DISTRIBUTED AND CLOUD COMPUTING

LAB7 LOAD BALANCING AND SERVICE REGISTRATION

WEB SERVICES VS. DISTRIBUTED OBJECT MODEL

Two separate influences led to the emergence of web services:

- The addition of service interfaces to web servers with a view to allowing the resources on a site to be accessed by client programs other than browsers and using a richer form of interaction
- The desire to provide something like RPC over the Internet based on the existing protocols.

	Distributed Object model	Web Services
Reference/URI	Remote object reference	URI (compared with the remote object reference of a single object.)
Object Creation	Objects can create remote objects dynamically and return remote references to them. The recipient of these remote references can use them to invoke operations in the objects	Cannot create instances of remote objects. A web service can be viewed as consisting of a single remote object and therefore both garbage collection and remote object referencing are irrelevant.
Servant	- The server program is generally modelled as a collection of servants (potentially remote objects). - Instantiating a new remote object means creating a servant.	- Do not support servants. - Cannot create servants when handling different server resources

WEB SERVICES VS. DISTRIBUTED OBJECT MODEL

Web Services	Distributed Objects
• **Document Oriented** – Exchange documents	• **Object Oriented** – Instantiate remote objects – Request operations on a remote object – Receive results – … – Eventually release the object
• **Document design is the key** – Interfaces are just a way to pass documents	• **Interface design is the key** – Data structures just package data
• **Stateless computing** – State is contained within the documents that are exchanged (e.g., customer ID)	• **Stateful computing** – Remote object maintains state

https://people.cs.rutgers.edu/~pxk/417/notes/pdf/02d-web-services-slides.pdf

RESOURCE-ORIENTED SERVICES

- Get parts info `HTTP GET //www.parts-depot.com/parts`
- Returns a document containing a list of parts

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink">
   <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
   <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
   <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
   <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

RESOURCE-ORIENTED SERVICES

- Get detailed parts info: `HTTP GET //www.parts-depot.com/parts/00345`
- Returns a document with information about a specific part

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
        xmlns:xlink="http://www.w3.org/1999/xlink">
    <Part-ID>00345</Part-ID>
    <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification
xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</p:Part>
```

Web service

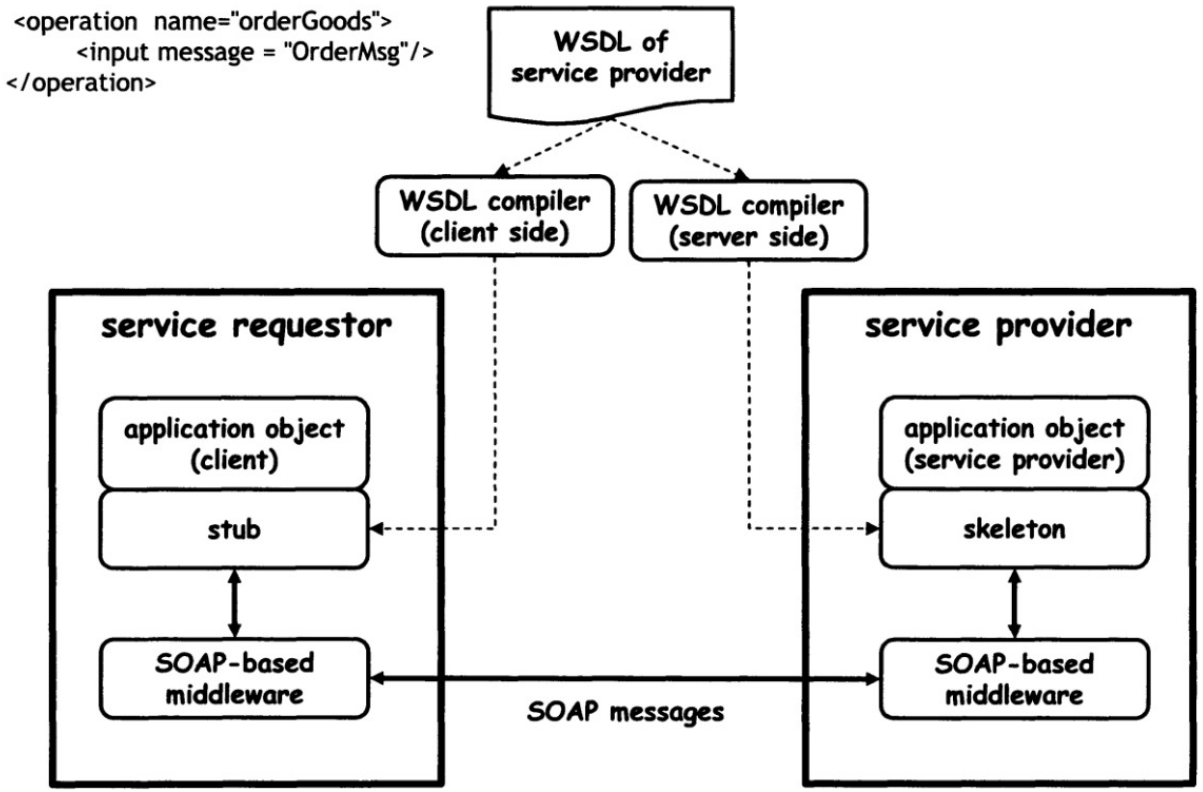
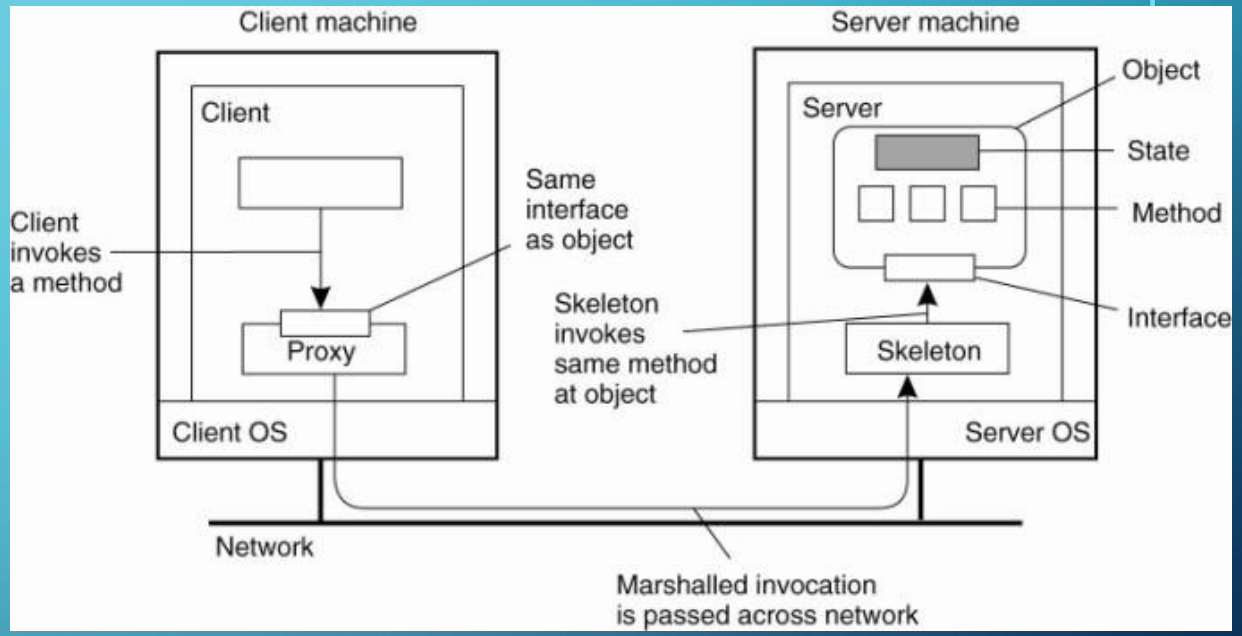


Fig. 6.2. WSDL specifications can be compiled into stubs and skeletons, analogously to IDL in conventional middleware. Dashed lines denote steps performed at development time, solid lines refer to run-time

https://link.springer.com/book/10.1007/978-3-662-10876-5

Distributed object



https://csis.pace.edu/~marchese/CS865/Lectures/Chap10/Chapter10.htm

Interfaces ShapeList and Shape

GraphicalObject is a class holds the state of the graphical object, e.g. type, position, line colour, fill colour, etc.

Distributed object

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
     int getVersion() throws RemoteException;
     GraphicalObject getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
     Shape newShape(GraphicalObject g) throws RemoteException;
     Vector allShapes() throws RemoteException;
     int getVersion() throws RemoteException;
}
```

Creating a new shape will return a reference

To get the version of a shape, first get its reference by *ShapeList.allShapes()*, then use that reference to get its version using getVersion()

To get the state of a shape, first get its reference, then use getAllState() to get an instance

Web services

```
import java.rmi.*;
public interface ShapeList extends Remote {
     int newShape(GraphicalObject g) throws RemoteException;
     int numberOfShapes() throws RemoteException;
     int getVersion() throws RemoteException;
     int getGraphicalObjectVersion(int i) throws RemoteException;
     GraphicalObject getAllState(int i) throws RemoteException;
}
```

When creating a new shape, do not return an object reference, but the index

Get the version of a shape directly by its index

Get the state of a shape directly use the index

Server

When creating a new shape…

Distributed object

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
      try{
            ShapeList aShapeList = new ShapeListServant();
            ShapeList stub = (ShapeList)
                        UnicastRemoteObject.exportObject(aShapeList,0);
            Naming.rebind("//bruno.ShapeList", stub);
            System.out.println("ShapeList server ready");
      }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
      }
}


import java.util.Vector;
public class ShapeListServant implements ShapeList {
      private Vector theList; // contains the list of Shapes
      private int version;

      public ShapeListServant(){...}
      public Shape newShape(GraphicalObject g) {
            version++;
            Shape s = new ShapeServant(g, version);
            theList.addElement(s);
            return s;
      }
      public Vector allShapes(){...}
      public int getVersion() { ... }
}
```

Create a servant of the graphical object

Web service

```
import java.util.Vector;
public class ShapeListImpl implements ShapeList{
      private Vector theList = new Vector();
      private int version = 0;
      private Vector theVersions = new Vector();

      public int newShape(GraphicalObject g) throws
      RemoteException{
            version++;
            theList.addElement(g);
            theVersions.addElement(new Integer(version));
            return theList.size();
      }
      public int numberOfShapes(){}
      public int getVersion() {}
      public int getGOVersion(int i){}
      public GraphicalObject getAllState(int i) {}
}
```

Do not create servants.

Add the graphical object directly to the list

Client

Distributed object

```
import java.rmi.*; import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
      public static void main(String args[]){
            System.setSecurityManager(new RMISecurityManager());
            ShapeList aShapeList = null;
            try{
                  aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList");
                  Vector sList = aShapeList.allShapes();
            }catch(RemoteException e) {System.out.println(e.getMessage());
            }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
      }
}
```

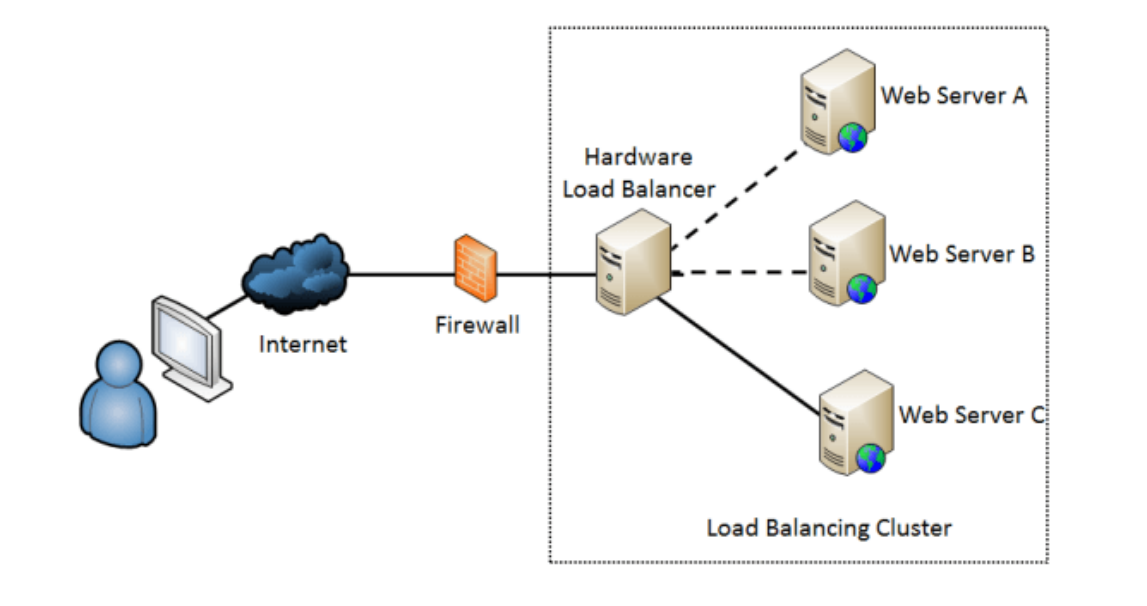
Web service

```
import javax.xml.rpc.Stub;
public class ShapeListClient {
      public static void main(String[] args) { /* pass URL of service */
            try {
                  Stub proxy = createProxy();
                  proxy._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
                  ShapeList aShapeList = (ShapeList)proxy;
                  GraphicalObject g = aShapeList.getAllState(0);
            } catch (Exception ex) { ex.printStackTrace(); }
      }

      private static Stub createProxy() {
            return (Stub) (new MyShapeListService_Impl().getShapeListPort());
      }
}
```

LOAD BALANCING

- Load balancing: With multiple instances, distribute workload to each instance
- Partitioning: One instance only holds one part of data, load balancer routes corresponding requests to the instance that has the data



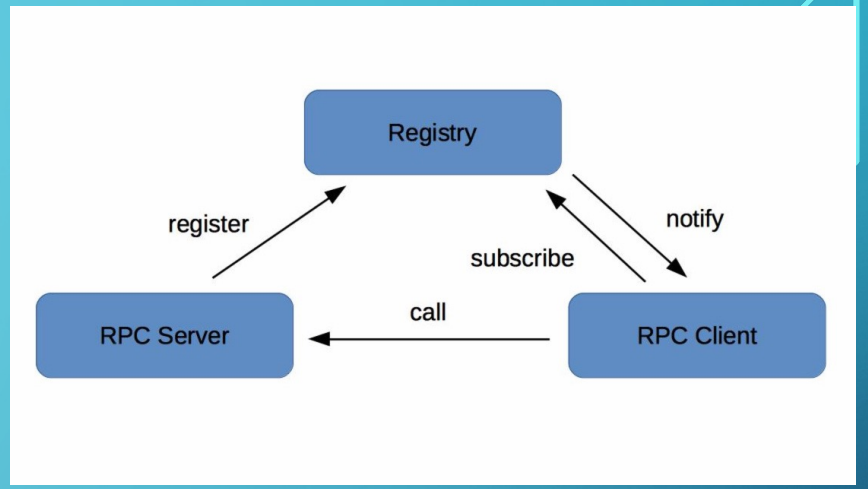
LOAD BALANCING POLICIES

- Random: Simplest, randomly select one from available list
- Round Rubin: set a value index, loop between instance [0,total-1]
- Weighted Round Robin: Add weight to RR, in case the resource is not even

- Hash: E.g. Modular hashing (hash=k mod n), Consistent Hash (Supports dynamic resource addition/deletion)

https://www.toptal.com/big-data/consistent-hashing

REGISTRY



- Registry: help to discover available service instances
  - Client: Lookup available services in registry
  - Server: Publish itself to registry, send heartbeat to report status
- In the load balancer: helps the client library to get available instances
- Other functionalities: monitoring, statistics, configuration, etc.

HEARTBEAT

- Why heartbeat: keepalive, detect link failure
- Typically, TCP connection will remain open if not explicitly closed
- However, connection may pass through NAT or firewall, their state table may expire
- Send heartbeat packet periodically to keep the connection alive, also tells the server that instance is available

TYPES OF HEARTBEAT

- ICMP Ping: Easy, could detect link failure, but don't know if server is still up
- TCP keepalive: use TCP option SO_KEEPALIVE, by default, send packet every 2 hours to detect if connection is still up. But is too low-level
- Application level keep alive: Use application to detect status, e.g. a simple RPC call ping(). Could keep connection alive also detect service failure.

RECAP – RPC/RMI FRAMEWORK

- Dynamic Proxy: stub, skeleton
- Network communication: I/O models
- Serialization: Cross-language serialization, IDL
- Load balancing: Different policies
- Registry: Service discovery and monitoring

PREVIEW

- https://hadoop.apache.org/