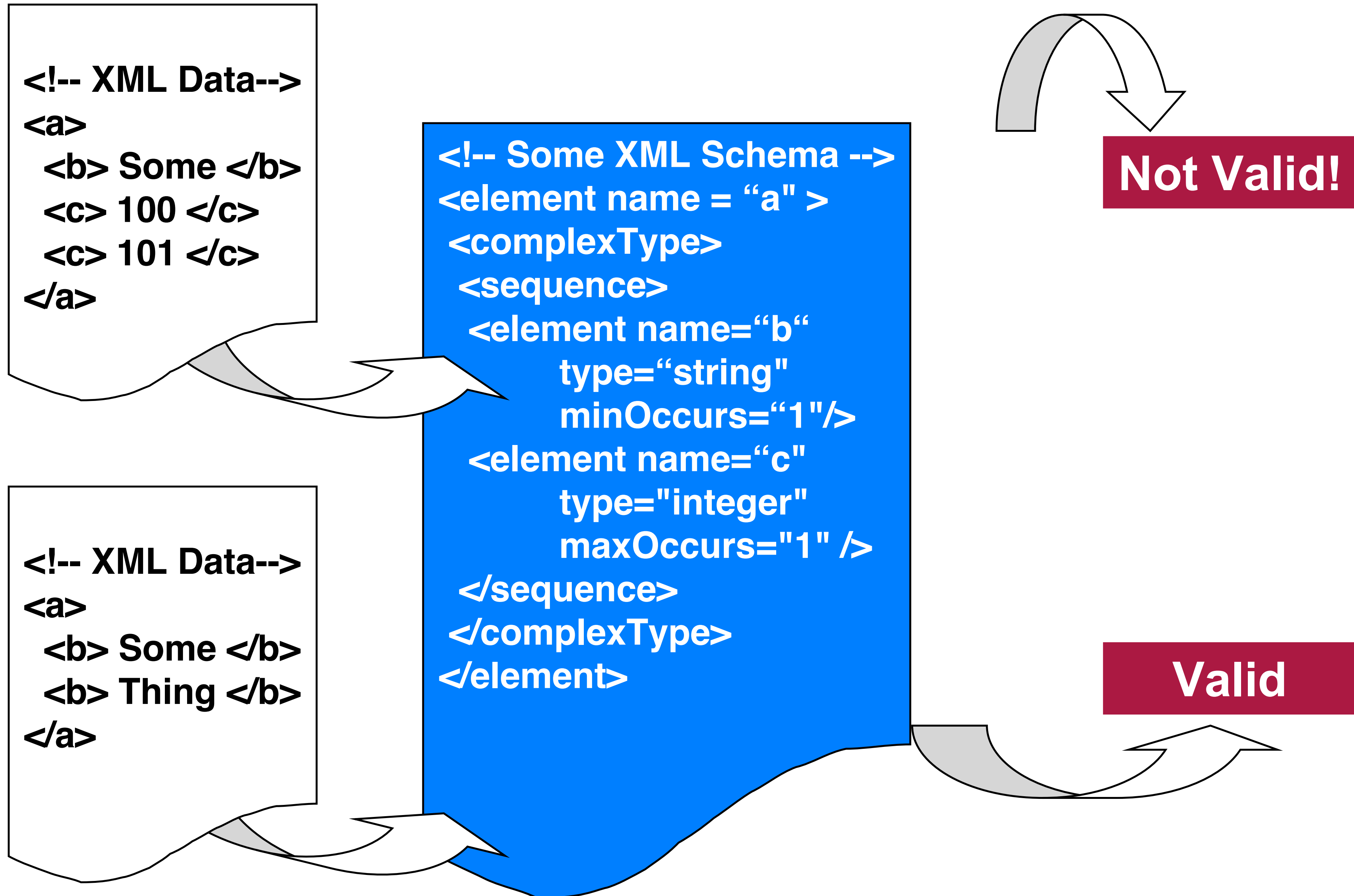
</marketing>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3c.org/2001/XMLSchema">

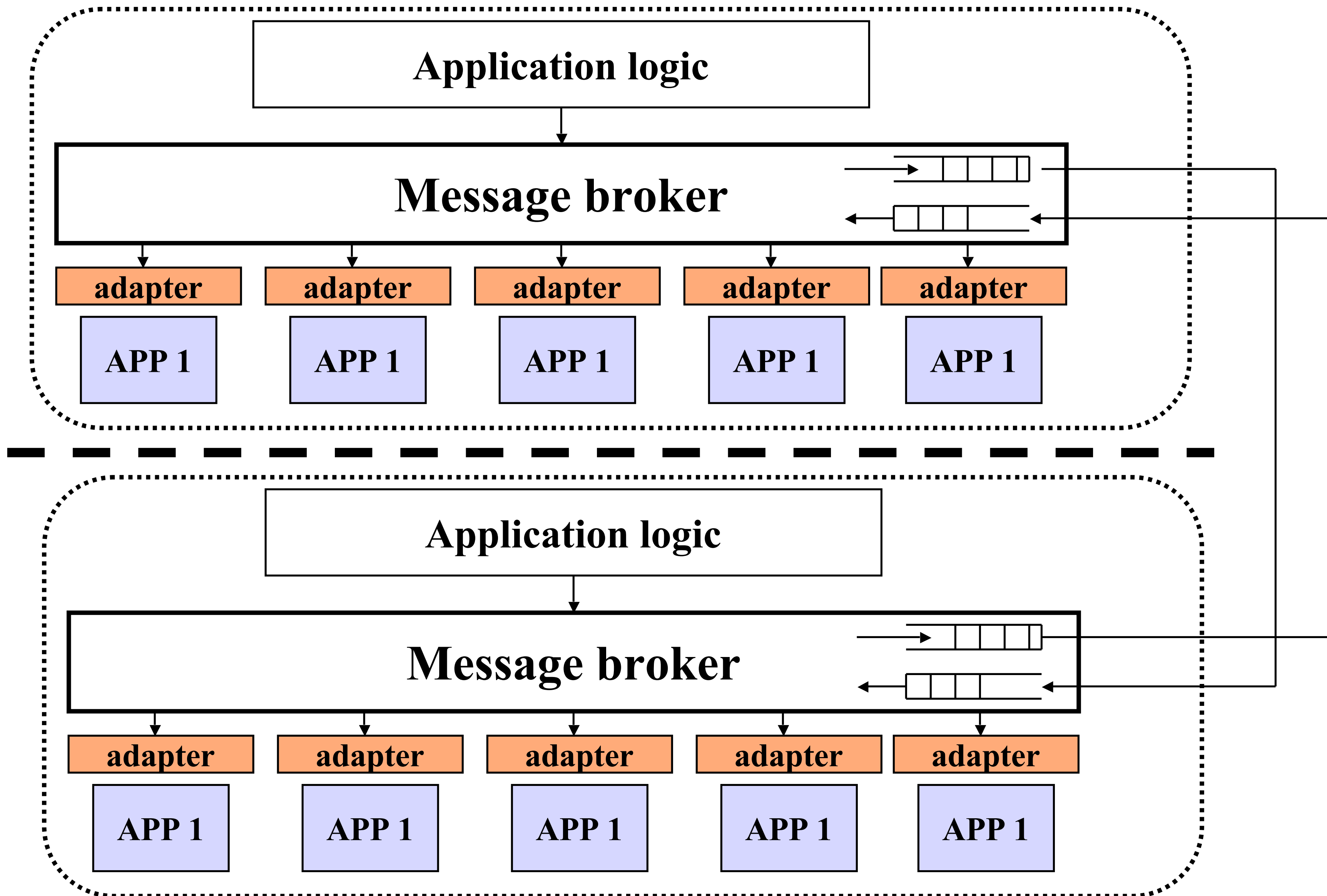
<xsd:element name = "marketing>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "employee" type = "xsd:string"
                   maxOccurs = "unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

XML Documents + XML Schema



Recap: Message Brokers



Web Services - Concepts, Architecture and Applications
Part 4: Internet Interaction (SOAP)

Gustavo Alonso, Cesare Pautasso
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
http://www.inf.ethz.ch/~alonso

What is SOAP?

- The Simple Object Access Protocol (SOAP) was initiated by W3C in 1999. SOAP 1.0 was entirely based on HTTP, and the following version, SOAP 1.1 (May 2000), was more generic since it included other transport protocols. The first draft of SOAP 1.2 was presented in July 2001 and was recently promoted to a “Recommendation”.
- SOAP covers the following four main areas:
  - ➡ A message format for one-way communication describing how a message can be packed into an XML document
  - ➡ A description of how (the XML document that makes up) a SOAP message should be transported through the Web (using HTTP) or e-mail (using SMTP).
  - ➡ A set of rules that must be followed when processing a SOAP message and a simple classification of the entities involved in that processing. It also specifies what parts of the messages should be read by whom and how to react in case the content is not understood
  - ➡ A set of conventions on how to turn an RPC call into a SOAP message and back as well as how to implement the RPC style of interaction (how the client side RPC call is translated into a SOAP message, forwarded, turned into a server side RPC call, the reply converted into a SOAP message and returned to the client)

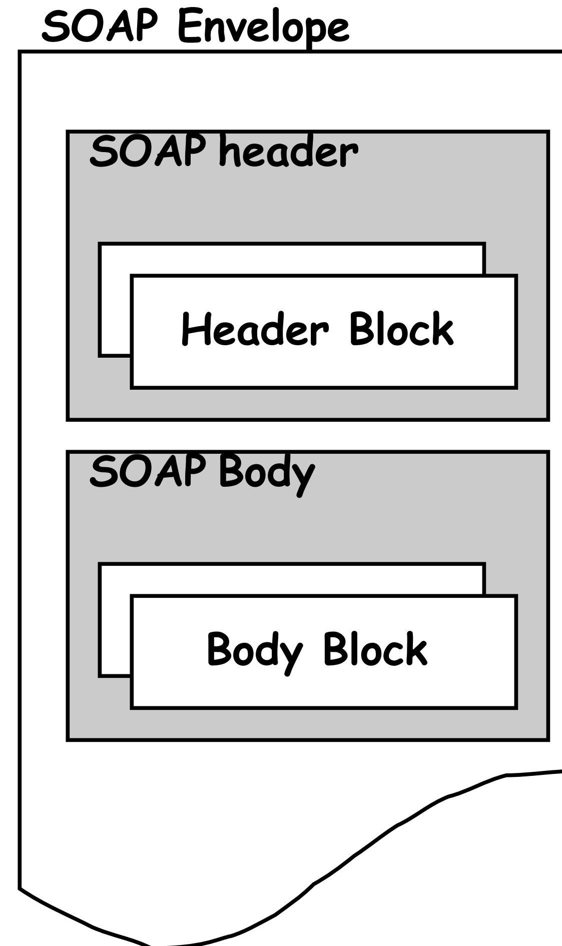
The background of SOAP

- SOAP was originally conceived as the minimal possible infrastructure necessary to perform RPC through the Internet:
  - ➥ use of XML as intermediate representation between systems
  - ➥ very simple message structure
  - ➥ mapping to HTTP for tunneling through firewalls and using the Web infrastructure
- The goal was to have an extension that could be easily layered on top of existing middleware platforms to allow them to interact through the Internet rather than through a LAN, as is typically the case.
- Hence the emphasis on RPC from the very beginning (essentially all forms of middleware use RPC at one level or another)
- Eventually SOAP started to be presented as a generic vehicle for computer driven message exchanges through the Internet and then it was open to support interactions other than RPC and protocols other than HTTP. This process, though, is still on-going.

Structure of a SOAP message

SOAP messages

- SOAP is based on message exchanges
- Messages are seen as envelopes where the application encloses the data to be sent
- A message has two main parts; header and body, which both can be divided into blocks
- SOAP does not specify what to do with the header and the body, it only states that the header is optional and the body is mandatory
- The use of header and body, however, is implicit. The body is for application level data. The header is for infrastructure level data



For the XML fans (SOAP, body only)

XML name space identifier for SOAP serialization

XML name space identifier for SOAP envelope

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <SOAP-ENV:Body>
            <m:GetLastTradePrice xmlns:m="Some-URI">
                <symbol>DIS</symbol>
            </m:GetLastTradePrice>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

From the: Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000

SOAP example, header and body

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    <SOAP-ENV:Header>
        <t:Transaction
            xmlns:t="some-URI"
            SOAP-ENV:mustUnderstand="1">
                5
        </t:Transaction>
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <symbol>DEF</symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

From the: Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000

The SOAP header

- The header is intended as a generic place holder for information that is not necessarily application dependent (the application may not even be aware that a header was attached to the message).
- Typical uses of the header are: coordination information, identifiers (for, e.g., transactions) and security information (e.g., certificates)
- SOAP provides mechanisms to specify who should deal with headers and what to do with them. For this purpose it includes:
  - ➥ SOAP role attribute (previously called “actor”): who should process that particular header entry (or header block). The “role” can be either: none, next, ultimateReceiver. ‘None’ is used to propagate information that does not need to be processed. ‘Next’ indicates that a node receiving the message can process that block. ‘ultimateReceiver’ indicates that the header is intended for the final recipient of the message
  - ➥ mustUnderstand attribute: with values true/false (previously 1/0), indicating whether it is mandatory to process the header. If a node can process the message (as indicated by the “role” attribute), the mustUnderstand attribute determines whether it is mandatory to do so.
  - ➥ New in SOAP 1.2 is also the relay attribute (forward header if not processed)

The SOAP body

- The body is intended for the application specific data contained in the message
- A body entry (or a body block) is syntactically equivalent to a header entry with attributes role = ultimateReceiver and mustUnderstand = 1
- Unlike for headers, SOAP does specify the contents of some body entries:
  - mapping of RPC to a collection of SOAP body entries
  - the Fault entry (for reporting errors in processing a SOAP message)
- The fault entry has four elements (in 1.1):
  - fault code: indicating the class of error (version, mustUnderstand, client, server)
  - fault string: human readable explanation of the fault (not intended for automated processing)
  - fault actor: who originated the fault
  - detail: application specific information about the nature of the fault

SOAP Fault element (v 1.2)

- In version 1.2, the fault element is specified in more detail. It must contain two mandatory sub-elements:
  - Code: containing a value (the code of the fault) and possibly a sub-code (for application specific information)
  - Reason: same as fault string in 1.1
    - This string can be provided in different languages:

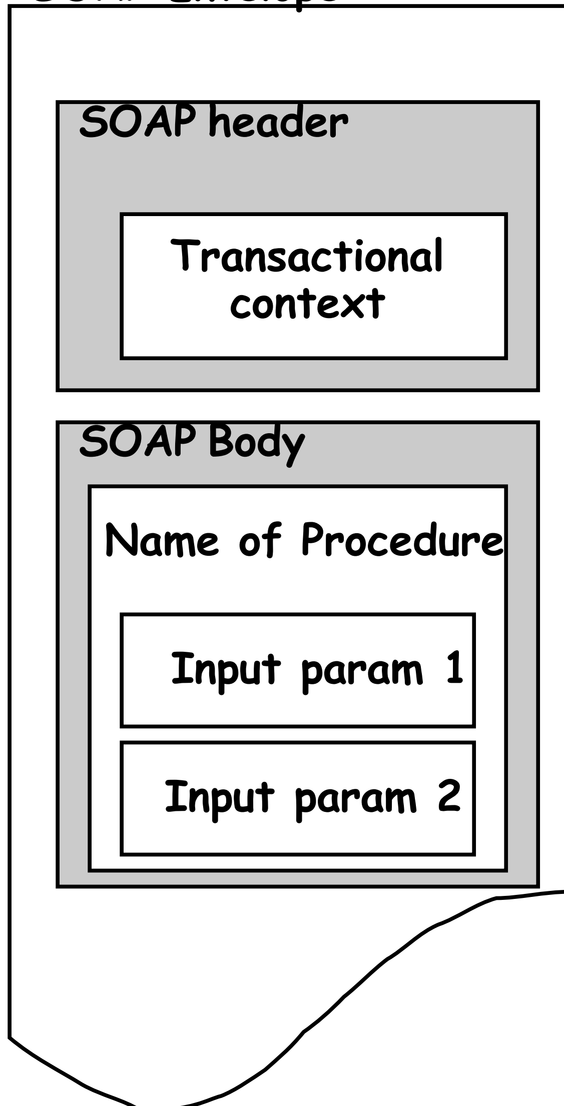
      <env:Text xml:lang="en-US">Header not understood</env:Text>

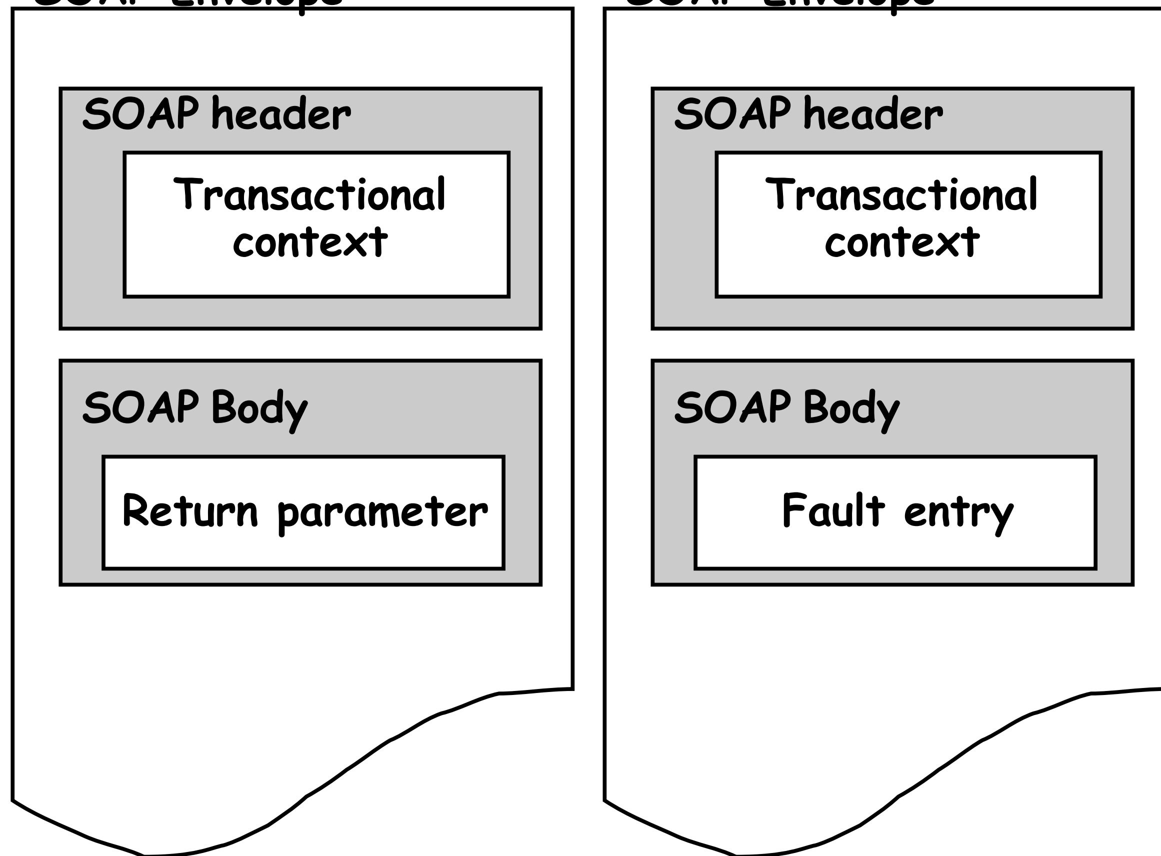
      <env:Text xml:lang="fr">En-tête non compris</env:Text>
- and may contain any of the following optional elements:
  - Node: the URI identifying the node producing the fault (if absent, it defaults to the intended recipient of the message)
  - Role: the role played by the node that generated the fault
  - Detail: as in 1.1
- Errors in understanding a mandatory header are responded using a fault element but also include a special header indicating which one of the original headers was not understood.

Message processing

- SOAP specifies in detail how messages must be processed (in particular, how header entries must be processed)
  - ➥ Each SOAP node along the message path looks at the role associated with each part of the message
  - ➥ There are 3 standard roles: none, next or ultimateReceiver (as mentioned above)
  - ➥ Applications can define their own message roles
  - ➥ The role determines who is responsible for each part of a message
- If a block doesn't have a role associated with it, it defaults to ultimateReceiver
- If a mustUnderstand flag is included, a node that matches the specified role <u>must</u> process that part of the message, otherwise it <u>must</u> generate a fault and not forward the message any further
- SOAP 1.2 includes a relay attribute. If present, a node that does not process that part of the message must forward it (i.e., it cannot remove the part)
- The use of the relay attribute, combined with the role next, is useful for establishing persistence information along the message path (like session information)

From TRPC to SOAP messages





Mapping SOAP to a transport protocol

HTTP as a communication protocol

- HTTP was designed for exchanging documents. It is almost like e-mail (in fact, it uses RFC 822 compliant mail headers and MIME types)
- Example of a simplified request (from browser):

```
GET  /docu2.html HTTP/1.0
Accept: www/source
Accept: text/html
Accept: image/gif
User-Agent:  Lynx/2.2  libwww/2.14
From:  eggs@spam.com
     * a blank line *
```

- Request methods: GET (retrieve data), POST (append information), PUT (send information), DELETE (remove information), ...

File being requested (docu2.html) and version of the protocol used

List of MIME types accepted by the browser

Information about the environment where the browser is running

E-mail or identifier of the user (provided by the browser)

End of request

HTTP server side

- Example of a response from the server (to the request by the browser):

HTTP/1.0 200 OK
Date: Wednesday, 02-Feb-94 23:04:12 GMT
Server: NCSA/1.1
MIME-version: 1.0
Last-modified: Monday, 15-Nov-93 23:33:16 GMT
Content-type: text/html
Content-length: 2345
* a blank line *
<HTML><HEAD><TITLE> . . . </TITLE> . . .etc.

Protocol version, code indicating request status (200=ok)

Date, server identification (type) and format used in the request

MIME type of the document being sent

Header for the document (document length in bytes)

Document sent

- The server is expected to convert the data into a MIME type specified in the request (“Accept:” headers)

Parameter passing

- The introduction of forms for allowing users to provide information to a web server required the modification of HTML (and HTTP), but it provided a more advanced interface than just retrieving files:

```
POST /cgi-bin/post-query HTTP/1.0
Accept: www/source
Accept: text/html
Accept: video/mpeg
Accept: image/jpeg
...
Accept: application/postscript
User-Agent:  Lynx/2.2  libwww/2.14
From:  mickey@mouse.com
Content-type: application/x-www-form-
    urlencoded
Content-length: 150

    * a blank line *

&name = Gustavo
&email= alonso@inf.ethz.ch
...
```

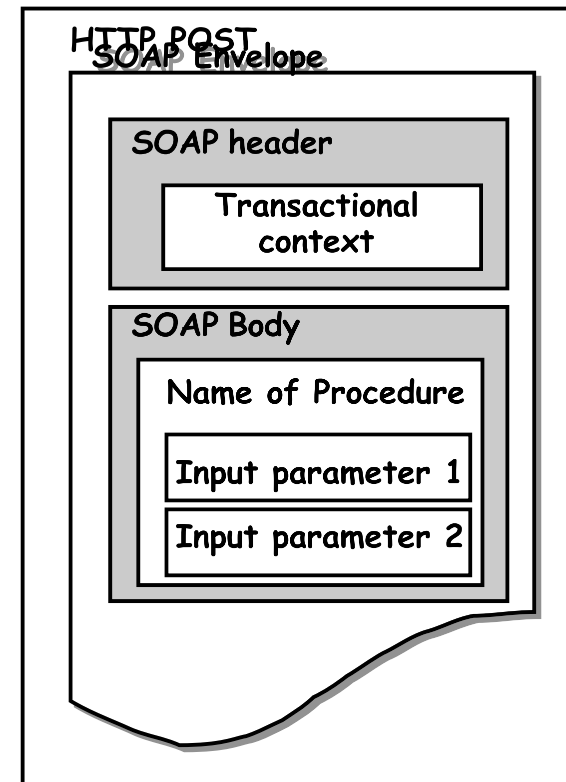
POST request indicating the CGI script to execute (post-query) GET can be used, but requires the parameters to be sent as part of the URL:

As before

Data provided through the form and sent back to the server

SOAP and HTTP

- A binding of SOAP to a transport protocol is a description of how a SOAP message is to be sent using that transport protocol
- The typical binding for SOAP is HTTP
- SOAP can use GET or POST. With GET, the request is not a SOAP message but the response is a SOAP message, with POST both request and response are SOAP messages (in v1.2, v1.1 mainly considers using POST).
- SOAP uses the same error and status codes as those used in HTTP so that HTTP responses can be directly interpreted by a SOAP module



In XML (a request)

From the: Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000

```
POST /StockQuote HTTP/1.1
        Host: www.stockquoteserver.com
        Content-Type: text/xml; charset="utf-8"
        Content-Length: nnnn
        SOAPAction: "Some-URI"
```

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In XML (the response)

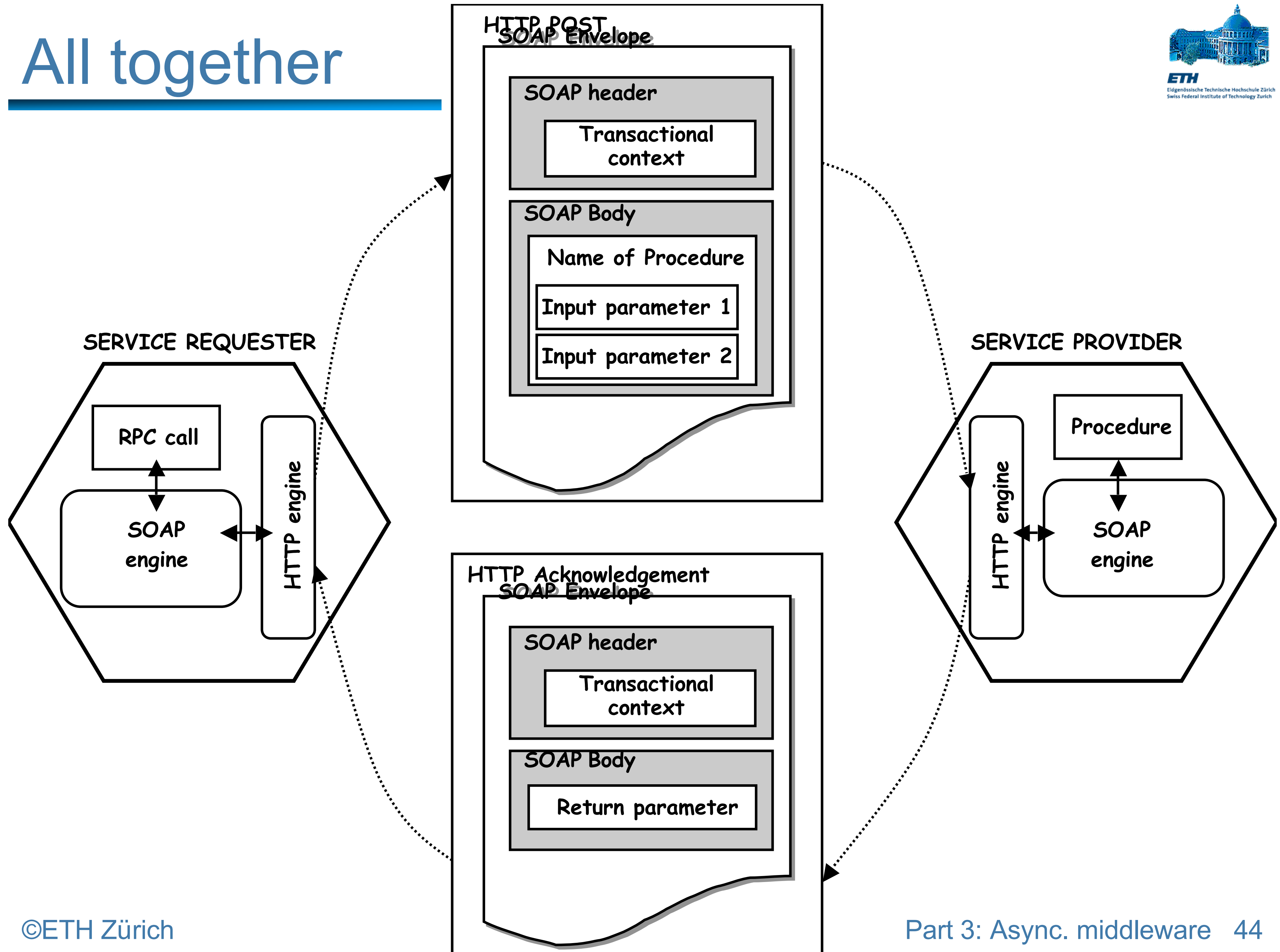
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

From the: Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000

```
HTTP/1.1 200 OK
        Content-Type: text/xml; charset="utf-8"
        Content-Length: nnnn
```

```xml
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

All together



SOAP summary

- SOAP, in its current form, provides basic mechanisms for:
  - ➥ encapsulating messages into an XML document
  - ➥ mapping the XML document to a SOAP message and turn it into an HTTP request
  - ➥ transforming RPC calls into SOAP messages
  - ➥ simple rules on how to process a SOAP message (rules have become more precise and comprehensive in v1.2 of the specification)
- SOAP takes advantage of the **standards** surrounding XML to resolve problems of **data representation** and **serialisation** (it uses XML Schema to represent data and data structures, and it also relies on XML for serialising the data for transfer). As XML becomes more powerful and additional XML standards appear, SOAP can take advantage of them by simply indicating what schema and encoding is used as part of the SOAP message. Current schema and encoding are generic but soon there will be **vertical standards** implementing schemas and encoding tailored to a particular application area (e.g., the efforts around EDI, Electronic Data Interchange and ebXML)
- SOAP is a very simple protocol intended for transferring data from one middleware platform to another. In spite of its claims to be open (which are true), current specifications are very **tied to RPC and HTTP**.