Architecture of the CORBA Component Model

Comparing CORBA and .NET: IIOP and SOAP

Interoperability

... the ability for a client on ORB A to invoke an OMG-IDL defined operation on an object on ORB B, where ORB A and ORB B are independently developed. [Interoperability RFP, 93-09-15]

Problems:

- Information model and validity of object references
- transport protocol
- Extensions (security, transactions)

Interoperability Domain

- Set of nodes/CORBA installations which can „directly“ exchange operation calls
- Domain Boundaries:
  - different networking technology (Internet vs. X.25)
  - network islands (private networks/firewalls)
  - different ORB protocol
- ORB domains

Interoperability means to overcome domain boundaries.

Inter-ORB-Bridges

- Mediate between ORB domains
- Client in domain A contacts bridge, which contacts domain B
- Different functions:
  - conversion of object references
  - conversion of message formats
  - validation of message authenticity
- Allows to integrate non-CORBA systems
  - COM/CORBA interworking
- Source of interoperability: GIOP

GIOP: General Inter-ORB Protocol

Prerequisites:

- connection-oriented transport
- full-duplex connections
- connection is symmetric w.r.t. shutdown
- transport is reliable
- transport transmits byte streams
- transport informs about connection loss (disorderly release)

GIOP (2)

- GIOP defines
  - Message format: Common Data Representation (CDR)
  - Message types
  - Structure of object references: Interoperable Object Reference (IOR)
- GIOP message format and IOR format are defined in IDL
  - will be transmitted through CDR
- IIOP
  - Transport is TCP
  - IOR format is specialized (IOR profile)

Common Data Representation

- Bi-endian
  - Endianness is typically „native“ for sender (JDK: always big-endian)
- Data are encoded as a sequence of primitive values
  - structures, parameter boundaries are not represented
- Each primitive type has a fixed size
  - char, (wchar), octet, boolean: 1
  - short, unsigned short: 2
  - long, unsigned long, float, enum: 4
  - long long, unsigned long long, double: 8
  - long double: 16

Common Data Representation (2)

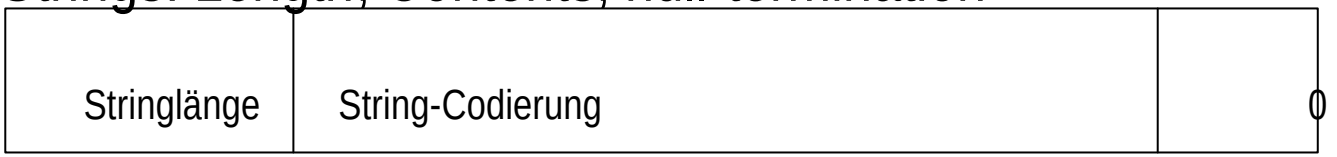
- Alignment: Each primitive value is aligned relative to the message start
  - usually a multiple of its size
  - long double: 8
  - Idea: allow direct copying from transport buffer into C structures

Encoding of Primitive Types

- Integral types: binary, signed types in two‘s complement
- Floating point types: IEEE-754
- octets: „as-is“
- boolean: TRUE==1, FALSE==0
- characters: character set according to „character set negotiation“

Encoding of complex types

- Strings: Length, Contents, null-termination



- Structures: Element for element, padding according to alignment

Encoding of Complex Types (2)

- Unions: Discriminator, union branch
- Array: Element for element
  - multi-dimensional: last index grows fastest
- Sequence: length, elements
- Enum: like unsigned long, enumerators are numbered starting with 0

Encoding of Complex Types (3)

- Any: typecode(type of value), value encoding
- Context: sequence<string>
- Exception: string(repos-id), struct(exception-members)

Interoperable Object References

```
module IOP {
   typedef unsigned long ProfileId;
   struct TaggedProfile {
       ProfileId tag;
       sequence <octet> profile_data;
   };
   struct IOR {
       string type_id;
       sequence <TaggedProfile> profiles;
   };
}
```

- IOR-String: hexified version of an encapsulation containing an IOP::IOR

IOR Profiles

- Define protocol independent contact information
- Defined in IDL
- encoded as an encapsulation

```
module IOP {
   const ProfileId TAG_INTERNET_IOP = 0;
   const ProfileId TAG_MULTIPLE_COMPONENTS = 1;
   const ProfileId TAG_SCCP_IOP = 2;
};
```

IIOP Profile

```
module IIOP {
   struct Version {
       octet major;
       octet minor;
   };
   struct ProfileBody_1_1 {// also used for 1.2
       Version iiop_version;
       string host;
       unsigned short port;
       sequence <octet> object_key;
       sequence <IOP::TaggedComponent> components;
   };
};
```

Tagged Components

- Define protocol-independent contact information
- represented as TAG_MULTIPLE_COMPONENTS or in the IIOP profile

```
module IOP{
typedef unsigned long ComponentId;
struct TaggedComponent {
   ComponentId tag;
   sequence <octet> component_data;
};
typedef sequence<TaggedComponent>
   TaggedComponentSeq;
};
```

Standard IOR Components

```
module IOP {
    const ComponentId TAG_ORB_TYPE = 0;
    const ComponentId TAG_CODE_SETS = 1;
    const ComponentId TAG_POLICIES = 2;
    const ComponentId TAG_ALTERNATE_IIOP_ADDRESS = 3;
    const ComponentId TAG_ASSOCIATION_OPTIONS = 13;
    const ComponentId TAG_SEC_NAME = 14;
    const ComponentId TAG_SPKM_1_SEC_MECH = 15;
    const ComponentId TAG_SPKM_2_SEC_MECH = 16;
    const ComponentId TAG_KerberosV5_SEC_MECH = 17;
    const ComponentId TAG_CSI_ECMA_Secret_SEC_MECH = 18;
    const ComponentId TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
    const ComponentId TAG_SSL_SEC_TRANS = 20;
    const ComponentId TAG_JAVA_CODEBASE = 25;
    ...
```

GIOP Messages

- Protocol assumes connection between client and server
- Connection management is invisible to the application (implicit binding)
- different protocol versions:
  - Version 1.0 (CORBA 2.0)
  - Version 1.1 (CORBA 2.1): Fragmentation
  - Version 1.2 (CORBA 2.3): bidirectional communication
- downwards compatible: old clients can talk to new servers

Message Types

Message Type	Initiator	Value	GIOP Version
Request	Client	0	1.0, 1.1, 1.2
Reply	Server	1	1.0, 1.1, 1.2
CancelRequest	Client	2	1.0, 1.1, 1.2
LocateRequest	Client	3	1.0, 1.1, 1.2
LocateReply	Server	4	1.0, 1.1, 1.2
CloseConnection	Server	5	1.0, 1.1, 1.2
MessageError	beide	6	1.0, 1.1, 1.2
Fragment	beide	7	1.1, 1.2

Structure of a GIOP Message

- Basic Structure:
  - GIOP message header
  - message-specific header
  - message-specific body

```
module GIOP {
   struct Version {octet major; octet minor; };
   enum MsgType_1_1 { Request, Reply, CancelRequest, ... };

   struct MessageHeader_1_1 {
        char magic [4]; // GIOP
        Version GIOP_version;
        octet flags;     // Bit 0: Endianness (0: big)
                         // Bit 1: more fragments
        octet message_type;
        unsigned long message_size;
};
```

Request

```
struct RequestHeader_1_1 {
   IOP::ServiceContextList service_context;
   unsigned long request_id;
   boolean response_expected;
   octet reserved[3];
   sequence <octet> object_key;
   string operation;
   CORBA::OctetSeq requesting_principal;
};
```

- followed by parameters (GIOP 1.2: 8-aligned)

Service Context

- Additional information transmitted from ORB to ORB

```
module IOP {
   typedef unsigned long ServiceId;
   struct ServiceContext {
        ServiceId context_id;
        sequence <octet> context_data;
   };
   typedef sequence <ServiceContext>ServiceContextList;
   const ServiceId TransactionService = 0;
   const ServiceId CodeSets = 1;
   const ServiceId ChainBypassCheck = 2;
   const ServiceId ChainBypassInfo = 3;
   const ServiceId LogicalThreadId = 4;
   const ServiceId BI_DIR_IIOP = 5;
   ...
```

Reply

```
enum ReplyStatusType_1_0 {
   NO_EXCEPTION,
   USER_EXCEPTION,
   SYSTEM_EXCEPTION,
   LOCATION_FORWARD
};
struct ReplyHeader_1_0 {
   IOP::ServiceContextList service_context;
   unsigned long request_id;
   ReplyStatusType_1_0 reply_status;
};
```

Reply (2)

- NO_EXCEPTION: followed by result, out-parameters
- USER_EXCEPTION: according to CDR
- SYSTEM_EXCEPTION:

```
module GIOP {
   struct SystemExceptionReplyBody {
       string exception_id;
       unsigned long minor_code_value;
       unsigned long completion_status;
   };
};
```

- LOCATION_FORWARD: IOR of the new location

Location Forwarding

- Permanent Object References: Client knows host/port
- Idea: Operator can move object
- Two procedures:
  - Client sends normal request, gets LOCATION_FORWARD, retries
  - Client sends LOCATE_REQUEST, gets LOCATE_REPLY

Location Forwarding (2)

```
struct LocateRequestHeader_1_0 {
   // Renamed LocationRequestHeader
   unsigned long request_id;
   sequence <octet> object_key;
};

enum LocateStatusType_1_0 {
   UNKNOWN_OBJECT,
   OBJECT_HERE,
   OBJECT_FORWARD
};
struct LocateReplyHeader_1_0 {
   unsigned long request_id;
   LocateStatusType_1_0 locate_status;
};
```

CancelRequest

```
module GIOP { // IDL
    struct CancelRequestHeader {
        unsigned long request_id;
    };
};
```

CloseConnection

- Client can close connection at any time
- server only if there are no pending replies
- CloseConnection consists only of MessageHeader

MessageError

- sent for erroneous or unrecognized message
- consists only of message header
- sender closes connection after transmission

Fragment

- Allows to split message into smaller chunks

```
module GIOP {
    struct FragmentHeader_1_2 {
        unsigned long request_id;
    };
};
```

SOAP

- http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
  - 1.2: http://www.w3.org/TR/soap12-part0/, -part1, -part2
- “lightweight protocol for exchange of information in a decentralized, distributed environment”
- three parts:
  - envelope structure for defining messages
  - set of encoding rules for encoding application-specific data types
  - convention for doing RPC
- Current version 1.2 (W3C XML Protocols group)
  - Version 1.1 widely used (Don Box, DevelopMentor)
- Supports potentially multiple transport mechanisms
  - only HTTP specified

SOAP Companions

- WSDL: Web Services Definition Language
  - http://www.w3.org/TR/wsdl
  - interface definition: endpoints, operations, and messages
  - XML vocabulary for describing services (SOAP, HTTP, MIME)
- UDDI: Universal Description Discovery and Integration
  - http://uddi.org/pubs/uddi-v3.0.1-20031014.htm
  - repository service for discovery of services
  - XML schemas for various information models
  - not covered here
- Web Services Interoperability
- Web Services Security

Usage Scenarios

- Fire-and-forget to single/multiple receiver (notifications)
- Request/response asynchronous communication
- RPC
- Request with acknowledgement
- Request with encrypted payload
- (Multiple) Third party intermediary
- Multiple asynchronous responses
- Caching
- Routing
- …

SOAP Non-Goals

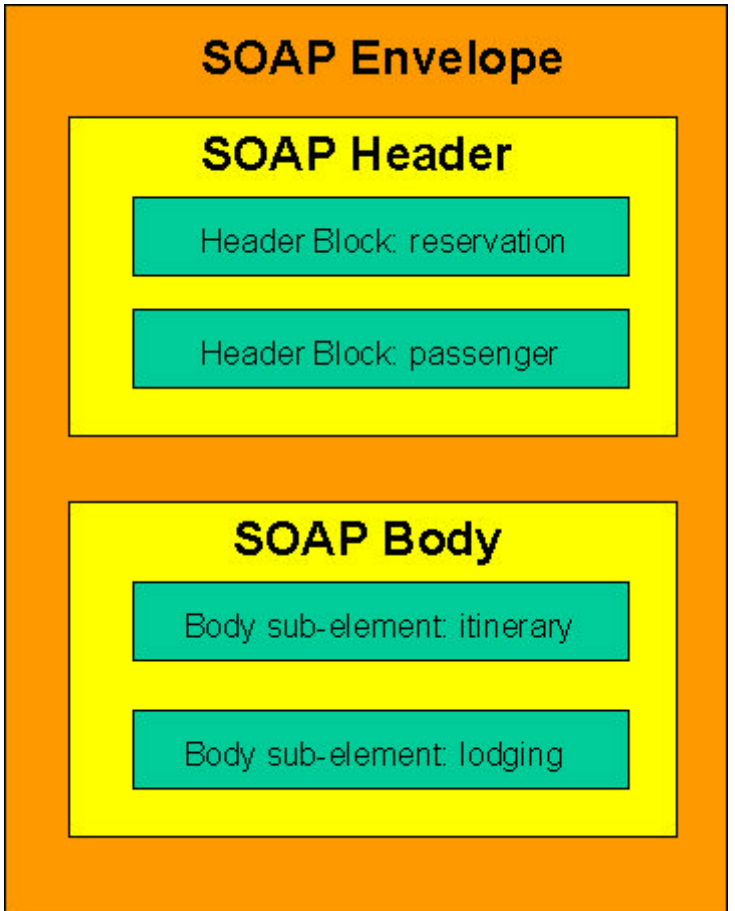
- Distributed Garbage Collection
- Boxcarring/batching of messages
- objects-by-reference
- activation

Namespaces

- xmlns:ENV="http://schemas.xmlsoap.org/soap/envelope/"
  - SOAP 1.2: http://www.w3.org/2003/05/soap-envelope
- xmlns:ENC="http://schemas.xmlsoap.org/soap/encoding/"
  - SOAP 1.2: http://www.w3.org/2003/05/soap-encoding

Messages

- Emitted by *sender*, targeted at *ultimate receiver*, through *intermediaries*



Envelope

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header> … </env:Header>
  <env:Body> … </env:Body>
</env:Envelope>
```

Header

```
<env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
         env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
          env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
         env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
          env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
```

Body

```
<env:Body>
  <p:itinerary  xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid-morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
</env:Body>
```

Roles

- Target roles specified in ENV:role of headers
  - SOAP 1.1: ENV:actor
- Possible roles
  - "http://www.w3.org/2003/05/soap-envelope/role/next"
  - "http://www.w3.org/2003/05/soap-envelope/role/none"
  - "http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"
- ENV:mustUnderstand specifies whether target is required to process the header
- Intermediary may drop, modify, or add headers
  - processed headers are dropped, unless ENV:relay is true

Body Processing

- Targeted at ultimate receiver
- Structure completely application-defined, except for faults
- Structure of body specified in ENV:encodingStyle
  - 1.1: typically specified on ENV:Envelope
  - 1.2: specified on header block, child of body that is not a Fault element, child of a Detail element, or descendents thereof

Faults

- ENV:Fault
- Mandatory children
  - ENV:Code
    - ENV:Value, with values Env:{VersionMismatch, MustUnderstand, DataEncodingUnknown, Sender, Receiver}
      - Additional Header elements may provide more detail
    - Optional ENV:Subcode child
  - ENV:Reason, with optional ENV:Text children
- Optional children
  - ENV:Node, with URI content
  - ENV:Role, with URI content
  - ENV:Detail, with arbitrary attributes and child elements

SOAP Data Model

- Optional part of SOAP
- Intended for RPC, accompanied by encoding style
- Data are represented as directed edge-labeled graph of nodes
  - outbound edges may be distinguished either by label or position
  - labels are QNames
  - nodes may be referenced just once, or have multiple references
- Structs are nodes with all outbound edges distinguished by label
- Arrays are nodes with all outbound edges distinguished by position

SOAP Encodings

- Support for multiple encoding styles
- Only one standard encoding
  - "http://www.w3.org/2003/05/soap-encoding"
- Graph edges represented by elements
  - label of the edge is element name
  - ENC:ref=“id” (type IDREF) can be used for multiple references
  - ENC:id=“id” (type ID) specifies target of the reference
- Simple values are encoded in element content
- Compound values are encoded through child elements
  - outgoing labelled edges again denoted through child element name

SOAP Encodings (2)

- ENC:nodeType can be used to specify one of “simple”, “struct”, “array”
- Arrays: Element names of child elements are not significant
  - ENC:itemType specifies the array element type
  - ENC:arraySize allows the specification of multi-dimensional arrays
- Integration with XML Schema
  - xsi:type can specify the node type
  - if not specified, ENC:itemType may apply
  - otherwise, graph node has unspecified type

Using SOAP for RPC

- Multiple bindings to protocols, only HTTP binding specified
- RPC usage must identify RPC resource
  - e.g. encoding of identified resources in URL query parameters
  - pure GET operations may not include any SOAP envelope
- RPC Invocation is a single struct, with outbound edges for parameters
  - name of struct node equals operation name
- RPC Response is a single struct, whose name is irrelevant
- Faults should be used to indicate errors in the RPC invocation

WSDL

- http://www.w3.org/TR/2001/NOTE-wsdl-20010315
- Definition of *services* as a collection of *ports* (network endpoints)
- Definition of *messages*: data that are exchanged
- Definition of *port types*: abstract collections of operations
- Definition of *bindings*: specification of protocol and data format for a type

Example

```
<?xml version="1.0"?>
<definitions name="StockQuote“
     targetNamespace="http://example.com/stockquote.wsdl"
         xmlns:tns="http://example.com/stockquote.wsdl"
         xmlns:xsd1="http://example.com/stockquote.xsd"
         xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>…</types>
   <message>…</message>
   <portType>…</portType>
   <binding>…</binding>
   <service>…</service>
</definitions
```

Example: Types

```
<types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>
```

Example: Messages

```
<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>
```

Example: Port Types

```
<portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
   </operation>
</portType>
```

Example: Bindings

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

Example: Services

```
<service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
          <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
```

WSDL Document Structure

- Specification uses informal grammar to describe document structure, e.g.

<definitions .... >

<types>

<xsd:schema .... />*

</types>

</definitions>

means: The “definitions” element contains a “types” element, which can contain multiple “xsd:schema” elements

WSDL Definitions

- Types: List of schema types
- Messages: Consist of multiple parts
  - each part specified either as an element or a type
  - actual (XML) representation depends on the binding
- port types: List of operations
  - Each operation has optional input, output, faults
- Bindings: Lists of inputs, outputs, faults, mixed with extension elements
- Service: List of ports
  - Each port has name, binding, optional extensions

WSDL SOAP Binding

- Extensions soap:binding, soap:operation, soap:body, soap:header, soap:headerFault, soap:address, …
- soap:binding specifies transport= (default HTTP) and style= (“document” | “rpc”)
- soap:operation specifies soapAction= target URL (SOAPAction: header), and optional style=
- soap:body specifies parts= included in the body, use= type of encoding (“literal” | “encoded”), optional encodingStyle=, and optional namespace=
- soap:address specifies location= of HTTP server

Interopability

- Sources of non-interoperability
  - Under-specification (optional features, implementation-defined behavior)
  - Non-compliance of implementation
- Web-Services Interoperability Organization aims at improving interoperability
  - through specifications
  - through testing technology
- Multiple profiles of WS-I, depending on application domain