CS 417:
the one-hour study guide for exam 1

Disclaimer: This study guide attempts to touch upon the most important topics that may be covered on the exam but does not claim to necessarily cover everything that one needs to know for the exam. Finally, don't take the *one hour* time window in the title literally.

Introduction

Why are distributed systems more interesting now than they may have been one or two dozen years ago? A number of advances in computing technology had a profound effect on distributed systems. Network connectivity increased by a factor of a thousand since the 1980s. Connectivity within a local area network (LAN) moved from shared to switched networking, allowing the network to scale without increasing congestion. On the wide area, Internet access has become available to the population at large, not just to researchers on Department of Defense projects. Processor performance, system memory, and disk capacity has also increased by more than a thousandfold over the past couple of decades.

Even though these improvements make it easier to store, process, and move data between systems, what are the motivations for distributing computing? There are several reasons. Performance does not scale linearly with an increase in price with a single computer; with a collection of computers this scaling may be possible. Secondly, distributing systems makes sense in certain environments: databases may reside in different locations than the user, for example. Thirdly, we may need to interact with other people or remote data that are geographically dispersed.

Taxonomy

One way of classifying system architectures is via Flynn's taxonomy, proposed by Michael J. Flynn in 1972. He categorized machines based on the number of instruction streams and the number of data streams. *SISD* (single instruction stream, single data stream) refers to conventional single processor systems. *SIMD* (single instruction stream, multiple data streams) refers to single processor machines where each instruction may process a collection of data. Vector and array processors fall into this category. This includes graphics processors, cell processors, Intel's SSE3 instructions in the Pentium 4, and the PowerPC's AltiVec instructions. Finally, *MIMD* (multiple instruction streams, multiple data streams) refer to any machines with multiple processors, where each processor operates on its own stream of data.

MIMD can be further categorized by identifying whether the system has a shared memory space or not. Systems with shared memory are known as *multiprocessor systems*. Examples are conventional PC with multi processors on a single system bus (e.g., a dual-processor Pentium 4). An architecture where multiple identical processors communicate with a single shared memory is called *SMP*, or *symmetric multiprocessing*. Systems without a shared memory are

collections of separate machines, each with its own memory. They have to rely on a network to communicate and are sometimes referred to as *multicomputers*.

Cache coherence

On a bus-based multiprocessor system, all the processors, system memory, and peripherals are connected to a shared system bus. Only one device can access the bus at any time. Since processors need to access memory continuously, there is constant contention for the system bus. Cache memory alleviates this, as it is a small amount of memory that is local to each processor with the expectation is that most programs exhibit a certain amount of locality, allowing most memory references to be satisfied from the cache.

The problem with using cached data is that if a processor modifies a memory location, it will only modify the local copy in the cache. Other processors will read the previous value. This is unacceptable but can be remedied by having write operations pass through to the system bus so that main memory can be updated. This is known as *write-through*. There is a performance penalty for doing this, but read operations usually far outnumber write operations. The second problem is that, even if a processor updates the data in main memory, another processor may still access its own cached data, which is now obsolete. This can be remedied by having each processor's cache controller monitor write operations on the system bus and detect whether others are modifying any cached addresses. This operation is known as *snooping*. Together, write-through and snooping comprise a *snoopy cache*. Virtually all multiprocessor systems employ a snoopy cache to ensure cache memory coherence.

Improving scalability

Because of its shared system bus, bus-based SMP systems face increasing bus congestion as the number of CPUs in the system increases. Beyond eight or so processors, the effects of this congestion can become quite severe, rendering the architecture impractical for highly parallel systems.

From a performance point of view, a *crossbar switch* is an ideal solution. It allows the switching of any processor to any chunk of memory without interfering with the traffic from any other processor accessing any other chunk of memory. With a sufficiently large crossbar switch, memory can be divided into a lot of chunks and delays will only occur when two processors need to access the same region of memory at the same time.

The problem with a crossbar switch is that it is expensive, particularly in the large sizes needed to support a large number of processors and a fine-grained partitioning of memory. An attempt to alleviate the cost of a crossbar switch is to decrease the aggregate number of switching elements (crosspoint switches) by increasing the number of switching stages. This is known as an *omega network*. Unfortunately, this slows down all memory accesses, as each request to memory has to pass through a number of crosspoint switches. A compromise solution is a non-uniform memory architecture, or *NUMA*. This architecture associates part of the global system memory with each processor. Every processor is able to access a portion of

the system memory at high speed through a local bus. The remaining memory is accessible through a shared (but slower) backplane. The idea is that a program may be loaded in such a way that most memory references are from local memory (and fast). The trick is for the operating system to place programs in memory and assign them to the correct processor.

AMD's HyperTransport technology (HTT) in its 64-bit Opteron CPUs is an example of NUMA support in current architectures. Each CPU has its own bank of DDR memory and communicates with inter-processor memory over the HTT link. Intel announced support for NUMA in 2007 with their Nehalem and Tukwila processors. These machines can be connected to shared memory via a Quick Path Interconnect (QPI).

In operating systems, a traditional scheduler, such as the one in the 2.4 Linux kernel, used a single run queue. A multi-queue scheduler with a separate run queue per processor was developed in the 2.5 kernel to support the NUMA model (2002). Microsoft added NUMA support in their Windows 2003 Server. The system attempts to improve memory access performance by scheduling threads on processors that are in the same node as the memory being used. It also tries to allocate memory within the same node if possible.

Software

One ideal of distributed systems software is something known as the *single-system image*. This is software that makes a collection of independent computers appear as a single system to the users.

Service models

The traditional computing model is a *centralized one*, where all computing takes place on a single system. A *client-server* architecture divides computing activity between *servers* (which provide some service) and *clients* (which access the service). This two-tiered model can extend to multiple-tiers (a service requests yet another service and so on). Another model is a *processor pool* model, where an arsenal of CPU servers can be accessed for computing needs. A generalization of this is *grid computing*, which provides users with a collection of processing power as well as storage resources, often supporting heterogeneous environments. *Thin clients* are those where the client software is minimal; often managing the user interface and nothing more (e.g., a system running only a web browser or an X-server). These are in contradistinction to *thick clients*, which are generally machines running entire applications with occasional reliance on services from servers (e.g., full PCs).

Networking

Networks fall into two broad categories: *circuit-switched* versus *packet switched*. A *circuit-switched* network is one where a dedicated path is established between two endpoints. It provides guaranteed bandwidth and constant latency. The telephone network is an example of circuit switching, providing a maximum delay of 150 msec. and digitizing voice to a constant 64 kbps data rate. *Packet-switched* networking uses a shared interconnect. Data is segmented into

packets and each packet must be identified and addressed. These networks generally cannot provide guaranteed bandwidth or constant latency. Ethernet is an example of a packet-switched network.

Data goes over a network in one of two ways: *baseband* or *broadband*. Only one node may transmit data at a time on a *baseband* network but, for that time, it has the full bandwidth of the network. A *broadband* network, on the other hand, has its available bandwidth divided into multiple channels, or frequency bands. Cable TV is an example of a broadband network. However, data services offered by cable providers are confined to two of those channels (one for downstream traffic and another for upstream), making IP access effectively baseband within broadband. Don't confuse these terms with the marketing community's use of *broadband* to refer to any relatively high-speed home networking.

Ethernet is the most widely used data network for local area networking. It uses a system called *carrier sense multiple access with collision detection* (CSMA/CD) to access the network. This is analogous to making a phone call on a party line system. The network interface card monitors the network. Only when it detects no traffic does it send out its packet. While doing so, it still monitors the network to detect a collision – the case where two cards decided to send a packet out simultaneously. If a collision took place, the transmission is reattempted again at a later time. Progressively longer back-off times are used when collisions are encountered to ensure that overall performance degrades gracefully.

Data networking is generally implemented as a stack of several *protocols* – each responsible for a specific aspect of networking. The OSI reference model defines seven layers of network protocols. Some of the more interesting ones are: the network, transport, and presentation layers. The *network layer* (3) manages the journey of packets from one machine to another. The *transport layer* (4) manages the communication of data from one application to another. The *presentation layer* (6) manages the representation of data and handles any necessary conversion of data types across architectures (for example, different byte ordering in integers, different character representations).

IP Networking

The *Internet Protocol* (IP) handles the interconnection of multiple local and wide-area networks. It is a logical network whose data is transported by physical networks (such as Ethernet, for example). Each machine on an IP network must have an IP address. The addressing scheme for IP divided addresses into two segments: a *network* part of the address, which is used in determining where to route the packet, and a *host* part, which is used in identifying the specific host within that local area network.

Instead of using a single network-host partition, IP was designed to use three distinct partitions, or *classes* of networks: A, B, and C. This allowed for a small number of huge networks and a large number of networks with a small number of machines. However, the allocation of machines to networks was still inefficient. An organization that needed addresses for 300 machines would be allocated a class B network, and over 65,000 addresses would go unused (a

class C network, accommodating only 254 machines, would have been too small). *Classless Inter-Domain Routing* (CIDR) was created to alleviate this inefficiency. Networks could be allocated to organizations on any power of two (arbitrary network-host partitioning). This made routing tables a bit more complex; they now need to have an extra datum: the number of leading bits that constitute the network part of the address.

Since IP is a logical network, any machine that needs to send out IP packets must do so via the physical network. Typically, this is Ethernet (which uses a 48-bit Ethernet address, unrelated to a 32-bit IP address). To send an IP packet out, the system needs to identify the *physical* destination address (MAC, or Media Access Control address) that corresponds to the desired IP destination. The *Address Resolution Protocol*, or *ARP*, accomplishes this. ARP works by broadcasting a request containing an IP address ("*do you know the corresponding MAC address for this IP address*?) and then waiting for a response from the machine with the corresponding IP address. To avoid doing this for every outgoing packet, it maintains a cache of most recently used addresses.

There are two transport-layer protocols on top of IP: TCP and UDP. TCP (*Transmission Control Protocol*) provides *virtual circuit* (*connection-oriented*) service. This layer of software ensures that packets arrive in order to the application and lost or corrupt packets are retransmitted. The transport layer keeps track of the destination so that the application can have the illusion of a connected data stream. UDP (*User Datagram Protocol*) provides *datagram* (*connectionless*) service. While UDP drops packets with corrupt data, it does not ensure in-order delivery or reliable delivery. *Port numbers* in both TCP and UDP are used to allow the operating system to direct the data to the appropriate application (or, more precisely, to the *socket* that is associated with the communication stream).

Sockets are an interface to the network provided to applications by the operating system. They are created with the *socket* system call and assigned an address and port number with the *bind* system call. For connection-oriented protocols, a socket on the server can be set to listen for connections with the *listen* system call. The *accept* call blocks until a connection is received, at which point the server receives a socket dedicated to that connection. A client can establish a connection with the *connect* system call. After this, sending and receiving data is compatible with file operations: the same *read/write* system calls can be used. When communication is complete, the socket can be closed with the *shutdown* or *close* system calls.

Quality of Service

As IP networks began to be used for carrying continuous media, such as voice and data, it became clear that the protocol has no provisions for controlling quality of service (QoS). There are two basic approaches for dealing with quality of service over networks: *hard QoS* and *soft QoS*. Soft QoS refers to prioritization without any reservation of resources from routers or endpoints or any *a priori* negotiation for a level of service. Hard QoS refers to a system where such negotiation may take place and the network is capable of providing a guaranteed level of service for a data stream.

IP was largely designed as a system that would provide best-effort packet delivery but with no guarantees on the path a packet will take, whether it gets dropped, or what order it arrives in. Several approaches were taken to attempt to provide better controls for the delivery of IP packets.

Flow detection: Many routers attempt to detect a *flow* of a stream of packets from one address/port to another address/port and then dropping or delaying packets to control the flow rate. Routers can also be programmed to drop TCP packets over UDP or vice versa or drop packets when traffic exceeds an allotted bandwidth. Routers can also manage several queues based on flows, connected networks, or source or destination addresses and ports. *Traffic shaping* is when a router queues packets in certain flows during peak usage for later retransmission when there is available bandwidth. With *traffic policing*, traffic that exceeds the allocated bandwidth for a particular flow is discarded.

Inefficient use of packets: Having a system send lots of small packets instead of a few larger ones is clearly inefficient. For example, the overhead of TCP, IP, and ethernet headers is approximately 58 bytes. Sending one byte of date requires 59 bytes – a 5,800% overhead! *Nagle's algorithm* adds any new transmitted TCP/IP data to a buffer rather than sending it immediately if there are any unacknowledged packets outstanding. Incidentally, Nagle's algorithm can be disabled on a socket with the TCP_NODELAY option to the *setsockopt* system call

Differentiated services: Flow control mechanisms are outside the purview of programmers or even the computers at either end: it is up to the router configuration to define the policies. Differentiated services provide a way for programmers to provide advisory information inside an IP header on how a packet should be processed by routers. A packet can be assigned a priority as well as high/low levels for reliability, throughput, and delay. It is entirely up to the routers to decide how to process this information or whether to process it at all. Differentiated services are referred to as *soft QoS*: there is no guarantee on the actual quality of service that will be delivered.

Hard QoS approach: The Reservation protocol, RSVP, has been developed to allow a flow of packets to be routed with rate and/or delay guarantees. The problem with providing this is that all routers in the path from the source to the destination must be configured to support RSVP: each intermediate router must commit to reserving the needed amount of routing resources to guarantee the desired level of service.

ATM

Bandwidth and latency are, of course, critical issues for real-time and streaming media applications. We can categorize the timing demands of traffic into three categories: *asynchronous* data has no timing requirements for message delivery; *synchronous* data must be delivered at strict deadlines; and *isochronous* data must meet specific bandwidth needs but may be delivered sooner than needed (streaming media with a receive buffer is an example).

ATM, or *Asynchronous Transfer Mode*, networking was created to bridge the synchronous needs of telephony (low bandwidth but precisely scheduled), asynchronous data networking (often high bandwidth), and isochronous streaming media applications. ATM is a packet based network that negotiates a circuit (route) when a connection is first established. Every router commits to having the requisite resources to provide for the service needs of the connection. The connection is created to provide ABR (available bit rate), CBR (constant bit rate), or VBR (variable bit rate) service.

ATM routes small (53-byte) cells in contrast to the variable-size packets of IP. This allows a router to express its service in cells per second, simplifies switching, and avoids the issue of large packets delaying small ones.

Naming

Names are used for identifying a variety of things. We often use a name server (offering a naming service) to perform a name to address mapping. An address is nothing more than the lower representation of a name. For example, you can't just look at a string like 192.168.1.5 and say it is an address. To an IP driver it is treated as a name and the address is the underlying ethernet address. *Binding* is the association of a name to an address. *Resolution* is the process of looking up that name to address binding.

Static binding refers to a hard-wired association between a name and an address (e.g., hard-coded in a program). *Early binding* refers to a resolution that is performed ahead of time and cached for future use. *Late binding* refers to performing the resolution just at the time a name-to-address binding is needed.

The *Domain Name Server* (DNS) is an example of a distributed name server for resolving domain names into IP addresses. Each server is responsible for answering questions about machines within its zone. A name server will do one of several things: (1) answer a request if it already knows the answer, (2) contact another name server(s) to search for the answer, (3) return an error fit he domain name does not exist, or (4) return a *referral* – the address of another name server that may know more of the answer. For example, searching for mail.pk.org (with no cached information) begins by querying one of several replicated root name servers. These keep track for name servers responsible for top-level domains. This query will give you a referral to a name server that is responsible the .org domain; querying that name server will give you a name server responsible for pk.org. Finally, the name server responsible for pk.org can provide the IP address or an authoritative "this host does not exist" response. Along the way, referrals are cached so that a name server need not go through the entire process again.

Case Study: The Google Cluster Architecture

The Google Cluster architecture is built atop over 10,000 unreliable commodity PCs running fault-tolerant software. The goal of the system is energy efficiency and the best price for the realized performance rather than maximizing processor performance.

The Google search service contains several clusters that are distributed worldwide. Each of these clusters has several thousand machines. A user's query is directed to a specific cluster via the DNS lookup of google.com. The DNS load-balancing system takes the user's round-trip time and system capacity into account to provide the address of a suiteable cluster. Once the request gets to the cluster, a hardware load balancer forwards the request to one of the web servers in the cluster.

The fundamental approach to exploiting distribution is to transform a query on a very large database into a much of queries on smaller databases, followed by a merge of the results.

The web server performs a query against several index servers. Since the index is many terabytes in size, it is divided into pieces (shards), each holding a subset of the documents from the full index. Each of these index servers is replicated, with a load balancer assigning query requests. If any of the replica servers goes down then performance is degraded proportionally. The index server returns a list of document IDs to the web server that made the query.

The next task for the web server is to take the result from the index queries and consult document servers that get the title, URL, and the in-context snippet from the document. As with index lookup, the documents are also randomly distributed among multiple document servers and each server has multiple replicas that are load balanced.

Most index and document lookup operations are read-only and updates to the data are infrequent. Replicas can be taken off-line during an update, so consistency problems are not an issue.

Since individual shards (pieces of the index or pieces of the document database) don't need to communicate with each other, performance can scale almost linearly with an increase in the number of machines. Because the performance scales so well, the emphasis on optimizing cost favors using commodity components instead of, say, quad-processor motherboards whose cost is disproportionately higher as well as using inexpensive and somewhat slower disks.

Remote Procedure Calls

One problem with the interface offered by sockets was that it encouraged a *send-receive* model of interaction. However, most programs use a functional (procedure call) model. Remote procedure calls are a programming language construct (something provided by the compiler, as opposed to an operating system construct such as sockets). They provide the illusion of calling a procedure on a remote machine. During this time, execution of the local thread stops until the results are returned. The programmer is alleviated from packaging data, sending and receiving messages, and parsing results.

The illusion of a remote procedure call is accomplished by generating *stub functions*. On the client side, the stub is a function with the same interface as the desired remote procedure. Its job is to take the parameters, *marshal* them into a network message, send them to the server, await a reply, and then *unmarshal* the results and return them to the caller. On the server side,

the stub (sometimes known as a skeleton) is responsible for being the main program that registers the service and awaits incoming requests for running the remote procedure. It unmarshals the data in the request, calls the user's procedure, and marshals the results into a network message that is sent back to the recipient.

Sun (ONC) RPC

Sun's RPC was one of the first RPC systems to achieve widespread use. It is still in use on virtually all Unix-derived systems (System V, *BSD, Linux, OS X). It uses a precompiler called *rpcgen* that takes input from an *interface definition language* (IDL) file. This is a file that defines the interfaces to the remote procedures. Fron this, *rpcgen* creates client stub functions and a server stub program. These can be compiled and linked with the client and server functions, respectively.

Every interface is assigned a unique 32-bit number, known as a program number. When the server starts up, it binds a socket to any available port and registers that port number and its program number with a name server, known as the *portmapper*, running on the same machine. A client, before invoking any remote procedure calls, contacts the portmapper on the desired server to find the port to which it needs to send its requests.

DCE RPC

The Distributed Computing Environment, defined by the Open Group created its own flavor of RPC, very similar to Sun's. They also had the programmer specify an interface in an IDL, which they called the *Interface Definition Notation* (IDN).

To avoid the problem of picking a "unique" 32-bit identifier for the interface, DCE RPC provides the programmer with a program called *uuidgen*. This generates a *unique universal ID* (UUID) – a 128-bit number that is a function of the current time and ethernet address.

The DCE also introduced the concept of a *cell*, which is an administrative grouping of machines. Each cell has a cell directory server that maintains information about the services available within the cell. Each machine in the cell knows how to contact the cell directory server. When a server program starts up under DCE RPC, it registers its port and the interface's UUID with a local name server (the DCE host dæmon, *dced*, which is similar to the *portmapper*). It also registers the UUID, host mapping with the cell directory server. This allows a degree of location transparency for services: a client does not need to know what machine a service lives on *a priori*.

As object oriented languages gained popularity in the late 1980s and 1990s, RPC systems like Sun's and DCE's proved incapable of handling some object oriented constructs, such as instances of objects or polymorphism (different functions sharing the same name, with the function distinguished by the incoming parameters). A new generation of RPC systems dealt with these issues.

Microsoft DCOM & ORPC

Microsoft already had a scheme in place for dynamically loading components into a process. This was known as COM, the Component Object Model and provided a well-defined mechanism for a process to identify and access interfaces within the component. The same model was extended to invoke remotely-located components to become the Distributed Component Object Model (*DCOM*). Because the components can no longer be loaded into the local process space (as they're on a remote system), they have to be loaded by *some* process. This process is known as a *surrogate process*. It runs on the server, accepting remote requests for loading components and invoking operations on them.

DCOM is implemented through remote procedure calls. Microsoft enhanced DCE RPC to create what they termed *Object RPC* (ORPC). This is essentially DCE RPC with the addition of support for an *interface pointer identifier* (IPID). The IPID provides the ability to identify a specific instance of a remote object. Interfaces are defined via the Microsoft Interface Definition Language (MIDL) and compiled into client and server side stubs. The client-side stub becomes the *local* COM object that is loaded when the object is activated.

Since objects can be instantiated and deleted remotely, the surrogate process needs to ensure that there isn't a build-up of objects that are no longer needed by any client. Microsoft accomplishes this via *remote reference counting*. This is an explicit action where the client can send requests to increment or decrement a reference count on the server. When their reference count drops to zero, the surrogate process deletes the object. To guard against programming errors or processes that terminated abnormally, a secondary mechanism exists, called *pinging*. The client must periodically send the server a *ping set* – a list of all the remote objects that are currently active. If the server does not receive this information within a certain period, it elides the objects.

CORBA

The *Common Object Request Broker Architecture* (CORBA) was created to provide a software platform for distributing objects that is architecture, language, and operating system independent. The core concept is the ORB – the *Object Request Broker*. This is the collection of stub functions and libraries that support the remote invocation of procedures and the management of objects. CORBA provides an Interface Definition Language (IDL) that is compiled with a pre-compiler to create client and server stubs.

CORBA provides a rich set of capabilities for the management of objects. These include the ability to start the server if it is not running, discover interfaces, persist objects to persistent media,

One of the biggest problems with CORBA (aside from its complexity) was the fact that, while the programming interfaces were defined, the underlying protocol was not. This meant that clients and servers would often not be able to communicate unless they used an implementation of CORBA from the same vendor. This was rectified in 1996 with the

introduction of the *Internet Inter-ORB Protocol* (IIOP). By this time, however, much of the momentum of using CORBA over the Internet was gone.

Java RMI

When Java was created, it was designed to be a language for deploying downloadable applets. In 1995, Sun extended Java to support *Remote Method Invocation* (RMI). This allows a programmer to invoke methods on objects that are resident on other JVMs.

Since RMI is designed for Java, there is no need for OS, language, or architecture interoperability. This allows RMI to have a simple and clean design. Classes that interact with RMI must simply play by a couple of rules. All parameters to remote methods must implement the *serializable* class. This ensures that the data can be serialized into a byte stream for transport over the network. All remote interfaces (methods that may be invoked from a remote Java virtual machine) must extend the *remote* class.

RMI provides a naming service called *rmiregistry* to allow clients to locate remote object references. These objects are given symbolic names and looked up via a URI naming scheme (e.g., "rmi://remus.rutgers.edu:2311/testinterface").

Java's distributed garbage collection is somewhat simpler than Microsoft's. There are two operations that a client can invoke: *dirty* and *free*. A client JVM sends a *dirty* call to the server when the object is in use and is refreshed periodically. When there are no more references to the object, the client sends a *clean* call to the server so that it can delete the object.

XML RPC

As people started looking beyond the LAN for hosting services via RPC, firewalls stood in the way. Most server-side RPC systems had a habit of asking the operating system to pick any available port. This required opening up a whole range of ports on a firewall. Moreover, people often could not get firewall rules modified in some environments. A workaround to this is to send RPC messages via HTTP ports (e.g., 80 and 443), and have the messages formatted as XML messages to get around any content-inspecting firewalls.

XML-RPC was created in 1998 as a simple protocol that marshals all requests and responses into XML messages. There are a lot of libraries to support this system but no IDL compiler or stub function generator thus far.

SOAP

XML RPC took an evolutionary fork and evolved (with the support of companies such as Microsoft and IBM) into something known as SOAP, the *Simple Object Access Protocol*. XML RPC is a subset of SOAP. In addition to remote procedure calls, SOAP added support for general purpose messaging (send, receive, asynchronous notification) of messages. SOAP invocations are XML messages sent via an HTTP protocol. SOAP services can be described via an XML

document formatted to conform to an interface specified by a corresponding Web Services Description Language (WSDL) document – another XML document.

.NET

A problem with Microsoft's DCOM was that it was a somewhat low-level implementation. Clients had to provide reference counting of their objects explicitly and the convenience of using DCOM depended very much on the language (easy in VB, difficult in C++). Moreover, the use of operating-system-selected random ports, and a binary data format made it difficult to use over the Internet where firewalls may block certain ports or requests need to in an XML format and be sent over HTTP.

With .NET, Microsoft provided (among other things) a runtime environment, a set of libraries, and a web server that provides inbuilt support for web services. For supporting function-based access to web services, .NET provides a *remoting* capability that allows a process to interact with objects that reside on other processes and other machines.

.NET Remoting was created to be a successor to DCOM that would work more easily over the Internet (no random ports, for example, as well as the ability to use XML over HTTP). As with other RPC systems, client and server stub functions (proxies) are created. Microsoft's Visual Studio application development environment hides the mechanics of this and integrates remote procedure calls directly into the language.

These stubs rely on the .NET runtime system to marshal parameters into one of several types, which include SOAP or binary formats. The .NET runtime system then sends the message over a particular *channel* that that was set up for transport. This channel may be HTTP using TCP/IP to send SOAP messages, TCP/IP with no wrapping protocol to send binary messages on a local-area network, or named pipes to send messages between processes on the same machine. Microsoft's web server, IIS, can be configured to directs certain URLs that contain the SOAP (encapsulated in HTTP) request to specific objects running under the .NET framework, which then sends a response back to the web server, which is then returned to the caller.

.NET manages object lifetime in three ways:

1. *Single call objects* instantiate a new instance of the object for a call and immediately clean it up upon return.

2. *Singleton objects* share the same instance of the object among all callers. This is much like traditional function calls in non-object-oriented systems.

3. Finally, *Client activated objects* are similar to objects in DCOM and created on demand (e.g., via a *new* operation). .NET manages object lifetime of client activated objects by setting a lease timer each time a method is called on an object. The object will be cleaned up unless the client either makes additional calls to reset the lease time or sends an explicit message

to renew the lease time. This is a very similar concept to Java RMI. There is no more reference counting as under DCOM.

Distributed file systems

To provide the same system call interface for supporting different local file systems as well as remote files, operating systems generally rely on a layer of abstraction that allows different file system-specific interfaces to coexist underneath the common system calls. On most Unix-derived systems (e.g., Linux, BSD, OS X, SunOS), this is known as the VFS layer (Virtual File System).

There are a couple of models for implementing distributed file systems: the *download/upload* model or the *remote procedure call* model. In a *stateful* file system, the server maintains varying amounts of state about client access to files (e.g., whether a file is open, whether a file has been downloaded, cached blocks, modes of access). In a *stateless* file system, the server maintains no state about a client's access to files. The design of a file system will influence the *access semantics* to files. *Sequential semantics* are what we commonly expect to see in file systems, where reads return the results of previous writes. *Session semantics* occur when an application "owns" the file for the entire access session, writing the contents of the file – hence making the updates visible to others – on close, and thereby overwriting any modifications made by others prior to that.

NFS

NFS was designed as a stateless, RPC-based model implementing commands such as *read bytes*, *write bytes*, *link files*, *create a directory*, and *remove a file*. Since the server does not maintain any state, there is no need for remote *open* or *close* procedures: these only establish state on the client. NFS works well in faulty environments – there's no state to restore if a client or server crashes. To improve performance, a client reads data a block (8 KB by default) at a time and performs *read-ahead* (fetching future blocks before they are needed). NFS suffers from ambiguous semantics because the server (or other clients) has no idea what blocks the client has cached and the client does not know whether its cached blocks are still valid. The system checks modification times if there are file operations to the server and otherwise invalidates the blocks after a few seconds. File locking could not be supported because of NFS's stateless design but was added through a separate lock manager that maintained the state of locks.