Web Services - Concepts, Architecture and Applications
Part 2: Synchronous middleware

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

- The Web service architecture proposed by IBM is a direct descendant of conventional middleware architectures:

“The Web Services architecture is the logical evolution of object-oriented analysis and design, and the logical evolution of components geared towards the architecture, design, implementation, and deployment of e-business solutions. Both approaches have been proven in dealing with the complexity of large systems. As in object-oriented systems, some of the fundamental concepts in Web Services are encapsulation, message passing, dynamic binding, and service description and querying. Fundamental to Web Services, then, is the notion that everything is a service, publishing an API for use by other services on the network and encapsulating implementation details.” (from the IBM Web Services Architecture Team)

- Hence, as a first step towards understanding Web services, we need to have a clear understanding of conventional middleware and how it got to be the component based, object oriented platform for application integration that IBM considers as the origin of Web services.
Remote Procedure Call (RPC): the basics of all middleware
Basics of distribution

There are four basic problems to solve in a distributed information system. Everything else derives from these basic aspects:

- Use the appropriate abstraction for distribution in view of the communication infrastructure available. The abstraction must hide the network stack and provide high level primitives.

- How to embed the service abstraction part of the programming language in a more or less transparent manner.
  - Don’t forget this important aspect: whatever you design, others will have to program and use it.

- How to exchange data between machines that might use different representations for different data types. This involves two aspects:
  - data type formats (e.g., byte orders in different architectures)
  - data structures (need to be flattened and then reconstructed)

- How to describe and find services:
  - so that clients can use them
  - to avoid tight integration
RPC as an abstraction

- The most accepted standard for network communication is IP (Internet Protocol) which provides unreliable delivery of single packets to one-hop distant hosts.
- IP was designed to be hidden behind other software layers:
  - TCP (Transport Control Protocol) implements connected, reliable message exchange.
  - UDP (User Datagram Protocol) implements unreliable datagram based message exchanges.
- TCP/IP and UDP/IP are visible to applications through sockets. The purpose of the socket interface was to provide a UNIX-like abstraction.
- Yet sockets are quite low level for many applications, thus, RPC (Remote Procedure Call) appeared as a way to:
  - hide communication details behind a procedural call.
  - bridge heterogeneous environments.
- RPC is the standard for distributed (client-server) computing.
RPC in programming languages

- The notion of distributed service invocation became a reality at the beginning of the 80’s when procedural languages (mainly C) were dominant.

- In procedural languages, the basic module is the procedure. A procedure implements a particular function or service that can be used anywhere within the program.

- It seemed natural to maintain this same notion when talking about distribution: the client makes a procedure call to a procedure that is implemented by the server. Since the client and server can be in different machines, the procedure is remote.

- Client/Server architectures are based on Remote Procedure Calls (RPC)

- Once we are working with remote procedures in mind, there are several aspects that are immediately determined:
  - data exchange is done as input and output parameters of the procedure call
  - pointers cannot be passed as parameters in RPC, opaque references are needed instead so that the client can use this reference to refer to the same data structure or entity at the server across different calls. The server is responsible for providing this opaque references.
Describing services IDL

- All RPC systems have a language that allows to describe services in an abstract manner (independent of the programming language used). This language has the generic name of IDL (interface definition language) (e.g., the IDL of SUN RPC is called XDR)
- The IDL allows to define each service in terms of their names, and input and output parameters (plus maybe other relevant aspects).
- An interface compiler is then used to generate the stubs for clients and servers (rpcgen in SUN RPC). It might also generate procedure headings that the programmer can then use to fill out the details of the implementation.

- Given an IDL specification, the interface compiler performs a variety of tasks:
  - It generates the client stub procedure for each procedure signature in the interface. The stub will be then compiled and linked with the client code
  - It generates a server stub. It can also create a server main, with the stub and the dispatcher compiled and linked into it. This code can then be extended by the designer by writing the implementation of the procedures
  - It might generate a *.h file for importing the interface and all the necessary constants and types
Machine independent representations

- Marshalling or serializing can be done by hand (although this is not desirable) using (in C) `sprintf` and `sscanf`:

  ```c
  Message= "Alonso" "ETHZ" "2001"
  char *name="Alonso", place="ETHZ";
  int year=2001;

  sprintf(message, "%d %s %s %d %d", strlen(name), name, strlen(place), place, year);
  ```

  Message after marshalling =
  "6 Alonso 4 ETHZ 2001"

- Remember that the type and number of parameters is known, we only need to agree on the syntax ...

- SUN XDR follows a similar approach:
  - messages are transformed into a sequence of 4 byte objects, each byte being in ASCII code
  - it defines how to pack different data types into these objects, which end of an object is the most significant, and which byte of an object comes first
  - the idea is to simplify computation at the expense of bandwidth

```
6 A l o n
s o
4
E T H Z
2 0 0 1
```
Discovering services

- A service is provided by a server located at a particular IP address and listening to a given port.

- Binding is the process of mapping a service name to an address and port that can be used for communication purposes.

- Binding can be done:
  - locally: the client must know the name (address) of the host of the server.
  - distributed: there is a separate service (service location, name and directory services, etc.) in charge of mapping names and addresses. This service must be reachable to all participants.

- With a distributed binder, several general operations are possible:
  - REGISTER (Exporting an interface): A server can register service names and the corresponding port.
  - WITHDRAW: A server can withdraw a service.
  - LookUP (Importing an interface): A client can ask the binder for the address and port of a given service.

- There must also be a way to locate the binder (predefined location, environment variables, broadcasting to all nodes looking for the binder).
Making it work in practice

- One cannot expect the programmer to implement all these mechanisms every time a distributed application is developed. Instead, they are provided by a so called RPC system.

- What does an RPC system do?
  - Provides an interface definition language (IDL) to describe the services
  - Generates all the additional codes necessary to make a procedure call remote and to deal with all the communication aspects
  - Provides a binder in case it has a distributed name and directory service system
RPC Application

SALES POINT CLIENT
IF no_customer_
THEN New_customer
ELSE Lookup_customer
Check_inventory
IF enough_supplies
THEN Place_order
ELSE ...

INVENTORY CONTROL
CLIENT
Lookup_product
Check_inventory
IF supplies_low
THEN Place_order
Update_inventory
...

Server 1
- New_customer
- Lookup_customer
- Delete_customer
- Update_customer

Server 2
- New_product
- Lookup_product
- Delete_product
- Update_product

Server 3
- Place_order
- Cancel_order
- Update_inventory
- Check_inventory

Server 1
- DBMS
- Customer database

Server 2
- DBMS
- Products database

Server 3
- DBMS
- Inventory and order database
RPC middleware: DCE architecture

[Diagram of DCE architecture with IDL source, IDL compiler, Interface header, Application client procedure, Language-specific call interface, Client stubs, RPC API, RPC run-time service library, RPC protocols, DCE directory services, DCE security services, Application server procedure, Language-specific call interface, Server stubs, RPC API, RPC run-time service library, Development Environment]
Building upon RPC: TP-monitors, object brokers, object monitors
Transactional RPC

- The limitations of RPC in terms of reliability can be resolved by making RPC calls transactional. In practice, this means that they are controlled by a 2PC protocol.
- An intermediate entity is needed to run 2PC (the client and server could do this themselves but it is neither practical nor generic enough).
- This intermediate entity is usually called a transaction manager (TM) and acts as intermediary in all interactions between clients, servers, and resource managers.
- When all the services needed to support RPC, transactional RPC, and additional features are added to the intermediate layer, the result is a TP-Monitor.
Basic TRPC (making calls)

1. Client stub: Get tid from TM
2. Transaction Manager (TM): Generate tid, store context for tid
3. Server stub: Get tid, register with the TM
4. Service procedure
5. Server: Invoke service, return
Basic TRPC (committing calls)

1. **Client**
   - ...Service_call...
   - EOT

2. **Client stub**
   - Send to TM
   - commit(tid)

3. **Transaction Manager (TM)**
   - Look up tid
   - Run 2PC with all servers associated with tid
   - Confirm commit

4. **Server stub**
   - Participant in 2PC

5. **Server**
One step beyond ...

- The previous example assumes the server is transactional and can run 2PC. This could be, for instance, a stored procedure interface within a database. However, this is not the usual model.
- Typically, the server invokes a resource manager (e.g., a database) that is the one actually running the transaction.
- This makes the interaction more complicated as it adds more participants but the basic concept is the same:
  - the server registers the resource manager(s) it uses
  - the TM runs 2PC with the resources managers, instead of the server (see OTS at the end)
Example: BEA Tuxedo (early versions)
Component based design

- The notion of component based design is nowadays associated with object orientation. However, RPC, TRP, or TP-Monitors are just alternative (perhaps more primitive) forms of component based design.

- The main difference is that with RPC, TRPC and conventional TP-Monitors, server and clients are typically developed together, as part of a single system and within the same developing (programming) effort.

- Component based design tries to take this idea one step further: the clients or servers do not need to be developed together and, ultimately, can be developed by different teams (vendors) based on standard interfaces.
The Common Object Request Broker Architecture (CORBA) is part of the Object Management Architecture (OMA) standard, a reference architecture for component based systems.

The key parts of CORBA are:

- **Object Request Broker (ORB):** in charge of the interaction between components.
- **CORBA services:** standard definitions of system services.
- A standardized IDL language for the publication of interfaces.
- Protocols for allowing ORBs to talk to each other.
Object Management Architecture

- The OMA defines the architecture and set of services that support object-based, heterogeneous distributed computing. Its components are:
  - ORB: communication infrastructure, server monitoring and management, marshalling, service location, etc.
  - CORBAservices: basic functionality of any middleware system (transactions, security)
  - CORBAdomains: vertical business domain services (financial, healthcare, manufacturing, etc.)
  - CORBAfacilities: services that are common to applications (printing services, internationalization, error control)
Object Monitors ...

OTS should not be confused with OTM (Object Transaction Monitors). An OTM is an attempt to bring together a lot of different technologies and bundle them into a coherent whole in an attempt to provide an “ideal” platform for developing distributed systems. OTMs are the basis of Enterprise Application Integration nowadays.

- **OBJECT REQUEST BROKERS (ORBs):** Reuse and distribution of components via a standard, object oriented interface and number of services that add semantics to the interaction between components.
- **TRANSACTION PROCESSING MONITORS:** An environment to develop components capable of interacting transactionally and the tools necessary to maintain transactional consistency.
- **PERSISTENT QUEUING SYSTEMS:** A communication mechanism that supports asynchronous interaction between components.

Object Monitor = ORB + TP-Monitor + queues + web
RPC over the Internet: Example of CORBA
In order for ORBs to be a truly universal component architecture, there has to be a way to allow ORBs to communicate with each other (we cannot have all components in the world under a single ORB).

For this purpose, CORBA provides a General Inter-ORB Protocol (GIOP) that specifies how to forward calls from one ORB to another and get the requests back.

The Internet Inter-ORB Protocol specifies how GIOP messages are translated into TCP/IP.

There are additional protocols to allow ORBs to communicate with other systems (an idea that will reappear with Web services).
From synchronous middleware to Web services
Middleware and Web services

RPC WORLD

Application client procedure
Language-specific call interface
Client stubs
RPC API
RPC run-time service library
RPC protocols
DCE directory services
DCE security services

IDL
IDL source
IDL compiler
Interface header

UDDI
SERVICE REGISTRY
Service description
FIND
PUBLISH

SOAP
SERVICE REQUESTER
BIND

SERVICE PROVIDER
Service description
Service Interface
Service

WSDL
Remote calls in CORBA

Identifying and locating services

Marshalling and serializing arguments

Client

Client stub
CORBA runtime

Interface repository

Implementation repository

Skeleton
Object adapter

ORB

ORB

TCP/IP socket

TCP/IP socket

MIDDLEWARE

MIDDLEWARE

NETWORK

Client (sever)

©ETH Zürich
SOAP as RPC mechanism

This could be RPC, CORBA, DCOM, using SOAP as protocol.
Web services architecture

- The Web services architecture represented by SOAP, UDDI, and WSDL is a direct descendant of conventional middleware platforms. They can be seen as the most basic extensions that are necessary to allow conventional synchronous middleware to interact with each other.

- The model and even the notation followed in this architecture mimics to a very large extent what has been done in RPC, TP-Monitors, CORBA, etc.

- This dependency gives a very good hint of what can be done with these technologies today and what is missing to obtain a complete platform for business to business electronic commerce.

- First implementations are just extensions of existing platforms to accept invocations through a Web service interface (e.g., database stored procedure published as Web services).