ID #14110

NAME :SALMAN AFRIDI

PROGRAME MS(CS)

UNIVERSITY INU PESHAWAR

PAPER FINAL

Spring Semester 2020 Final Exam
Course: - Distributed Computing

Deadline: - Mentioned on SIC **Marks: - 50**

Program: - MS (CS) **Dated: 24 June 2020**

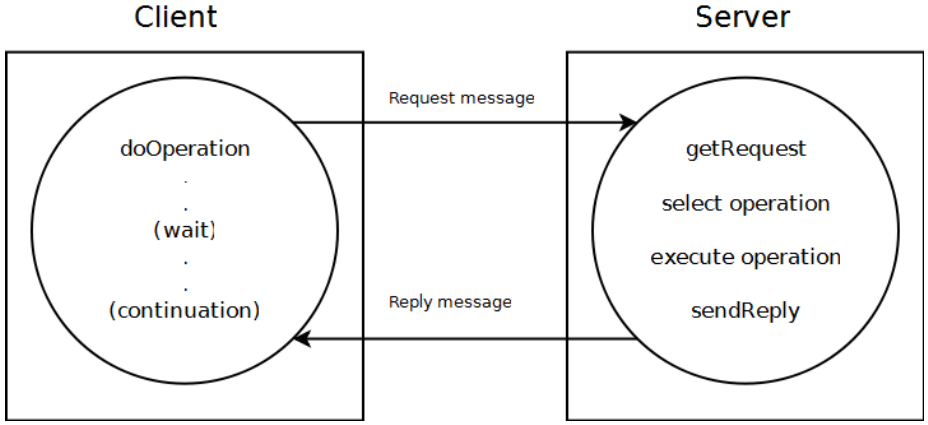
Student Name: SALMAN AFRIDI Student ID#:14110

Class and Section:_____

Section: Remote Invocation

Q1. Describe briefly the purpose of the three communication primitives in request-reply protocols. (6)

- **ANSWER: Normally in request-reply communication the client process blocks until the reply arrives from the server (synchronous).**
- **It can also be reliable because the reply from the server is effectively an acknowledgement to the client.**
- **Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later.**
- **The request-reply protocol we describe here is based on a trio of communication primitives, *doOperation*, *getRequest* and *sendReply*, as shown in Figure Are as below:**



- **The *doOperation* method is used by clients to invoke remote operations.**

- **Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation.**

Below 5.3 Figure outlines the three communication primitives:

Figure 5.3
Operations of the request-reply protocol

***public byte**[] doOperation (RemoteRef s, **int** operationId, **byte**[] arguments)*
sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

***public byte**[] getRequest ();*
acquires a client request via the server port.

***public void** sendReply (**byte**[] reply, InetAddress clientHost, **int** clientPort);*
sends the reply message reply to the client at its Internet address and port.

- **The doOperation method sends a request message to the server whose Internet address**
- **port are specified in the remote reference given as an argument. The caller of doOperation is blocked until the server performs the requested operation and transmits a reply message to the client process.**
- **getRequest is used by a server process to acquire service requests, as shown in Figure 5.3**
- **When the server has invoked the specified operation, it then uses.**
- **sendReply to send the reply message to the client. When the reply message is received by the client the original doOperation is unblocked and execution of the client program continues.**

Q2. Explain the technical difference between RPC and RMI? (4)

ANSWER: Remote Procedure Call (RPC): is a programming language feature devised for the distributed computing and based on semantics of(**local procedure)** calls. It is the most common forms of remote service and was designed as a way to abstract the procedure call mechanism to use between systems connected through a network. It is similar to IPC mechanism where the operating system allows the processes to manage shared data and deal with an environment where different processes are executing on separate systems and necessarily require message-based communication.
Remote Method Invocation (RMI): is similar to RPC but is language specific and a feature of java. A thread is permitted to call the method on a remote object. To maintain the transparency on the client and server side, it implements remote object using stubs and skeletons. The stub resides with the client and for the remote object it behaves as a proxy. When a client calls a remote method, the stub for the remote method is called. The client stub is accountable for creating and sending the

parcel containing the name of a method and the marshaled parameters, and the skeleton is responsible for receiving the parcel.

Section: Indirect Communication

Q:3 In contrast to Direct Communication, which two important properties are present in Indirect Communication? (6)

ANSWER: Direct communication may be used when there is no room for discussion or compromise. This style usually doesn't allow the listener to respond with an opinion or viewpoint. For example, your supervisor may say to you, 'You need to get to work on time every day. You must not be late again.

INDIRECT COMMUNICATION: Define as the the communication between entities in distributed system through intermediary with no direct coupling between the sender and receiver's.

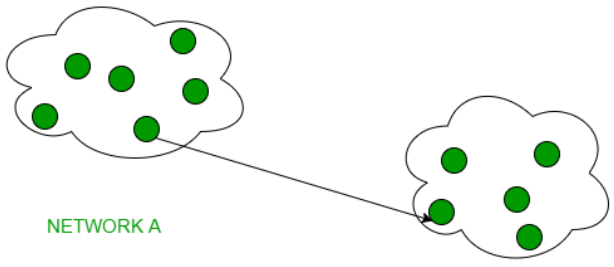
Key properties Space uncoupling: The sender does not know or need to know the identity of the receiver(s), and vice versa.

Time uncoupling :The sender and the receiver(s) can have independent lifetimes. Indirect communication is often used in distributed systems where change is anticipated.

Examples • Mobile environments where users may rapidly connect to and disconnect from the network • Managing event feeds in financial systems

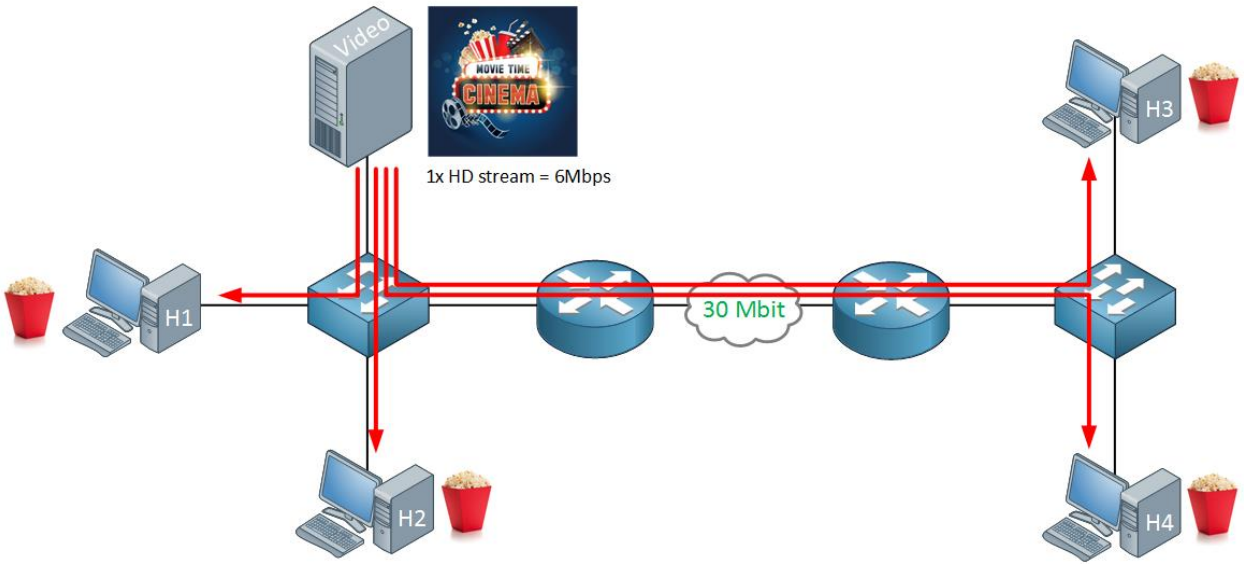
Q:4 Provide three reasons as why group communication (single multicast operation) is more efficient than individual unicast operation? (9)

ANSWER: **1. Unicast –This type of information transfer is useful when there is a participation of single sender and single recipient. So, in short you can term it as a one-to-one transmission. For example, a device having IP address 10.1.2.0 in a network wants to send the traffic stream(data packets) to the device with IP address 20.12.4.2 in the other network,then unicast comes into picture. This is the most common form of data transfer over the networks.**



UNICAST EXAMPLE

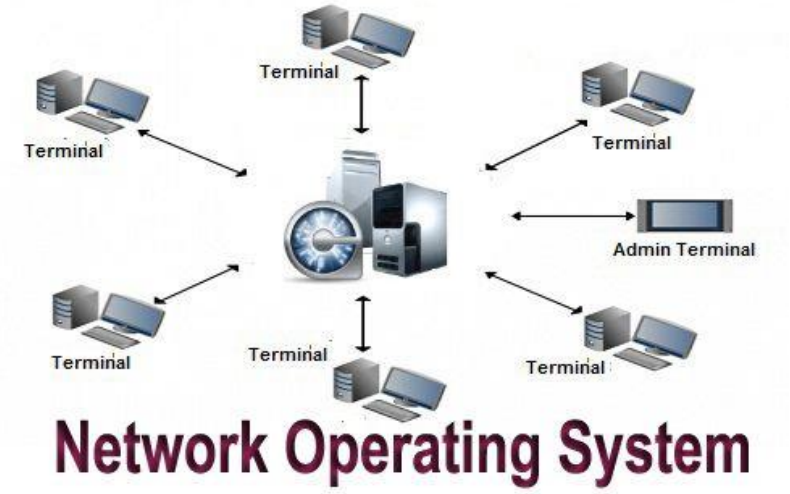
In multicasting:one/more senders and one/more recipients participate in data transfer traffic. In this method traffic recline between the boundaries of unicast (one-to-one) and broadcast (one-to-all). Multicast lets server's direct single copies of data streams that are then simulated and routed to hosts that request it. IP multicast requires support of some other protocols like IGMP (Internet Group Management Protocol), Multicast routing for its working. Also in Classful IP addressing Class D is reserved for multicast groups



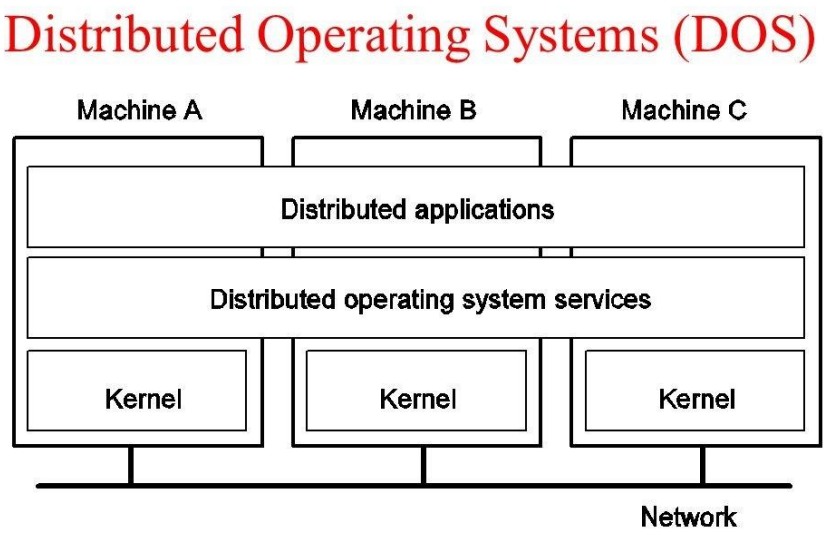
Section: OS Support

Q5. Differentiate a between a network OS and distributed OS. (6)

ANSWE: Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. } The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other network.



Distributed Operating System is a model where distributed applications are running on multiple computers linked by communications. } Distributed system is a collection of independent computers that appear to the user of the system as a single computer. A distributed operating system is an extension of the network operating system that



Network Operating Systems :- Contains N copies of Operating Systems, communication between machines is via shared files.

Distributed OS : - Contains N copies of Operating systems, communication between nodes is via messages over a network. These messages pass the necessary parameters for the task and on completion messages return the results. It as if computers sends emails to other computers with request and answer.

NOS vs. DO/S

Network operating system (NOS)	**Distributed operating system (DO/S)**
Resources owned by local nodes	Resources owned by global system
Local resources managed by local OS	Local resources managed by a global DO/S
Access performed by local OS	Access performed by DO/S
Requests passed from one local OS to another via NOS	Requests passed directly from node to node via DO/S

Q6. Describe briefly how the OS supports middleware in a distributed system by providing and managing (6)

a) **Process and threads**
b) **System Virtualization**

ANSWER6: Middleware in the context of distributed applications is software that provides services beyond those provided by the operating system to enable the various components of a distributed system to communicate and manage data. Middleware supports and simplifies complex distributed applications .

A)process and threads:

A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

A *thread pool* is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads.

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run i Operating system abstracts operating system components to guest operating systems such as memory access, file system, and network access. One key component of this type of virtualization is that the kernel of the parent operating system is the same kernel used in each guest operating system. This type of virtualization avoids emulation since the same system call interface is shared by each guest. Memory and CPU resources can be managed very effectively because load balancing is more efficient since there is not a

boundary that must be crossed to perform process execution.

Since all guests hosted using OS virtualization share the same kernel, they also share any problems within the shared kernel including stability or security problems. Upgrades can be problematic because upgrading one virtual machine requires updating all virtual machines on the same host system, which can cause logistical problems.

n the context of the threads that schedule them.

B)

Section: Distributed Objects and Components

Q7. Write in your own words the issues with Object (distributed) oriented middlewares. (13)

ANSWER: Two fundamental trends influence the way we conceive and construct new computing and information systems. The first is that information technology of all forms is becoming highly commoditized i.e., hardware and software artifacts are getting faster, cheaper, and better at a relatively predictable rate. The second is the growing acceptance of a network-centric paradigm, where distributed applications with a range of quality of service (QoS) needs are constructed by integrating separate components connected by various forms of communication services. The nature of this interconnection can range from 1. The very small and tightly coupled, such as avionics mission computing systems to 2. The very large and loosely coupled, such as global telecommunications systems. The interplay of these two trends has yielded new architectural concepts and services embodying layers of middleware. These layers are interposed between applications and commonly available hardware and software infrastructure to make it feasible, easier, and more cost effective to develop and evolve systems using reusable software. Middleware stems from recognizing the need for more advanced and capable support–beyond simple connectivity–to construct effective distributed systems. A significant portion of middleware-oriented R&D activities over the past decade have focused on 1. The identification, evolution, and expansion of our understanding of current middleware services in providing this style of development and 2. The need for defining additional middleware layers and capabilities to meet the challenges associated with constructing future network-centric systems. These activities are expected to continue forward well into this decade to address the needs of next-generation distributed applications. During the past decade we've also benefited from the commoditization of hardware (such as CPUs and storage devices) and networking elements (such as IP routers). More recently, the maturation of programming languages (such as Java and C++), operating environments (such as POSIX and Java Virtual Machines), and enabling fundamental middleware based on previous middleware R&D (such as CORBA, Enterprise Java Beans, and .NET) are helping to commoditize many software components and architectural layers. The quality of commodity software has generally lagged behind hardware, and more facets of middleware are being conceived as the complexity of application requirements

increases, which has yielded variations in maturity and capability across the layers needed to build working systems. Nonetheless, recent improvements in frameworks [John97], patterns [Gam95, Bus96, Sch00b], and development processes [Beck00, RUP99] have encapsulated the knowledge that enables common off-the-shelf (COTS) software to be developed, combined, and used in an increasing number of real-world applications, such as e-commerce web sites, consumer electronics, avionics mission computing, hot rolling mills, command and control planning systems, backbone routers, and high-speed network switches. The trends outlined above are now yielding additional middleware challenges and opportunities for organizations and developers, both in deploying current middleware-based solutions and in inventing and shaping new ones. To complete our overview, we summarize key challenges and emerging opportunities for moving forward, and outline the role that middleware plays in meeting these challenges. • Growing focus on integration rather than on programming – There is an ongoing trend away from programming applications from scratch to integrating them by configuring and customizing reusable components and frameworks [John97]. While it is possible in theory to program applications from scratch, economic and organizational constraints– as well as increasingly complex requirements and competitive pressures–are making it infeasible to do so in practice. Many applications in the future will therefore be configured by integrating reusable commodity hardware and software components that are implemented by different suppliers together with the common middleware substrate needed to make it all work harmoniously. • Demand for end-to-end QoS support, not just component QoS – The need for autonomous and time-critical behavior in next-generation applications necessitates more flexible system infrastructure components that can adapt robustly to dynamic end-to-end changes in application requirements and environmental conditions. For example, next-generation applications will require the simultaneous satisfaction of multiple QoS properties, such as predictable latency/jitter/throughput, scalability, dependability, and security. Applications will also need different levels of QoS under different configurations, environmental conditions, and costs, and multiple QoS properties must be coordinated with and/or traded off against each other to achieve the intended application results. Improvements in current middleware QoS and better control over underlying hardware and software components–as well as additional middleware services to coordinate these–will all be needed. • The increased viability of open systems – Shrinking profit margins and increasing shareholder pressure to cut costs are making it harder for companies to invest in long-term research that does not yield short-term pay offs. As a result, many companies can no longer afford the luxury of internal organizations that produce completely custom hardware and software components with proprietary QoS support. To fill this void, therefore, standards-based hardware and software researched and developed by third parties–and glued together by common middleware–is becoming increasingly strategic to many industries. This trend also requires companies to transition away from proprietary architectures to more open systems in order to reap the benefits of externally developed components, while still maintaining an ability to compete with domain-specific solutions that can be differentiated and customized.

How Middleware Addresses Distributed Application Challenges Requirements for faster development cycles, decreased effort, and greater software reuse motivate the creation and use of middleware and middleware-based architectures. Middleware is

systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to 1. Functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate and 2. Enable and simplify the integration of components developed by multiple technology suppliers. When implemented properly, middleware can help to: • Shield software developers from low-level, tedious, and error-prone platform details, such as socket-level network programming. • Amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use. • Provide a consistent set of higher-level network-oriented abstractions that are much closer to application requirements in order to simplify the development of distributed and embedded systems. • Provide a wide array of developer-oriented services, such as logging and security that have proven necessary to operate effectively in a networked environment. Over the past decade, various technologies have been devised to alleviate many complexities associated with developing software for distributed applications. Their successes have added a new category of systems software to the familiar operating system, programming language, networking, and database offerings of the previous generation. Some of the most successful of these technologies have centered on distributed object computing (DOC) middleware. DOC is an advanced, mature, and field-tested middleware paradigm that supports flexible and adaptive behavior. DOC middleware architectures are composed of relatively autonomous software objects that can be distributed or collocated throughout a wide range of networks and interconnects. Clients invoke operations on target objects to perform interactions and invoke functionality needed to achieve application goals. Through these interactions, a wide variety of middleware-based services are made available off-the-shelf to simplify application development. Aggregations of these simple, middleware-mediated interactions form the basis of large-scale distributed system deployments

Structure and Functionality of DOC Middleware: Just as networking protocol stacks can be decomposed into multiple layers, such as the physical, data-link, network, transport, session, presentation, and application layers, so too can DOC middleware be decomposed into multiple layers, such

APPLICATIONS

DOMAIN-SPECIFIC MIDDLEWARE SERVICES

COMMON MIDDLEWARE SERVICES

DISTRIBUTION MIDDLEWARE

HOST INFRASTRUCTURE MIDDLEWARE

OPERATING SYSTEMS & PROTOCOLS

HARDWARE DEVICES