Distributed Systems – Theory
2. Design and middleware

Loosely coupled vs. Tightly coupled softw.

Loosely coupled:

- Allows machines & users of DS to be independent one another
- Examples:
  1. PCs sharing some resources, s.a. printers or databases, over a LAN
  2. Network operating systems: shared file system, each computer its OS, obey user requests

Tightly coupled:

- Single time-sharing system
- Users are not aware of the existence of multiple CPUs in the system
- Examples:
  1. Cluster systems

Process management & file system in DS

- Single, global interprocess communication mechanism so that any process can talk to any other process
- How processes are created, destroyed, started, and stopped must not vary from machine to machine.
- File system must look the same everywhere (global file system)
- Same system call interface everywhere => (?) identical kernels run on all the CPUs in the system
- When a process has to be started up, all the kernels have to cooperate in finding the best place to execute it

Design issues: 1. Transparency

- DS: a set of cooperating processes
- Complexity resulting from the distribution should de made transparent (i.e., invisible) to the applications programmer
- A time-sharing system that achieve single-system image is said to be transparent
- Example:

1. User of make in Unix to compile a large no. of file does not need to know if all the compilators are proceeding in parallel on different machines
2. Read remote files in the manner as the local ones

Transparency types

Kind	*Meaning*
Location transparency	The users cannot tell where resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	The user cannot tell how many copies exist
Concurrency transparency	Multiple users can share resources automatically
Parallelism transparency	Activities can happen in parallel without users knowing
Access transparency	Identical ways in which access takes place to local/remote components
Failure transparency	Users are unaware of a failure of a component.
Technology transparency	Different technologies, such as programming languages and operating systems, are hidden from the user.

Design issues: 2. Flexibility

- Usage of monolithic or micro-kernels systems?
- Monolithic usually: a centralized system + network facilities
- Micro-kernels are more flexible because it does almost nothing

Design issues: 3. Reliability

- One of the DS goals: make DS more reliable than single-processor systems
- E.g. : one machine goes down, some other takes over the job
- Boolean OR of the components reliability: one machine has probability 95% up, possibility that all four ones of a DS are down: (5%)^4=0.0006 %
- Counter-say: Lamport “definition” a DS is a system “on which I cannot get any work done because some machine I have never heard of has crashed."

Aspects of reliability

- Availability: fraction of time that the system is usable – can be enhanced by:
  - a design that does not require simultaneous functioning of a substantial no. of critical components
  - Redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up
    - More copies -> better availability -> greater the chance for the copies to be inconsistent
- Security – resources protected from unauthorized usage (more severe in DS than in single proc. sys)
- Fault tolerance
  - e.g. effects of server crashing and quick rebooting
  - DS must designed to mask failures: hide them from the user

Design issues: 4. Performance

- Running an application on a DS it should not be appreciably worse than running the same application on a single processor
- Metrics:
  - Response time
  - *Throughput* (number of jobs per hour),
  - system utilization,
  - amount of network capacity consumed.
- Performance problem: communication is typically quite slow -> optimize performance by minimizing the no. messages.
- ? Best way: gain performance is many activities running in parallel on different processors, but this requires sending many messages
  - Starting up a small computation remotely, such as adding two integers, is rarely worth it
  - Starting up a long compute-bound job remotely may be worth the trouble

See Parallel Computing lecture from second semester!

Design issues: 5. Scalability

- Usual DS: few hundred of CPUs.
- Grids: several thousands
- Electronic reservation soon tens of millions
- The problem: solution that work well for 200 machines will fail miserably for 200,000,000
- Guiding principle:
  - Avoid centralized components
    - Counter-example: a single mail server for 50 million users (non-fault tolerant, network bootleneck etc)
  - Avoid centralized tables
    - Counter-example: a single data-base keep track of the telephone numbers and addresses of 50 million people
  - Avoid centralized algorithms
    - Example: a large distributed system, an enormous no. of messages have to be routed over many lines.
      - Bad way: optimal way to do this is collect complete information (to one server) about the load on all machines and lines, and then run a graph theory algorithm (on one server) to compute all the optimal routes; then spread info around the system to improve the routing.

Decentralized algorithms - characteristics

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists
    - Counter-example: Any algorithm that starts out with "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized.

Middleware

- offers general services that support distributed execution of applications
- it is software positioned between the operating system and the application
- a “tablecloth” that spreads itself over a heterogeneous network, concealing the complexity of the underlying technology from the application being run on it.

Middleware Tasks in case of a OO appl. (1)

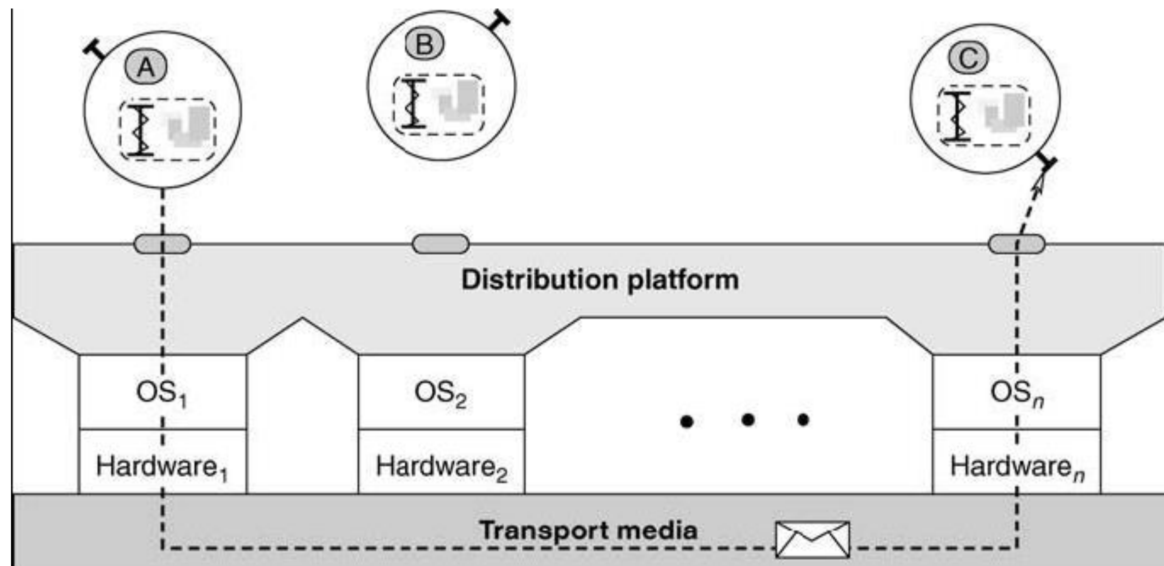
- Object:
  - encapsulates state and behavior and can only be accessed via a well-defined interface.
  - interface hides the details that are specific to the implementation, thereby helping to encapsulate different technologies.
  - a unit of distribution
  - communicate with each other by exchanging messages.

Middleware Tasks in case of a OO appl. (2)

- Object model support:
  - Middleware should offer mechanisms to support the concepts incorporated in the object model.
- Operational interaction:
  - Middleware should allow the operational interaction between two objects. The model used is the method invocation of an object-oriented programming language.
- Remote interaction:
  - Middleware should allow the interaction between two objects located in different address spaces.
- Distribution transparency:
  - From the standpoint of the program, interaction between objects is identical for both local and remote interactions.
- Technological independence:
  - The middleware supports the integration of different technologies.

Structure of a Middleware Platform

- middleware is conceptually located between the application and OS
- the application is represented as a set of interacting objects
- each object is explicitly allocated to a hardware platform



Middleware hides the heterogeneity

Heterogeneity exists at different places:

- Programming languages:
  - Different objects can be developed in different programming languages.
- Operating system:
  - Operating systems have different characteristics and capabilities.
- Computer architectures:
  - Computers differ in their technical details (e.g., data representations).
- Networks:
  - Different computers are linked together through different network technologies.

How middleware deals with heterogeneity

- Offer equal functionality at all access points:
  - applications have access to its functionality through an API; APIs are adapted to the conditions of each programming language that is supported by the middleware.
- An applications programmer typically sees middleware as a program library and a set of tools.
  - depends on the development environment that the programmer is using.
  - affected also by the actual compiler/interpreter used to develop a distributed application.

Standardization of a Middleware

- When project a middleware to a global, worldwide network, we would find special characteristics that differ from those of a geographically restricted DS.
  - middleware spans several technological and political domains,
  - it can no longer be assumed that a homogenous technology exists within a DS
- Cannot assume that one vendor alone is able to supply middleware in the form of products for all environments
  - avoid the monopoly!
  - Innovation through competition
  - implementation of middleware through several competing products can result in partial solutions that are not compatible.
- Compatibility is only possible if all vendors of middleware adhere to a standard

Standard

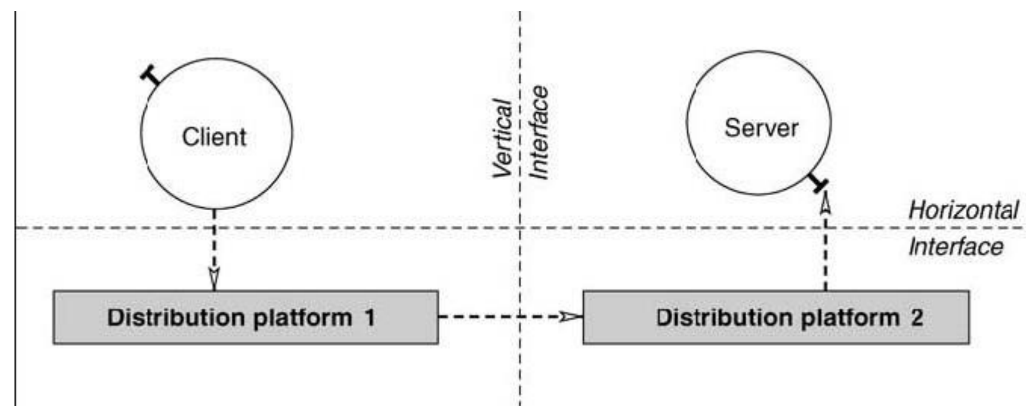
- Stipulate the *specification* for a product—
  - an abstract description of a desired behavior that allows a degree of freedom in the execution of an implementation.
- Serves as a blueprint according to which different products (i.e., implementations) can be produced.
- A specification identifies the verifiable characteristics of a system.
  - if a system conforms to a standard, it is fulfilling these characteristics.
- Advantages:
  - guarantees vendor independence, thereby enabling a customer to select from a range of products without having to commit to one particular vendor.
  - protection of the user investment in order to use a product.

Characteristics of open standards

- Nonproprietary:
  - The standard itself is not subject to any commercial interests
- Freely available:
  - Access to the standard is available to everyone
- Technology independent:
  - The standard represents an abstraction of concrete technical mechanisms and only defines a system to the extent that is necessary for compatibility between products
- Democratic creation process:
  - The creation and subsequent evolution of the standard is not ruled by the dominance of one company but takes place through democratic processes
- Product availability:
  - A standard is only effective if products exist for it.

Interfaces between DS components

- In the context of middleware, a standard has to establish the interfaces between different components to enable their interaction with one another.
- Two types of interface: horizontal and vertical



Horizontal interface / API / Portability

- exists between an application and the middleware
- defines how an application can access the functionality of the middleware
- is also referred to as an *Application Programming Interface* (API)
- the standardization of the interface between middleware and application results in the *portability* of an application to different middleware:
  - the same API exists at each access point.
- applications programmers are typically only interested in the horizontal interface because it defines the point of contact to their appls.

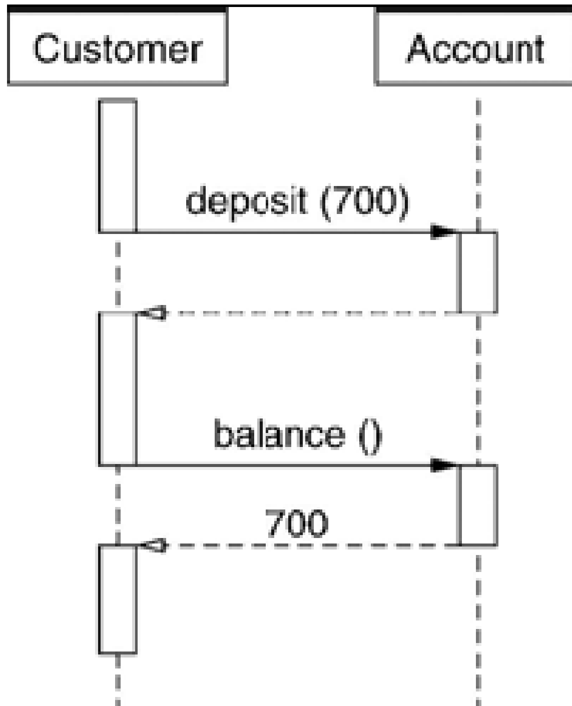
Vertical interface / Interoperability

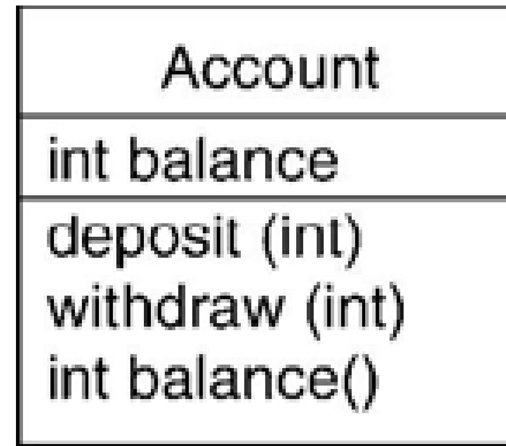
- defines the interface between two instances of a middleware platform
- is typically defined through a protocol on the basis of messages, referred to as *protocol data units* (PDUs).
  - A PDU is a message sent over the network.
  - Both client and server exchange PDUs to implement the protocol.
- separates technological domains
- ensures that applications can extend beyond the influence area of the product of middleware
- the standardization of this interface allows *interoperability* between applications
- of minor importance for the development of an application.
- implicit dependency exists between vertical and horizontal interfaces.
  - For example, coding rules for the PDUs have to exist in the vertical interface for all data types available in the horizontal interface of an application.

Sample application: account example

- customer wishes to do operations on a bank account.
- we are not concerned with different types of accounts, or how accounts are created by a bank.
- for simplicity sake, we assume that only one customer and one account exist, each represented through an object
- the account maintains a balance,
- the customer can deposit and withdraw money through appropriate operations
- the customer can furthermore inquire as to the balance of the account.

Sequence diagram for account use case & UML class diagram





Distribution of the Sample Application

- Layers:
  - The server layer contains the account object.
  - The client layer accesses this object through references (i.e., C++ pointers).
- The separation of client and server into different address spaces: assumes that:
  - the actual parameters are being transmitted between processes since a common address space no longer exists
  - all data belonging to the parameters of an interaction between a client and a server must therefore be transmitted explicitly to the address space of the server
  - the data must be self-contained; that is, it is not allowed to contain a pointer that is only valid in the context of the client.

Proxies

- A proxy of the server exists on the client side to
  - offers the same API as the server itself
  - transmit all current parameters over a communications channel to the remote address space
- In the remote address space, a proxy of the client
  - accepts the data and
  - executes the actual invocation on the server.
- it is not possible to distinguish the proxies from their "originals," so the distribution of client and server is *transparent.*
- the proxies are used to fill the gaps on either side so that client and server are unaware of the separation.