AOT LAB

Agent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma

Distributed and Agent Systems

Middleware

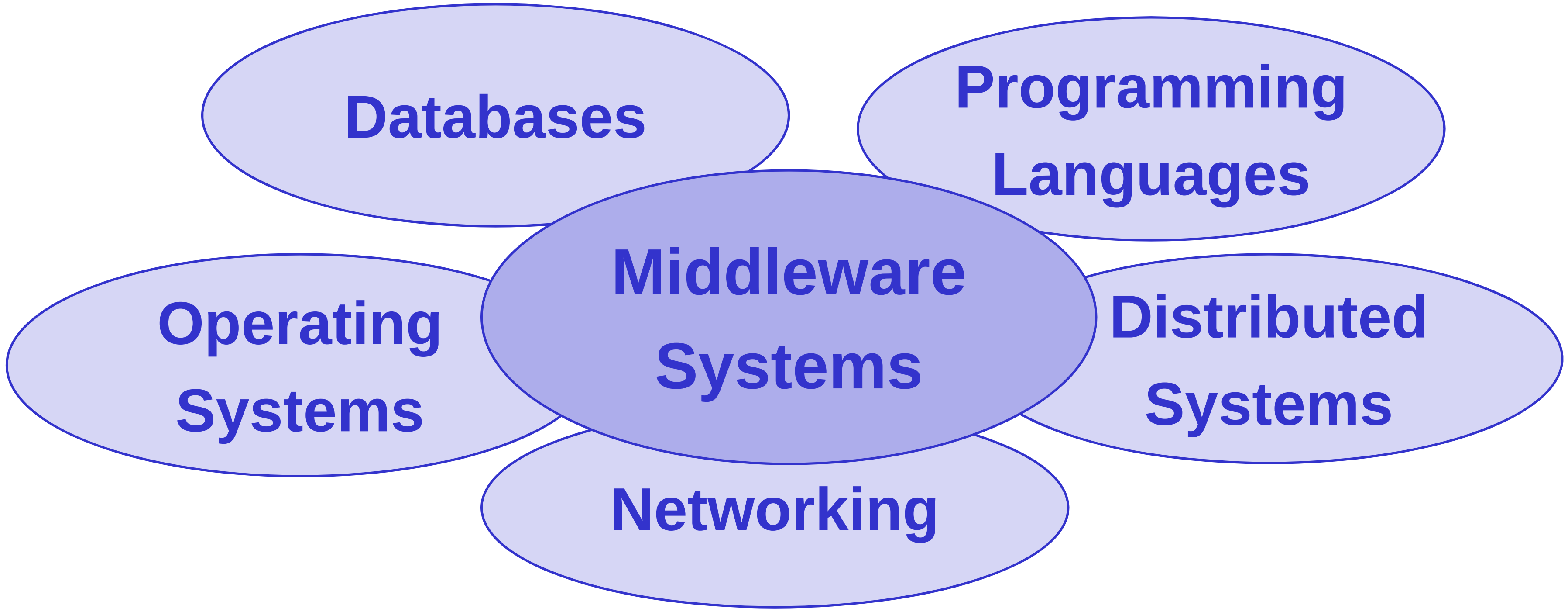
Prof. Agostino Poggi

AOT LAB

What is a Middleware?

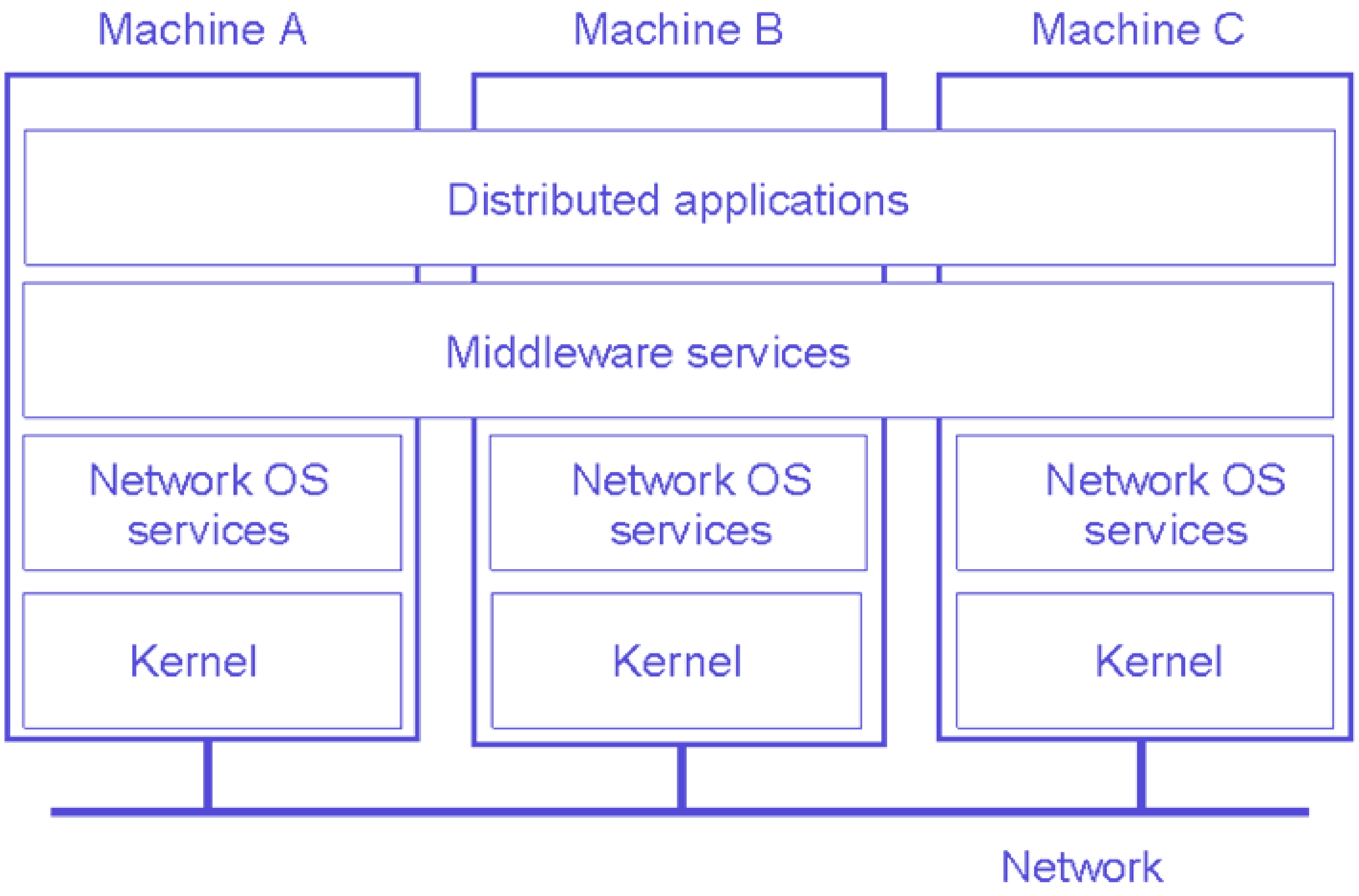
- The glue which connects objects which are distributed across multiple heterogeneous computer systems
- An extension of the operating system which provides a transparent communication layer to the applications
- A software layer that serves to shield the application of the heterogeneity of the underlying computer platforms and networks

What is Middleware ?



AOT LAB

What is a Middleware?



Application Programming Interface

- A middleware provides a more functional Application Programming Interface (API) than the operating system and network services to allow an application to:
  - Locate transparently across the network, providing interaction with another application or service
  - Be independent from network services
  - Be reliable and available
  - Scale up in capacity without losing function
- A middleware API decouples the applications from technologies used in the realized system

AOT
LAB

Middleware Evolution

- Middleware technology allowed the migration of mainframe applications to client/server applications and to providing for communication across heterogeneous platforms
- This technology has evolved during the 1990s to provide for interoperability in support of the move to client/server architectures
- Now it provides support to the most innovative distributed systems (e.g., mobile, ubiquitous, …)

Middleware Goals

- Integrate existing components into a distributed system
  - Components may be off-the-shelf
  - Components may be on incompatible hardware and OS platforms
  - Facilitate scalability through an appropriate architecture
- Resolve heterogeneity
  - Facilitate communication and coordination of distributed components
  - Build systems distributed across heterogeneous networks
  - Provide adaptability and reconfigurability

AOT LAB

Middleware Desiderata

- Network communication
  - Higher level primitives than network operating system primitives
  - Transport complex data structures over the network (marshalling / unmarshalling)
- Coordination
  - Three communication ways
    - Synchronous: client waits for the result
    - Deferred synchronous: client asks for the result (e.g., by polling)
    - Asynchronous: server autonomously sends the result
  - Component activation / deactivation
  - Group and concurrent requests management

Middleware Desiderata

- Reliability
  - Error detection and correction mechanisms on top of network protocols
- Scalability
  - Access to a component independent of whether it is local or remote
  - Migration transparency
  - Replication transparency
- Heterogeneity
  - Primitive data encoding
  - Different programming languages

AOT LAB

Middleware Limits

- There is a gap between principles and practice because many popular middleware services use proprietary implementations
- The large number of middleware services is a barrier to using them because developers have to select a small number of services that meet their needs for functionality and platform coverage
- Middleware still leave the application developer with hard design choices. In particular, the developer must still decide what functionality to put on the different components of a system

Middleware Types

- **R**emote **P**rocedure **C**all (RPC)
  - XML-RPC, SOAP, …
- **O**bject **R**equest **B**roker (ORB)
  - Corba, DCOM, RMI, …
- **T**ransaction **P**rocessing (TP) Monitor
  - IBM CSCS, BEA Tuxedo
- **M**essage-**O**riented **M**iddleware (MOM)
  - Java Message Queue, IBM MQSeries, …

Remote Procedure Call

- RPC middleware enable the logic of an application to be distributed across the network
  - Program logic on remote systems can be executed as simply as calling a local routine
  - The input and output parameters of the procedure call are used for exchanging data, but pointers cannot be passed as parameters
- RPC is appropriate for client/server applications in which the client can:

1. Issue a request
2. Wait for the server's response before continuing its own processing

AOT LAB

RPC Features

- Network communication
  - Server exports parameterized procedures
  - Clients call these across the network
  - Marshalling and unmarshalling by client and server stubs (generated by the compiler)
- Coordination
  - Synchronous interaction between one client and one server
  - Startup on demand possible (daemon needs a table that maps RPC names to program locations in the file system)

RPC Features

- No scalability
- Reliability
  - At-most-once semantics (exception if RPC fails)
- Heterogeneity
  - Can be used between different programming languages
  - Across different hardware and OS platforms

Interoperability Problems

- When exchanging data between clients and servers residing in different environments (hardware or software), care must be taken that the data is in the appropriate format:
  - Byte order
  - Data structures
- This is done using an intermediate representation format and data are transformed to this format and back through a process named:
  - Marshalling/unmarshalling
  - Serializing/deserializing

AOT LAB

Interface Description Language

- All RPC systems come with a language that allows to describe services in an abstract manner

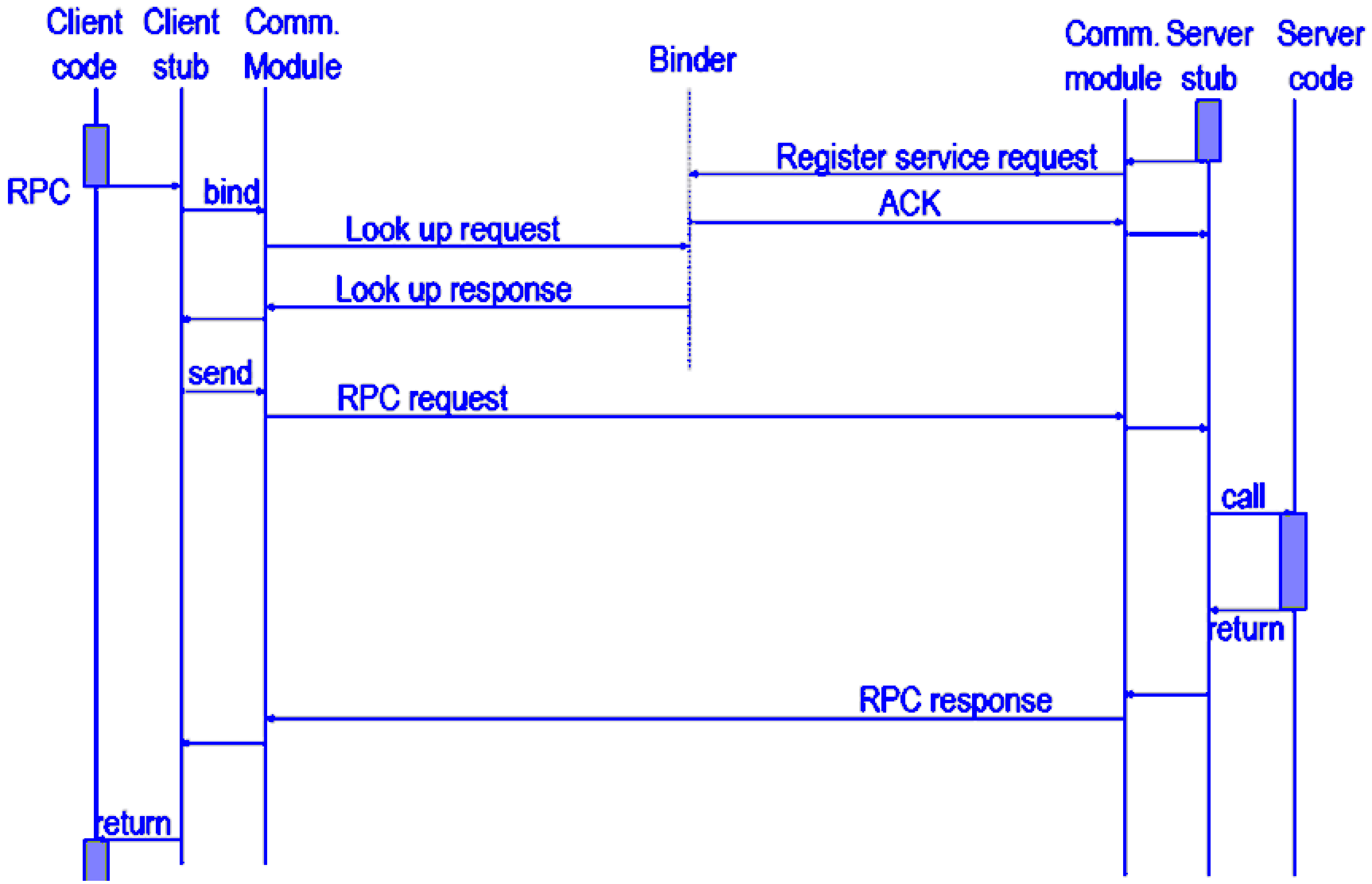
- This language has the generic name of IDL and defines each service in terms of their names and input and output parameters

- An interface compiler is then used to generate the stubs for clients and servers

Binding

- A service is provided by a server located at a particular IP address and listening to a given port
- Binding is the process of mapping a service name to an address and port that can be used for communication. Binding can be done:
  - Locally: the client must know the address of the server
  - Distributed: there is a separate service in charge of mapping names and addresses
- With a distributed binder:
  - The server registers and withdraws its services
  - The client lookups services

RPC Interaction



AOT
LAB

RPC Advantages

- RPC provided a mechanism to implement distributed applications in a simple and efficient manner
- RPC followed the programming techniques of procedural languages and fitted quite well with the most typical programming languages
- RPC allowed the modular and hierarchical design of large distributed systems:
  - Clients and servers are separate
  - Servers encapsulate and hide the details of the back end systems

AOT
LAB

RPC Disadvantages

- RPC is not a standard, it is an idea that has been implemented in many different ways
- RPC allows designers to build distributed systems but does not solve many of the problem distribution creates
  - It is only a low level construct
- RPC was designed for the client/server systems
- When there are more entities interacting with each other RPC treats the calls as independent of each other, but, they may be not independent
  - Recovering from partial system failures is very complex

Object Request Broker

- ORB middleware is based on RPC and supports distributed objects and the realization of component-based systems
- The basic difference between a RPC and ORB is that RPC uses standard programming procedure methods and ORB uses object oriented methods
- ORB enable the objects (components) that comprise an application to be distributed and shared across heterogeneous networks
- ORB promotes the goal of object communication across machine, software, and vendor boundaries

ORB Features

- Network communication
  - Client objects call methods of exported server objects
  - Marshalling and unmarshalling by client and server stubs (generated by the compiler)
- Coordination
  - Synchronous is the default
  - Some provide deferred synchronous and asynchronous

ORB Features

- Scalability
  - Support for load-balancing, replication is rather limited
- Reliability
  - At-most-once semantics (exceptions on failure) is the default
  - Usually requests may be clustered into transactions
- Heterogeneity
  - Some provide multi-language binding
  - Some may interoperate

Transaction Processor Monitor

- A TP Monitor allows building a common interface to several applications while maintaining or adding transactional properties
- A TP Monitor extends the transactional capabilities of a database beyond the database domain
  - It provides the mechanisms and tools necessary to build applications in which transactional guarantees are provided
- TP Monitor multiplexes client transaction requests onto a controlled number of servers that support particular services

AOT LAB

TP Monitor Features

- Network communication
  - Client and servers may reside on different hosts
- Coordination
  - Synchronous and asynchronous
- Scalability
  - Load balancing and replication of server components

TP Monitor Features

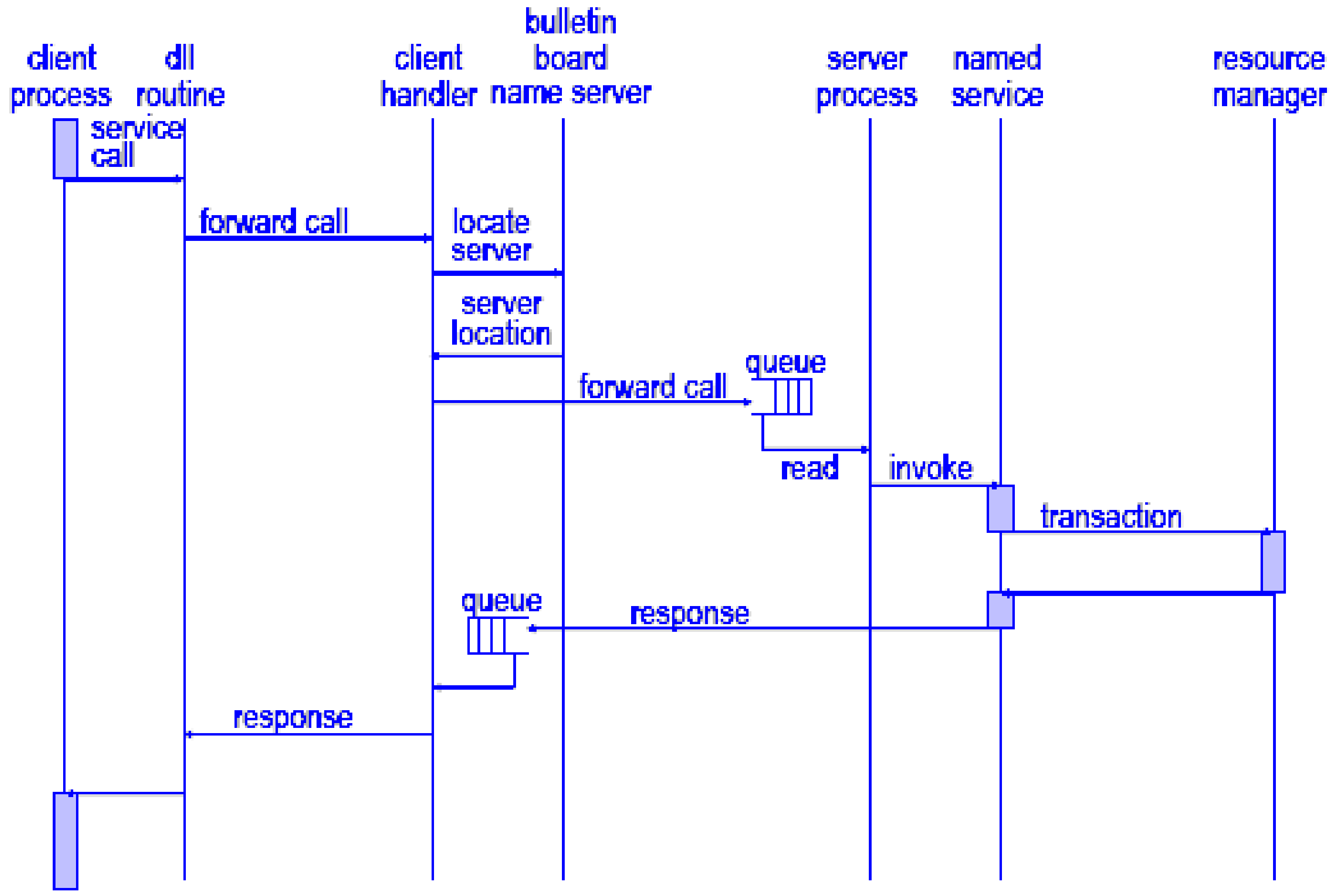
- Reliability
  - DTP (Distributed Transaction Protocol) based on the two phase commit protocol (2PC)
  - ACID properties:
    - Atomic: transaction is either complete or not
    - Consistent: system always in consistent state
    - Isolation: transaction is independent of other transactions
    - Durable: committed transaction survives system failures
- Heterogeneity
  - Different hardware and operating systems platforms
  - No data heterogeneity

Two Phase Commit

- Phase 1
  - The commit manager node sends prepare-to-commit commands to all its direct subordinate nodes
  - All the direct subordinate nodes answer with a ready-to-commit signal
- Phase 2
  - The commit manager node informs all its direct subordinate nodes to commit
  - The two-phase commit is successful if all subordinate nodes return commit confirmation, aborts if any of them returns refuse information

AOT
LAB

TP Monitor Interaction



Development and Execution

- Development of a TP Monitor application is very similar to an RPC one:
  - Define the services to implement and describe them in IDL
  - Specify which services are transactional
  - Use an IDL compiler to generate the client and server stubs
- Execution requires a bit more control since now interaction is no longer point to point:
  - Transactional services maintain context information and call records in order to guarantee atomicity
  - Stubs also need to support more information like transaction identifier and call context
  - Complex call hierarchies are typically implemented with a TP Monitor and not with plain RPC

AOT
LAB

TP Monitor Advantages

- Components are kept in consistent states (due to ACID properties of transaction)
- TP Monitor is very reliable
- TP Monitor performs better than message-oriented and procedural middleware
- TP Monitor can dispatch, schedule and prioritize multiple application requests concurrently, thus reducing CPU overhead, response times and CPU cost for large applications

TP-Monitor Disadvantages

- TP Monitor has often unnecessary or undesirable guarantees according to ACID
  - If a client is performing long-lived activities, then transactions could prevent other clients from being able to continue
- Marshalling and unmarshalling have to be done manually in many products
- The lack of common standard for services definition reduces the portability of an application between different TP monitors
- TP Monitor runs on less amount of platforms than other middleware types

AOT LAB

Synchronous Interaction Problems

- Synchronous invocations require to maintain a session between the caller and the receiver and maintaining sessions is expensive
  - There is also a limit on how many sessions can be active at the same time (thus limiting the number of concurrent clients connected to a server)
- If the client or the server fail, the context is lost and resynchronization might be difficult
  - Finding out when the failure took place may not be easy. Worse still, if there is a chain of invocations, the failure can occur anywhere

Solutions

- Transactional interaction
  - To enforce exactly once execution semantics and enable more complex interactions with some execution guarantees
- Service replication and load balancing
  - To prevent the service from becoming unavailable when there is a failure (however, the recovery at the client side is still possible)
- Asynchronous interaction
  - the caller sends a message that gets stored somewhere until the receiver reads it and sends a response
  - The response is sent in a similar manner

Message Oriented Middleware

- MOM implements asynchronous interaction through the exchange of messages
- MOM is analogous to email
  - It is asynchronous
  - Requires the recipients of messages to interpret their meaning and to take appropriate action
- MOM offers a natural way to implement complex interactions between heterogeneous systems
- There are two models supported by the MOM:
  - Point-to-point
  - Message queuing

Message Queuing

- Message queuing is a way for communicating across heterogeneous networks and systems that guarantees that messages are there even after some failures occur
- Message queuing offers significant advantages over traditional solutions in terms of fault tolerance and overall system flexibility
- Message queuing supports sophisticated distribution modes (multicast, transfers, replication, coalescing messages) an it also helps to handle communication sessions in a more abstract way

Transactional Queuing

- MOM may provide transactional interaction:
  - Message are written in the queue using 2PC
  - Messages between queues are exchanged using 2PC
  - Messages are read from a queue, processed and the reply is written in another queue using 2PC
- This introduces a significant overhead but it also provides considerable advantages because the processing of messages and sending and receiving can be tied together into one single transactions so that atomicity is guaranteed

MOM Features

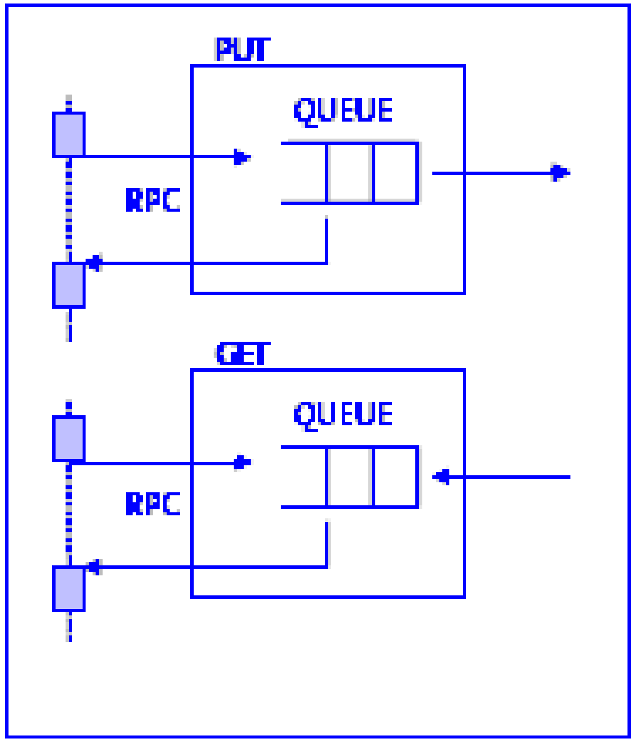
- Network communication
  - Client sends a request message and server replies with result message
  - Well suited for event notification and publish / subscribe
- Coordination
  - Asynchronous
  - Synchronous has to be coded by client

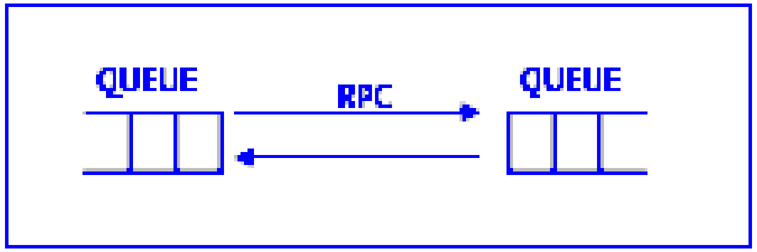
MOM Features

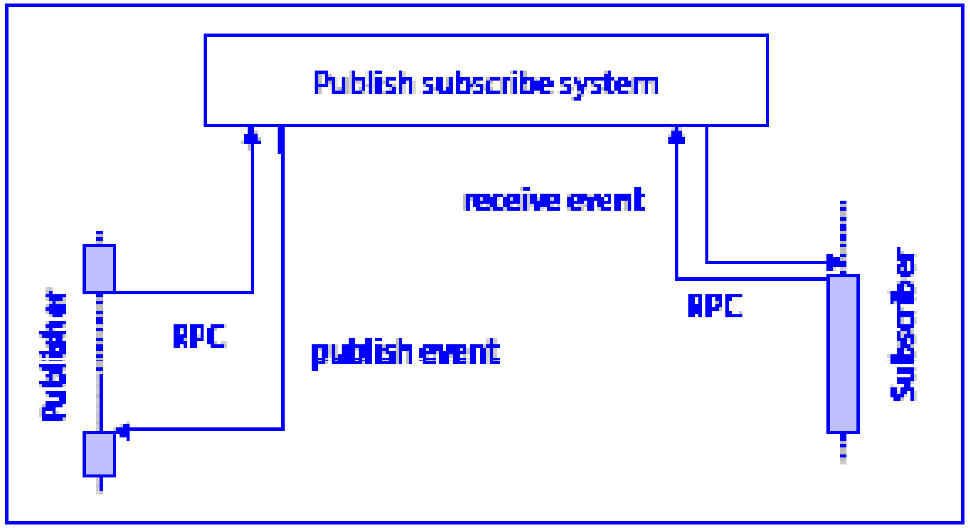
- Scalability
  - Local / remote differs
- Reliability
  - Message queues are stored on persistent memory
  - At-least-once semantics possible
- Heterogeneity
  - Marshalling code has to be written by hand

AOT LAB

MOM Interaction







AOT
LAB

MOM Advantages

- MOM supports group communication, which is atomic (either all clients receive a delivery or none)
  - A process doesn't have to worry about what to do, if some clients don't receive a message
- MOM supports more network protocols than RPC
- The use of persistent queues increases reliability in MOM products
- Transactional message queues provide high reliability
- MOM can send the message exactly-once due to QoS, thus increasing it's reliability

AOT
LAB

MOM Disadvantages

- MOM has limited scalability and heterogeneity support
- There is a bad portability support, because MOM products don't support any standards
  - Applications that are made for one MOM product are not compatible to another MOM product
  - There is no interface definition language and no marshalling and unmarshalling of message contents
- Since clients do not block after a message is put into the queue, clients can make requests faster than servers can respond to the requests