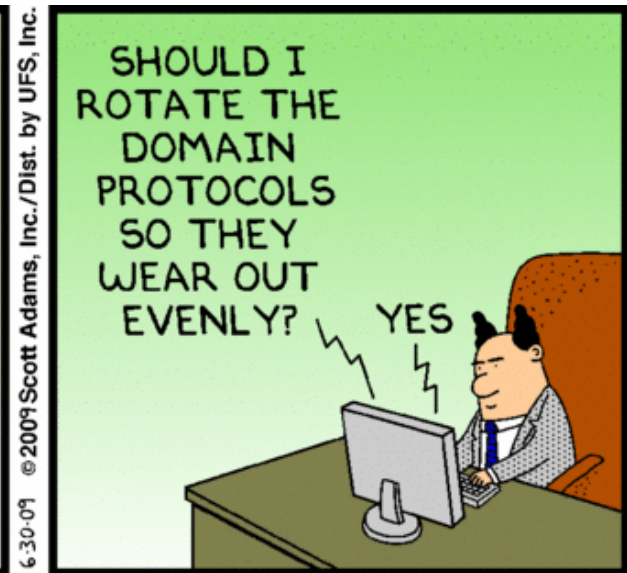


Lecture 27: Distributed Systems

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day



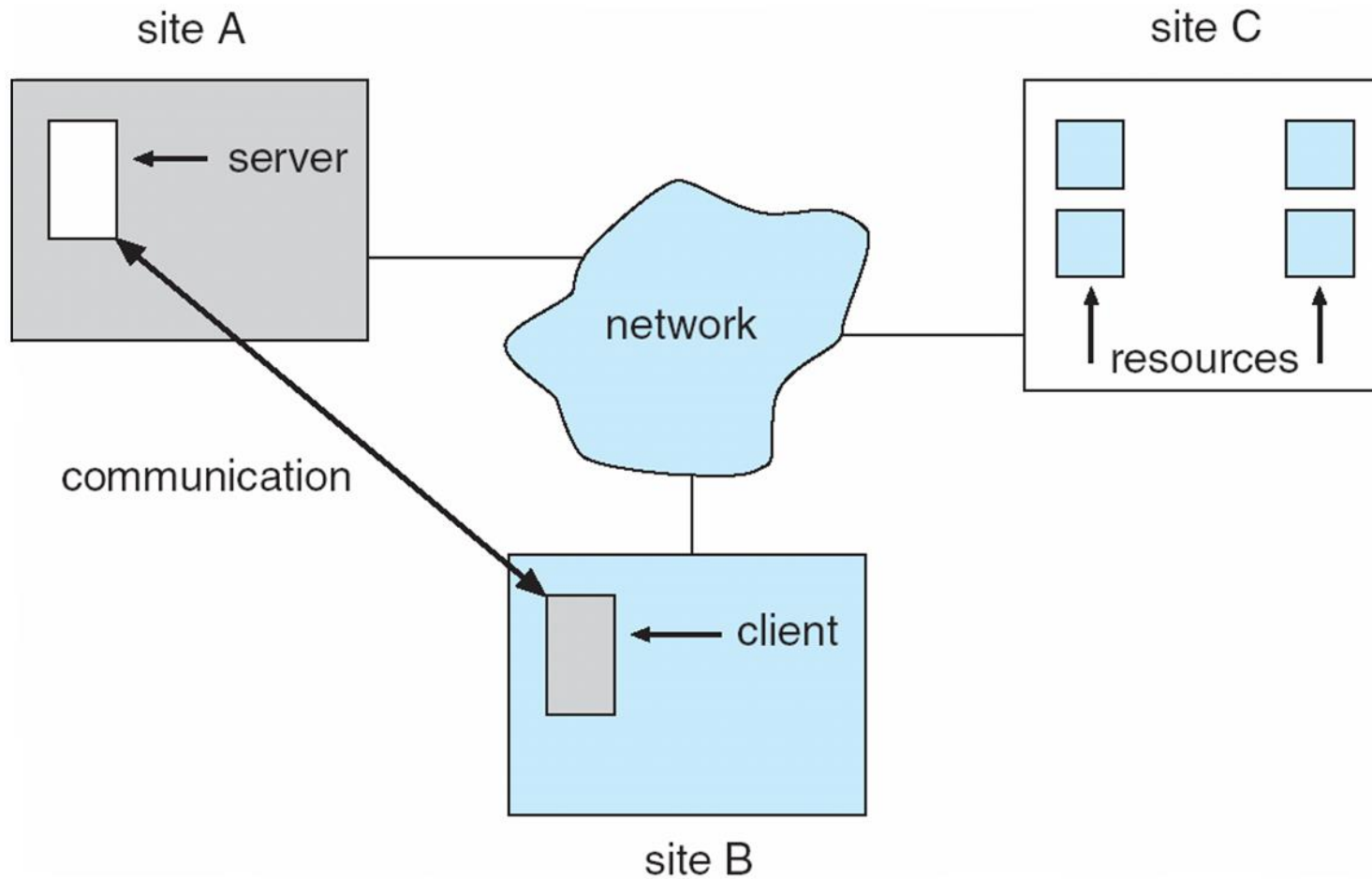
Chapter 17: Distributed Systems

- Motivation
- Types of Network-Based Operating Systems
- Network Structure
- Communication Structure
- Communication Protocols
- TCP/IP
- Java Sockets Programming
- Cloud Computing

Motivation

- **Distributed system** is collection of loosely coupled processors interconnected by a communications network
- Processors variously called *nodes*, *computers*, *machines*, *hosts*
 - *Site* is location of the processor
- Reasons for distributed systems
 - Resource sharing
 - ▶ sharing and printing files at remote sites
 - ▶ processing information in a distributed database
 - ▶ using remote specialized hardware devices
 - Computation speedup – **load sharing**
 - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
 - Communication – message passing

A Distributed System



Distributed Operating Systems

- Users not aware of multiplicity of machines
 - Access to remote resources similar to access to local resources
- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
- Computation Migration – transfer the computation, rather than the data, across the system

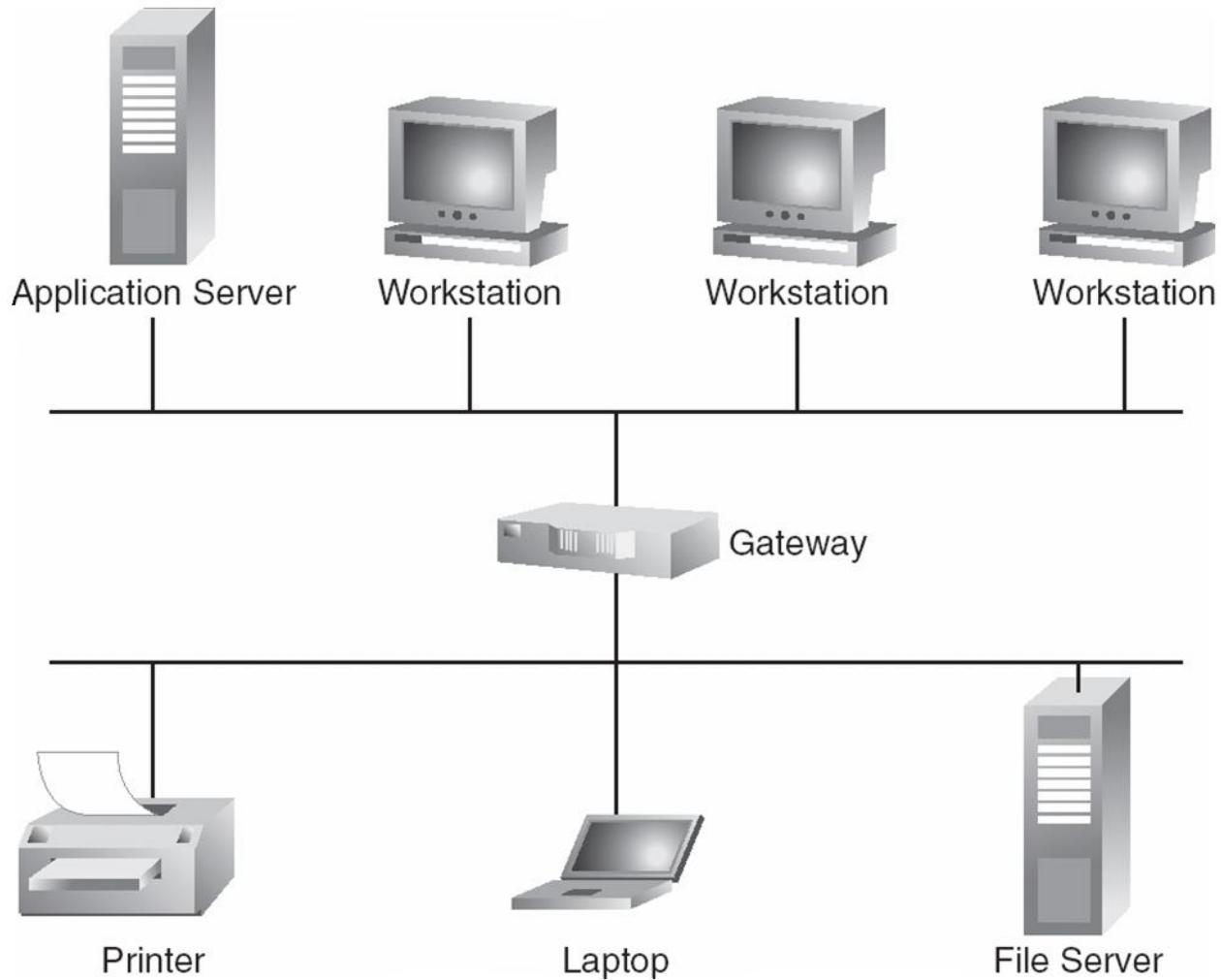
Distributed Operating Systems (cont.)

- Process Migration – execute an entire process, or parts of it, at different sites
 - **Load balancing** – distribute processes across network to even the workload
 - **Computation speedup** – subprocesses can run concurrently on different sites
 - **Hardware preference** – process execution may require specialized processor
 - **Software preference** – required software may be available at only a particular site
 - **Data access** – run process remotely, rather than transfer all data locally

Network Structure

- **Local-Area Network (LAN)** – designed to cover small geographical area.
 - Multiaccess bus, ring, or star network
 - Speed $\approx 10 - 100$ megabits/second
 - Broadcast is fast and cheap
 - Nodes:
 - ▶ usually workstations and/or personal computers
 - ▶ a few (usually one or two) mainframes

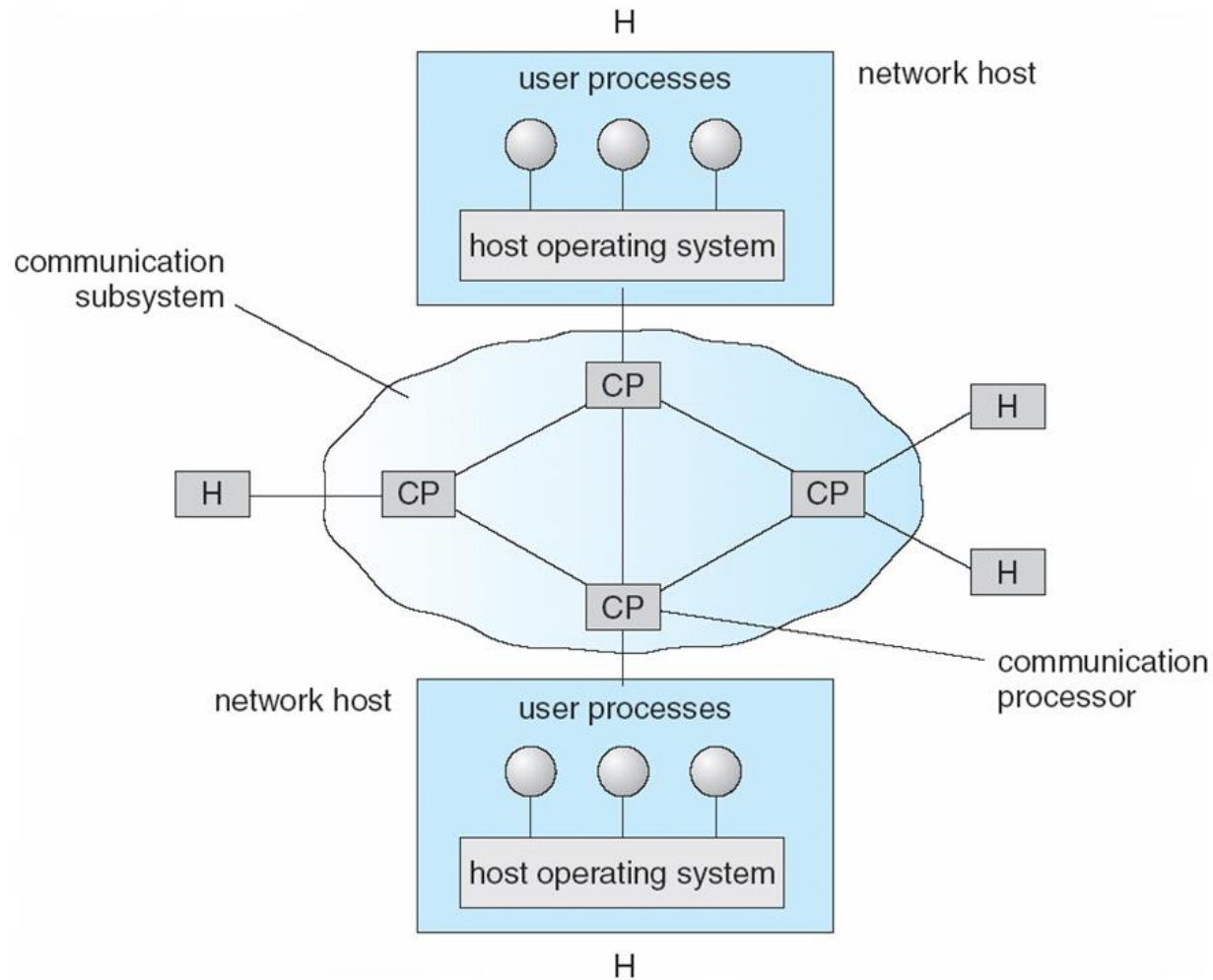
Depiction of typical LAN



Network Types (Cont.)

- **Wide-Area Network (WAN)** – links geographically separated sites
 - Point-to-point connections over long-haul lines (often leased from a phone company)
 - Speed \approx 1.544 – 45 megbits/second
 - Broadcast usually requires multiple messages
 - Nodes:
 - ▶ usually a high percentage of mainframes

Communication Processors in a Wide-Area Network



Communication Structure

The design of a *communication* network must address four basic issues:

- **Naming and name resolution** - How do two processes locate each other to communicate?
- **Routing strategies** - How are messages sent through the network?
- **Connection strategies** - How do two processes send a sequence of messages?
- **Contention** - The network is a shared resource, so how do we resolve conflicting demands for its use?

Naming and Name Resolution

- Name systems in the network
- Address messages with the process-id
- Identify processes on remote systems by
 <host-name, identifier> pair
- **Domain name service (DNS)** – specifies the naming structure of the hosts, as well as name to address resolution (Internet)

Routing Strategies

- **Fixed routing** - A path from *A* to *B* is specified in advance; path changes only if a hardware failure disables it
 - Since the shortest path is usually chosen, communication costs are minimized
 - Fixed routing cannot adapt to load changes
 - Ensures that messages will be delivered in the order in which they were sent

- **Virtual circuit** - A path from *A* to *B* is fixed for the duration of one session. Different sessions involving messages from *A* to *B* may have different paths
 - Partial remedy to adapting to load changes
 - Ensures that messages will be delivered in the order in which they were sent

Routing Strategies (Cont.)

- **Dynamic routing** - The path used to send a message from site *A* to site *B* is chosen only when a message is sent
 - Usually a site sends a message to another site on the link least used at that particular time
 - Adapts to load changes by avoiding routing messages on heavily used path
 - Messages may arrive out of order
 - ▶ This problem can be remedied by appending a sequence number to each message

Connection Strategies

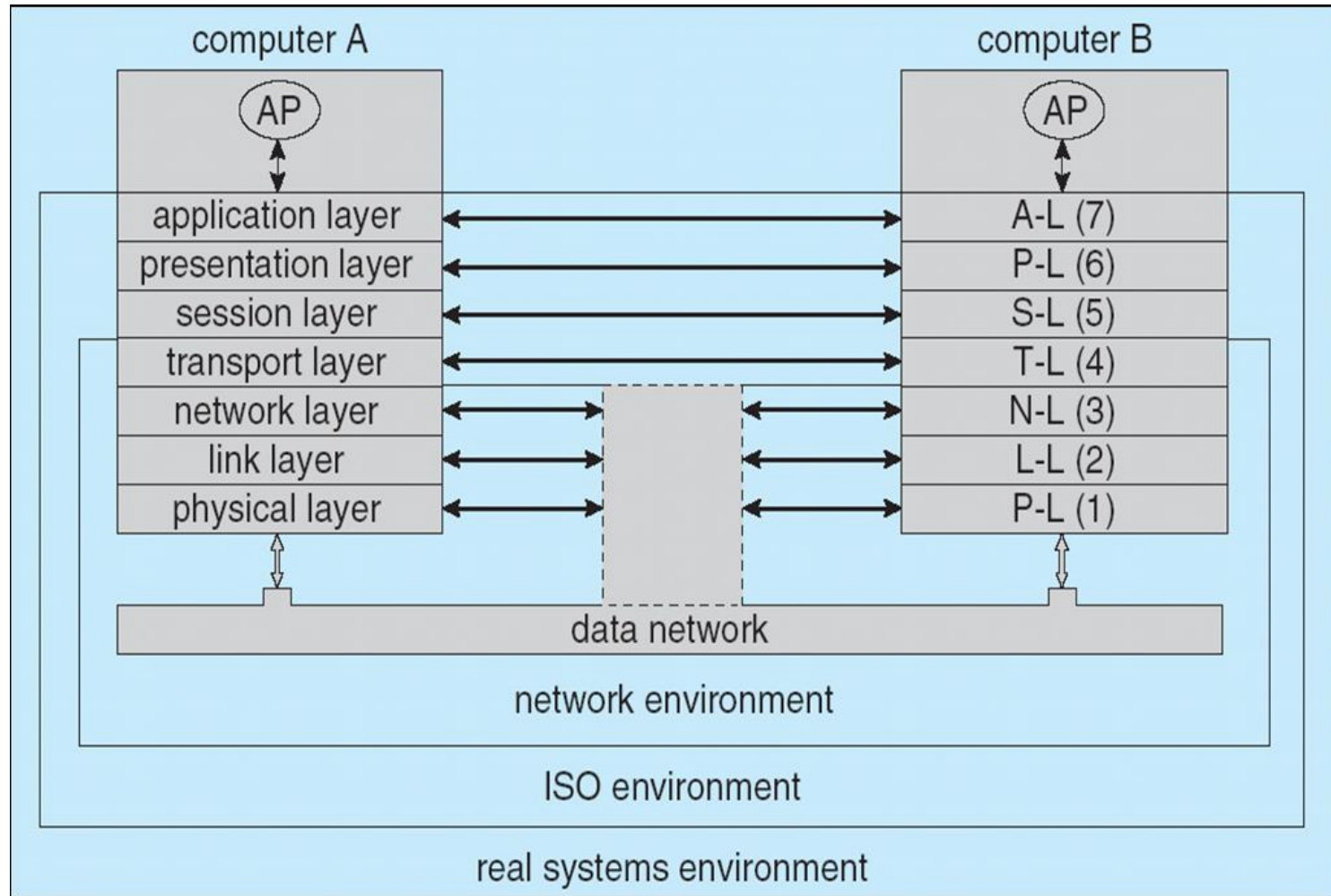
- **Circuit switching** - A permanent physical link is established for the duration of the communication (i.e., telephone system)
- **Message switching** - A temporary link is established for the duration of one message transfer (i.e., post-office mailing system)
- **Packet switching** - Messages of variable length are divided into fixed-length packets which are sent to the destination
 - Each packet may take a different path through the network
 - The packets must be reassembled into messages as they arrive
- Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth
 - Message and packet switching require less setup time, but incur more overhead per message

Contention

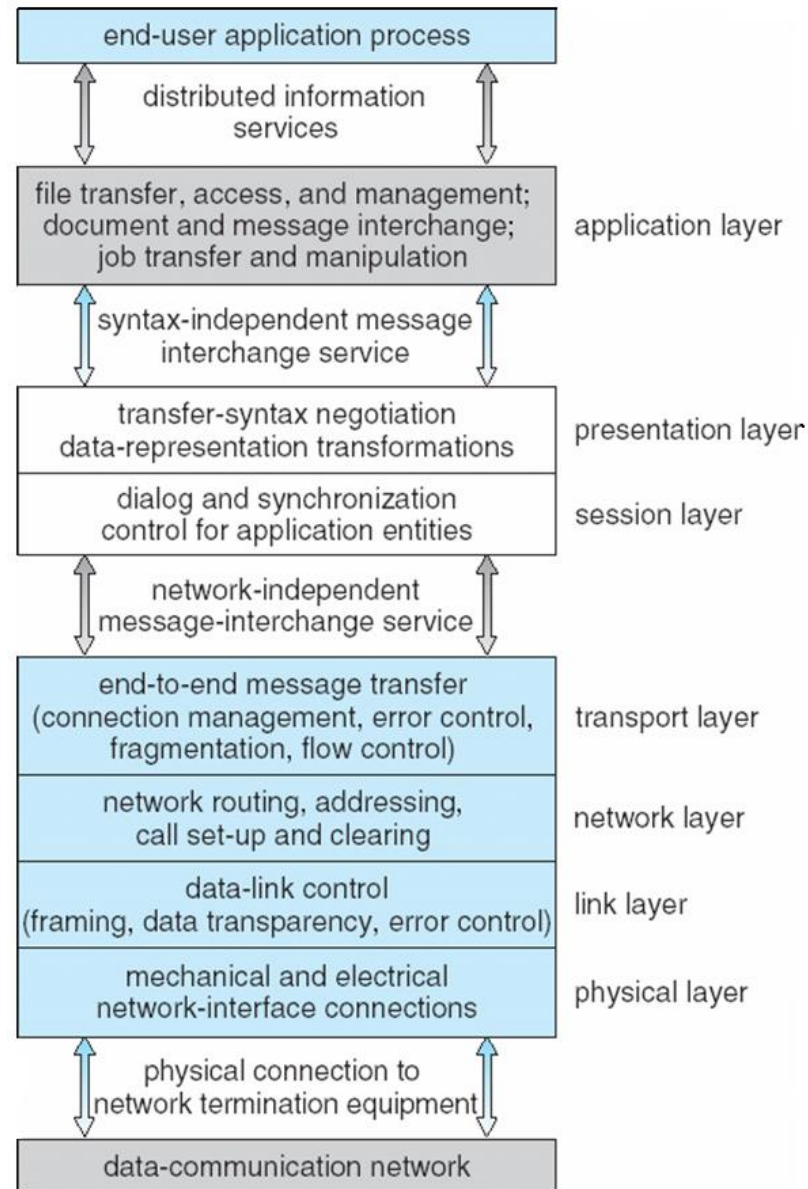
Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

- **CSMA/CD** - Carrier sense with multiple access (CSMA); collision detection (CD)
 - A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting
 - When the system is very busy, many collisions may occur, and thus performance may be degraded
- CSMA/CD is used successfully in the Ethernet system, the most common network system

OSI ISO 7-Layer Reference Model

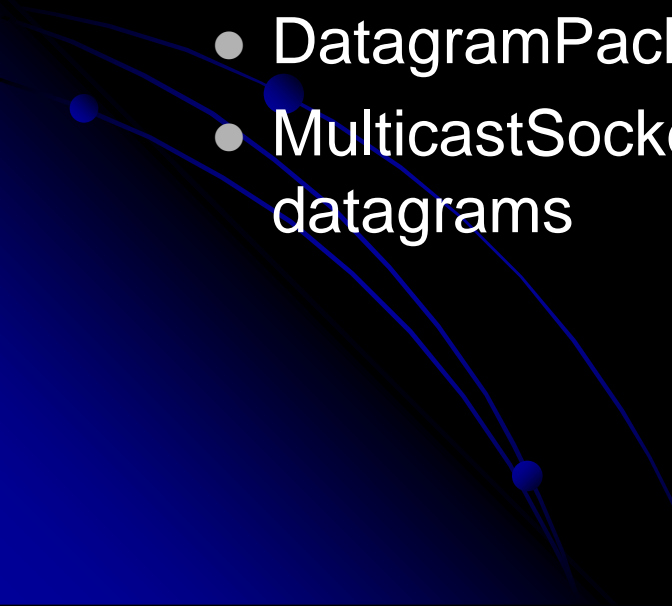


The ISO Protocol Layer



Java Sockets Programming

Primary Classes

- TCP (stream) interface
 - ServerSocket – used by a server
 - Socket – used by a client
 - UDP (datagram) interface
 - DatagramSocket – for sending / receiving
 - DatagramPacket – the data that is sent / received
 - MulticastSocket – for sending / receiving multicast datagrams
- 

TCP Client/Server Interaction

Server (running on `hostid`)

Client

create socket for
incoming requests

`serverSocket =
new ServerSocket()`

wait for incoming
connection request
`connectionSocket =
serverSocket.accept()`

read request from
`connectionSocket`

write reply to
`connectionSocket`

close
`connectionSocket`

TCP

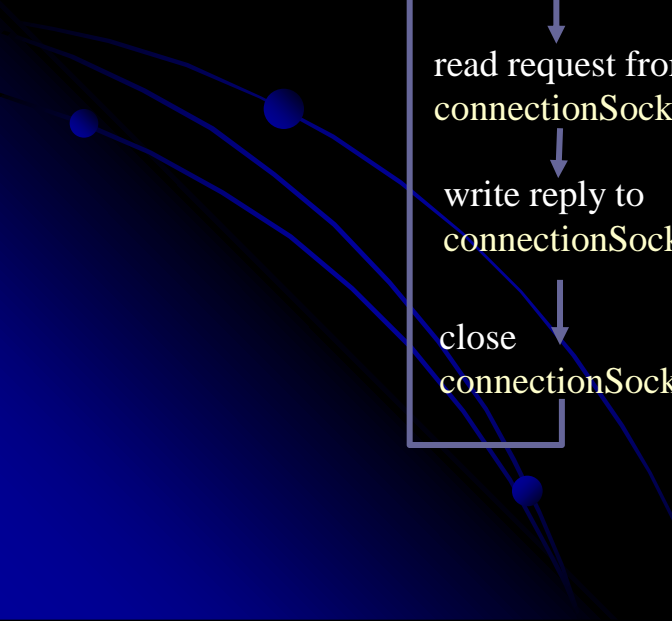
connection setup

create socket,
connect to **server**
`clientSocket =
new Socket()`

send request using
`clientSocket`

read reply from
`clientSocket`

close
`clientSocket`



TCP: ServerSocket

- ServerSocket() – constructor
- Accept() – wait for and accept a connection
- Close() – close the socket
- getInetAddress() – get local IP address
- getLocalPort() – get local port number

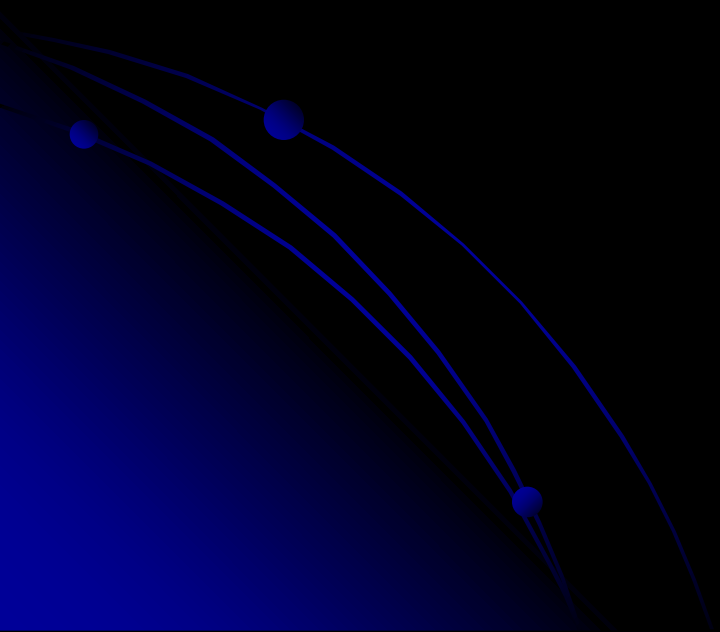


TCP: Creating a ServerSocket

- `Public ServerSocket(int port)` throws `IOException`
- `Public ServerSocket(int port, int backlog) ..`
 - Specify the local port number and the number of pending connection requests that are allowed (backlog)
- `Public ServerSocket(int port, int backlog, InetAddress addr)`
 - Specify the IP address (interface) to be used

TCP: `ServerSocket.Accept()`

- Public `Socket accept()` throws `IOException`
- Retrieves the first waiting connection request from the queue of pending connection requests



TCP: Socket

- Socket() – constructor
- Close()
- getInputStream()
- getOutputStream()
- getInetAddress()
- getLocalAddress()
- getLocalPort()
- getPort() – get the remote port
- ...other options for later..

TCP: Socket

- Specify only remote endpoint
 - `Socket(String host, int dstPort)` throws `UnknownHostException`, `IOException`
 - `Socket(InetAddress dstAddr, int dstPort)`
- Specify remote and local endpoints
 - `Socket(String host, int dstPort, InetAddress localAddr, int localPort)`
 - `Socket(InetAddress host, int dstPort, InetAddress localAddr, int localPort)`
- If `localPort` is 0, the socket is bound to any available port.

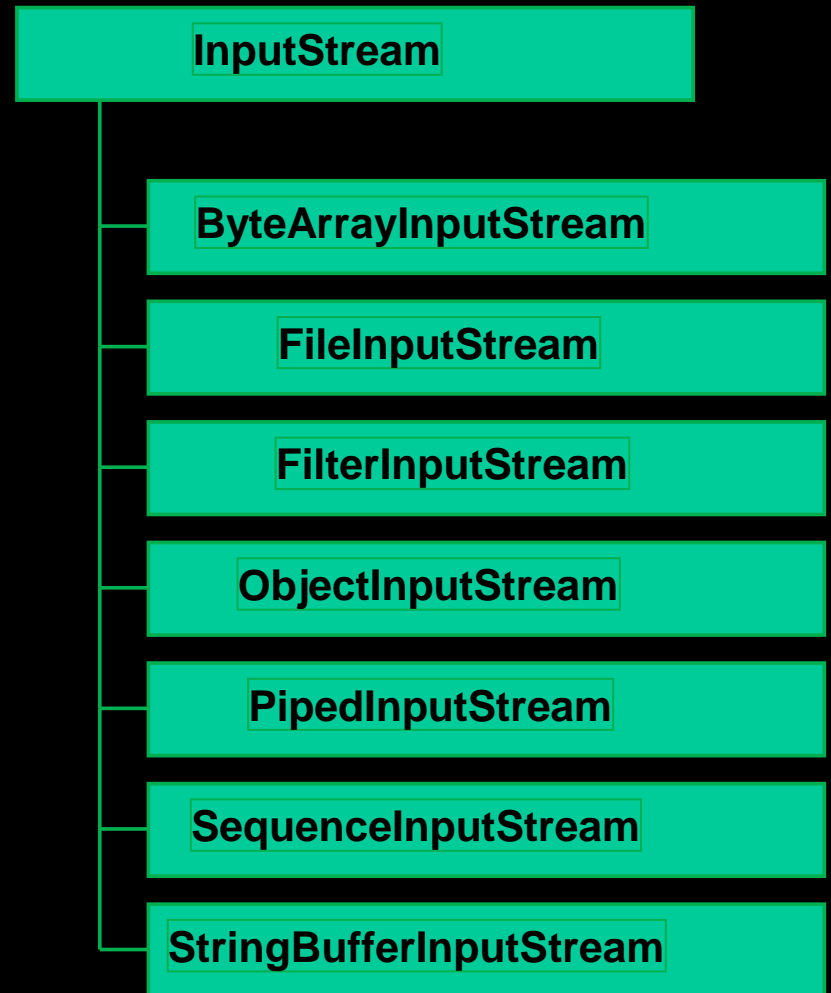
TCP: getInputStream

- Public InputStream getInputStream() throws IOException
- Example:

```
Socket client = new Socket(dst,port);  
InputStream in = client.getInputStream();  
for (int ch = in.read(); ch > 0 ; ch = in.read())  
{  
    System.out.print( (char) ch );  
}
```

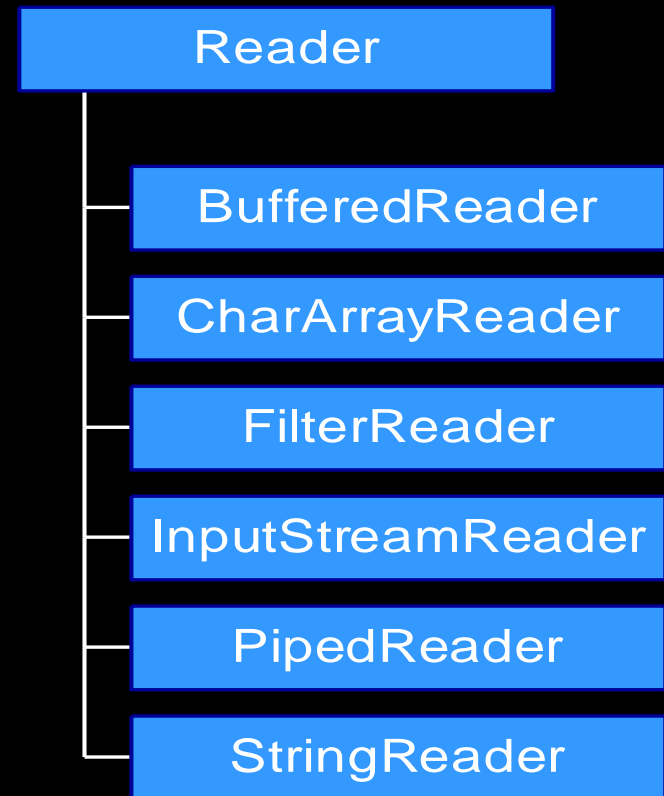
TCP: getInputStream

- InputStream reads byte streams
- Different subclasses to read from different sources; e.g., files, byte arrays, string buffers, etc.
- The Reader class reads character streams



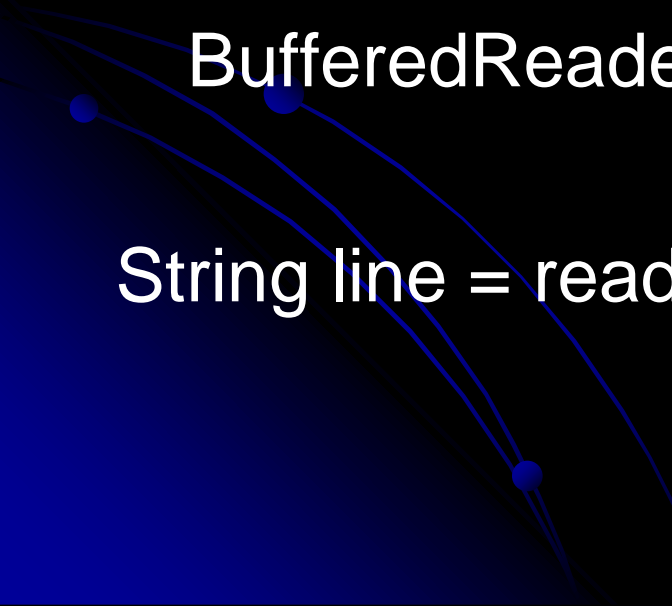
Reader

- The Reader class reads character streams
 - Read
 - Skip
 - Mark
 - Reset
 - Close
 - Ready – check for block
- A BufferedReader supports readLine()



Reading a line of input

```
Socket client = new Socket(dst,port);  
InputStream input = client.getInputStream();  
InputStreamReader inReader = new  
    InputStreamReader(input);  
BufferedReader reader = new  
    BufferedReader(inReader);  
  
String line = reader.readLine();
```



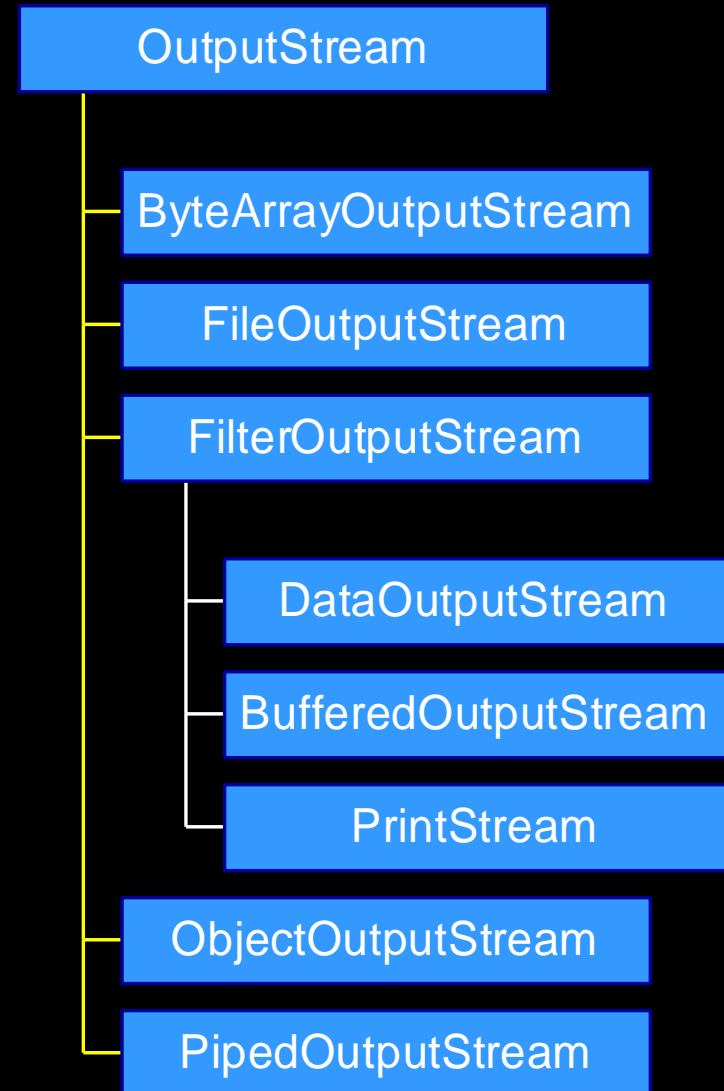
TCP: `getOutputStream()`

- Public `OutputStream getOutputStream()` throws `IOException`
- Example:

```
Socket client = new Socket(dst,port);  
OutputStream out = client.getOutputStream();  
String str = "data to send";  
byte[] obj = msg.getBytes();  
out.write(obj);  
out.flush();
```

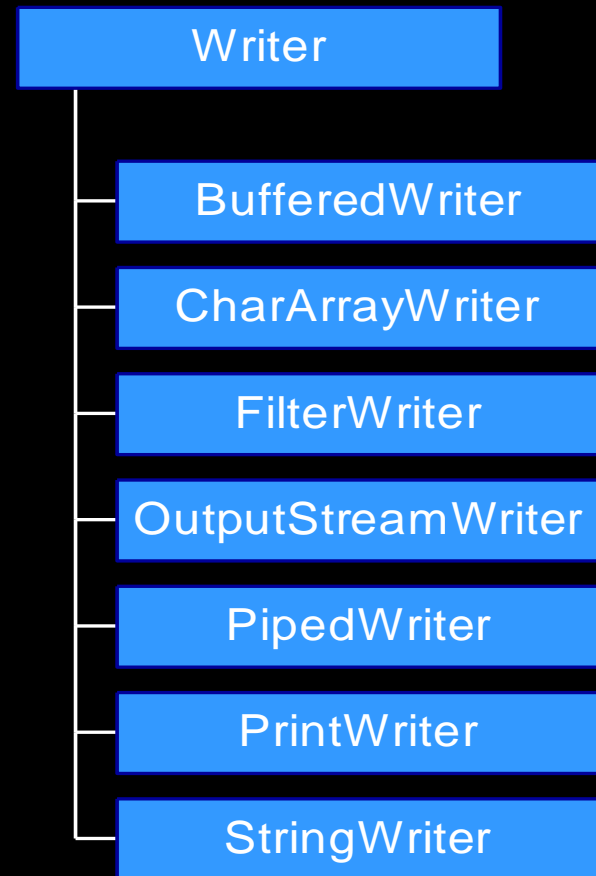
TCP: getOutputStream

- OutputStream writes byte streams
- Different subclasses write them from different sources (files, byte arrays, stringbuffers, etc)



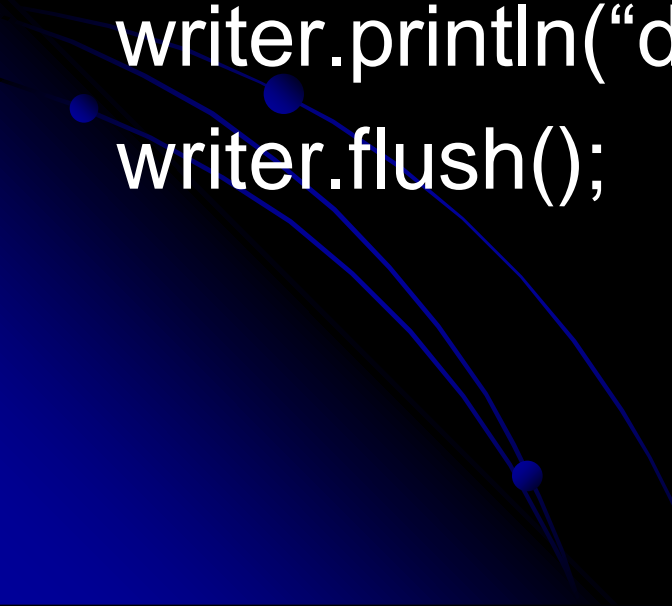
Writer

- The Writer class writes character streams
 - Read
 - Close
 - flush
- A PrintWriter supports print() and println()



Writing a line of output

```
Socket client = new Socket(dst,port);  
OutputStream output =  
    client.getOutputStream();  
PrintWriter writer = new PrintWriter(output);  
writer.println("data to send");  
writer.flush();
```



Example: Java client (TCP)

```
import java.io.*;  
import java.net.*;  
class TCPClient {
```

```
    public static void main(String argv[]) throws Exception  
    {  
        String sentence;  
        String modifiedSentence;
```

Create
input stream

```
        BufferedReader inFromUser =  
            new BufferedReader(new InputStreamReader(System.in));
```

Create
client socket,
connect to server

```
        Socket clientSocket = new Socket("hostname", 6789);
```

Create
output stream
attached to socket

```
        DataOutputStream outToServer =  
            new DataOutputStream(clientSocket.getOutputStream());
```

Example: Java client (TCP) (cont.)

Create
input stream
attached to socket

```
BufferedReader inFromServer =  
    new BufferedReader(new  
        InputStreamReader(clientSocket.getInputStream()));
```

```
sentence = inFromUser.readLine();
```

Send line
to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line
from server

```
modifiedSentence = inFromServer.readLine();  
  
System.out.println("FROM SERVER: " + modifiedSentence);  
  
clientSocket.close();
```

```
}
```

```
}
```

Example: Java server (TCP)

```
import java.io.*;  
import java.net.*;
```

```
class TCPServer {
```

```
    public static void main(String argv[]) throws Exception  
    {
```

```
        String clientSentence;
```

```
        String capitalizedSentence;
```

Create
welcoming socket
at port 6789

```
        ServerSocket welcomeSocket = new ServerSocket(6789);  
        while(true) {
```

Wait, on welcoming
socket for contact
by client

```
            Socket connectionSocket = welcomeSocket.accept();
```

Create input
stream, attached
to socket

```
            BufferedReader inFromClient =  
                new BufferedReader(new  
                    InputStreamReader(connectionSocket.getInputStream()));
```

Example: Java server (TCP) (cont.)

Create output stream, attached to socket → `DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());`

Read in line from socket → `clientSentence = inFromClient.readLine();`

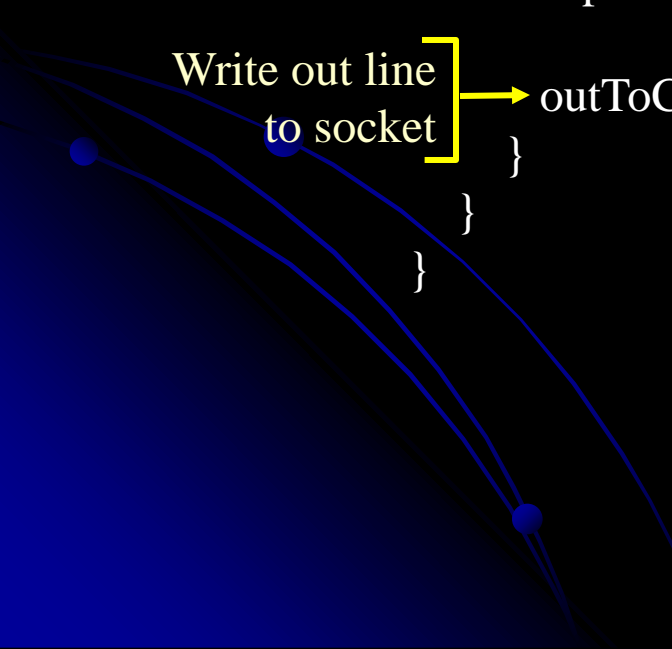
`capitalizedSentence = clientSentence.toUpperCase() + '\n';`

Write out line to socket → `outToClient.writeBytes(capitalizedSentence);`

`}`

`}`

`}`



Client/server socket interaction: UDP

Server (running on **hostid**)

Client

create socket,
port=**x**, for
incoming request:
serverSocket =
DatagramSocket()

read request from
serverSocket

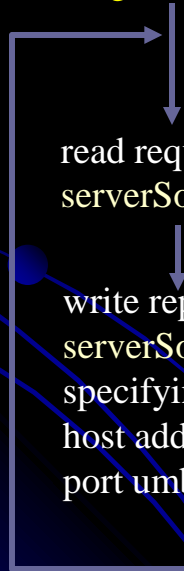
write reply to
serverSocket
specifying client
host address,
port number

create socket,
clientSocket =
DatagramSocket()

Create address (server hostid, port=**x**),
send datagram request
using clientSocket

read reply from
clientSocket

close
clientSocket



Example: Java client (UDP)

```
import java.io.*;  
import java.net.*;
```

```
class UDPClient {
```

```
    public static void main(String args[]) throws Exception  
    {
```

Create
input stream



```
        BufferedReader inFromUser =
```

```
            new BufferedReader(new InputStreamReader(System.in));
```

Create
client socket



```
        DatagramSocket clientSocket = new DatagramSocket();
```

Translate
hostname to IP
address using DNS



```
        InetAddress IPAddress = InetAddress.getByName("hostname");
```

```
        byte[] sendData = new byte[1024];
```

```
        byte[] receiveData = new byte[1024];
```

```
        String sentence = inFromUser.readLine();
```

```
        sendData = sentence.getBytes();
```


Example: Java client (UDP) (cont.)

Create datagram with
data-to-send,
length, IP addr, port

```
DatagramPacket sendPacket =  
new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram
to server

```
clientSocket.send(sendPacket);
```

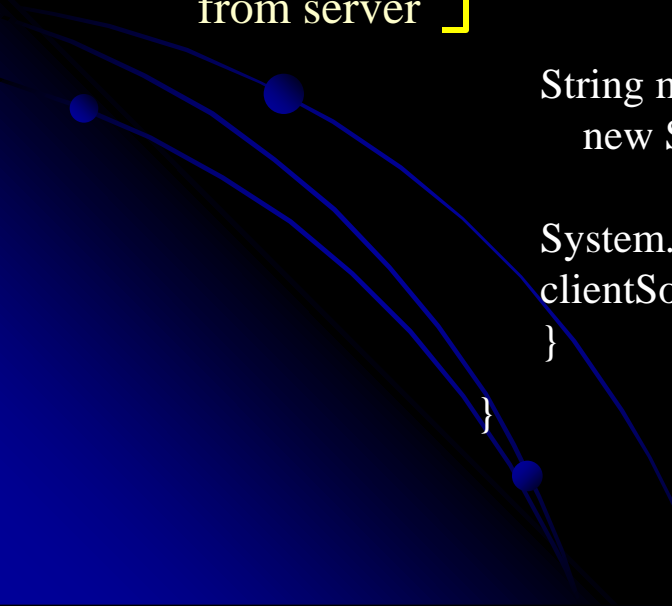
Read datagram
from server

```
DatagramPacket receivePacket =  
new DatagramPacket(receiveData, receiveData.length);
```

```
clientSocket.receive(receivePacket);
```

```
String modifiedSentence =  
new String(receivePacket.getData());
```

```
System.out.println("FROM SERVER:" + modifiedSentence);  
clientSocket.close();  
}
```



Example: Java server (UDP)

```
import java.io.*;  
import java.net.*;
```

```
class UDPServer {  
    public static void main(String args[]) throws Exception
```

Create
datagram socket,
bind to port 9876

```
{  
    DatagramSocket serverSocket = new DatagramSocket(9876);
```

```
    byte[] receiveData = new byte[1024];  
    byte[] sendData = new byte[1024];
```

```
    while(true)  
    {
```

Create space for
received datagram

```
        DatagramPacket receivePacket =  
            new DatagramPacket(receiveData, receiveData.length);
```

Receive datagram

```
        serverSocket.receive(receivePacket);
```

Example: Java server (UDP) (cont.)

```
String sentence = new String(receivePacket.getData());

Get IP addr and port number of sender ] InetAddress IPAddress = receivePacket.getAddress();
                                         int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

Create datagram to send to client ] DatagramPacket sendPacket =
                                   new DatagramPacket(sendData, sendData.length, IPAddress,
                                                         port);

Write out datagram to socket ] serverSocket.send(sendPacket);
                              }
                              }
                              }
```

The diagram illustrates the execution flow of the Java UDP server code. A blue curved line with dots represents the sequence of operations. Yellow arrows point from descriptive text to specific code lines. Brackets group related code lines under a single description.

- Get IP addr and port number of sender**: This description is linked to the lines `InetAddress IPAddress = receivePacket.getAddress();` and `int port = receivePacket.getPort();`.
- Create datagram to send to client**: This description is linked to the lines `DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);`.
- Write out datagram to socket**: This description is linked to the line `serverSocket.send(sendPacket);`.

Using sockets in php script

```
<?php
```

```
function checkSudokuString()
{
    $hostName = $_POST["hostName"];
    $fp = fsockopen("tcp://".$hostName, 9234, $errno, $errstr);
    if (!$fp) {
        echo "ERROR: $errno - $errstr<br/>\n";
        return "";
    }
    else
    {
        ...
        fwrite($fp,"3".$s);
        $contents = fread($fp, 100);
        fclose($fp);
        return $contents;
    }
}
?>
```

Android Sockets

- Same as Java sockets
- Need to give the application INTERNET permissions (just like WebView)
- Also, many network operations throw exceptions (as they can fail). In Java code they can be ignored, but in Android we have to handle them – add try/catch blocks:

```
try {  
    Ops that may fail and throw an exception;  
} catch (Exception e) {  
    Log.e(TAG, "Caught UDP Exception: " +  
        e.getMessage());  
    Toast.makeText(HelloNets.this, "UDP Error"+  
        e.getMessage(), Toast.LENGTH_LONG).show();  
}
```

Two Technologies for Agility

Virtualization:

*The ability to run multiple operating systems on a single physical system and share the underlying hardware resources**

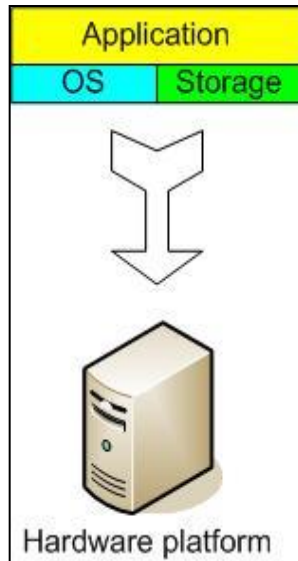
Cloud Computing:

*“The provisioning of services in a timely (near on instant), on-demand manner, to allow the scaling up and down of resources”***

* VMware white paper, *Virtualization Overview*

** Alan Williamson, quoted in *Cloud BootCamp March 2009*

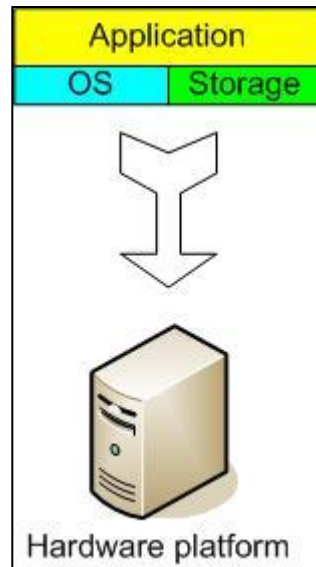
The Traditional Server Concept



Web Server

Windows

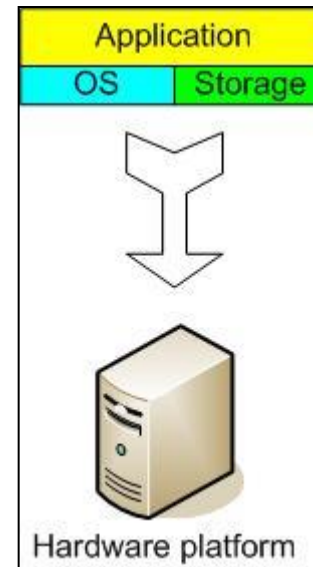
IIS



App Server

Linux

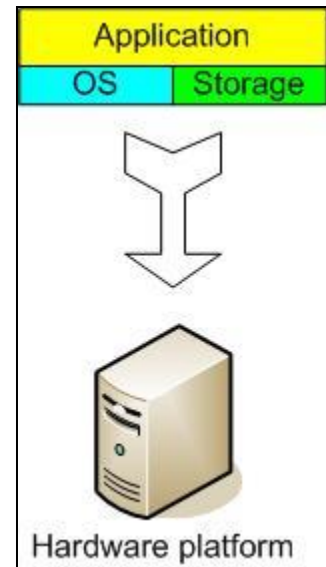
Glassfish



DB Server

Linux

MySQL

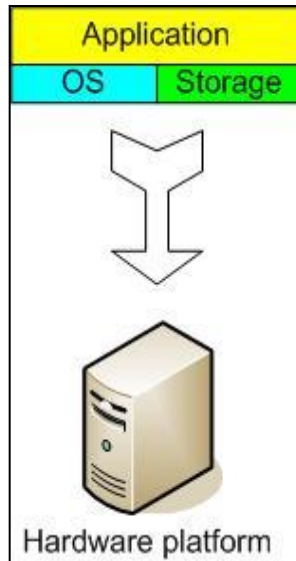


EMail

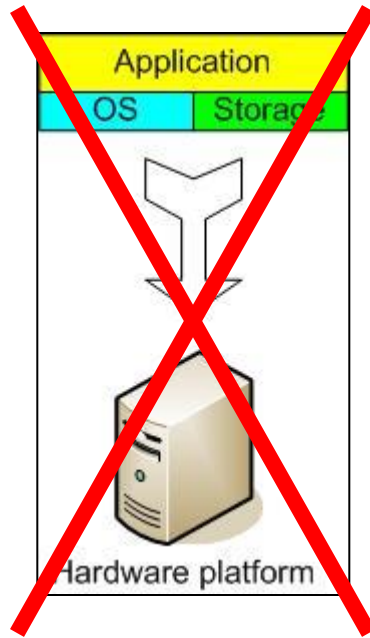
Windows

Exchange

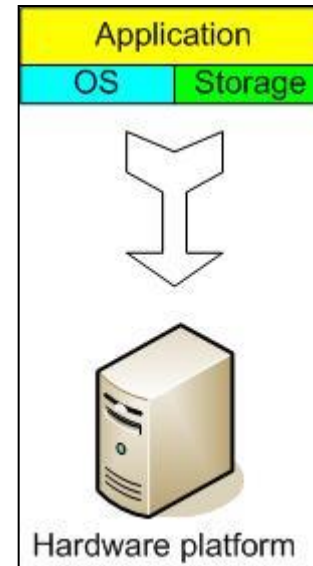
And if something goes wrong ...



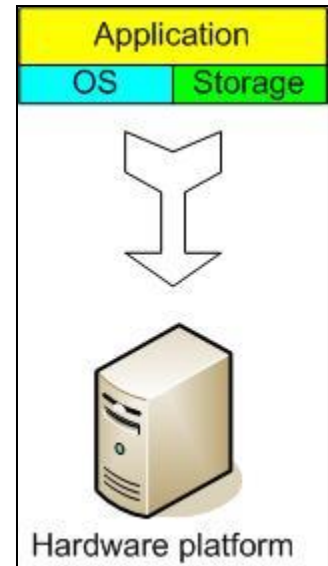
Web Server
Windows
IIS



App Server
DOWN!



DB Server
Linux
MySQL



EMail
Windows
Exchange

The Traditional Server Concept

- System Administrators often talk about servers as a whole unit that includes the hardware, the OS, the storage, and the applications.
- Servers are often referred to by their function; e.g., the SQL database server, the file server, etc.
- If the file server fills up, or the database server becomes overtaxed, then the System Administrators must add a new server.

The Traditional Server Concept

- Unless there are multiple servers, if a service experiences a hardware failure, then the service is down.
- System Administrators can implement clusters of redundant servers to make them more fault tolerant. However, even clusters have limits on their scalability, and not all applications work in a clustered environment.

The Traditional Server Concept

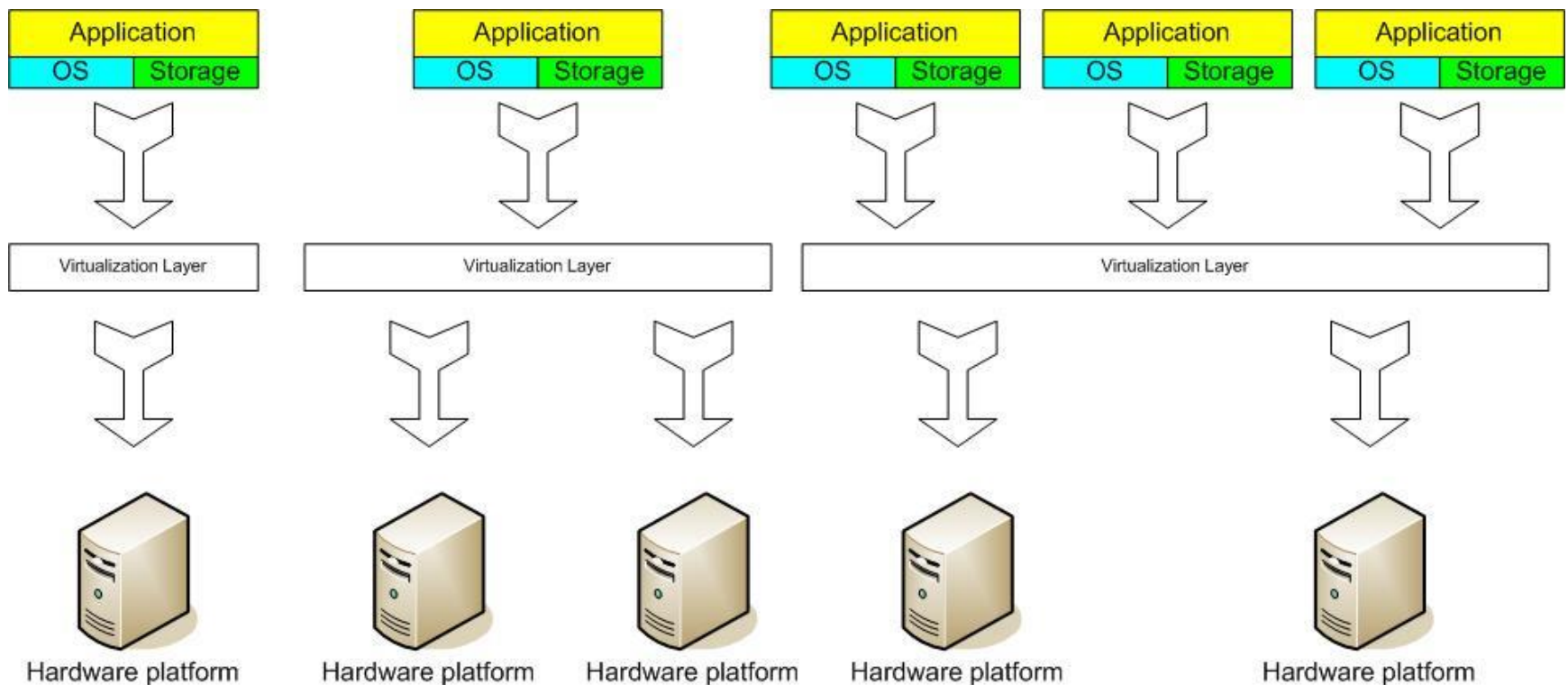
Pros

- Easy to conceptualize
- Fairly easy to deploy
- Easy to backup
- Virtually any application/service can be run from this type of setup

Cons

- Expensive to acquire and maintain hardware
- Not very scalable
- Difficult to replicate
- Redundancy is difficult to implement
- Vulnerable to hardware outages
- In many cases, processor is under-utilized

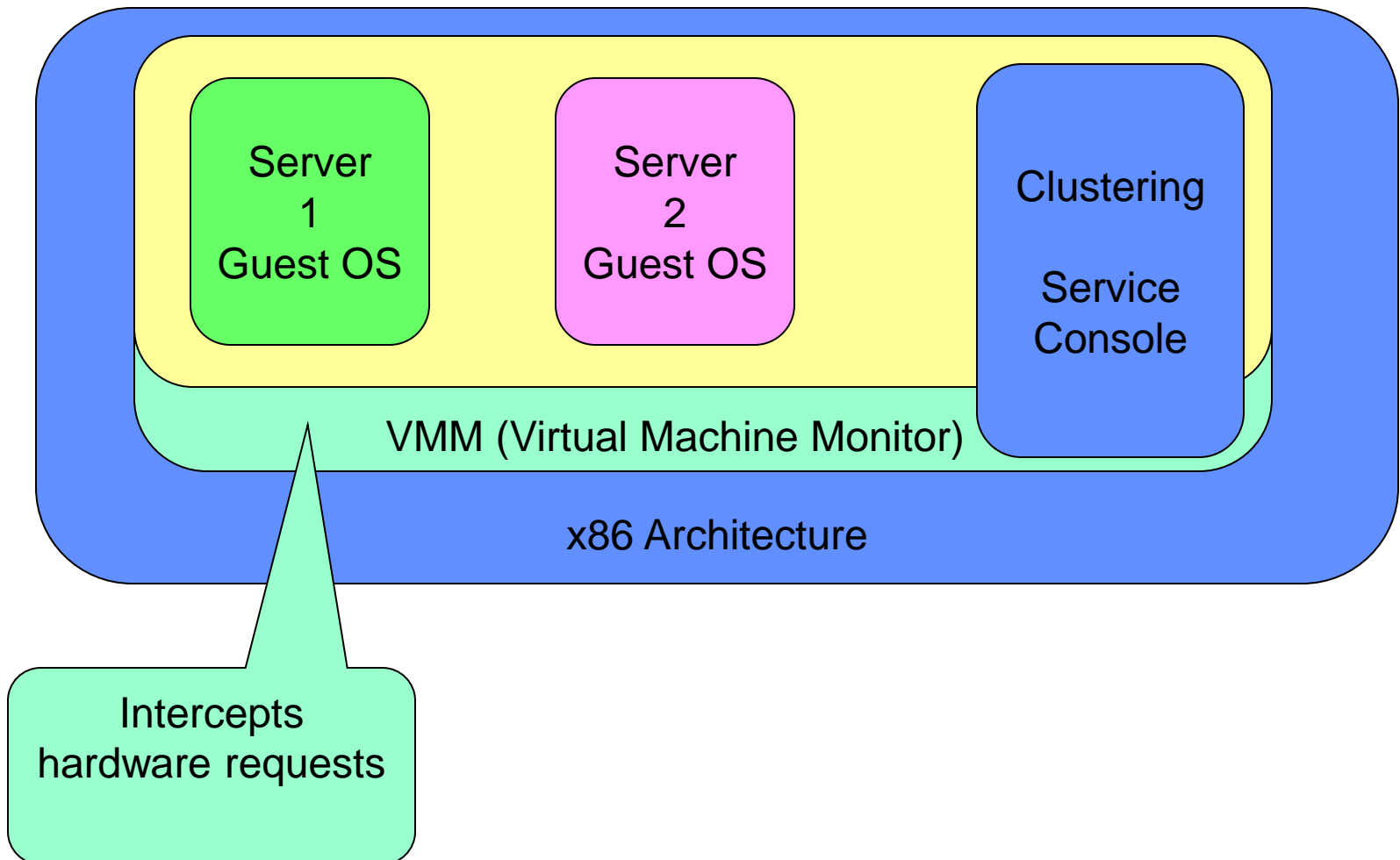
The Virtual Server Concept



Virtual Machine Monitor (VMM) layer between *Guest OS* and hardware

Close-up*

* adapted from a diagram in VMware white paper, *Virtualization Overview*



The Virtual Server Concept

- Virtual servers seek to encapsulate the server software away from the hardware. This includes the OS, the applications, and the storage for that server.
- Servers end up as mere files stored on a physical box, or in enterprise storage.
- A virtual server can be serviced by one or more hosts, and one host may house more than one virtual server.

The Virtual Server Concept

- Virtual servers can still be referred to by their function i.e. email server, database server, etc.
- If the environment is built correctly, virtual servers will not be affected by the loss of a host.
- Hosts may be removed and introduced easily to accommodate maintenance.

The Virtual Server Concept

- Virtual servers can be scaled out easily. If the administrators find that the resources supporting a virtual server are being taxed too much, they can adjust the amount of resources allocated to that virtual server.
- Server templates can be created in a virtual environment to be used to create multiple, identical virtual servers.
- Virtual servers themselves can be easily migrated from host to host.

The Virtual Server Concept

Pros

- Resource pooling
- Highly redundant
- Highly available
- Rapidly deploy new servers
- Easy to deploy
- Reconfigurable while services are running
- Optimizes physical resources by doing more with less

Cons

- Slightly harder to conceptualize
- Slightly more costly (must buy hardware, OS, Apps, and the abstraction layer)

So what about Cloud Computing?



Cloud computing takes virtualization to the next step

You don't have to own the hardware

You “rent” it as needed from a cloud

There are public clouds

- e.g. Amazon EC2, and now many others (Microsoft, IBM, Sun, and others ...)

A company can create a private one

- With more control over security, etc.

Goal 1 – Cost Control

Cost

- Many systems have variable demands
 - Batch processing (e.g. New York Times)
 - Web sites with peaks (e.g. Forbes)
 - Startups with unknown demand (e.g. the *Cash for Clunkers* program)
- Reduce risk
 - Don't need to buy hardware until you need it

Goal 2 - Business Agility

More than scalability - *elasticity*!

- Ely Lilly in rapidly changing health care business
Used to take 3 - 4 months to give a department a server cluster,
then they would hoard it!
- Using EC2, about 5 minutes!
And they give it back when they are done!

Scaling back is as important as scaling up

Goal 3 - Stick to Our Business

Most companies don't WANT to do system administration

- Forbes says:

We are is a publishing company, not a software company

But beware:

- *Do you really save much on sys admin?*
- *You don't have the hardware, but you still need to manage the OS!*

How Cloud Computing Works

Various providers let you create virtual servers

- Set up an account, perhaps just with a credit card

You create virtual servers ("virtualization")

- Choose the OS and software each "instance" will have
- It will run on a large server farm located somewhere
- You can instantiate more on a few minutes' notice
- You can shut down instances in a minute or so

They send you a bill for what you use

Any Nasty Details?


(loads of them!)

- How do I pick a provider?
- Am I locked in to a provider?
- Where do I put my data?
- What happens to my data when I shut down?
- How do I log in to my server?
- How do I keep others from logging in (security)?
- How secure is the data?
- How do I get an IP address?
- Etc.

And One Really Important Caveat*

Remember though ...

- These solutions will not auto-scale themselves
- They are merely providing you with the platform
- You must manage the scaling as if you had them running in your own data centre



Database in the Cloud

www.aw20.co.uk

Slide 12 / 26

Google Docs Menu

* *Cloud BootCamp March 2009*

References

(links are current as of September, 2009)

VMware Inc., *Virtualization Overview*, <http://www.vmware.com/pdf/virtualization.pdf>

Todd Hoff, *Amazon Architecture*, <http://highscalability.com/amazon-architecture>, Sept. 18, 2007

Intel Corp., *Technology brief: Understanding Intel® Virtualization Technology*,
<http://download.intel.com/technology/virtualization/320426.pdf>

aw2.0 Ltd, *Cloud BootCamp March 2009*,
http://www.aw20.co.uk/help/cloudbootcamp_march2009.cfm

also online: [CloudBootCamp2009.ppt](#)