8/8/19 DC Paradigms (M.L. Liu)

characteristics of DC applications:
i) Interprocess communication
ii) Event synchronization

Level of abstraction:

Level		
5	Object & pace	high
4	Network services	
3	RPC, RMI	
2	Client-server	
1	Message passing	

↓ More dependent on the underlying hardware

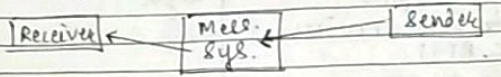
Message passing is the most fundamental & simplest form of DC. Basic operations involve send & receive

Client-server
- Best known paradigm for network app.
- Assigns asymmetric roles to two collaborating processes.
- Single server can host multiple service
- Single point failure.
- Centralized system
- Hardware limitations (Adaptability to newer tech.)

Peer-to-peer (resource)
- Replica of single server/process is kept on multiple machines.
- 3 versions/variations available

- Placement of replica affects the response time.

Message - System (Level - 2)
- Also called Message - Oriented Middleware (MOM)



Ex - WhatsApp

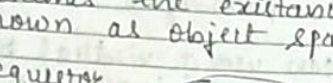
i) Point - to - point
Middleware system cannot read it. It just acts as a depository.

ii) Publish / Subscribe
The subscribers are notified whenever a certain event occurs for which the user subscribed to before.
Middleware system publishes the message to all the subscribers.

Remote Procedure Call (Level - 3)
- RMI is the public oriented version of RPC.
- Marshalling - unmarshalling comes into picture.
- One process calls another process (procedure) in a procedural way.

Object Request broker Paradigm
- Ex - CORBA (data is transferred as binary which is an overhead)
- can be implemented by any lang. supporting POJO.

Object Space
- the objects are stored at a global level and anyone across the globe can access it
- assumes the existance of logical entities known as object spaces.



object space

→ Sometimes, service providers aren't willing to expose their services outside their premise. A portable program (called mobile agent) travels from host to host according to an ~~itinerary~~ itinerary that it carries. as a solution Agent carries with it input & output from every host. At every host, agent accesses necessary resources and performs the necessary tasks to accomplish its mission.
Restrictions such as time window or whether an agent can come back once it

left or not, can be applied. This comes under configuring the agent.

Collaborative (Groupware)
Ex - seti@home (website)
Used for terrestrial signalling. If your machine is inactive for certain amount of time everyday, the resources (CPU) on your machine can be used. This is possible due to collaborative paradigm.

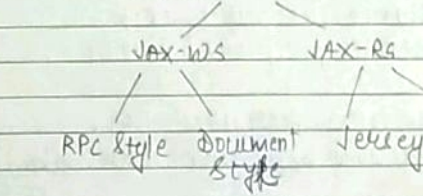
Processes can use a virtual board where every participant (in the group) can read/write.

20/8/14 RESTful web services.
- Based on REST (Representational State Transfer)
- Focusing on the state info of a particular web service.
- Uses HTTP protocol (Thus, stateless service)
- In REST arch, a REST server simply provides access to resources and REST client accesses and modifies the resources.
- WS is a collection of open protocols and standards used for exchanging data between app or systems
- While communicating, content is more important than its appearance
- This interoperability is due to the use

of open standards

- A RESTful WS usually defines a URI.

SOAP	REST
* Protocol	* Architectural style
* Can't use REST as it is a protocol	* Can use SOAP
* Uses services interfaces to expose the business logic	* Uses URI to expose business logic
* JAX-WS is the java API for SOAP WS	* JAX-RS is the java API for RESTful WS
* Defines standards to be strictly followed	* Does not define too much standards
* Requires more bandwidth & resources	* Requires less bandwidth and resources
* Defines its own security	* Inherits security measures from underlying transport
* Permits XML only	* Permits Plain text, JSON, HTML, XML
* Is more preferred	* Is less preferred



Package for annotations : javax.ws.rs.

Annotations

Path - Identifies the URI path (can be applied to class file and method)

PathParam

GET

POST

PUT

HEAD

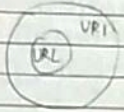
DELETE - Deletes a particular resource

Produces - Defines media type for the responses such as XML, PLAIN, JSON etc

⇒ Every entity is a resource in REST

⇒ Resource in REST is similar to Object in OOP

⇒ Once resource is identified (the temp of Nadiad when requested on website) its representation is to be decided using a standard

⇒ 

Name - URI → Always identifies

Address - URL & URI

Always locates

⇒ Instance resource URI:
weather/ziprode/380001

Collection resource URI:
in.bookmyshow.com/ahmedabad/movies/

⇒ Web pages are created at runtime as

(Returns entire HTML page which can be rendered on a browser without any modification)

RESTful WS can respond in HTML (Not possible in standard WS)

→ Ex- When a person is logged in on FB & wants to edit the 3rd comment on the 5th post. A URI is used. (URI used)

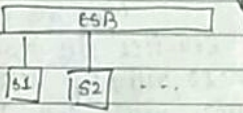
⇒ Resource can be identified by different URIs
i) bookmyshow/Ahmedabad/movies/theatre
ii) bookmyshow/theatre/movies

22/9/19 ESB (Enterprise Service Bus)
It is flexible connectivity infrastructure for integrating systems, applications and services to power your SOA

Why ESB is needed?

- Service are autonomous & availability of service at any given time cannot be guaranteed.
- Reduce point-to-point ~~comm~~ connections

... 10 Services.
Every service requires 9 connections for each of the other service.

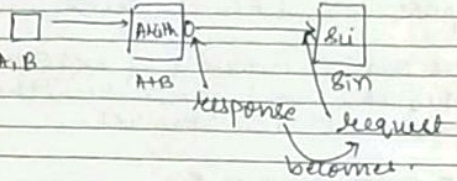
ESB: 

(Loose)

- Easier to monitor failure & misbehavior

- Every client directs all its requests through ESB instead of directly contacting the potential server

Management capabilities.



- ROUTING
- Transforming

Services

- Mediation Services
  1. Routing
  2. Transformation
- Event
  1. Publish
  2.
- Transport
  1. Synchronous / Asynchronous
  2. Persistent / Non-persistent
  3. Loosely-coupled / Tightly coupled

May be possible that services are independent but, business activity requires that a certain service must be performed only after 3 other services have been called. This is tightly-coupled.

- Mediation capabilities
  - transformations (XML-to XML translation)
  - Message validation
  - Content & quality selection

ESB Implementation
1) Java Business Integration (JBI)
2) Open ESB
3) WSO2

SOA Lifecycle of Project
1) Model
   - Gather requirements
   - Model & simulate
   - Design
2) Assemble
   - Discover (whether it is already available)
   - Construct & Test (if not available)
   - Compose
3) Deploy
   - Integrate people
   - Integrate processes
   - Integrate information
4) Manage
   - IT resources
   - Services
   - business metrics (Monitor) ← non-functional requirements
   - Monitor services & underlying resources

5) Governance & best practices
- Sharing & reuse of services
- Establish decision rights
- Policies, measurements and controls

27/8/19 Implementation of RESTful WS (Netbeans &
i) Add JAR files for JAX-RS

ii) New → Java class (User.java)
Package : com.rest

```
package com.rest;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlElement
                                    XmlRootElement
@XmlRootElement(name = "user")
public class User implements Serializable
{
    private static final long serialVersionUID = 1L
    private int id;
    private String name;
    private String profession;

    public User()
    {}

    public User(int id, String name, String
                                    profession)
    {
        this.id = id;
        this.name = name;
```

.txt - Plain text
.dat - Byte Stream

```
        this.profession = profession;
    }

    // getter - setter methods for all elements
}
```

iii) Neo → Java class (UserDao) // data connection code here
package : com.test

```
public class UserDao
{
    public List<User> getAllUsers() {
        List<User> userList = null;
        try {
            File file = new File("Users1.dat");
            if (!file.exists())
            {
                User user = new User(1, "Anand",
                                     "Professor");
                userList = new ArrayList<User>();
                userList.add(user);
                saveUserList(userList);
            }
            else
            {
                FileInputStream fis = new FileInputStream
                                      (file);
                ObjectInputStream ois = new
                                        ObjectInputStream(fis)
                userList = (List<User>) ois.readObject();
```

oii.close();
} catch (Exception e) { }
return userList;
}
public void saveUserList (List<User>) {
try {
File file = new File("User.dat");
FileOutputStream fos = new FileOutputStream(f
ObjectOutputStream oos = new ObjectOutputStream
(fos);

iv) New → Java class (UserService)
Package : com.rest

@Path ("/UserService")
Public class UserService {
UserDao userDao = new UserDao();
@GET
@Path ("/~~name~~ user")
@Produces (MediaType.APPLICATION_XML) -JSON
public List<User> getUsers()
{
return userDao.getAllUsers();
}
}

v) WEB-INF → New → XML Document
<display-name> RestfulService Hi </display-name>
<servlet>
<servlet-name> JerseyRESTfulApplication
</servlet-name>

<servlet-class> org.glassfish.jersey.servlet.
ServletContainer
</servlet-class>
<init-param>
<param-name> jersey.config.server.
provider.packages </param-name>
<param-value> com.rest </param-value>
</init-param>
</servlet>
<servlet-mapping>
<servlet-name> jersey REST fulApplication </servlet-name>
<url-pattern> /rest/* </url-pattern>
</servlet-mapping>

(Create a servlet to act as a container. This step wasn't req. for traditional WS)

→ @GET
@Path("/anand")
@Produces(MediaType.APPLICATION_XHTML_XML) → text/html
public String getUser()
{
return "<html><body><h1> Hello </h1>
</body></html>";
}

→ build
→ Deploy
→ Postman is http://localhost:8080/"Appname"/"Servicename"/

29/8/19 File → New Project → Web App (Restful Web Service)

Database → Turn on connection (derby sample) (Connect)

Right click Proj. name → New → Other → WS → Restful Service from database
Data source → Select appropriate DB (All tables appear)
Add tables) [✓ Include Related Tables]
Give Package name (entity)

⇒ IDE creates all the class & service files on its own.

⇒ @POST @Path("/addusers") @Produces(MediaType.APPLICATION_JSON)
public int AddUsers()
{
List<User> Nulist = userDao.getAllUsers();
User U = new User(2, "Dave", "Age Prof");
Nulist.add(U);
return 1; → userDao.saveUserList(Nulist);
}

// End all java.exe processes from task manager before clean & build

AddUsers(@PathParam("id") int id1)
// Now id can be accessed as id1 in the code.
// In URL ···/addusers?{id}={value}