SAIF ULLAH ID:15598 MS(CS)

Section: Remote Invocation

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

Answer 1: Three communication primitives in request-reply protocols:

Request-reply protocols represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing. In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC and RMI.

The three main communication primitive in request reply protocol (RRP) is

1. doOperation
2. getRequest
3. sendReply

The protocol we describe here is based on a trio of communication primitives, doOperation, getRequest and sendReply.

1. **doOperation:**

   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. Its result is a byte array containing the reply. It is assumed that the client calling doOperation marshals the arguments into an array of bytes and unmarshals the results from the array of bytes that is returned. The first argument of doOperation is an instance of the class RemoteRef, which represents references for remote servers. This class provides methods for getting the Internet address and port of the associated server. The doOperation method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument. After sending the request message, doOperation invokes receive to get a reply message, from which it extracts the result and returns it to the caller. The caller of doOperation is blocked until the server performs the requested operation and transmits a reply message to the client process.

2. **getRequest:**

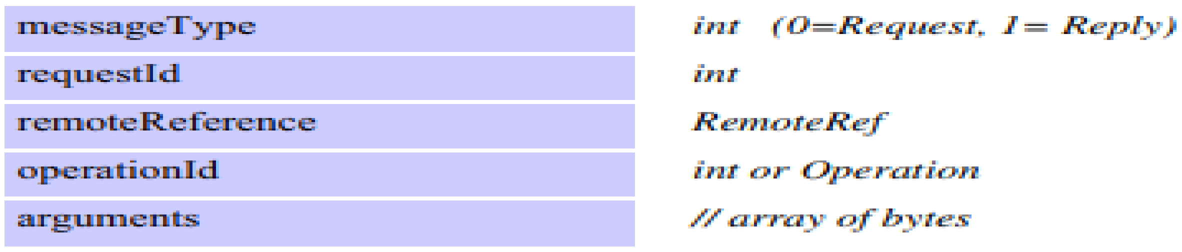
getRequest is used by a server process to acquire service requests, When the server has invoked the specified operation, it then uses sendReply to send the reply message to the client. When the reply message is received by the client the original doOperation is unblocked and execution of the client program continues.

3. **Send Reply:**

When the server has invoked the specified operation, it then uses sendReply to send the reply message to the client.

The information to be transmitted in a request message or a reply message is shown in Figure. Request-reply message structure messageType int (0=Request, 1= Reply) requestId int remoteReference RemoteRef operationId int or Operation arguments // array of bytes .

The first field indicates whether the message is a Request or a Reply message. The second field, requestId, contains a message identifier. A doOperation in the client generates a requestId for each request message, and the server copies these IDs into the corresponding reply messages. This enables doOperation to check that a reply message is the result of the current request, not a delayed earlier call. The third field is a remote reference. The fourth field is an identifier for the operation to be invoked. For example, the operations in an interface might be numbered 1, 2, 3, ... , if the client and server use a common language that supports reflection, a representation of the operation itself may be put in this field.



messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

Q2. Explain the technical difference between RPC and RMI?

Answer 2: Technical difference between RPC and RMI

Difference between RPC and RMI:

RPC and RMI are the systems which empower a customer to summon the technique or strategy from the server through building up correspondence among customer and server. The basic contrast among RPC and RMI is that RPC just backings procedural programming while RMI bolsters object-oriented programming.

Remote treatment call (RPC) is a system correspondence process with server and purchaser design and the thought behind RPC is to call executed code remotely as though we were simply reaching a capacity. Actually the main contrast among RMI and RPC is if there should arise an occurrence of RPC capacities are summoned by utilizing an intermediary work, and on the off chance that there is RMI we conjure strategies by utilizing an intermediary thing.

RMI is java approach to RPC, with availability to existing frameworks utilizing local strategies. RMI may take a characteristic, direct, and completely controlled way to deal with give an undertaking circulated preparing innovation that permits us to incorporate Java activity all through the machine. To accomplish the cross-stage transportability that Java gives, RPC requires significantly a greater number of overheads than RMI. RPC must change over the contentions between design with the goal that every PC can utilize its nearby information type.

Java-RMI is safely combined with the Java expressions. While RPC isn't explicit to any sole language and you can execute RPC utilizing distinctive wording.

Since RMI can executed utilizing Java, its get all points of interest like article arranged, equal figuring, plan style, simple to compose and re use, protected and secure, Write once and run anyplace. Be that as it may, in reality of RPC, to achieve any of these points of interest you need to compose usage code.

Key Differences between RPC and RMI:

1. RPC supports procedural programming paradigms thus is C based, while RMI supports object-oriented programming paradigms and is java based.

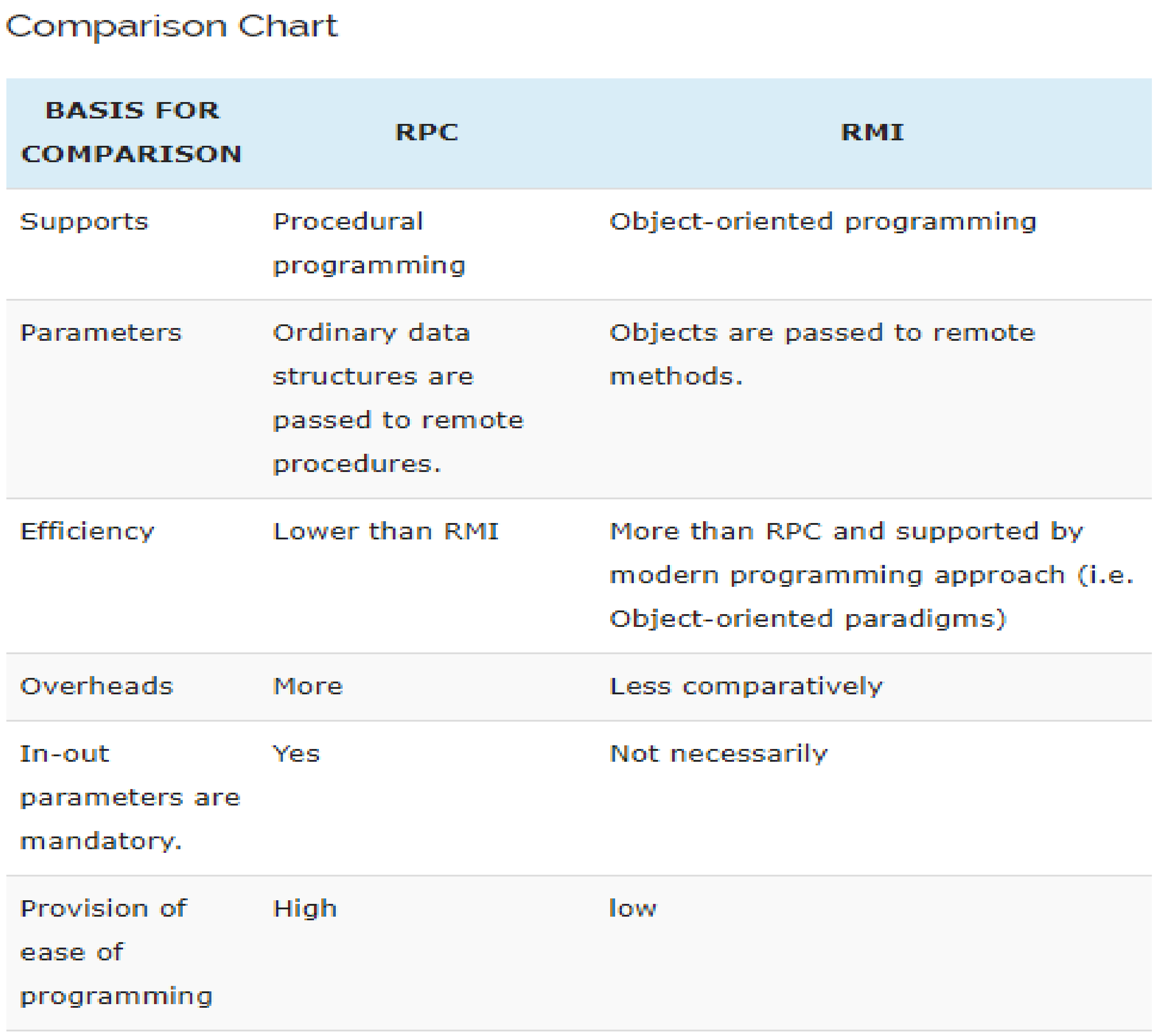
2. The parameters passed to remote procedures in RPC are the ordinary data structures. On the contrary, RMI transits objects as a parameter to the remote method.

3. RPC can be considered as the older version of RMI, and it is used in the programming languages that support procedural programming, and it can only use pass by value method. As against, RMI facility is devised based on modern programming approach, which could use pass by value or reference. Another advantage of RMI is that the parameters passed by reference can be changed.

4. RPC protocol generates more overheads than RMI.

5. The parameters passed in RPC must be “in-out” which means that the value passed to the procedure and the output value must have the same datatypes. In contrast, there is no compulsion of passing “in-out” parameters in RMI.

6. In RPC, references could not be probable because the two processes have the distinct address space, but it is possible in case of RMI.



Comparison Chart

BASIS FOR COMPARISON	RPC	RMI
Supports	Procedural programming	Object-oriented programming
Parameters	Ordinary data structures are passed to remote procedures.	Objects are passed to remote methods.
Efficiency	Lower than RMI	More than RPC and supported by modern programming approach (i.e. Object-oriented paradigms)
Overheads	More	Less comparatively
In-out parameters are mandatory.	Yes	Not necessarily
Provision of ease of programming	High	low

<u>Section: Indirect Communication</u>

<u>Q:3</u> In contrast to Direct Communication, which two important properties are present in Indirect Communication?

Answer 3: The technique considered are all based on a direct coupling between a sender and a receiver, and this lead to a certain amount of rigidity in the system in term of dealing with change. To illustrate this, consider a simple client-server interaction. Because of the direct coupling, it is more difficult to replace a server within alternative one offering equivalent functionality. Similarly, if the server fails, this directly affect the client, which must explicitly deal with the failure. In contrast in direct communication avoid this direct coupling and hence inherits interesting properties. The literature refers to two key properties stemming from the use of an intermediary.

1. Space Uncoupling:

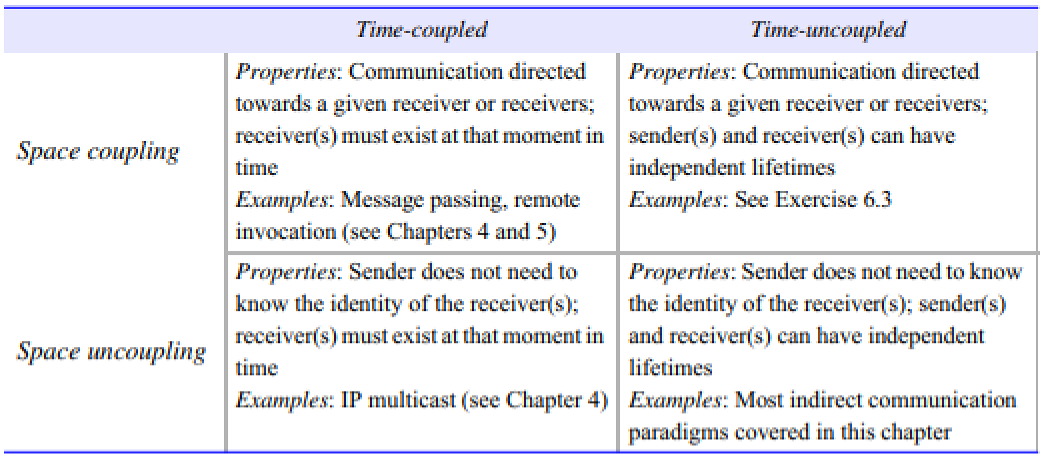
`   Space uncoupling in which the sender does not know or need to know the identity of the receiver and vice versa.Because of this space uncoupling, the system developer has many degree of freedom in dealing with change: participant (senders or receivers) can be replaced, updated, replicated, or migrated.

2. Time Uncoupling:

Time uncoupling in which the sender and receiver can have independent lifetime. In other words the sender and receiver donot need to exist at the same time to communicate. This has important benefits, for example in more volatile environment where sender and receiver may come and go.

For these reasons, indirect communication is often used in distributed systems where change is anticipated – for example, in mobile environments where users may rapidly connect to and disconnect from the global network – and must be managed to provide more dependable services. Indirect communication is also heavily used for event dissemination in distributed systems where the receivers may be unknown and liable to change – for example, in managing event feeds in financial systems.

The discussion below charts the advantages associated with indirect communication. The main disadvantage is that there will inevitably be a performance overhead introduced by the added level of indirection. Indeed, the quote above on indirection is often paired by the following quote, attributable to Jim Gray: There is no performance problem that cannot be solved by eliminating a level of indirection. In addition, systems developed using indirect communication can be more difficult to manage precisely because of the lack of any direct (space or time) coupling



	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 6.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

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

Answer 4: The main Reason of multicast mechanisms versus other possibilities is real time transmission for multiple clients. By using multicast mechanism in these cases, you save an enormous amount of network resources and additionally improve the multicast content transmission.

- Managing group membership. Mean in unicast group membership is easy every node or client can join and leave the group easily.
- Failure detection is easy for a specific group we have define in unicast because if fault occur we can't check or manage the whole network or process we can only detect the error or failure in the specific area mean specific group.
- It save us time because if we use unicast we can send message separately for separate nodes so in multicast it is easy to create a specific group and send the message to all specific nodes hence time is reduced.

Multicast allows an IP network to support more than just the unicast model of data delivery that prevailed in the early stages of the Internet. Multicast, originally defined as a host extension in RFC 1112 in 1989, provides an efficient method for delivering traffic flows that can be characterized as one-to-many or many-to-many.

Unicast traffic is not strictly limited to data applications. Telephone conversations, wireless or not, contain digital audio samples and might contain digital photographs or even video and still flow from a single source to a single destination. In the same way, multicast traffic is not strictly limited to multimedia applications. In some data applications, the flow of traffic is from a single source to many destinations that require the packets, as in a news or stock ticker service delivered to many PCs. For this reason, the term receiver is preferred to listener for multicast destinations, although both terms are common.

If unicast were employed by radio or news ticker services, each radio or PC would have to have a separate traffic session for each listener or viewer at a PC (this is actually the method for some Web-based services). The processing load and bandwidth consumed by the server would increase linearly as more people “tune in” to the server. This is extremely inefficient when dealing with the global scale of the Internet. Unicast places the burden of packet duplication on the server and consumes more and more backbone bandwidth as the number of users grows.

For radio station or news ticker traffic, multicast provides the most efficient and effective outcome, with none of the drawbacks and all of the advantages of the other methods. A single source of multicast packets finds its way to every *interested* receiver. As with broadcast, the transmitting host generates only a single stream of IP packets, so the load remains constant whether there is one receiver or one million. The network routing devices replicate the packets and deliver the packets to the proper receivers, but only the replication role is a new one for routing devices. The links leading to subnets consisting of entirely uninterested receivers carry no multicast traffic. Multicast minimizes the burden placed on sender, network, and receiver..

` **Section: OS Support**

Q5. Differentiate a between a network OS and distributed OS.

Answer 5: Network OS:

A network operating system is a specialized operating system for a network device such as a router, switch or firewall.

A network operating system (NOS) is a PC working framework (OS) that is structured basically to help workstations, PCs and, in certain cases, more established terminals that are associated on a neighborhood (LAN). The product behind a NOS permits different gadgets inside a system to convey and impart assets to one another.

Distributed OS:

A distributed operating system is a software over a collection of independent, networked, communicating, and physically separate computational nodes.

Distributed Operating System is where Distributed applications are running on numerous PCs connected by interchanges. A disseminated working framework is an augmentation of the system working framework that underpins more significant levels of correspondence and mix of the machines on the system.

Difference between Network Operating System and Distributed Operating System:

The principle contrast between organize working framework and distributed working framework is that a system working framework gives network related functionalities while a distributed working framework interfaces various autonomous PCs by means of a system to perform tasks like a single PC.

A working framework(operating system) fills in as the interface between the client and the equipment. It controls the execution of projects and supports an assortment of tasks. It performs record the executives, gadget taking care of, memory the board, making sure about information and assets, controlling the framework execution, processor taking care of and some more. Along these lines, a working framework is a fundamental part of the PC framework. There are different kinds of working frameworks. System working framework and dispersed working framework are two of them.

A network operating system is a special operating system that provides network-based functionalities. A distributed operating system is an operating system that manages a group of distinct computers and makes them appear to be a single computer.

Usage

Network operating system helps to manage data, users, groups, security and other network related functionalities. A distributed operating system helps to share resources and collaborate via a shared network to accomplish tasks.

Examples

Artisoft's LANtastic, Novell's NetWare, and Microsoft's LAN Manager are examples for network operating systems. LOCUS and MICROS are some examples for distributed operating systems.

Conclusion

An operating system is an interface between the user and the hardware. There are various types of operating systems. Two of them are network operating system and distributed operating system. The difference between network operating system and distributed operating system is that a network operating system provides network related functionalities while a distributed operating system connects multiple independent computers via a network to perform tasks similar to a single computer.

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

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

Answer 6: Middleware and network operating systems:

In fact, there are no distributed operating systems in general use just system working frameworks, for example, UNIX, Mac OS and
Windows. This is probably going to remain the case, for two primary reasons. The first is that clients have much put resources into their application programming, which regularly meets their current critical thinking needs; they won't embrace another working framework that won't run their applications, whatever effectiveness preferences it offers. Endeavors have been made to copy UNIX and other working framework bits on new pieces, however the copies exhibition has not been good. Anyway, keeping imitations of all the major working frameworks state-of-the-art as they develop would be an immense endeavor.

a) Process and threads:

A process is series or set of activities that interact to produce a result; it may occur once-only or be recurrent or periodic.
A threat can be either "intentional" (i.e. hacking: an individual cracker or a criminal organization) or "accidental" (e.g. the possibility of a computer malfunctioning, or the possibility of a natural disaster such as an earthquake, a fire, or a tornado) or otherwise a circumstance, capability, action, or event.

A procedure comprises of an execution
condition along with at least one threads. A threadsis the working framework deliberation of an action (the term gets from the expression 'string of execution'). An execution condition is the unit of asset the board: an assortment of nearby kernelmanaged assets to which its strings approach. An execution domain environment primarily consists of:
• an address space;
• thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets);
• higher-level resources such as open files and windows.

Threads can be created and destroyed dynamically, as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors. This can be particularly helpful within servers, where concurrent processing of clients’ requests can reduce the tendency for servers to become bottlenecks. For example, one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

b) **System Virtualization**

The objective of framework(system) virtualization is to give various virtual machines (virtual equipment pictures) over the fundamental physical machine design, with each virtual machine running a different working framework occurrence. The idea comes from the perception that cutting edge PC designs have the fundamental execution to bolster possibly enormous quantities of virtual machines and multiplex assets between them. Various occasions of the equivalent working framework can run on the virtual machines or on the other hand a scope of various working frameworks can be bolstered. The virtualization framework apportions the physical processor(s) and different assets of a physical machine between every single virtual machine that it supports.

To fully understand the motivation for virtualization at the operating system level, it is useful to consider different use cases of the technology:

• On server machines, an organization assigns each service it offers to a virtual machine and then optimally allocates the virtual machines to physical servers. Unlike processes, virtual machines can be migrated quite simply to other physical machines, adding flexibility in managing the server infrastructure. This approach has the potential to reduce investment in server computers and to reduce energy consumption, a key issue for large server farms

. • Virtualization is very relevant to the provision of cloud computing. As described in Chapter 1, cloud computing adopts a model where storage, computation and higher-level objects built over them are offered as a service. The services offered range from low-level aspects such as physical infrastructure (referred to as infrastructure as a service), through software platforms, (platform as a service), to arbitrary applicationlevel services (software as a service). Indeed, the first is enabled directly by virtualization, allowing users of the cloud to be provided with one or more virtual machines for their own use.

• The developers of virtualization solutions are also motivated by the need for distributed applications to create and destroy virtual machines readily and with little overhead. This is required in applications that may need to demand resources dynamically, such as multiplayer online games or distributed multimedia applications. Support for such applications can be enhanced by adopting appropriate resource allocation policies to meet quality of service requirements of virtual machines.

• A quite different case arises in providing convenient access to several different operating system environments on a single desktop computer. Virtualization can be used to provide multiple operating system types on one physical architecture. For example, on a Macintosh OS X computer, the Parallels Desktop virtual machine monitor enables a Windows or a Linux system to be installed and to coexist with OS X, sharing the underlying physical resources.

Section: Distributed Objects and Components

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

Answer 7: Issues with Object (distributed) oriented middlewares:

Implicit dependencies:

Object interfaces do not describe what the implementation of an object depends on, making object-based systems difficult to develop (especially for third-party developers) and subsequently manage.

Programming complexity:

Programming distributed object middleware leads to a need to master many low-level details associated with middleware implementations.

Lack of separation of distribution concerns:

Application developers are obliged to consider details of concerns such as security, failure handling and concurrency, which are largely similar from one application to another.

Component-based solutions can best be understood as a natural evolution of objectbased approaches, building on the strong heritage of this earlier work. this rationale in more detail and introduces the key features of a componentbased approach. T presents two contrasting case studies of componentbased solutions, Enterprise JavaBeans and Fractal, with the former offering a comprehensive solution that abstracts over many of the key issues in developing distributed applications and the latter representing a more lightweight solution often used to construct more complex middleware technologies.

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.

Benefits of DOC Middleware:

Middleware in general–and DOC middleware in particular–provides essential capabilities for developing distributed applications. In this section we summarize its improvements over traditional non-middleware oriented approaches, using the challenges and opportunities described in Section 1 as a guide:

Growing focus on integration rather than on programming:

This visible shift in focus is perhaps the major accomplishment of currently deployed middleware. Middleware originated because the problems relating to integration and construction by composing parts were not being met by either

1. Applications, which at best were customized for a single use,

2. Networks, which were necessarily concerned with providing the communication layer, or

3. Host operating systems, which were focused primarily on a single, self-contained unit of resources.

Demand for end-to-end QoS support, not just component QoS:

This area represents the next great wave of evolution for advanced DOC middleware. There is now widespread recognition that effective development of large-scale distributed applications requires the use of COTS infrastructure and service components. Moreover, the usability of the resulting products depends heavily on the properties of the whole as derived from its parts. This type of environment requires visible, predictable, flexible, and integrated resource management strategies within and between the pieces.

The increased viability of open systems:

The increased viability of open systems architectures and open-source availability – By their very nature, systems developed by composing separate components are more open than systems conceived and developed as monolithic entities. The focus on interfaces for integrating and controlling the component parts leads naturally to standard interfaces. In turn, this yields the potential for multiple choices for component implementations, and open engineering concepts. Standards organizations such as the OMG and The Open Group have fostered the cooperative efforts needed to bring together groups of users and vendors to define domain-specific functionality that overlays open integrating architectures, forming a basis for industry-wide use of some software components. Once a common, open structure exists, it becomes feasible for a wide variety of participants to contribute to the off-the-shelf availability of additional parts needed to construct complete systems. Since few companies today can afford significant investments in internally funded R&D, it is increasingly important for the information technology industry to leverage externally funded R&D sources, such as government investment. In this context, standards-based DOC middleware serves as a common platform to help concentrate the results of R&D efforts and ensure smooth transition conduits from research groups into production systems.

Increased leverage for disruptive technologies leading to increased global competition:

Middleware supporting component integration and reuse is a key technology to help amortize software life-cycle costs by:

1. Leveraging previous development expertise, e.g., DOC middleware helps to abstract commonly reused low-level OS concurrency and networking details away into higher-level, more easily used artifacts and

2. Focusing on efforts to improve software quality and performance, e.g., DOC middleware combines various aspects of a larger solution together, e.g., fault tolerance for domain-specific objects with real-time QoS properties.

Potential complexity cap for next-generation complex systems:

As today's technology transitions run their course, the systemic reduction in long-term R&D activities runs the risk of limiting the complexity of next-generation systems that can be developed and integrated using COTS hardware and software components. The advent of open DOC middleware standards, such as CORBA and Java-based technologies, is hastening industry consolidation towards portable and interoperable sets of COTS products that are readily available for purchase or open-source acquisition. These products are still deficient and/or immature, however, in their ability to handle some of the most important attributes needed to support future systems. Key attributes include end-to-end QoS, dynamic property tradeoffs, extreme scaling (large and small), highly mobile environments, and a variety of other inherent complexities. Complicating this situation over the past decade has been the steady flow of faculty, staff, and graduate students out of universities and research labs and into startup companies and other industrial positions. While this migration helped fuel the global economic boom in the late '90s, it does not bode well for long-term technology innovation.

An increasing number of next-generation applications will be developed as distributed "systems of systems," which include many interdependent levels, such as network/bus interconnects, local and remote endsystems, and multiple layers of common and domain-specific middleware. The desirable properties of these systems of systems include predictability, controllability, and adaptability of operating characteristics for applications with respect to such features as time, quantity of information, accuracy, confidence, and synchronization. All these issues become highly volatile in systems of systems, due to the dynamic interplay of the many interconnected parts. These parts are often constructed in a similar way from smaller parts. To address the many competing design forces and runtime QoS demands, a comprehensive methodology and environment is required to dependably compose large, complex, interoperable DOC applications from reusable components. Moreover, the components themselves must be sensitive to the environments in which they are packaged. Ultimately, what is desired is to take components that are built independently by different organizations at different times and assemble them to create a complete system. In the longer run, this complete system becomes a component embedded in still larger systems of systems. Given the complexity of this undertaking, various tools and techniques are needed to configure and reconfigure these systems hierarchically so they can adapt to a wider variety of situations. An essential part of what is needed to build the type of systems outlined above is the integration and extension of ideas that have been found traditionally in network management, data management, distributed operating systems, and object-oriented programming languages. The payoff will be reusable DOC middleware that significantly simplifies the building of applications for systems of systems environments. The following points of emphasis are embedded within that challenge to achieve the payoff: • Toward more universal use of common middleware.Today, it is too often the case that a substantial percentage of the effort expended to develop applications goes into building ad hoc and proprietary middleware substitutes, or additions for missing middleware functionality. As a result, subsequent composition of these ad hoc capabilities is either infeasible or prohibitively expensive. One reason why redevelopment persists is that it is still often relatively easy to pull together a minimalist ad hoc solution, which remains largely invisible to all except the developers. Unfortunately, this approach can yield substantial recurring downstream costs, particularly for complex and long-lived distributed systems of systems. Part of the answer to these problems involves educating developers about software lifecycle issues beyond simply interconnecting individual components. In addition, there are many different operating environments, so providing a complete support base for each takes time, funding, and a substantial user community to focus the investment and effort needed. To avoid this often repeated and often wasted effort, more of these "completion" capabilities must be available off-the-shelf, through middleware, operating systems, and other common services. Ideally, these COTS capabilities can eliminate, or at least minimize, the necessity for significant custom augmentation to available packages just to get started. Moreover, we must remove the remaining impediments associated with integrating and interoperating among systems composed from heterogeneous components. Much progress has been made in this area, although at the host infrastructure middleware level more needs to be done to shield developers and end-users from the accidental complexities of heterogeneous platforms and environments.

• Common solutions handling both variability and control – It is important to avoid "all or nothing" point solutions. Systems today often work well as long as they receive all the resources for which they were designed in a timely fashion, but fail completely under the slightest anomaly. There is little flexibility in their behavior, i.e., most of the adaptation is pushed to end-users or administrators. Instead of hard failure or indefinite waiting, what is required is either reconfiguration to reacquire the needed resources automatically or graceful degradation if they are not available. Reconfiguration and operating under less than optimal conditions both have two points of focus: individual and aggregate behavior. Moreover, there is a need for interoperability of control and management mechanisms. As with the adoption of middleware-based packages and shielding applications from heterogeneity, there has been much progress in the area of interoperability. Thus far, however, interoperability concerns have focused on data interoperability and invocation interoperability. Little work has focused on mechanisms for controlling the overall behavior of integrated systems, which is needed to provide "control interoperability." There are requirements for interoperable control capabilities to appear in individual resources first, after which approaches can be developed to aggregate these into acceptable global behavior. Before outlining the research that will enable these capabilities, it is important to understand the role and goals of middleware within an entire system. The middleware resides between applications and the underlying OS, networks, and computing hardware. As such, one of its most immediate goals is to augment those interfaces with QoS attributes. It is important to have a clear understanding of the QoS information so that it becomes possible to:

1. Identify the users' requirements at any particular point in time and

2. Understand whether or not these requirements are being (or even can be) met. It is also essential to aggregate these requirements, making it possible to form decisions, policies, and mechanisms that begin to address a more global information management organization. Meeting these requirements will require flexibility on the parts of both the application components and the resource management strategies used across heterogeneous systems of systems.

A key direction for addressing these needs is through the concepts associated with managing adaptive behavior, recognizing that not all requirements can be met all of the time, yet still ensuring predictable and controllable end-to-end behavior. Within these general goals, there are pragmatic considerations, including incorporating the interfaces to various building blocks that are already in place for the networks, operating systems, security, and data management infrastructure, all of which continue to evolve independently.

Ultimately, there are two different types of resources that must be considered:

1. Those that will be fabricated as part of application development and

2. Those that are provided and can be considered part of the substrate currently available.

While not much can be done in the short-term to change the direction of the hardware and software substrate that's installed today, a reasonable approach is to provide the needed services at higher levels of (middleware-based) abstraction. This architecture will enable new components to have properties that can be more easily included into the controllable applications and integrated with each other, leaving less lower-level complexity for application developers to address and thereby reducing system development and ownership costs. Consequently, the goal of next-generation middleware is not simply to build a better network or better security in isolation, but rather to pull these capabilities together and deliver them to applications in ways that enable them to realize this model of adaptive behavior with tradeoffs between the various QoS attributes. As the evolution of the underlying system components change to become more controllable, we can expect a refactoring of the implementations underlying the enforcement of adaptive control.

The End