Chapter 17 – Introduction to Distributed Systems

Outline
17.1 Introduction
17.2 Attributes of Distributed Systems
17.2.1 Performance and Scalability
17.2.2 Connectivity and Security
17.2.3 Reliability and Fault Tolerance
17.2.4 Transparency
17.2.5 Network Operating Systems
17.2.6 Distributed Operating Systems
17.3 Communication in Distributed Systems
17.3.1 Middleware
17.3.2 Remote Procedure Call (RPC)
17.3.3 Remote Method Invocation (RMI)
17.3.4 CORBA (Common Object Request Broker Architecture)
17.3.5 DCOM (Distributed Component Object Model)
17.3.6 Process Migration in Distributed Systems
17.4 Synchronization in Distributed Systems
17.5 Mutual Exclusion in Distributed Systems
17.5.1 Mutual Exclusion without Shared Memory

Outline (continued)
17.5.2 Agrawala and Ricart’s Distributed Mutual Exclusion Algorithm
17.6 Deadlock in Distributed Systems
17.6.1 Distributed Deadlock
17.6.2 Deadlock Prevention
17.6.3 Deadlock Detection
17.6.4 A Distributed Resource Deadlock Algorithm
17.7 Case Study: The Sprite Distributed Operating System
17.8 Case Study: The Amoeba Distributed Operating System
Objectives

• After reading this chapter, you should understand:
  – the need for distributed computing.
  – fundamental properties and desirable characteristics of distributed systems.
  – remote communication in distributed systems.
  – synchronization, mutual exclusion and deadlock in distributed systems.
  – examples of distributed operating systems.

17.1 Introduction

• Distributed systems
  – Remote computers cooperate via a network to appear as a local machine
  – Users are given the impression that they are interacting with just one machine
  – Spread computation and storage throughout a network of computers
  – Applications are able to execute code on local machines and remote machines and to share data, files and other resources among these machines
17.2 Attributes of Distributed Systems

- Importance of distributed systems has been stressed for decades
- Explosion of the Internet has made distributed systems common
- Attributes of distributed systems:
  - Performance
  - Scalability
  - Connectivity
  - Security
  - Reliability
  - Fault tolerance

17.2.1 Performance and Scalability

- Centralized system
  - A single server handles all user requests
- Distributed system
  - User requests can be sent to different servers working in parallel to increase performance
- Scalability
  - Allows a distributed system to grow (i.e., add more machines to the system) without affecting the existing applications and users
17.2.2 Connectivity and Security

- **Distributed systems**
  - Susceptible to attacks by malicious users if they rely on insecure communications media

- **To improve security:**
  - Allow only authorized users to access resources
  - Ensure that information transmitted over the network is readable only by the intended recipients
  - Provide mechanisms to protect resources from attack

17.2.3 Reliability and Fault Tolerance

- **Fault tolerance**
  - Implemented by providing replication of resources across the system

- **Replication**
  - Offers users increased reliability and availability over single-machine implementations
  - Designers must provide mechanisms to ensure consistency among the state information at different machines
17.2.4 Transparency

- **Access transparency**
  - Hides the details of networking protocols that enable communication between distributed computers

- **Location transparency**
  - Builds on access transparency to hide the location of resources in the distributed system

- **Failure transparency**
  - Method by which a distributed system provides fault tolerance
    - Checkpointing
      - Periodically stores the state of an object such that it can be restored if a failure in the distributed system results in the loss of the object
    - Replication
      - A system provides multiple resources that perform the same function

- **Replication transparency**
  - Hides the fact that multiple copies of a resource are available in the system

- **Persistence transparency**
  - Hides the information about where the resource is stored—memory or disk

- **Migration and relocation transparency**
  - Hide the movement of components of a distributed system
    - Migration transparency
      - Masks the movement of an object from one location to another in the system
    - Relocation transparency
      - Masks the relocation of an object from other objects that communicate with it

- **Transaction transparency**
  - Allows a system to achieve consistency by masking the coordination among a set of resources
  - Hides the implementation of checkpointing and other consistency mechanisms
17.2.5 Network Operating Systems

- **Network OS**
  - Accesses resources on remote computers that run independent operating systems
  - Not responsible for resource management at remote locations
  - Distributed functions are explicit rather than transparent
    - A user or process must explicitly specify the resource’s location to retrieve a networked file or remotely execute an application
  - Lack of transparency in network OSs
    - Disadvantage: Does not provide some of the benefits of distributed OSs
    - Advantage: Easier to implement than distributed OSs

17.2.6 Distributed Operating Systems

- **Distributed OSs**
  - Manage resources located in multiple networked computers
  - Employ many of the same communication methods, file system structures and other protocols found in network operating systems
  - Transparent communication
    - Objects in the system are unaware of the separate computers that provide the service (unlike network operating systems)
  - Rare to find a “truly” distributed system because the high level of transparency is difficult to achieve
17.3 Communication in Distributed Systems

- Designers must establish interoperability between heterogeneous computers and applications
- Interoperability
  - Permits software components to interact among different
    - hardware and software platforms
    - programming languages
    - communication protocols
- Standardized interface
  - Allows each client/server pair to communicate using a single, common interface that is understood by both sides

17.3.1 Middleware

- Software in distributed systems helps provide:
  - Portability
    - Enables the movement of a system or component from one environment (including both hardware and software) to another without changing the system or component being moved
  - Transparency
  - Interoperability
- Provides standard programming interfaces to enable interprocess communication between remote computers
17.3.2 Remote Procedure Call (RPC)

- RPC
  - Allows a process executing on one computer to invoke a procedure in a process executing on another computer
  - Goal of RPC
    - To simplify the process of writing distributed applications by preserving the syntax of a local procedure call while transparently initiating network communication

17.3.2 Remote Procedure Call (RPC)

- To issue an RPC:
  - Client process makes a call to the procedure in the client stub
  - Client stub performs marshaling of data to package procedure arguments along with the procedure name into a message for transmission over a network
  - Client stub passes the message to the server
  - Server transmits the message to the server stub
17.3.2 Remote Procedure Call (RPC)

- To issue an RPC (Cont.):
  - Message is unmarshaled
  - Stub sends the parameters to the appropriate local procedure
  - When the procedure has completed, the server stub marshals the result and sends it back to the client
  - Finally, the client stub unmarshals the result, notifies the process and passes it the result
17.3.3 Remote Method Invocation (RMI)

- **RMI**
  - Enables a Java process executing on one computer to invoke a method of an object on a remote computer using the same syntax as a local method call
  - Similar to RPC, the details of parameter marshaling and message transport in RMI are transparent to the calling program
  - The stub/skeleton layer of RMI contains parameter-marshaling structures analogous to the client and server stubs of RPC
  - The stub employs object serialization
    - Enables programs to pass Java objects as parameters and receive objects as return values

- **RMI (cont.)**
  - The remote reference layer (RRL) and the transport layer of RMI work together to send the marshaled message between the client and the server
  - The skeleton unmarshals the parameters, identifies the object on which the method is to be invoked and calls that method
  - Upon completion of the method, the skeleton marshals the result and returns it to the client via the RRL and stub
17.3.4 CORBA (Common Object Request Broker Architecture)

- CORBA
  - Open standard designed to enable interoperation among programs in heterogeneous as well as homogeneous systems
  - Supports objects as parameters or return values in remote procedures during interprocess communication

- CORBA implementation
  - The process on the client passes the procedure call along with the required arguments to the client stub
  - The client stub marshals the parameters and sends the procedure call through its Object Request Broker (ORB), which communicates with the ORB on the server
  - CORBA provides programmers language independence with the Interface Definition Language (IDL), which allows them to strictly define the procedures that can be called on the object
17.3.5 DCOM (Distributed Component Object Model)

- DCOM
  - Designed to allow software components residing on remote computers to interact with one another
  - As in CORBA, objects in DCOM are accessed via interfaces
  - Unlike CORBA, however, DCOM objects may have multiple interfaces
  - When a client requests a DCOM object from a server, the client must also request a specific interface of the object

17.3.6 Process Migration in Distributed Systems

- Process migration
  - Transfers a process between two computers in a distributed system
  - Allows processes to exploit a remote resource
  - A complicated task that often reduces the performance of the process that is being migrated

- Process cloning
  - Similar to process migration
  - Instead of transferring a process to a remote location, a new copy of the process is created on the remote machine
17.4 Synchronization in Distributed Systems

- Determining the order in which events occur is difficult
  - Communication delays in a distributed network are unpredictable
- Causal ordering
  - Ensures that all processes recognize that a causally dependent event must occur only after the event on which it is dependent
  - Implemented by the happens-before relation:
    - If events $a$ and $b$ belong to the same process, then $a \rightarrow b$ if $a$ occurred before $b$
    - If event $a$ is the sending of a message and event $b$ is the receiving of that message, then $a \rightarrow b$
    - This relation is transitive
  - Only a partial ordering
    - Events for which it cannot be determined which occurred earlier are said to be concurrent

17.4 Synchronization in Distributed Systems

- Total ordering
  - Ensures that all events are ordered and that causality is preserved
  - Can be implemented through a logical clock that assigns a timestamp to each event that occurs in the system
  - Scalar logical clocks synchronize the logical clocks on remote hosts and ensure causality
17.5 Mutual Exclusion in Distributed Systems

- Various synchronization methods implemented to enforce mutual exclusion in distributed systems
  - Message passing
  - Agrawala and Ricart’s distributed mutual exclusion algorithm

17.5.1 Mutual Exclusion without Shared Memory

- In environments with no shared memory, mutual exclusion must be implemented via message passing
- To synchronize the system, message-passing systems use clock synchronization concepts to employ:
  - FIFO broadcast
    - Guarantees that when two messages are sent from one process to another, the message that was sent first will arrive first
  - Causal broadcast
    - Ensures that when message \( m_1 \) is causally dependent on message \( m_2 \), then no process delivers \( m_1 \) before delivering \( m_2 \)
  - Atomic broadcast
    - Guarantees that all messages in a system are received in the same order at each process
17.5.2 Agrawala and Ricart’s Distributed Mutual Exclusion Algorithm

- Before a process can enter its critical section:
  - The process first must send a request message to all other processes in the system
  - The process must receive a response from each of these processes
- When a process receives a request to enter a critical section and has not sent a request of its own, it sends a reply

17.5.2 Agrawala and Ricart’s Distributed Mutual Exclusion Algorithm

- If the process has sent its own request, it compares the timestamps of the two requests
  - If the process’s own request has a later timestamp than the other request, it sends a reply
  - If the process’s own request has an earlier timestamp than the other request, it delays its reply
  - Finally, if the timestamps of the requests are equal, the process compares its process number to that of the requesting process
    - If its own number is higher, it sends a response, otherwise it delays its response
17.6 Deadlock in Distributed Systems

- Distributed deadlock
  - Occurs when processes spread over different computers in a network wait for events that will not occur

17.6.1 Distributed Deadlock

- Three types of distributed deadlock:
  - Resource deadlock
    - As discussed in Chapter 7
  - Communication deadlock
    - Circular waiting for communication signals
  - Phantom deadlock
    - Due to the communications delay associated with distributed computing, it is possible that a deadlock detection algorithm might detect a deadlock (called phantom deadlock, a perceived deadlock) that does not exist
    - Although this form of deadlock cannot immediately cause the system to fail, it is a source of inefficiency
17.6.2 Deadlock Prevention

- Two algorithms designed to prevent deadlock
  - Rely on ordering processes based on when each process was started
  - Wound-wait strategy
    - Breaks deadlock by denying the no-preemption condition
    - A process will wait for another process if the first process was created before the other
    - A process will wound (restart) another process if the first process was created after the other
  - Wait-die strategy
    - Breaks deadlock by denying the wait-for condition
    - A process will wait for another process if the first process was created after the other process
    - A process will die (restart) itself if it was created before the other process

Figure 17.2 Wound-wait strategy.
17.6.2 Deadlock Prevention

**Figure 17.3** Wait-die strategy.

17.6.3 Deadlock Detection

- **Central deadlock detection**
  - Entire system monitored by one dedicated site
  - Whenever a process requests or releases a resource it informs the central site, which continuously checks the system for cycles
  - DDAs for central detection are simple to implement and are efficient for LANs
  - Disadvantages:
    - The system may experience decreased performance (the central site becomes a bottleneck)
    - Not fault tolerant—the central site becomes the system’s single point of failure
17.6.3 Deadlock Detection

• Hierarchical deadlock detection
  – Arranges each site in the system as a node in a tree
  – Each node, except a leaf node, collects the resource allocation information for its children
  – Tree structure helps to improve fault tolerance
  – More efficient
    • Because deadlock detection is divided into hierarchies and clusters, sites that do not introduce the possibility of deadlock for a resource do not have to participate in deadlock detection for that resource

• Distributed deadlock detection
  – Places the responsibility of deadlock detection with each site
  – Each site in the system queries all other sites to determine whether any other sites are involved in deadlock
  – This is the most fault-tolerant method of deadlock detection
    • Failure of one site will not cause any other site to fail
17.6.4 A Distributed Resource Deadlock Algorithm

- Johnston, et al.’s simple algorithm for deadlock detection in distributed systems:
  - Each process keeps track of the transaction wait-for graph (TWFG) of which they are involved
  - When a process requests a resource that is being held by another process, the requesting process blocks and the TWFG is updated
  - As this happens, any deadlocks are detected and removed
17.6.4 A Distributed Resource Deadlock Algorithm

Figure 17.5 Deadlock is introduced to the system.

Figure 17.6 System after deadlock has been eliminated.
17.7 Case Study: The Sprite Distributed Operating System

• Sprite
  – Large numbers of personal workstations are connected and many computers could be idle at any given time
  – Idle workstations allow Sprite to use process migration to balance the workload of the system
  – When the central migration server is notified that a workstation is idle, it will migrate a process to that target computer
  – When the user of the target computer returns, the workstation notifies the central migration server about the return, and the process is migrated back to the home computer

• Sprite (cont.)
  – The Sprite kernel provides more location-independent calls by providing the exact same view of the file system for each workstation
  – When a location-dependent call is required:
    • The system either forwards the call to the home computer for evaluation
    or
    • The system transfers the process’s state information from the home computer to the target computer
  – The Sprite file system also caches files on both the server and client sides
17.8 Case Study: The Amoeba Distributed Operating System

• Amoeba:
  – Users share processors located in one or more processor pools
  – When a user issues a command to execute a process, the processor pool dynamically allocates the processors for the user
  – When the user process terminates, the user returns the allocated processors to the processor pool

• Amoeba (cont.)
  – Provides transparency by hiding the number and location of processors from the user
  – Amoeba supports two forms of communication:
    • Point-to-point communication
      – A client stub sends a request message to the server stub and blocks, awaiting the server reply
    • Group communication
      – Messages are sent to all receivers in exactly the same order
17.8 Case Study: The Amoeba Distributed Operating System

• The Amoeba file system
  – Standard file server called the bullet server which has a large primary memory
  – The files stored in the bullet server are immutable
  – If a file is modified, a new file is created to replace the old one, and the old one is deleted from the server
  – The bullet server also stores files contiguously on the disk so that it can transfer files faster than Sprite