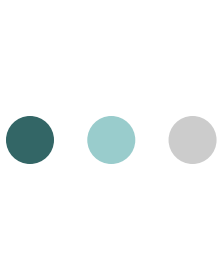


Chapter 1

Introduction

Rendezvous Problem

Imagine a person trying to start two programs at the same time

- on two separate computers
- wanting them to communicate with each other

Rendezvous Problem—Client/Server

The client/server model solves the rendezvous problem by determining that one side (the server) must start execution, and wait indefinitely for the other side (the client) to contact it

- Server
  - a program that waits for incoming communication requests from a client
  - When the server receives a communication from the client, the server provides some useful service to the client, and sends the client the results
- Client
  - a program that initiates communication with the server
  - The client has need of some service that the server can provide

Client/Server

- Clients are often easier to build than servers
  - commonly require no special system privileges
- Servers often need to access data and/or routines or resources that are provided by an operating system and protected by the operating system
  - servers often need special system privileges

Client/Server (cont'd)

- Servers are concerned with:
  - Authentication – verifying that the client is who it claims to be
  - Authorization – determining whether the given client is permitted to access any of the services the server supplies
  - Providing services
  - Data security – guaranteeing that a client is not allowed to access data that the client is not allowed to access, preventing data from being stolen

Client/Server (cont'd)

Servers typically come in two different versions:

- Stateless – server does not save any information about the status of ongoing interaction with clients
- Stateful – server saves information about the status of ongoing interactions with clients

What is Middleware?

According to Oracle (2016):

> “Middleware is the software that connects software components or enterprise applications. Middleware is the software layer that lies between the operating system and the applications on each side of a distributed computer network. Typically, it supports complex, distributed business software applications.”

What is Middleware? (cont'd)

This definition is from Techopedia (2016) :

"Middleware is a software layer situated between applications and operating systems. Middleware is typically used in distributed systems where it simplifies software development by doing the following:

Hides the intricacies of distributed applications

Hides the heterogeneity of hardware, operating systems and protocols

Provides uniform and high-level interfaces used to make interoperable, reusable and portable applications

Provides a set of common services that minimizes duplication of efforts and enhances collaboration between applications"

What is Middleware? (cont'd)

This definition is from Apprenda (2016) :

"A simple middleware definition: software that connects computers and devices to other applications. It can also be referred to as the slash or connecting point in client/server. Another way to define middleware is to say that it is software that acts as a liaison between applications and networks. The term is often used in the context of cloud computing, such as public or private cloud."

What is Middleware? (cont'd)

Middleware

- is used to hide the lower level complexity of networks and operating systems from the application programmer
- can allow a client running on one kind of computer with one kind of operating system to talk to a server on a different kind of computer running a different operating system
- allows the definition of clearly defined interfaces to servers

Sockets

- The Berkeley Software Distribution of UNIX back in 1983 defined an application interface called “sockets”
  - This eventually became a Portable Operating System Interface (POSIX) specification
  - So they’re commonly known as BSD sockets or as POSIX sockets

Sockets (cont'd)

- There is a Windows version of sockets, called Winsock that was originally based on the POSIX sockets
  - However, there are quite a few differences in terms of include files, names of library routines, etc.
  - According to MSDN (2016) there are also some implementation differences based on differences in Windows compared to UNIX
- It is usually possible to port socket applications between Linux and Windows

Sockets (cont'd)—Socket Data Structures

- With POSIX/BSD sockets, a socket is treated similarly to a file
  - Sockets are stored in the file descriptor table
  - thus, an application cannot have both a file descriptor and a socket descriptor with the same value

Sockets (cont'd)—Socket Data Structures

- To perform communication, sockets choose a family of protocols to use
  - which family to use is defined in the *sockaddr* struct
    - which is defined in the file "sys/socket.h"

Sockets (cont'd)—Socket Data Structures

```
struct sockaddr
 {
        unsigned char  sa_len;
                        // length of address
        sa_family_t     sa_family;
                        // the address family
        char                sa_data[14];
                        // the address
 };
```

Sockets (cont'd)—Socket Data Structures

Several different address families are supported

- based on the kinds of protocols being used
- Some of the predefined address families included in the sockets specifications were for protocols that are not used a lot nowadays
  - remember that sockets have been around since the early 80s
- we will focus only on TCP/IP here

Sockets (cont'd)—Socket Data Structures

- The sockaddr_in struct is defined in the file "netinet/in.h"
- The sockaddr_in struct will contain the following members (at a minimum):
  - sa_family_t sin_family;
  - in_port_t sin_port;
  - struct in_addr sin_addr;

Sockets (cont'd)—Socket Data Structures

- The sin_family is set to AF_INET
  - AF_INET is a constant value defined as the number 2
  - which selects TCP/IP using IPv4 addressing.
    - this corresponds to a protocol family called PF_INET
    - the internet protocol family for IPv6, which is also number 2
- sin_port is a port number
- sin_addr is a typical IP address

Sockets (cont'd)—Socket Data Structures

- The sockaddr_in6 struct is also defined in the file "netinet/in.h".
- The sockaddr_in6 struct will contain the following members (at a minimum):
  - sa_family_t sin6_family;
  - in_port_t sin6_port;
  - uint32_t sin6_flowinfo;
  - struct in6_addr sin6_addr;
  - uint32_t sin6_scope_id;

Sockets (cont'd)—Socket Data Structures

- The sin6_family is set to AF_INET6
  - AF_INET6 is a constant defined as the number 28 in base 10
  - selects TCP/IP using IPv6 addressing
  - corresponds to a protocol family called PF_INET6
    - the internet protocol family for IPv6, which is also number 28
- sin6_port value is used to store an IPv6 port number (same as IPv4 port numbers)
- sin6_addr stores the IPv6 address
- sin6_flowinfo is for IPv6 traffic class and flow information
- sin6_scope_id is the set of interfaces for a scope

Sockets (cont'd)—Socket Data Structures

- socklen_t
  - a 32 bit integer
  - used to define the size of an address (among other things)

Sockets (cont'd)—Socket Data Structures

- instead of using either sockaddr_in or sockaddr_in6, you could use *sockaddr_storage*
  - can be used to hold and pass around either IPv4 or IPv6 addresses
  - likely to have to do a lot of typecasting, since the various socket API calls (connect, accept, etc.) depend on the sockaddr struct

Sockets (cont'd)—Socket Library Calls

- An application calls a "socket" to create a new socket for network communication
- returns a descriptor for the newly created socket
- arguments to the socket call include:
- protocol family
- type of service (stream or datagram).
  - stream means a data connection is being used where several blocks will be sent (this is how TCP/IP works)
  - datagram means a block of data is sent all by itself without a regular connection being established (this is how UDP works)

Sockets (cont'd)—Socket Library Calls

Summary of main socket calls:

- Socket—creates a descriptor for use in network communications
- connect—connect to a remote peer (client)
- write—send outgoing data across a connection
- read—acquire incoming data from a connection
- close—terminate communication and deallocate a descriptor
- bind—bind a local IP address and protocol port to a socket
- listen—set the socket listening on the given address and port for connections from the client and set the number of incoming connections from a client (backlog) that will be allowed in the listen queue at any one time
- accept—accept the next incoming connection (server)
- recv—receive the next incoming datagram
- recvmsg—receive the next incoming datagram (variation of recv)
- recvfrom—receive the next incoming datagram and record its source endpoint address
- send—send an outgoing datagram
- sendmsg—send an outgoing datagram (variation of send)
- sendto—send an outgoing datagram, usually to a prerecorded endpoint address
- shutdown—terminate a TCP connection in one or both directions
- getpeername—after a connection arrives, obtain the remote machine's endpoint address from a socket
- getsockopt—obtain the current options for a socket
- setsockopt—change the options for a socket

Sockets (cont'd)—Socket Library Calls

- write()—writes to an operating system buffer, and blocks when the buffer is full
- When read() is used with the TCP/IP protocol, it extracts bytes and copies them to user's buffer. It blocks if there is no input data. When there is data, it fills the receiving buffer, then stops.
- When read() is used with the UDP protocol, it extracts one incoming UDP message. If the buffer cannot hold the entire message, read() fills the buffer and discards all remaining data in the UDP message
- Note that you can also use a recv instead of a read, and you could also do recvmsg, recvfrom, sendmsg, sendto. These would allow you to send a message without waiting for a response to tell whether the message was received by the other side or not
  - This is datagram type operation

Sidebar Big Endian vs. Little Endian

- Let's choose the number 70A32C17 base 16, and assume it is stored starting at byte address 100.
- In big endian this would be stored as follows:
  - 100: 70
  - 101: A3
  - 102: 2C
  - 103: 17

- whereas in little endian this would be stored as follows:
  - 100: 17
  - 101: 2C
  - 102: A3
  - 103: 70

Network Byte Order with Sockets

- Network byte order—a standard representation specifies a standard representation for binary integers in protocol headers in TCP/IP
  - specifies integers with the most significant byte first (big endian).

- Although the protocol software hides headers from application programs, a socket programmer has to understand network byte order
  - since some socket routines require arguments to be stored in network byte order
  - For example, the protocol port field of a sockaddr_in structure uses network byte order

Network Byte Order with Sockets

- socket routines include several functions that convert integers between network byte order and the local host's byte order
  - programs must explicitly call the conversion routines
  - and always should call them, even when not necessary, for portability
- Short conversion routines – operate on 16 bit integers (unsigned):
  - Host to network short (htons)
  - Network to host short (ntohs)
- Long conversion routines – operate on 32 bit integers (unsigned):
  - Host to network long (htonl)
  - Network to host long (ntohl)

Sockets (cont'd)—Flow of Operations

Client Flow	**Server Flow**
	socket(…);
	bind (IP address, protocol port);
	listen(…);
socket(…);	while not end of time allowed
	{
connect (…);	
	accept(…);

Sockets (cont'd)—Flow of Operations (cont'd)

Client Flow	**Server Flow (cont'd)**
write (…);	
	read(…);
write(…)	
	read (…);
close ();	
	close();
	}

Sockets (cont'd)—Simple Socket Server

```
server_socket_fd = socket(      AF_INET,
                                SOCK_STREAM,
                                0
                        );
```

Sockets (cont'd)—Simple Socket Server (cont'd)

```
bind(   server_socket_fd,
        (struct sockaddr *) &server_addr,
        sizeof(server_addr))
```

```
listen(server_socket_fd,  3);
```

Sockets (cont'd)—Simple Socket Server (cont'd)

```
client_conn_fd = accept(  server_socket_fd,

                          (struct sockaddr *) &client_addr,

                          (socklen_t *) &client_addr_size

                );
```

Sockets (cont'd)—Simple Socket Server (cont'd)

```
client_message_length = read(   client_conn_fd,
                                message_from_client,
                                  255
                                );
```

Sockets (cont'd)—Simple Socket Client

```
inet_pton(          AF_INET,
                    IP_address,
                    &serv_addr.sin_addr
          )
```

Sockets (cont'd)—Simple Socket Client (cont'd)

```
connect(      socket_fd,

              (struct sockaddr *) &serv_addr,

              sizeof(serv_addr)

        )
```

What are Remote Procedure Calls?

- The first attempts at middleware were things like sockets and Open Network Computing (ONC) Remote Procedure Calls (RPCs)
- With Remote Procedure Calls (such as ONC RPC), the idea is that you treat a call across the network the same as you would treat a call to a local procedure

What are Remote Procedure Calls?

- RPCs are functionally-oriented because they came along before the object-oriented paradigm was widely used in industry
  - Functionally-oriented means they focused on procedure calls—data was separate
- Note, however, that how various technologies that employ Distributed Objects work is often also called, in a generic way, a remote procedure call

What are Remote Procedure Calls?

- RPCs are functionally-oriented because they came along before the object-oriented paradigm was widely used in industry
  - Functionally-oriented means they focused on procedure calls—data was separate
- Note, however, that how various technologies that employ Distributed Objects work is often also called, in a generic way, a remote procedure call

What are Remote Procedure Calls?

ONC RPCs are a well-known RPC-based technology

- originally from Sun Microsystems
- date from the mid-1980s
- the remote procedures that are to be called across the network are defined in a file with a “.x” extension
- Then a program called “rpcgen” is used to translate this .x file into handlers in the C language
  - a stub file is created for the client side and later linked with client code
  - a skeleton file is created for the server side and later linked with server code

Synchronous vs. Asynchronous

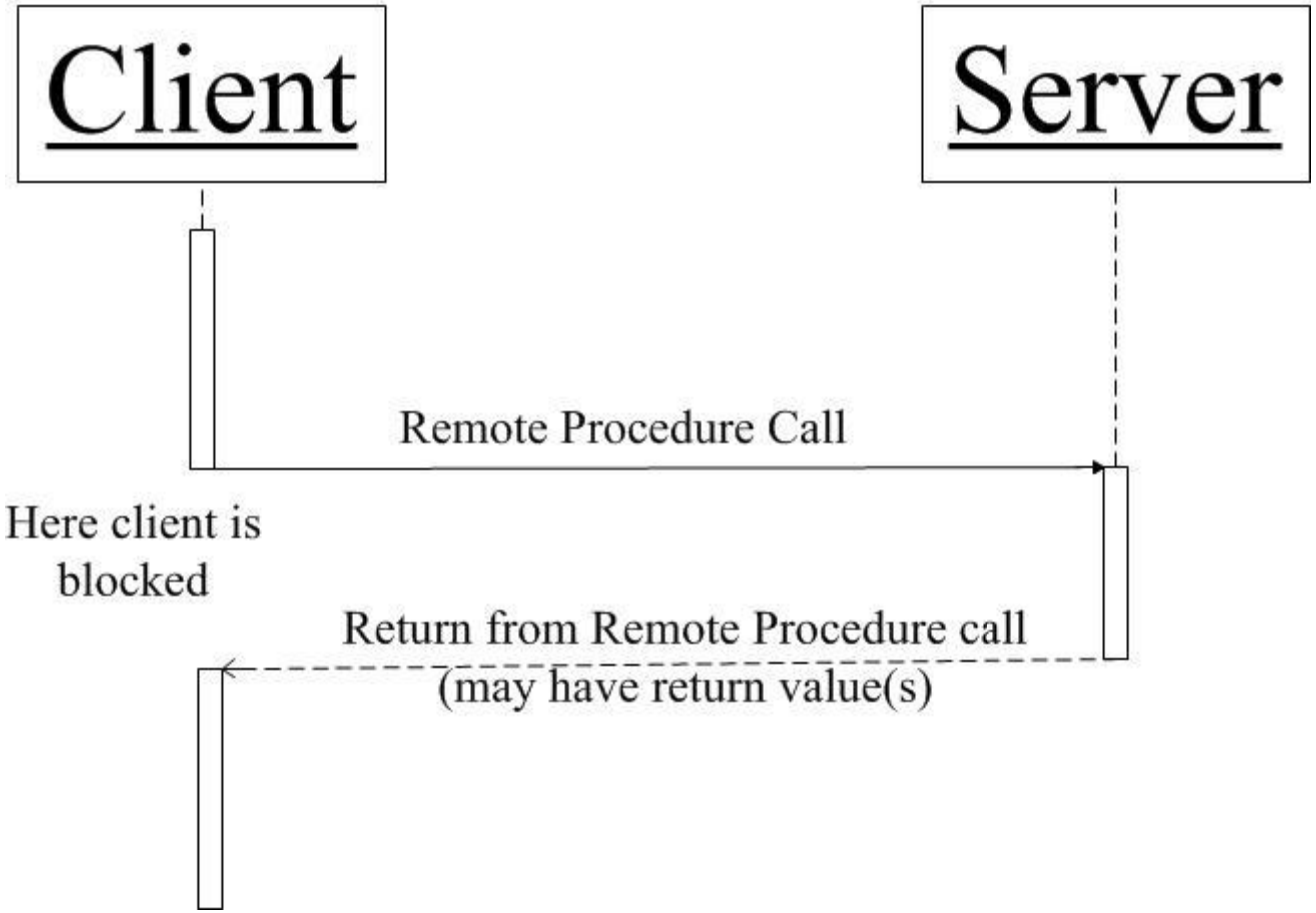
In synchronous communication

- the client calls the server
- then the client blocks and waits for the server to finish

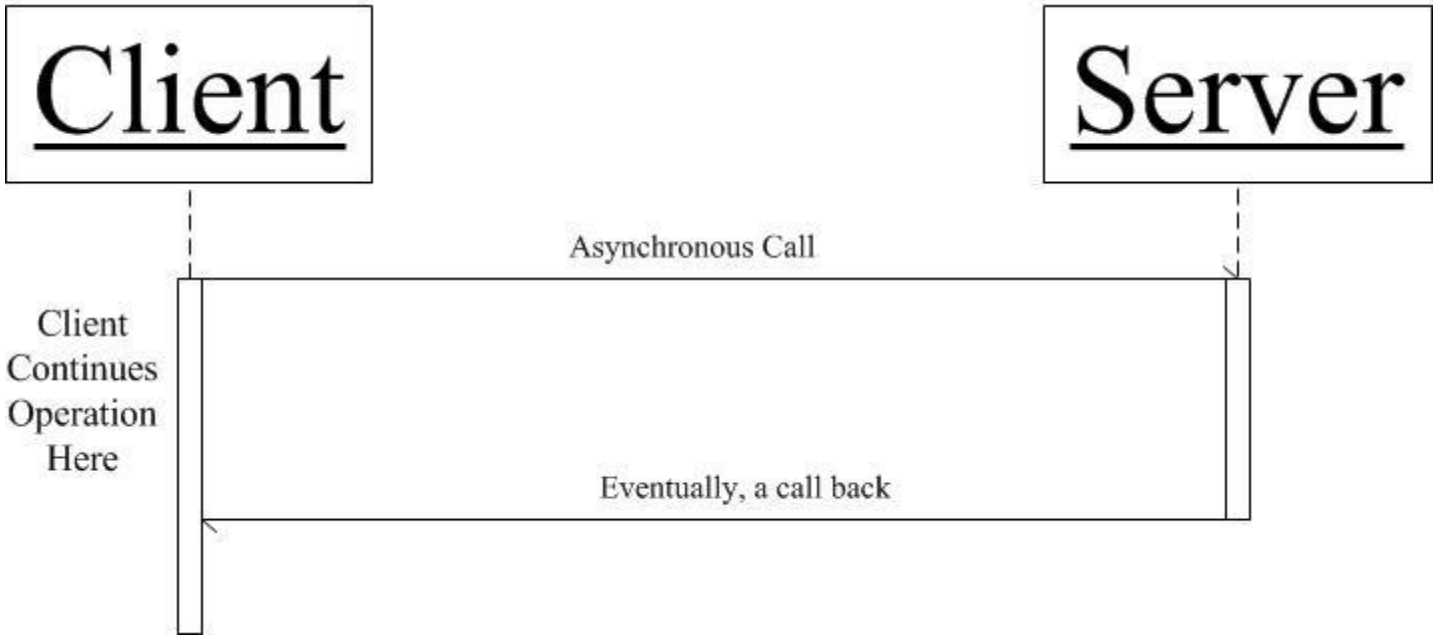
In asynchronous communication

- the client calls the server
- instead of blocking and waiting for the server's response, it goes on about its business

Synchronous vs. Asynchronous (cont'd)—Synchronous



Synchronous vs. Asynchronous (cont'd)—Asynchronous—Callback Version



What are Distributed Object-Oriented Components?

In the object-oriented paradigm, of course data and procedure calls are encapsulated inside an object

- So distributed objects just refer to objects that are located on different computers

What are Distributed Object-Oriented Components?

Generally speaking,component-based software engineering consists of combining loosely coupled components into systems

What are Distributed Object-Oriented Components?

There have been arguments about whether there is, in fact, a difference between component-based systems and object-oriented systems

- objects in an object-oriented system focus on modeling real world situations
- components in a component-based system are focused solely on combining existing components into systems
- but where do the existing components come from in the first place goes the other argument

What are Distributed Object-Oriented Components?

Gomaa (2011) defines a distributed component as follows:

> “a distributed component is a concurrent object with a well-defined interface, which is a logical unit of distribution and deployment. A well-designed component is capable of being reused in applications other than the one for which it was originally developed.”

What is Message Oriented Middleware?

Message oriented middleware (MOM)

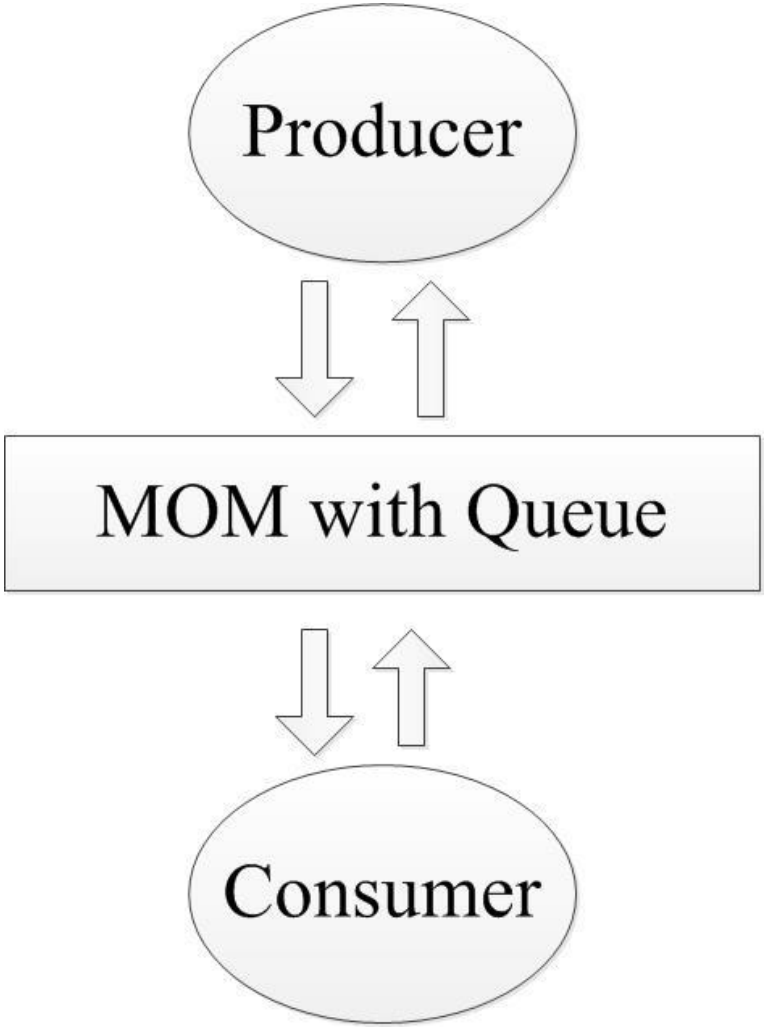
- purpose is to provide a middleman in between the client and the server
  - may do additional processing, perhaps Quality of Service-type processing
- the client talks to the middleman
- the server talks to the middleman,
- the client doesn't have to talk directly to the server
- the server doesn't have to talk directly to the client.
- nearly always uses a message queue to store messages between message producer and message consumer
- asynchronous

What is Message Oriented Middleware?

Message oriented middleware (MOM)

- purpose is to provide a middleman in between the client and the server
  - may do additional processing, perhaps Quality of Service-type processing
- the client talks to the middleman
- the server talks to the middleman,
- the client doesn't have to talk directly to the server
- the server doesn't have to talk directly to the client.
- nearly always uses a message queue to store messages between message producer and message consumer
- asynchronous

What is Message Oriented Middleware? (cont'd)

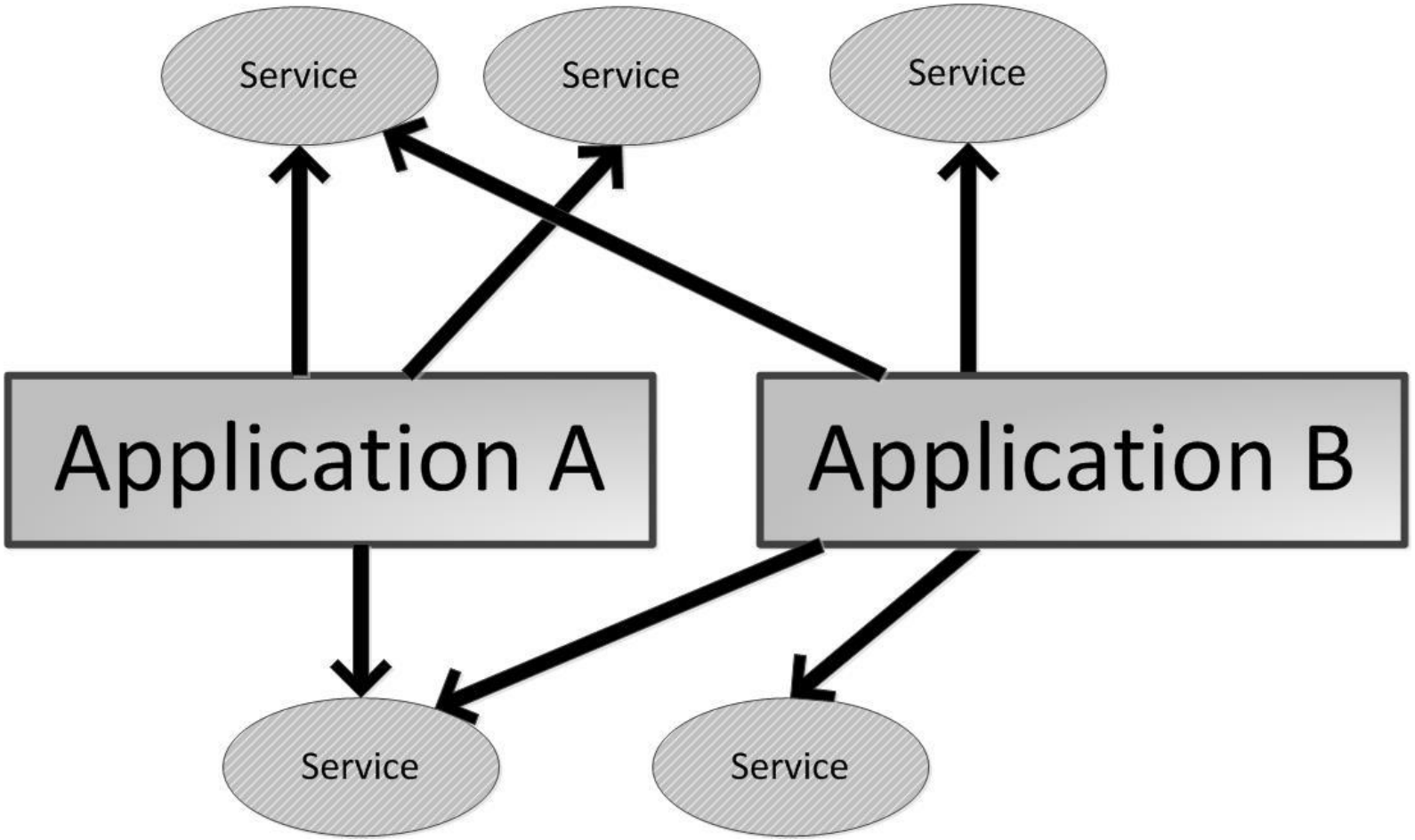


What are Service Oriented Architectures?

According to the Open Group a *Service Oriented Architecture (SOA)* is an architectural style that supports service orientation

- Service orientation is a way of thinking in terms of the outcomes of services, and how they can be developed and combined
- a *service* is a repeatable business activity that can be logically represented, the Open Group gives the example: “check customer credit”
- a service is self-contained
- a service may be composed of other services
- consumers of the service treat the service as a black box

What are Service Oriented Architectures? (cont'd)



What are Service Oriented Architectures? (cont'd)

- Services may be located on different computers
- Each service has a well defined, well documented interface
- Services are independent and may be reused by different applications
- Services are loosely coupled with any of the calling applications
- It is common to implement service oriented architectures with web services
  - It is possible to alternately use other implementations, such as distributed object components

What are Web Services?

- Web services are applications that typically expect to make use of the world wide web to provide application services
  - We distinguish the “world wide web” from the “internet” The internet is connected using the TCP/IP protocols
  - The world wide web is a collection of information in the form of web pages, that are connected using hypertext (clickable) links

What are Web Services?

- Web services are applications that typically expect to make use of the world wide web to provide application services
  - We distinguish the “world wide web” from the “internet” The internet is connected using the TCP/IP protocols
  - The world wide web is a collection of information in the form of web pages, that are connected using hypertext (clickable) links

What are Web Services? (cont'd)

- There are two kinds (architectural styles) of technologies that are commonly used for web services:
  - non-RESTful web services implemented with WSDL and SOAP technologies that treat HTTP as an independent lower protocol layer
  - the RESTful web services that use HTTP directly

What is Cloud Computing?

- IBM (2016) defines cloud computing as follows:

  “Cloud computing, often referred to as simply “the cloud,” is the delivery of on-demand computing resources—everything from applications to data centers—over the Internet on a pay-for-use basis.”

- Applications running on the cloud often employ a service oriented architecture

What is Cloud Computing?

- Public cloud:
  - instead of doing your own computing and storing your data on the computer on your desktop yourself
    - you hire a company to do the computing and store the data on their big computer servers that you access via the web
- Private cloud:
  - instead of having computing and data storage on employees' desks
  - the company can buy its own big servers
    - then the employees do their computing and store their data on those servers
- Hybrid cloud:
  - Part of a company's computing is done in house
  - Part of a company's computing is done by a public cloud

What is Cloud Computing?

- There are three different paradigms for cloud computing:
  - Infrastructure as a Service (IaaS)
    - you or your company pays a cloud provider for computing resources
    - You provide your own operating system and application software
  - Platform as a Service (PaaS)
    - you or your company pays a cloud provider for an environment provided by the company that provides everything you need in order to develop and run your applications
      - This environment includes operating system, development tools, web site hosting, among other things
  - Software as a Service (SaaS)
    - you or your company pays a cloud provider for the use of their software application

Environmental Monitoring Project

- A program used to illustrate various technologies in this textbooks
- Sensors will monitor Lake Guntersville for
  - Temperature
  - air quality
  - water quality
- Data is sent back to the University of Alabama in Huntsville (UAH) for analysis

Sailboat Marina Project

- A program used to illustrate various technologies in this textbooks
- A web-based application will be used to manage a large sailboat marina that
  - employs a database to keep track of:
    - Sailboats
    - Owners
    - Whether owners have paid slip rental or not
  - Handles sailboat maintenance requests from owners