MONASH University
Information Technology

FIT3142 Distributed Computing 2016 Semester 2

Topic 1: Introduction: IPC and Distributed Computing Schemes

Dr Carlo Kopp
Clayton School of Information Technology
Monash University

References/Reading

1. Ch.1; Ch.3; Ch.4; Coulouris, Dollimore, Kindberg and Blair, *Distributed Systems: Concepts and Design*, Edition 5, © Addison-Wesley, 2012.
2. Pawel Plaszczak and Richard Wellner, Jr.; *Grid Computing*, Elsevier, 2005.
3. http://www.globus.org/
4. http://www.csse.monash.edu.au/~carlo/SYSTEMS/IPC-Introduction-0595.htm
5. http://www.csse.monash.edu.au/~carlo/SYSTEMS/RPC-Intro-0796.htm
6. http://www.csse.monash.edu.au/~carlo/WALNUT/msc.thesis.ck.pdf
7. http://www.javacoffeebreak.com/articles/rmi_corba/index.html
8. http://java.sun.com/javase/technologies/core/corba/index.jsp
9. http://www.cs.utexas.edu/~wcook/Drafts/2006/WSvsDO.pdf
10. http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html
11. http://www.csse.monash.edu.au/~carlo/SYSTEMS/Cluster-Practical-1299.html
12. http://www.omg.org/gettingstarted/corbafaq.htm

Topic 1 Content

- What are distributed systems?
- Challenges in distributed systems?
- Distributed computing models?
- Processes in operating systems (revision);
- IPC in operating systems (revision);
- Stream-oriented IPC in operating systems (revision);
- Parallelism vs. distributed computing;
- Clusters, Grids, Clouds, Distributed Storage;

Why Study Distributed Computing?

- **Traditional "centralised" computing has been displaced largely by distributed computing schemes over the last 20 years – centralised mainframes are used much less;**
- **Software applications are increasingly being developed for distributed computing environments, and older applications are being ported to such;**
- **For a software application to both perform well and be reliable, it must be designed around the environments it is expected to be operated in;**
- ***Programmers and software architects therefore need a good understanding of distributed computing environments to be effective and successful in current and future industry;***
- **Many badly behaved applications in distributed environments are a result of insufficient understanding by programmers.**

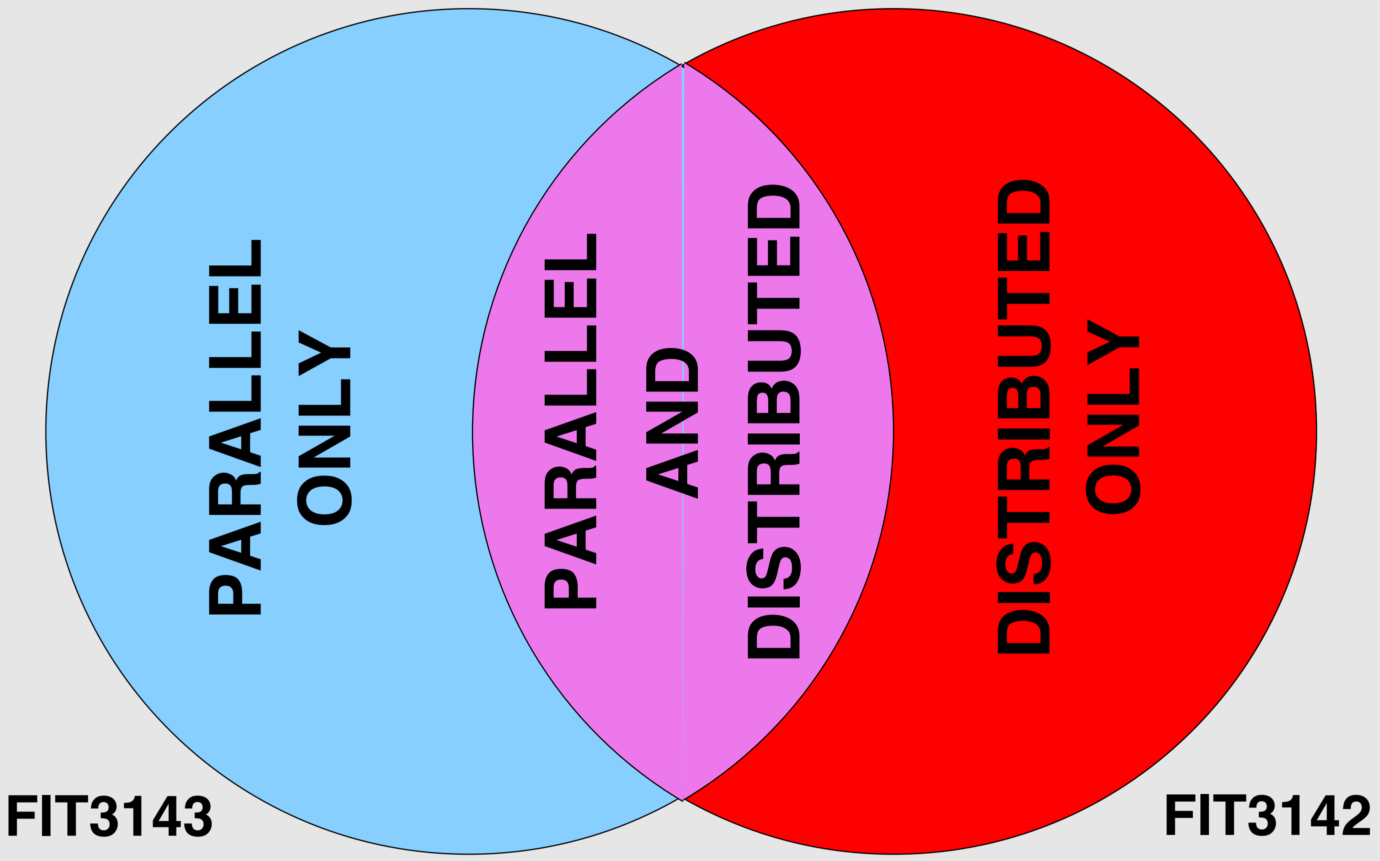
Key Unit Objectives

- This unit has been designed to provide a balanced mix between broader understanding of distributed computing, and a more detailed focus on grid computing, as grids are now reasonably mature and widely deployed.
- *As distributed computing is a very wide and deep area, it cannot be covered in full depth across its whole scope within a single one semester duration unit.*
- This unit is therefore partly focused on distributed computing fundamentals, and partly focused on the detail of grids, as the latter provides the best platform for reinforcing understanding, and also gaining some practical skills.
- A student who successfully completes this unit will have a good understanding of distributed computing concepts and what is needed to build viable distributed applications.

What is Distributed Computing?

- *Distributed computing involves any computing system, where computational activity is divided across multiple host machines, linked by a digital communications medium of some type.*
- While a parallel computing system may be distributed, many distributed systems do not closely fit the model of parallel computing, as the components of the distributed system and applications may be dissimilar in many ways.
- The advent of wide area networks, especially the Internet, has been the principal enabler for distributed computing systems.
- The first "modern" distributed computing systems emerged during the 1980s, exploiting high speed Local Area Networks (LAN) to provide connectivity.
- Growth of the Internet has stimulated distributed computing.

Parallel versus Distributed Computing



Definition of Distributed Systems

- *“A distributed system is a collection of independent computers that appears to its users as a single coherent system.”*
- **The definition has several important aspects**
  - Autonomous components
  - Users (whether people or program) think they are dealing with a single system
- **A distributed system is a system in which components located at networked computers communicate and coordinate their actions only by passing messages.**

Advantages of Distributed Systems

A. Reliability: If 5% of the machines are downed, the system as a whole can still survive with a 5% degradation of performance.
B. Incremental growth: Computing power can be added in small increments
C. Sharing: Allow many users access to a common database and peripherals.
D. Communication: Make human-to-human communication easier.
E. Effective Resource Utilization: Spread the workload over the available machines in the most cost effective way.

Disadvantages of Distributed Systems

A. Software: It is more difficult to develop distributed software than centralized software .

B. Networking: The network can saturate and slow down dramatically or cause other problems.

C. Security: Ease of access also applies to secret data.

Challenges in Distributed Systems

A. Heterogeneity - Within a distributed system, we have variety in networks, computer hardware, operating systems, programming languages, etc.

B. Openness - New services are added to distributed systems. To do that, specifications of components, or at least the interfaces to the components, must be published.

C. Transparency - One distributed system made to look like a single computer by concealing the distribution mechanism.

D. Performance - One of the objectives of distributed systems is achieving high performance while using cheap computers.

Challenges in Distributed Systems

E. Scalability - A distributed system may include thousands of computers. Whether the system works is the question in that large scale.

F. Failure Handling - One distributed system is composed of many components. That results in high probability of having failure in the system.

G. Security - Because many stake-holders are involved in a distributed system, the interaction must be authenticated, the data must be concealed from unauthorized users, and so on.

H. Concurrency - Many programs run simultaneously in a system and they share resources. They should not interfere with each other

INTERPROCESS COMMUNICATIONS

Processes in Operating Systems [Stallings Ch.3]

- *A program in execution*
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions
- A process comprises:

1. Program code (possibly shared)
2. A set of data
3. A number of attributes describing the state of the process

Process Identity and State Information [Stallings Ch.3]

- At a minimum, a process must have the following attributes so it can be identified and its state known:

1. Identifier
2. State
3. Priority
4. Program counter
5. Memory pointers
6. Context data
7. I/O status information
8. Accounting information

- This data is usually kept in a Process Control Block (PCB)

Process Control Block [Stallings Ch.3]

- Contains the process elements;
- Created and managed by the operating system;
- Allows support for multiple processes;

- The structure and format of a PCB is unique to an operating system type, and usually not interchangable between operating systems;

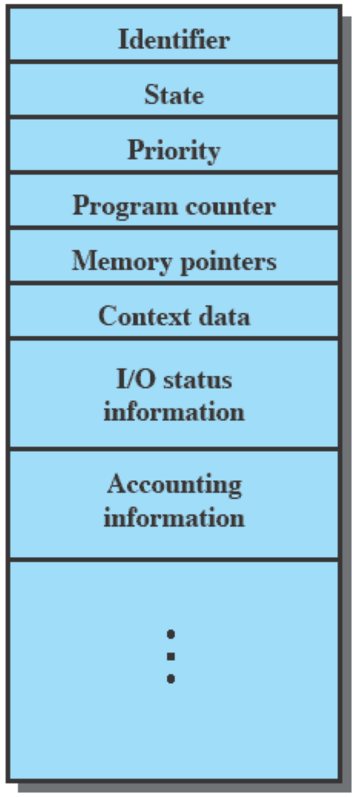


Figure 3.1 Simplified Process Control Block

Processes in Main Memory [Stallings Ch.3]

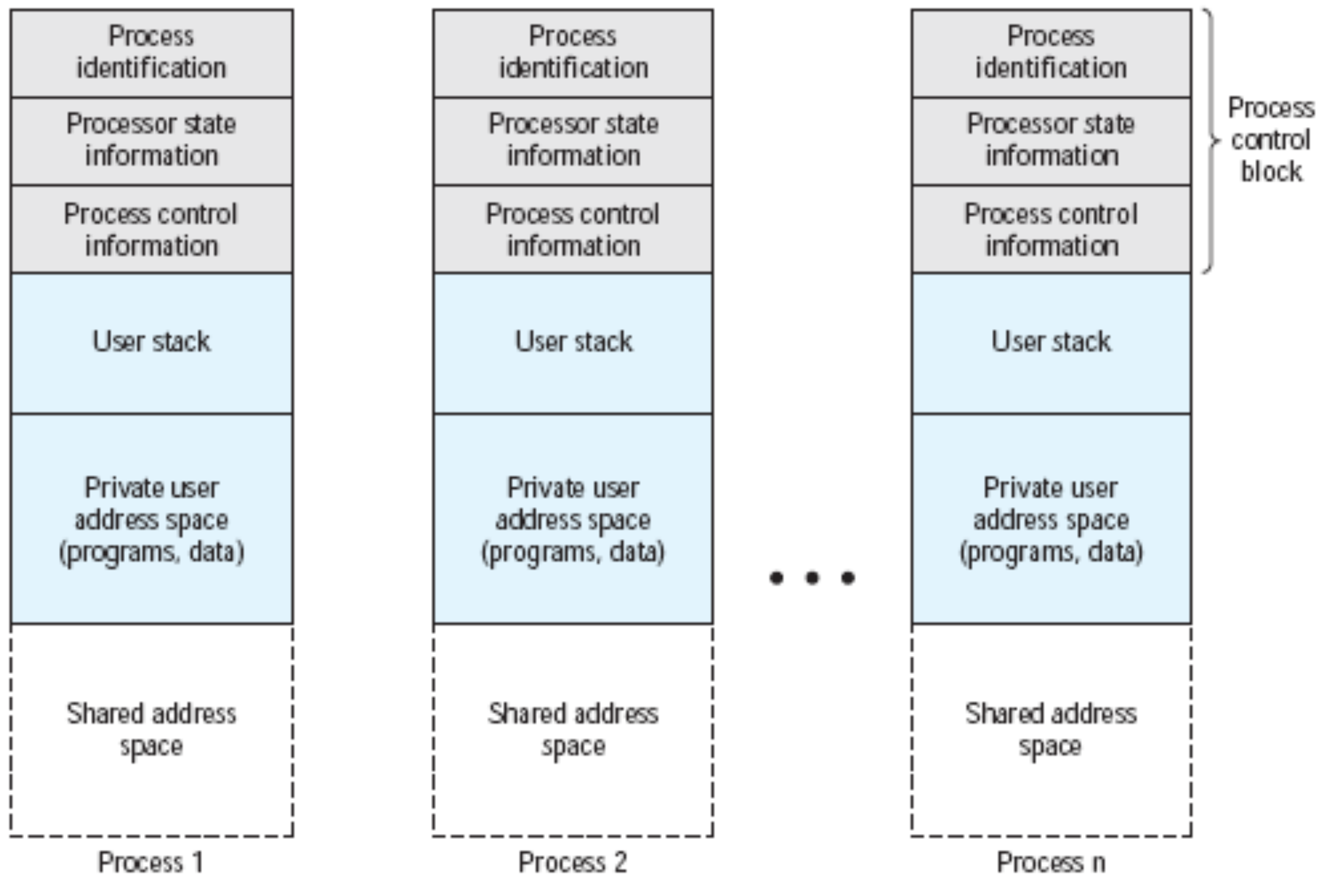
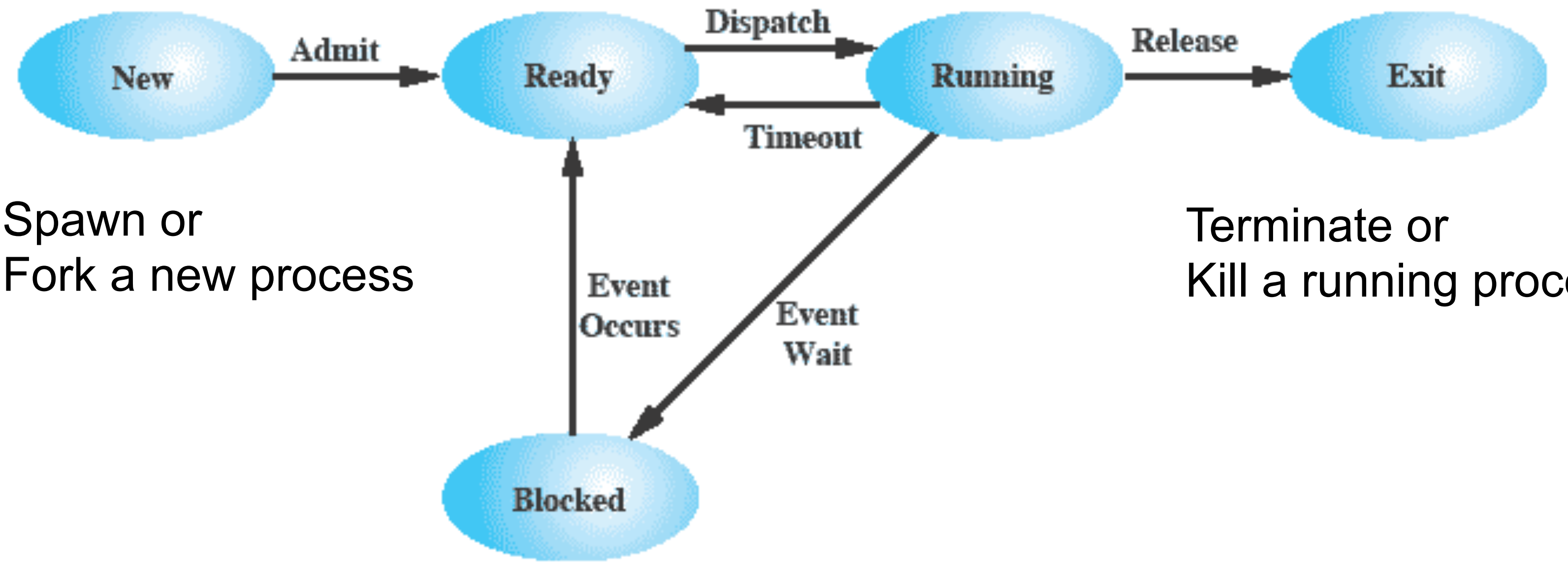


Figure 3.13 **User Processes in Virtual Memory**

Process States [Stallings Ch.3]



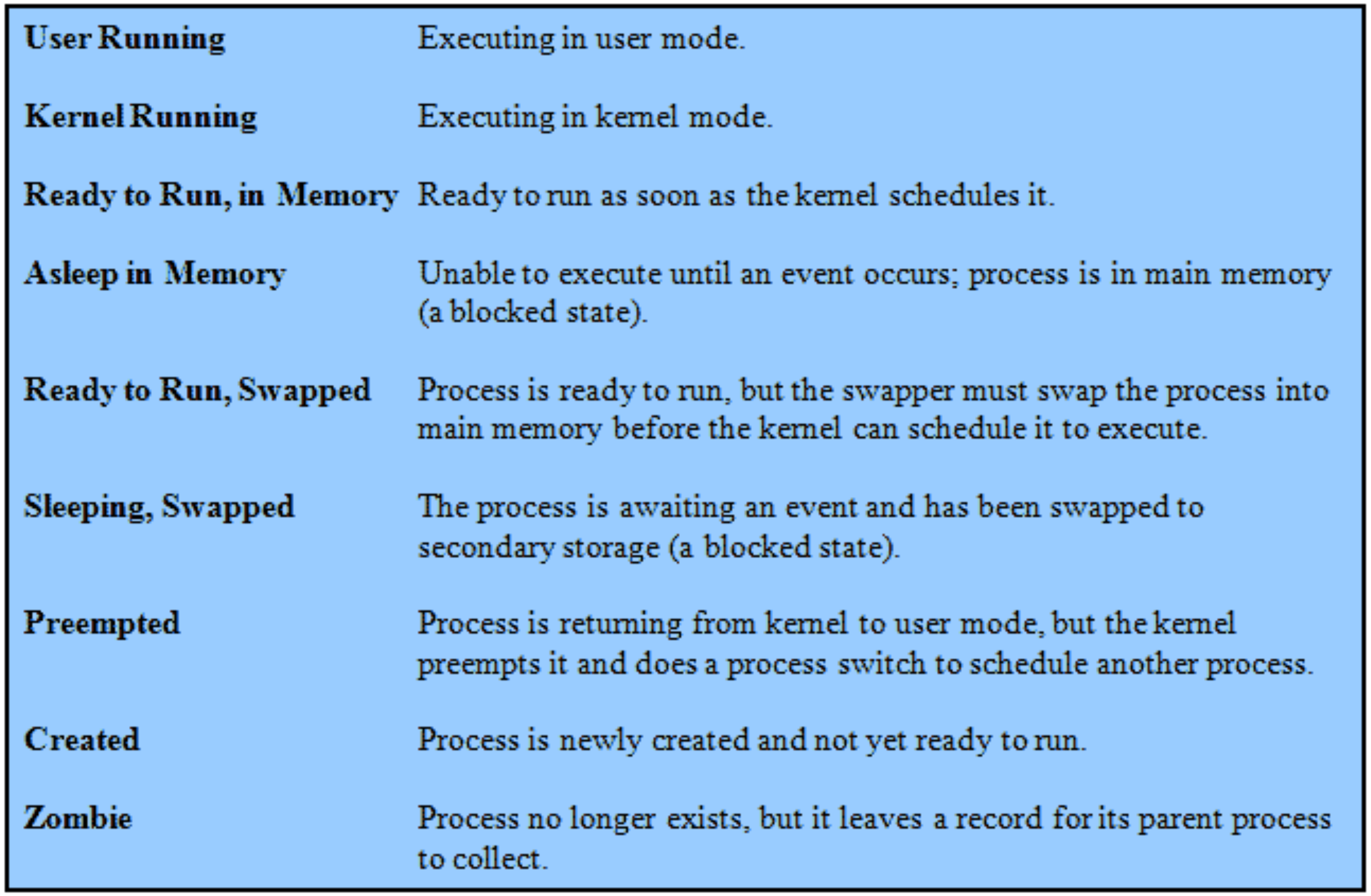
Spawn or
Fork a new process

Terminate or
Kill a running process

Inter Process Communications [IPC]
may or may not cause a process to block

Figure 3.6 Five-State Process Model

Unix Process States [Stallings Ch.3]



User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

Inter-Process Communications

- Inter-Process Communications (IPC) are all mechanisms which permit two or more processes to exchange messages;
- IPC mechanisms may be restricted to communications within a host machine's operating system, or may permit communications between processes on different host machines connected via a network;
- IPC mechanisms restricted to a host operating system include:

1. *Shared Memory*
2. *Message Passing*
3. *Unix Signals and analogues*

- *Shared Memory* imposes no structure on messages, but *Signals* and *Message Passing* typically force discrete message structures.

Shared Memory IPC – Limited to Local Host

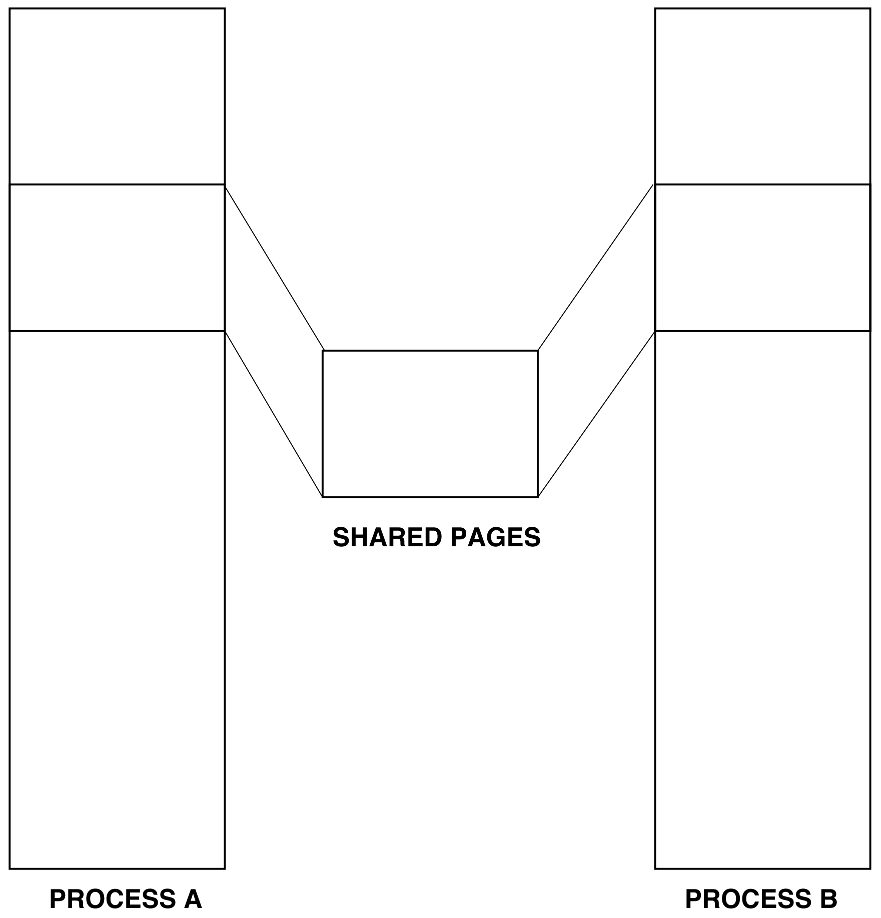


FIG.1 SHARED MEMORY – PAGES ARE MAPPED INTO TWO OR MORE PROCESSES

Stream Oriented Inter-Process Communications

- ***Stream Oriented IPC*** **is especially important since it is the most commonly used scheme for IPC within operating systems and between networked hosts;**
- **A stream connection imposes no implicit structure on the data being sent – character mode "streams" of data are handled in a FIFO manner, arriving typically at the destination in the order they were sent;**
- **The two most widely used stream oriented IPC schemes are the BSD Socket mechanism, and the SVR4 STREAMS mechanism; Application protocols treat the stream as a transparent pipe.**
- **Both are designed around standard software Application Programming Interfaces (API), although most recent STREAMS implementations also include an interface to emulate the BSD Socket API;**

Stream Oriented IPC – Channel Properties

- The channel is "reliable" when the stream connects two processes on a single host;
- The channel is "reliable" when the stream connects two processes on different hosts, using TCP protocol;
- The channel is "unreliable" when the stream connects two processes on different hosts, using UDP protocol;
- A stream IPC connection usually delivers messages "in order" thus behaving like a FIFO or pipe;
- Depending on the type of stream IPC, the API, and its configuration, the interface may be "blocking" or "non-blocking", thus determining process behaviour during message transfers;
- *BSD Sockets and SVR4 STREAMs are the most common APIs in use for distributed applications;*

SVR4 STREAMS vs BSD Sockets

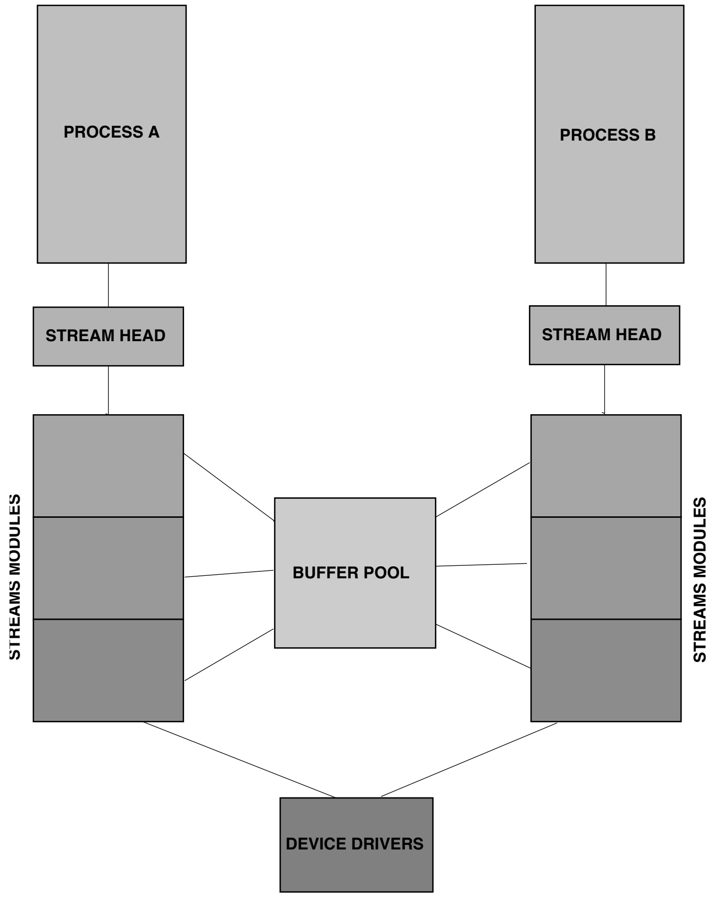


Fig.2.2 STREAMS Transport - Conceptual Model

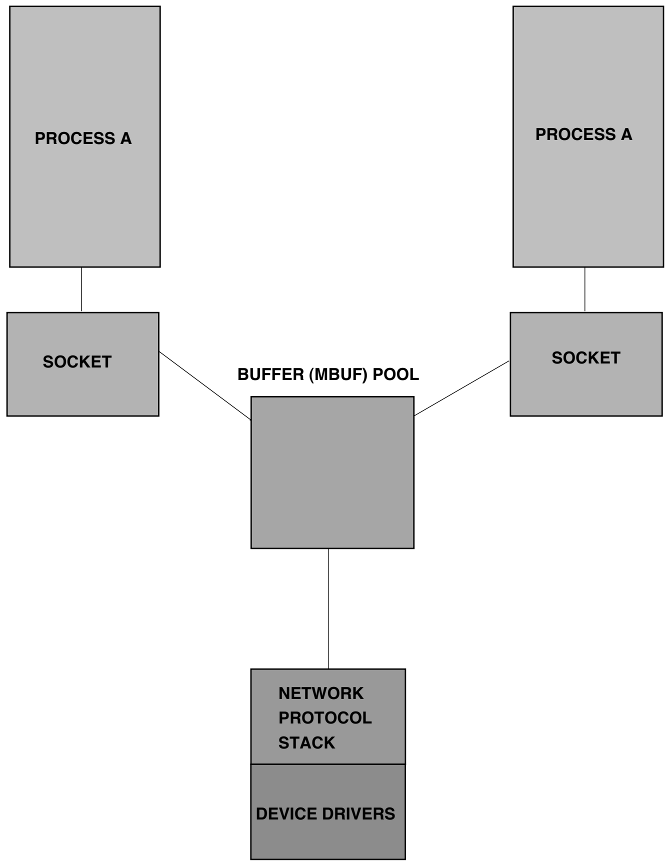
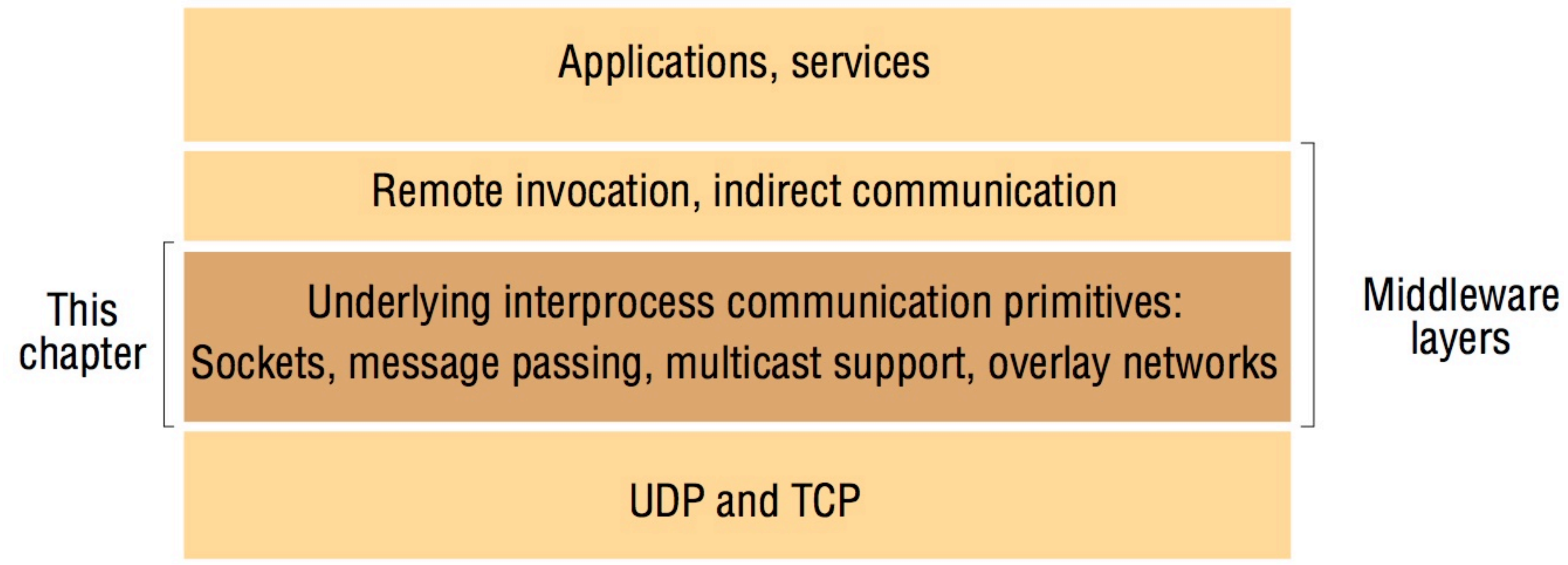


Fig 2.4 BSD Socket Transport - Conceptual Model

Stream Oriented IPC – Integration


Applications, services
Remote invocation, indirect communication
Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks
UDP and TCP
This chapter
Middleware layers

Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5, © Pearson Education 2012

- The "middleware" provides the API interface for the user program running in a process;
- The middleware interfaces to the protocol stack software, which is usually embedded in the operating system kernel;

Example: Linux Socket Integration [Stallings Ch.17]

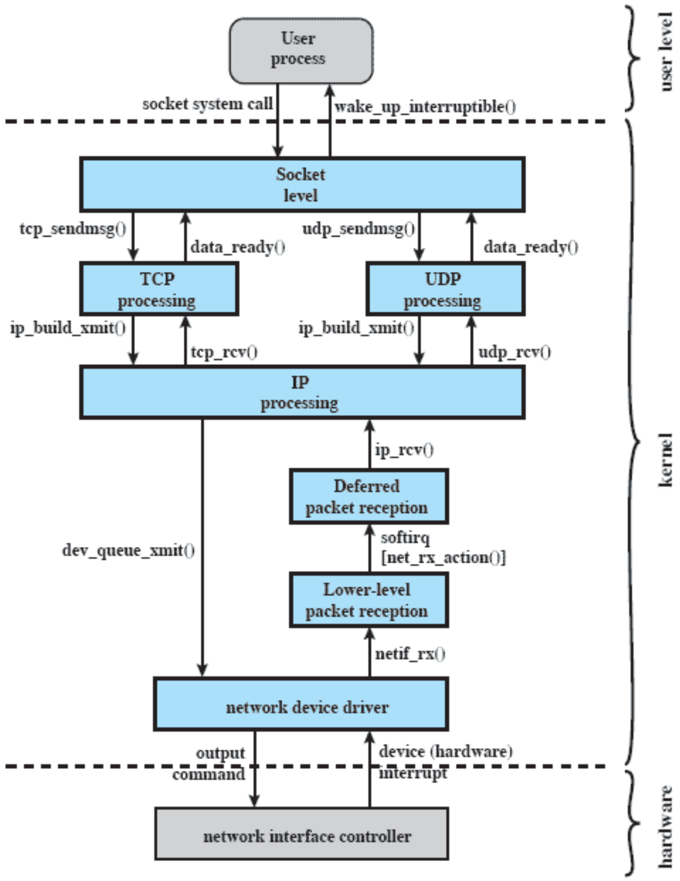


Figure 17.7 Linux Kernel Components for TCP/IP Processing

MONASH University
Information Technology

www.infotech.monash.edu

DISTRIBUTED COMPUTING MODELS

Distributed Computing Models

- Client-Server – well established protocols e.g. HTTP, running over socket connections.
- Client-Server – *Remote Procedure Calls* (RPC) where client can execute program on server, NFS over RPC.
- Client-Server – OO schemes such as *CORBA* (Common Object Request Broker) where client can create, destroy, execute objects on a remote or local server.
- Parallel Distributed Schemes – *Clusters, Grids, Clouds*.
- *Clusters* – large numbers of homogenous or inhomogenous machines working on a typically parallel problem; *Grids* generalise and extend the cluster scheme through standard interfaces over wide area networks.
- *Clouds* – provide an environment where application environment can be decoupled from the native platform.

Client Server Computing

- Introduced during the 1980s with early Local Area Networks;
- Extension of existing protocols for host-host connectivity such as Telnet, FTP;
- Primary connections are simple streams using either the BSD Unix Socket programming interface, or the SVR4 Unix Streams programming interface;
- Higher level protocols are typically application specific, and rigidly split functions between a client and a server;
- The client makes requests to the server, which in turn services them;
- The two most widely used examples of protocols are the Hypertext Transfer Protocol used for the W3 and the X.org X11 X-Window System protocol used for graphics displays on Unix, Linux and BSD systems.

Client Server Environment [Stallings Ch.16]

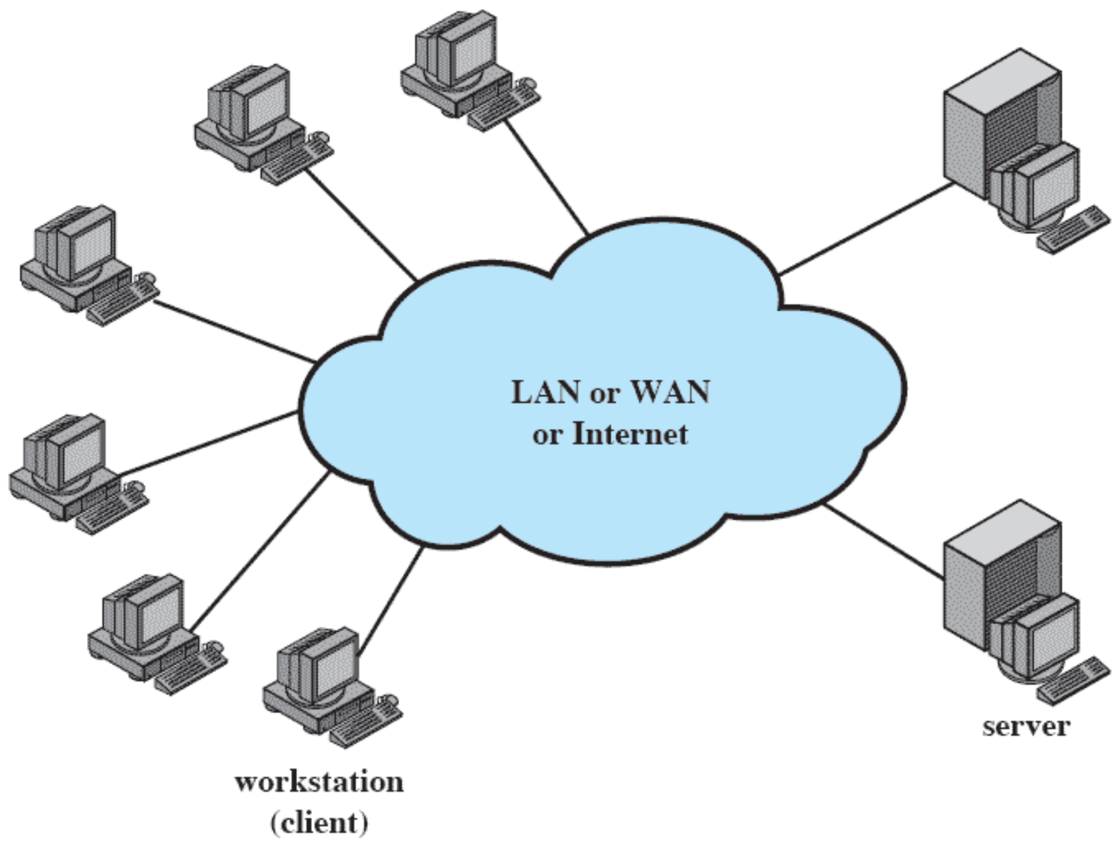
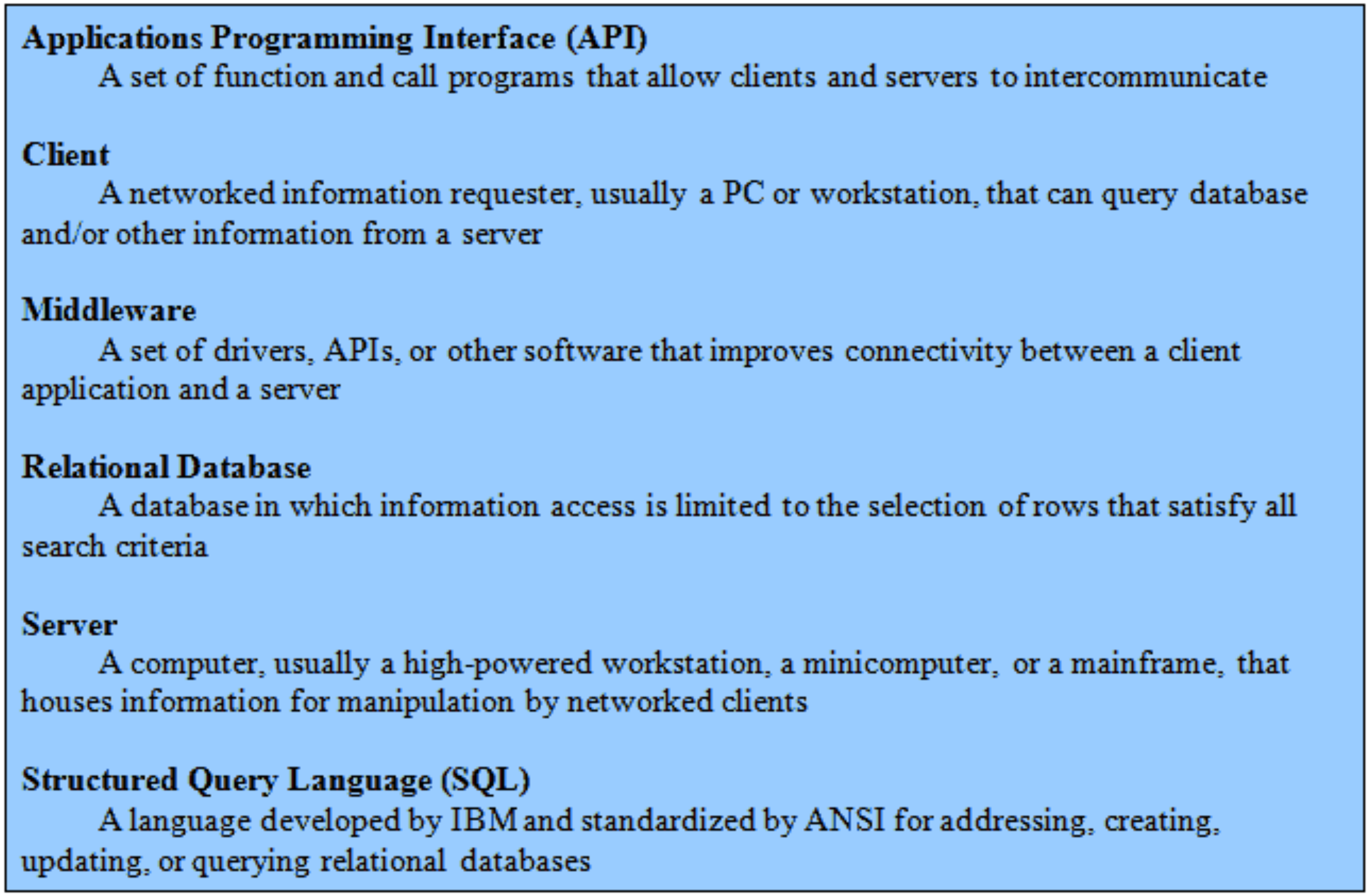


Figure 16.1 Generic Client/Server Environment

Client Server Terminology [Stallings Ch.16]



Applications Programming Interface (API)
A set of function and call programs that allow clients and servers to intercommunicate

Client
A networked information requester, usually a PC or workstation, that can query database and/or other information from a server

Middleware
A set of drivers, APIs, or other software that improves connectivity between a client application and a server

Relational Database
A database in which information access is limited to the selection of rows that satisfy all search criteria

Server
A computer, usually a high-powered workstation, a minicomputer, or a mainframe, that houses information for manipulation by networked clients

Structured Query Language (SQL)
A language developed by IBM and standardized by ANSI for addressing, creating, updating, or querying relational databases

Client Server Architecture [Stallings Ch.16]

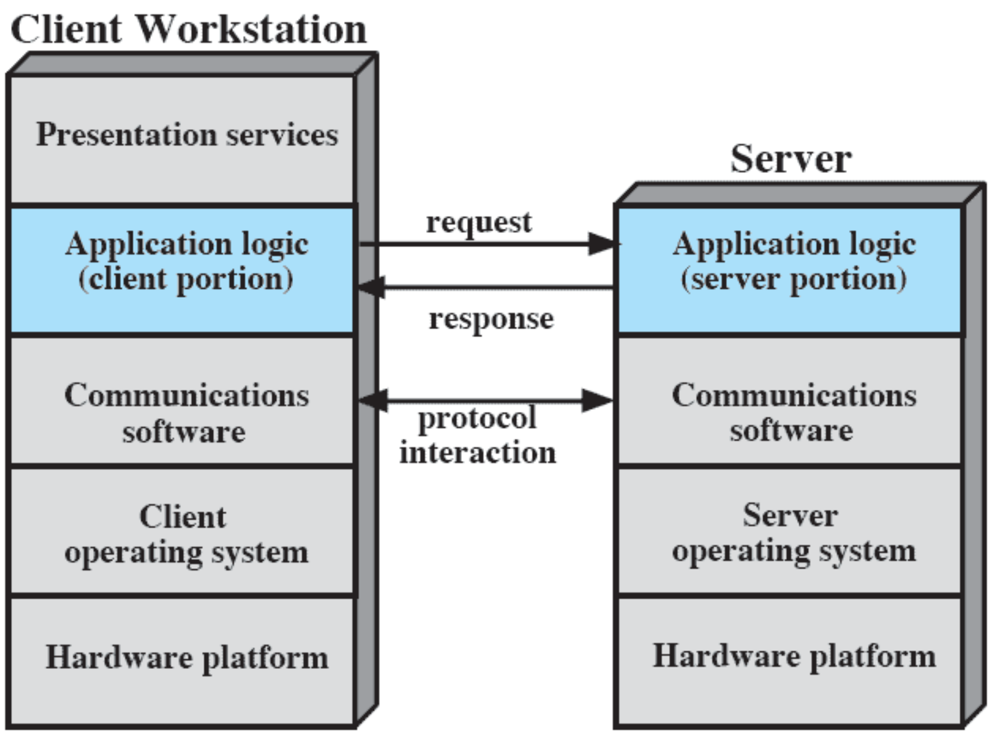


Figure 16.2 Generic Client/Server Architecture

Client Server Middleware [Stallings Ch.16]

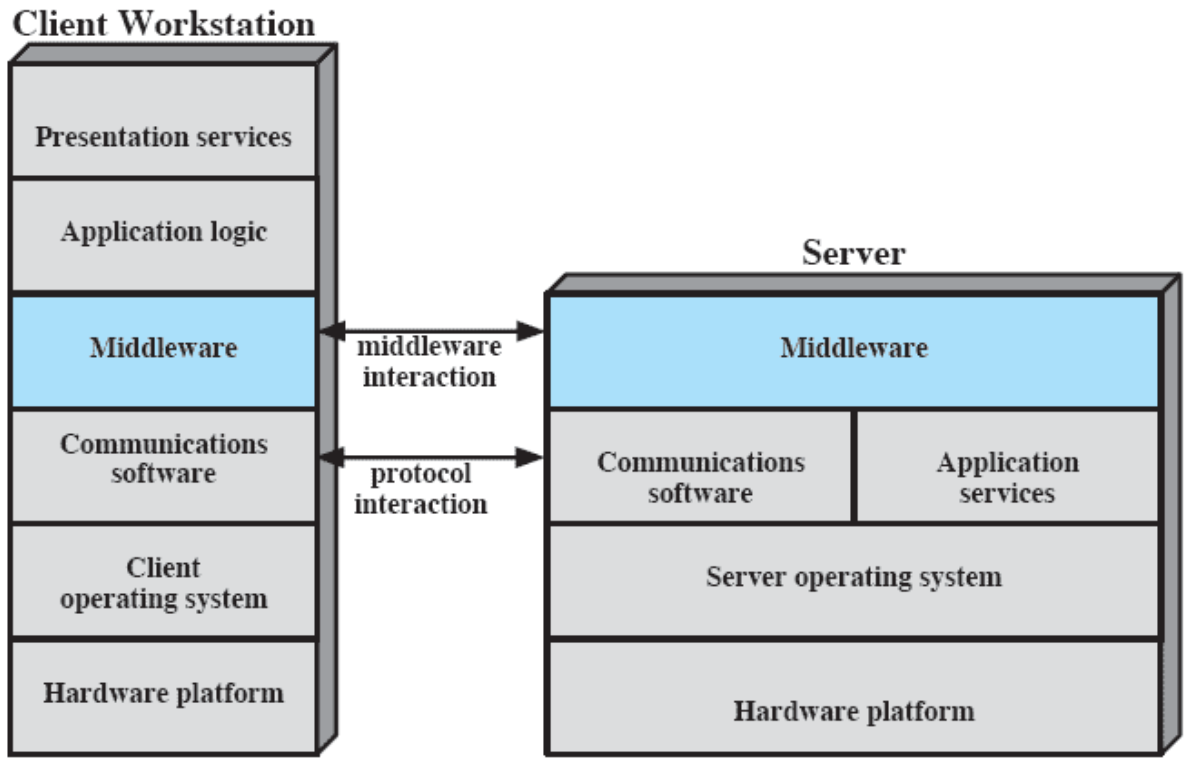
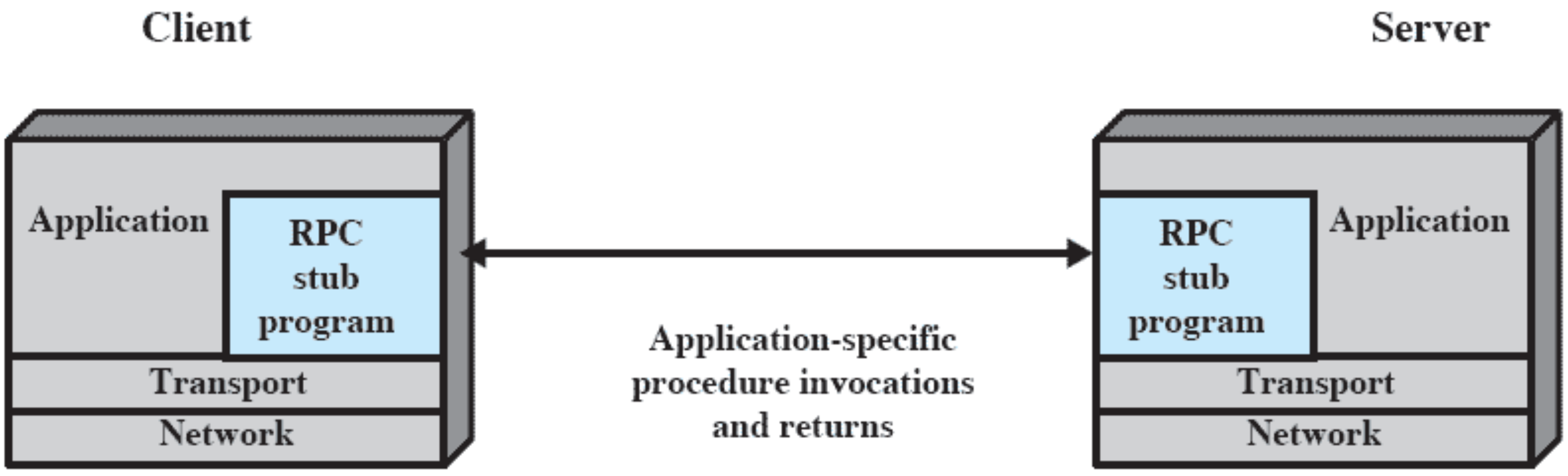


Figure 16.8 The Role of Middleware in Client/Server Architecture

Remote Procedure Calls (RPC)

- The very widely used ONC RPC (RFC1831) protocol is a good example of a more advanced distributed computing scheme;
- The central idea in all RPC schemes is that a procedure (i.e. function call) may be executed locally on a host, or on a remote server host;
- The client host will make a request upon a server which involves a procedure identifier (name) and some list of arguments; the server then returns the results of the call;
- An important issue in all RPC systems is data representation, as the client and server may have different “endianness” and thus data being sent as arguments, or procedure results must be formatted in a compatible fashion;
- Another issue is whether the protocol is “stateful” or “stateless”, i.e. whether the server remembers state.

Remote Procedure Calls [Stallings Ch.16]



(b) Remote Procedure Calls

Remote Procedure Calls [Stallings Ch.16]

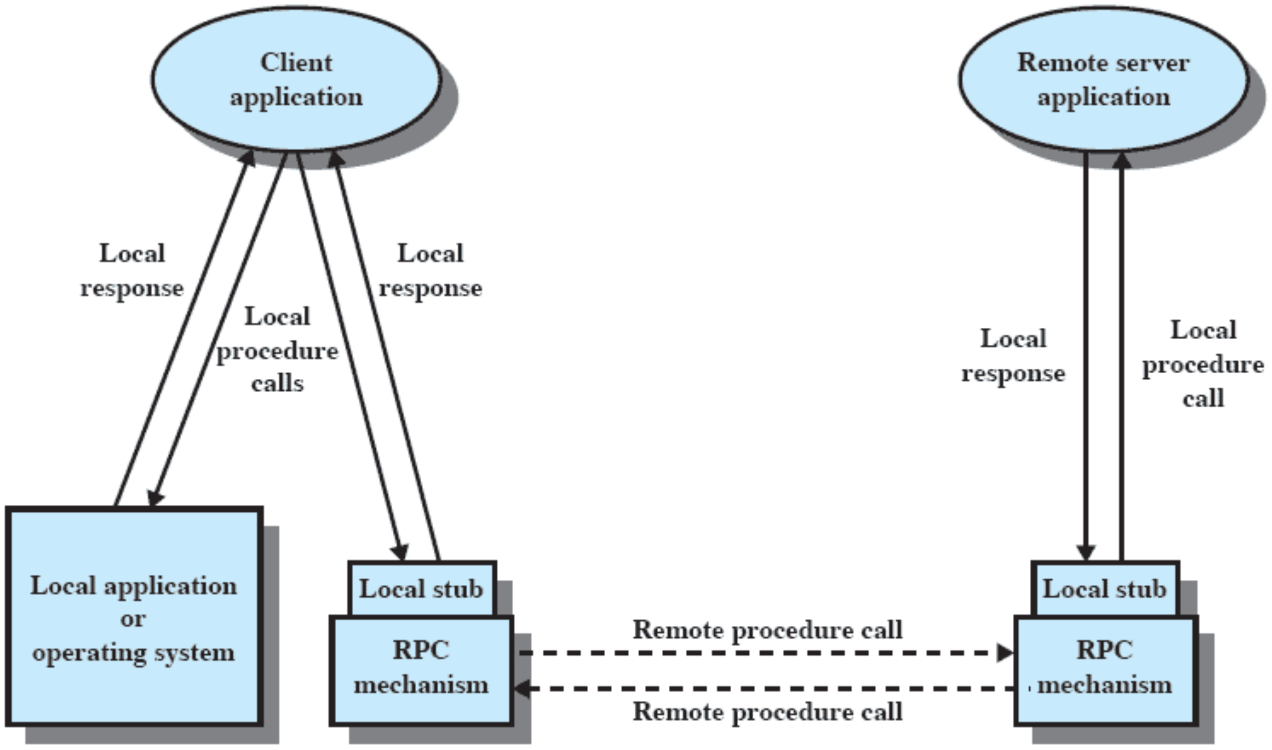


Figure 16.12 Remote Procedure Call Mechanism

Object Oriented Distributed Schemes

- **While the RPC scheme proved very effective and useful, it was syntactically modelled on procedural programming, and did not fit cleanly into increasingly popular Object Oriented programming languages such as C++ and later Java;**
- **The most widely used RO systems are CORBA and .NET both of which extend the RPC model to permit a client to create, run or destroy an object on a remote or local server process;**
- **The intent of both schemes is to provide a programmer with a completely transparent OO programming environment where a specific service to the called can be local to user host system, or located on a distant server host;**
- **Both CORBA and .NET were initially architected around a simple client server model, where typically many clients were serviced by a single server.**

Multiprocessing vs Cluster Host Systems

- When Client-Server systems were introduced during the 1980s, most client systems were desktops with a single CPU chip, and servers were single or multiple CPU workgroup servers, superminicomputers, or mainframe computers;
- By the 1990s the increasing performance and declining cost of 32-bit microprocessors resulted in the introduction of the first "clusters";
- Rather than connecting a large number of CPU chips via a large shared fast parallel bus and memory in a single backplane host housed in a single chassis, a cluster used a high speed LAN to interconnect initially dozens, and later hundreds or thousands of individual machines to form a parallel computing system;
- Early clusters were built from ordinary desktop chassis.

Multiprocessing vs Cluster Host Systems …

- Since the 1990s hardware manufacturers have shifted to racked CPU chassis specifically designed to permit the construction of very large clusters;
- Clusters resulted in two important changes in the programming environment for distributed computing;
- A.InterProcess Communications (IPC) shifted to Socket / Stream techniques as a networked cluster did not have a common shared memory or other IPC mechanisms which are specific to an operating system kernel or upper layers;
- B.Programming models had to be capable of handling possibly large numbers of CPUs, not just the 2, 4, 8, 16 CPUs typical for standalone multiple CPU hosts.
- Clusters were initially used for large web servers and supercomputers, the latter mainly for scientific computing.

Cluster Architecture [Stallings Ch.16]

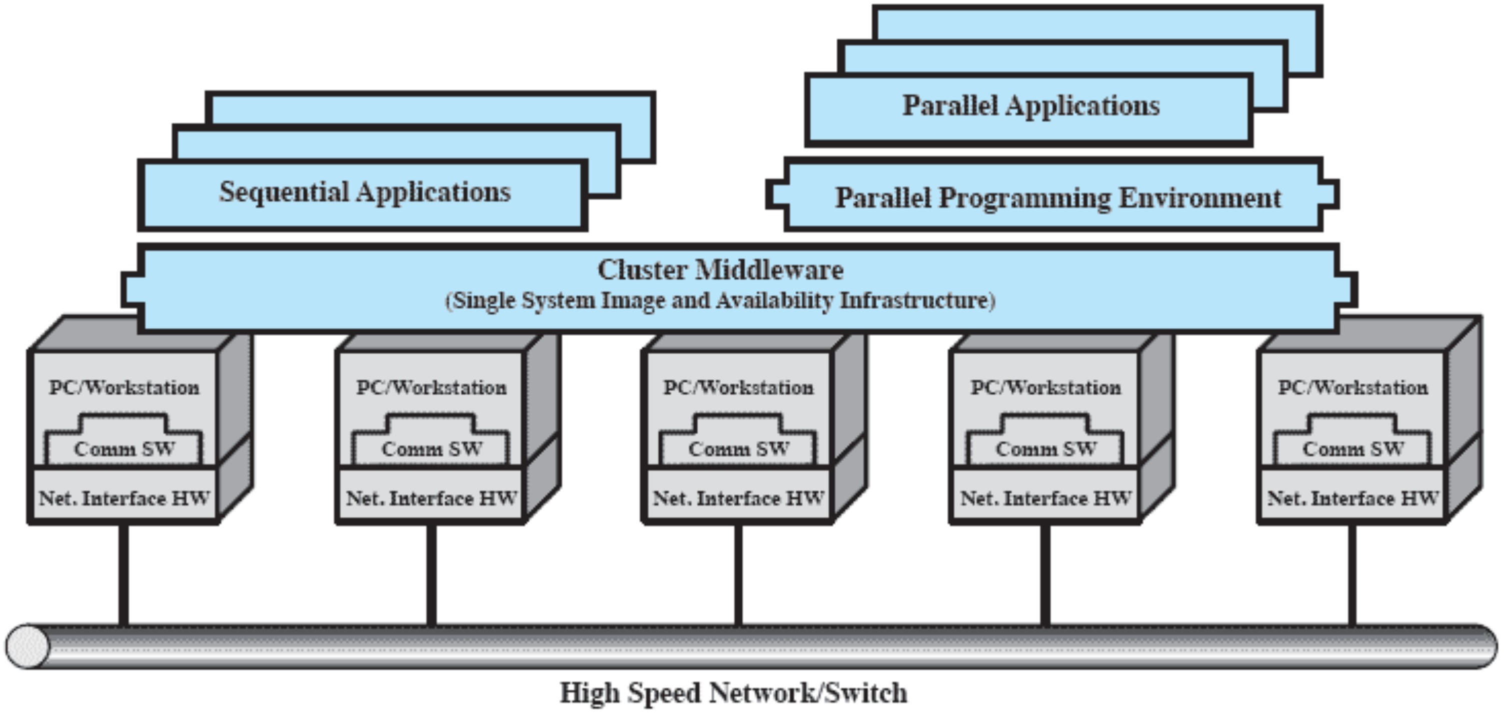


Figure 16.14 Cluster Computer Architecture [BUYY99a]

Beowulf Cluster Architecture [Stallings Ch.16]

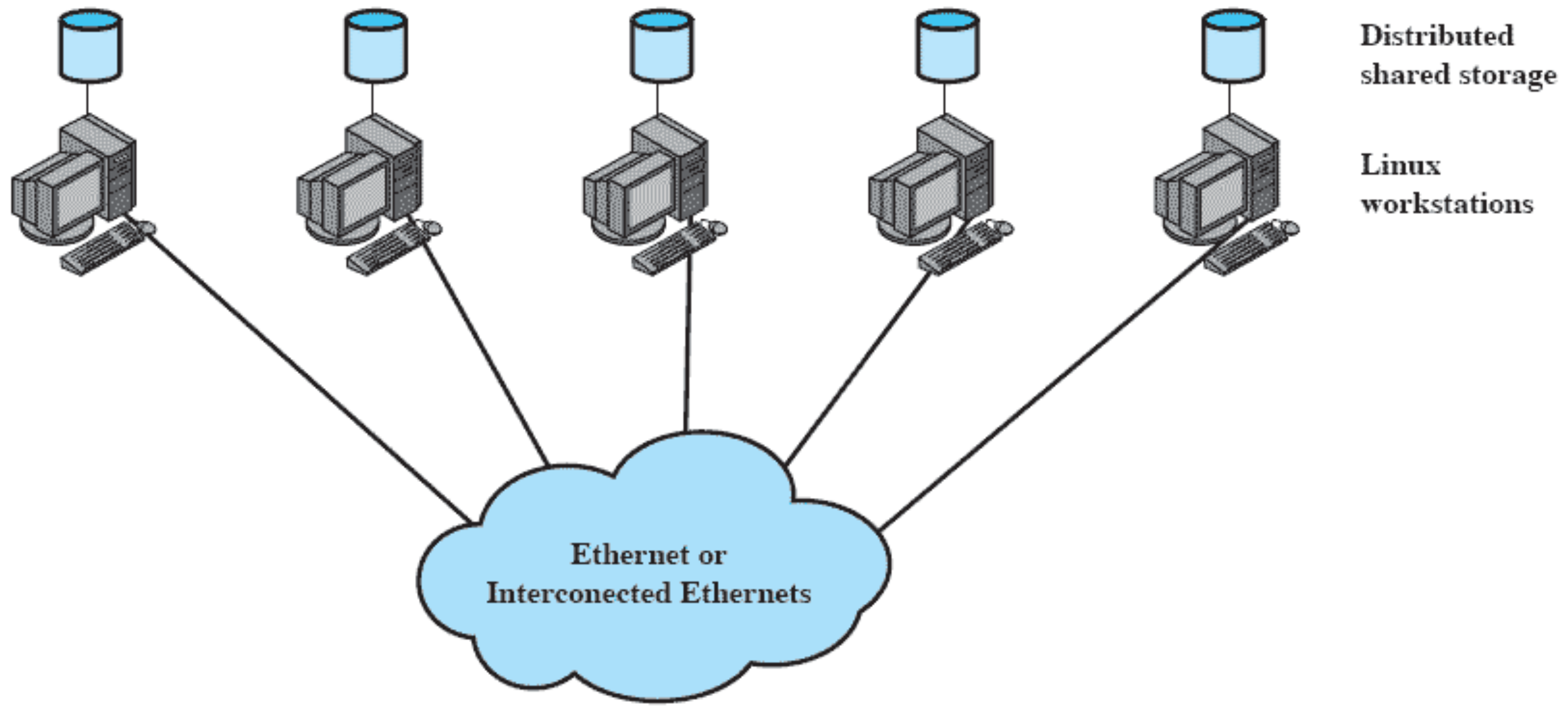

Distributed shared storage
Linux workstations
Ethernet or Interconected Ethernets

Figure 16.18 Generic Beowulf Configuration

Post 2000 – Divergence in Distributing Computing

- The decade following the 1990s saw Moore's Law effects driving the performance of CPUs upward, and costs downward, making clustering more affordable;
- User needs and expectations also changed as the W3 expanded strongly to cover commercial applications such as retailing and search engines;
- Different user communities with different needs influenced growth in distributed applications and techniques;
- This is important, insofar as commercial users, research users and social networking users all have different needs in a distributed processing environment;
- A multimedia server network like *Youtube* has very different requirements to a generic web server, or a retail server scheme like *Amazon* – different applications and protocols.

Large Scale Distributed Computing Schemes

- At this time "traditional" small and medium scale client-server schemes continue to be widely used – based on X11, NFS/RPC, CORBA and .NET and other established schemes;
- Large scale distributed computing schemes are filling other niches using protocols, software interfaces and applications specifically built for large numbers of CPUs; these are based on clusters, grids and web service or cloud schemes;
- *Cluster Schemes*: Web servers and scientific supercomputers
- *Grid Schemes*: Scientific supercomputing arrangements
- *Web Services / Cloud Schemes*: Commercial search engines, social networking media, retail and wholesale systems, outsourced storage and computing services, and a range of other possible applications.
- *What does this mean?*

Impact of Diversity in Distributed Computing

- **At this time there is greater diversity in available distributed computing technology than ever before;**
- **Diversity will continue to increase as the technology matures, but also as new applications are devised, built to address different needs;**
- **A programmer may have to develop software within more than one distributed computing environment, and sometimes interface across one or more;**
- **To best exploit such an environment, a programmer must have a good understanding of the fundamental concepts which are unique to each of the these schemes, but also understand what problems and limitations all share;**
- ***The underlying networks and operating systems will always influence behaviour and performance in such systems.***

Client-Server Schemes vs Parallelism

- Client-server schemes, encompassing BSD Sockets, ONC RPC, and CORBA all share common features, but reflect evolving expectations in functionality;
- All are based on the idea of many clients accessing typically a single server;
- All are designed around the favoured programming model and languages of the period when they were devised;
- All have limited instrumentation and facilities for managing or balancing loads on servers;
- Most importantly, they are not designed around the expectation than dozens or hundreds of servers may be accessible, and that applications might exploit parallelism across large numbers of host machines.
- Clusters, Grids, Clouds, etc evolved to exploit parallelism.

Clusters

- The central idea underpinning all clusters was to increase available computing power for an application by aggregating a large number of cheap machines via a fast shared interface;
- Early clusters used proprietary or custom fast busses to provide high speed interfaces, or ordinary 10-Base-T and later 100-Base-T Ethernet to provide low cost interfaces;
- Aggregating a large number of machines provides raw computing power, and fast interconnection provides bandwidth between the CPUs in the machines – *but neither address the problem of how to spread the computing load evenly across the CPUs, or how to construct applications to run on such a system;*
- Software remains the critical design issue in all distributed computing environments!

Programming Clusters

- Clusters will remain widely used for the foreseeable future, in scientific computing, and commercial computing;
- Some clusters will run “traditional” clustering software environments, but some will run grid middleware, or cloud runtime software – in a sense many grids and cloud systems are effectively clusters operating under layers of middleware code to provide a grid or cloud programming interface;
- Traditional clustering software falls into three broad categories:

1. Parallel Computing APIs such as MPI, typically for Finite Element Modelling engineering/scientific applications;
2. Parametric Computing environments such as Nimrod/Enfuzion;
3. Load sharing environments like PVM and LVS.

Grids

- The intent behind the development of grids was to overcome many of the basic limitations seen in clusters by providing a high standardised software API implemented as "middleware":

1. Grids would permit the aggregation of much greater numbers of machines than clusters;
2. Grids would permit the aggregation of machines across geographically distant sites if required;
3. Grids would decouple the application from the specifics of the underlying hardware, and provide security mechanisms;
4. Grids would provide a much more flexible programmer interface to permit a wider range of applications to be run;
5. Grids would provide "on demand" computing power in a manner analogous to an electricity or other utility "grid".

Programming Grids

- Grid applications can be architected around the open source Globus grid middleware, or a range of application specific proprietary middleware products;
- Grid middleware is intended to provide a “clean” API for the programmer, which conceals the details of the grid hardware and operating systems as much as possible, and provides embedded job management, load management, instrumentation and security mechanisms;
- Current grid middleware addresses most of the functional requirements well, but is immature in some areas, especially those involving network Quality of Service mechanisms;
- *While optimal application performance often requires new application designs for grid execution, many legacy applications have been successfully ported to grids.*

Clouds

- **NIST Cloud Definition:**

  Cloud computing is a "model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."
- **The intent behind *Cloud Computing* is to extend the ideas used in *Grid Computing* to provide a "utility" computing services scheme, where compute power, storage, or other services can be used flexibly on demand, and their usage metered and charged for;**
- **Cloud Computing is currently dominated by proprietary products rather than open standards.**

Programming Clouds

- **Most Cloud products currently available involve providing access to extant pools of servers built for other applications, such as bulk storage, retailing services, web searches and like;**
- **The Cloud product is employed to exploit and earn return on investment on surplus computing or storage capacity in an existing pool of servers;**
- **Therefore, there is no suite of standard APIs for Clouds, with providers often offering proprietary APIs or applications;**
- **Many cloud products are based on "virtual machines" where an emulator such as *VMWare* or *Parallels* is provided, and the end user must provide the operating system, applications and other tools – often this is called a "bare metal" API;**
- **Open Cloud Computing Interface (OCCI) is in development.**

Revision / Notes

- What are distributed systems?
- Challenges in distributed systems?
- Distributed computing models?
- Processes in operating systems (revision);
- IPC in operating systems (revision);
- Stream-oriented IPC in operating systems (revision);
- Parallelism vs. distributed computing;
- Clusters, Grids, Clouds, Distributed Storage;