Distributed Systems

Lectured by Emil C Lupu

Typed by Aris Zhu Yi Qing

April 18, 2022

Contents

1 Characteristics

1.1 Distribution Transparencies

Realize a coherent system by *hiding distribution* from the user where possible.

- **Access**: uniform access whether local or remote
- **Location**: access without knowledge of location
- **Concurrency**: sharing without interference (requires synchronization)
- **Replication**: hides use of redundancy (e.g. for fault tolerance)
- **Failure**: conceal failures by replication or recovery
- **Migration**: hides migration of components (e.g. for load balancing)
- **Performance**: hide performance variations (e.g. through use of scheduling and reconfiguration)
- **Scaling**: permits expansion by adding more resources (e.g. cloud)

1.2 Challenges

- **Heterogeneity**: different OS, data representation, implementations, etc.
- **Openness**: need to define *interfaces* for components to easily scale up systems
- **Security**: control access to preserve integrity and confidentiality
- **Concurrency**: inconsistencies may arise with interleaving requests
- **Failure handling**: transient/permanent failures could occur at any time. It is difficult detect them and to maintain consistency.
- **Scalability**: size of the system makes it difficult to maintain information about *system state*.

1.3 Wrong Assumptions

- The network is reliable, secure & homogeneous.
- The topology does not change.
- The latency is zero.
- The bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

1.4 Terminology

- **Client**: an entity initiating an interation
- **Server**: a componenet responds to interactions usually implemented as a process
- **Service**: a componenet of a computer system that manages a collection of resources and presents their functionality to users.
- **Middleware**: software layer between the application and the OS masking the heterogeneity of the underlying system.

2 Architecture

2.1 Layered architecture

- e.g. Network stack. Control flows downwards, results flow upwards.

+ framework is simple and easy to learn and implement
+ reduced dependency due to layer separation
+ testing is easier with such modularity
+ cost overheads are fairly low
- scalability is difficult due to fixed framework structure
- difficult to maintain, since a change in a single layer can affect the entire system because it operates as a single unit
- parallel processing is not possible

2.2 Object-based and service-oriented architectures

- e.g. RMI

+ reusability, easy maintainability and greater reliability due to modularity
+ improved scalability and availability: multiple instances of a single service can run on different servers at the same time.
- Increased overhead: Service interactions require validations of inputs, thereby increasing the response time and machine load, and reducing the overall performance

2.3 Message-based architectures

	Temporally coupled	Temporally decoupoled
Referentially coupled	Direct process messaging	Messaging via mailbox
Referentially decoupled	Event-based (publish-subscribe)	Shared data spaces

- *Referentially coupled*: processes name sender/receiver in their communication.
- *Temporally coupled*: both sender and receiver need to be up and running.

2.4 Peer-to-peer

- structured: Each node is indexed so that the location is known, and messages are routed according to the topology.
- unstructured: *flooding* or *random walks* or both.

+ no server needed since individual workstations are used to access files
+ resilient to computer failures, since it does not disrupt any other part of the network
+ very scalable
- poor performance with larger networks since each computer is being accessed by other users
- no central file system, hard to look up or backup
- ensuring that viruses are not introduced into the network is the responsibility of each individual user.
- There is no security other than assigning permissions.

3 Message-passing and IPC

- Asynchronous send: sender continues its execution once the message has been copied out of its address space
  - + mostly used with *blocked receive*
  - + underlying system must provide buffering for receiving messages independently of receiver processes
  - + *loose* coupling: sender does not know when message will be received, does not suspend execution until the message has been received
  - - *Buffer exhaustion* (no flow control)
  - - formal verification is more difficult, as need to account for the state of the buffers
- Synchronous send: *blocked send*, where the sender is held up until actual receipt of the message by the destination.
  - + usually used with blocking receive, where receiver execution is suspended until a message is received.
  - + synchronization between sender and receiver
  - + generallyl easier to formally reason about synchronous systems
  - - what if no receivers? message loss?
  - - No multi-destination, requiring synchronization with all receivers.
  - - implementation more complicated
  - - The underlying communication service is expected to be *reliable*, i.e. to guarantee in order message delivery.
- Asynchronous receive: process continues execution if there are no messages. hardly provided as primitives
- Blocked receive: the destination process blocks if no message is available, and receives it into a target variable when available.
- Please check the coursework for how UDP client/server is implemented in Java, e.g. how datagram, socket, port are used.