Middleware - introduction

CPE 545

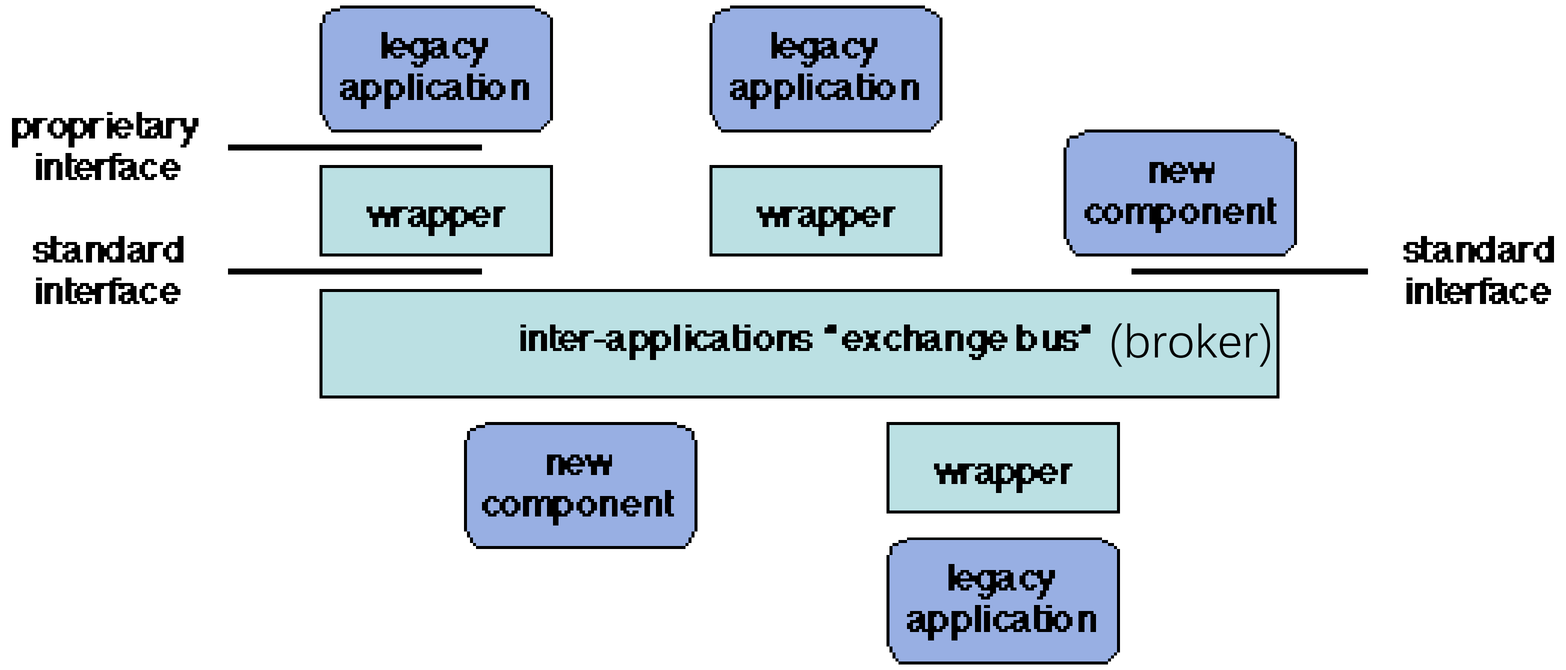
What is Middleware?

- Software that mediates between an application program and a network.
- Middleware is a distributed software layer that sits above the network operating system and below the application layer and abstracts the heterogeneity of the underlying environment.

Motivation for Middleware

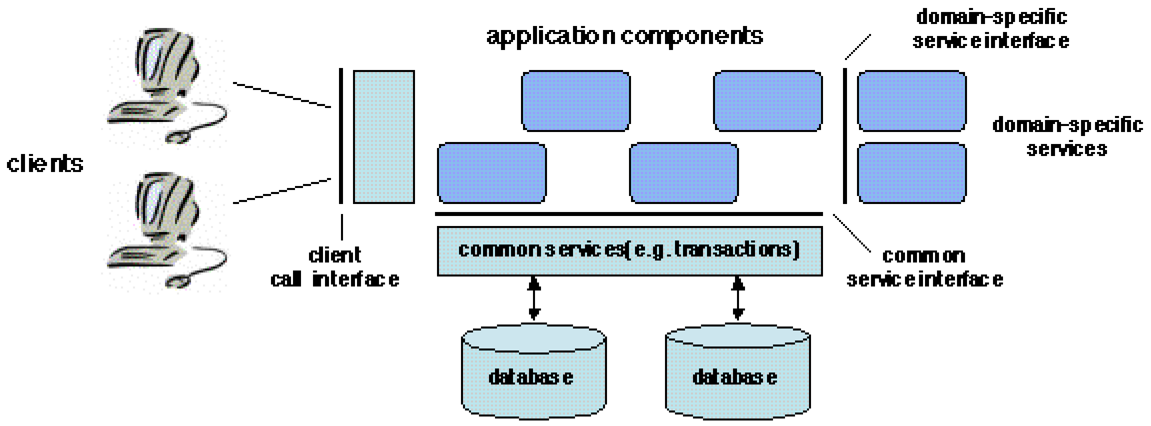
- Reusing Legacy Software. We would like to:
  - Adopt a *common language* to interconnect with other applications
  - Define the *interface* and *interchange protocols* implemented in a software acting as broker
  - Employ a *user wrapper* to bridge the legacy interface with the new interface

Motivation for Middleware



Motivation for Middleware

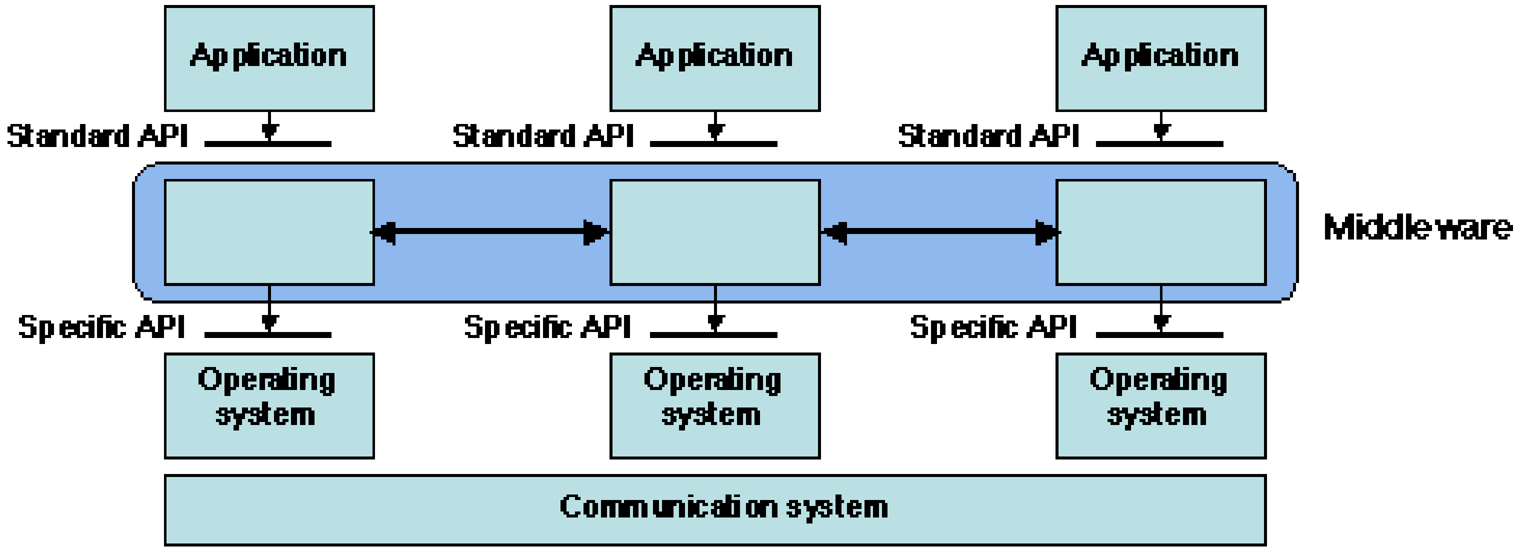
- Component Based Software
  - Applications composed of software components
    - Client user interface (presentation layer)
    - Information management (Database)
    - Application specific functionality (business logic)



Middleware

- Middleware is an *intermediate software* that resides on top of the *operating systems* and *communication protocols* to perform the following functions:
  - Hiding *distribution*.
  - Hiding the *heterogeneity* of the various hardware components, operating systems and communication protocols.
  - Providing uniform, standard, high-level <u>*interfaces*</u> to the application developers and integrators,
    - Applications can easily interoperate and be reused.
  - Supplying a set of <u>*common services*</u> to perform various general purpose functions
    - Avoids duplicating efforts and facilitates collaboration between applications.

Motivation for Middleware



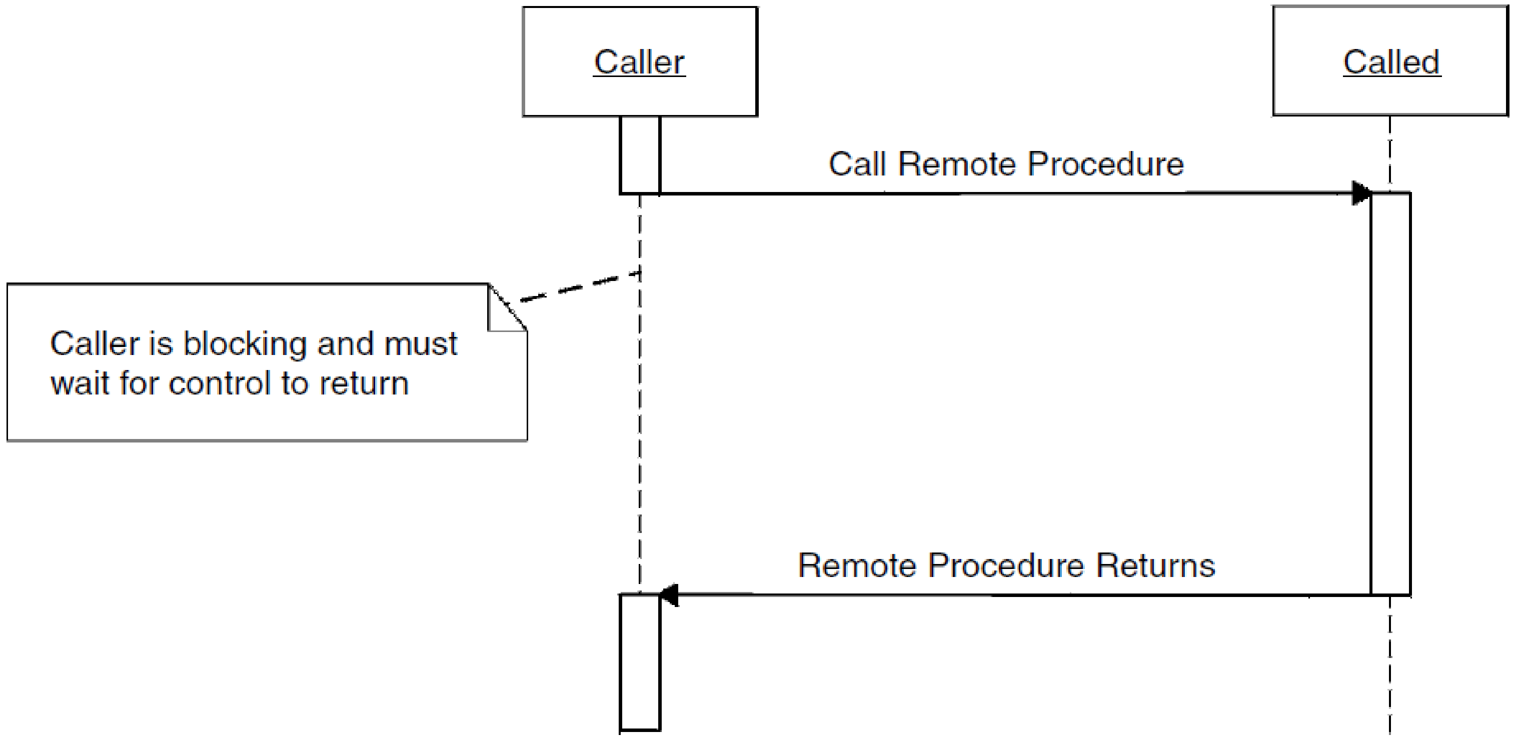
Middleware- architecture

- The overall architecture of a middleware system may be classified according to the following properties:
  - *Managed entities* - Manage different kinds of entities which differ by their definition, properties such as:
    - Objects, agents, and components.
  - *Structure* - Manage entities that may have predefined roles such as:
    - *client* (service requester) and *server* (service provider),
    - *publisher* (information supplier) and *subscriber* (information receiver).
    - *peer to peer* - all entities are at the same level and a given entity may indifferently assume different roles
  - *Interfaces*- Provide communication primitives that may follow the synchronous or asynchronous paradigm.

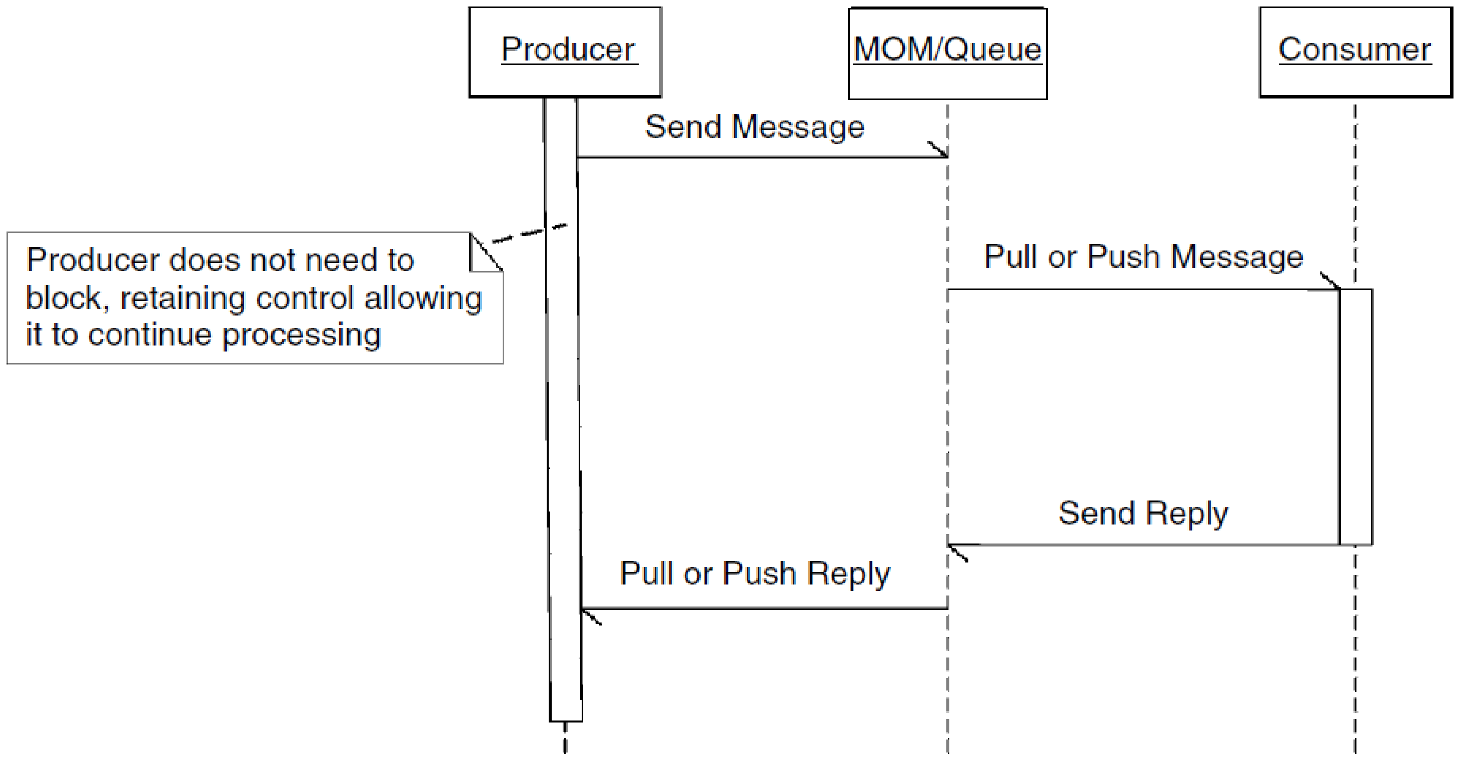
Middleware- architecture

- The interface properties:
  - Synchronous communication
    - When a method is called using the synchronous interaction model, the caller code must block and wait (suspend processing) until the called code completes execution and returns control to it
  - Asynchronous communication.
    - The caller code does not need to block and wait for the called code to return. This model allows the caller to continue processing regardless of the processing state of the called procedure/function/method.

Synchronous Interaction Model



Asynchronous Interaction Model


Producer
MOM/Queue
Consumer
Send Message
Producer does not need to block, retaining control allowing it to continue processing
Pull or Push Message
Send Reply
Pull or Push Reply

- While more complex than the synchronous model, the asynchronous model allows all participants to retain processing independence.

RPC – Synchronous Model

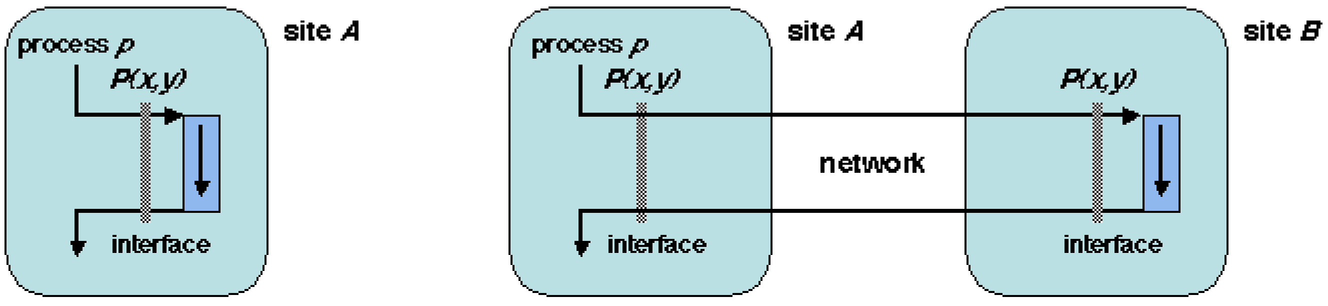
- The traditional RPC model is a fundamental concept of distributed computing.
- It is utilized in middleware platforms including CORBA, Java RMI, Microsoft DCOM, and XMLRPC.
- The objective of RPC is to allow two processes to interact.

RPC- Requirement

- Procedural *abstraction* is the key concept
- A procedure, is a "black box" that performs a specified task by executing an *encapsulated* sequence of code.
  - Encapsulation means that the procedure may only be called through *an interface* that specifies its *parameters* and *return values* as a set of typed holders.

RPC- Requirement

- On a site *A*, consider a process *p* that executes a local call to a procedure *P*.
- Design a mechanism that would allow process *p* on site A (*client*) to perform the same call, with the execution of *P* taking place on a remote site *B (server)*, while *preserving the overall effect* of the call.



(a) Local call (b) Remote call

RPC- Motivations and challenges

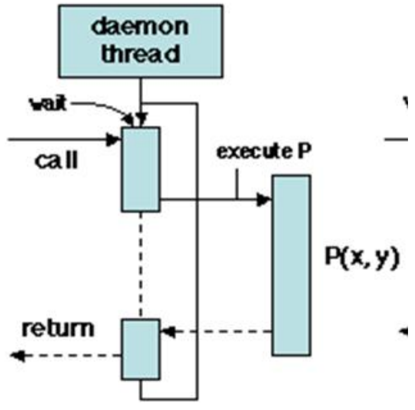
- Preserving the semantics between local and remote call
  - Procedural abstraction is preserved;
  - Portability is improved - the application is independent of the underlying communication protocols.
    - Application may easily be ported, without changes, between a local and a distributed environment.
- Preserving semantics is no easy task, for two main reasons:
  - The failure modes are different in the distributed case - the server procedure failure, network timeouts, server timeout, etc.
  - The semantics of parameter passing is different (e.g. passing a pointer as a parameter does not work in the distributed case because the calling process and the procedure execute in distinct address spaces).

RPC- Implementation Principles

- RPC relies on two pieces of software, the client stub and the server stub
  - The client stub acts as a local representative of the server on the client site;
  - The server stub has a symmetrical role. Thus both the calling process (on the client side) and the procedure body (on the server side) keep the same interface as in the centralized case.
  - The client and server stubs rely on a communication subsystem to exchange messages. In addition, they use a *naming service* in order to help the client locate the server

RPC- synchronization

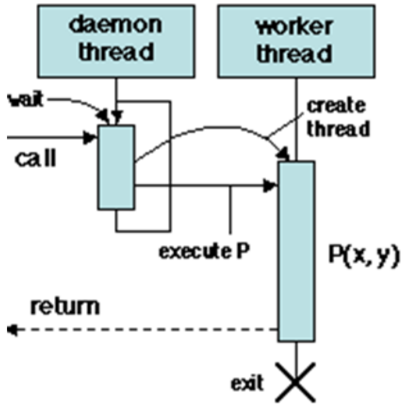
- On the client side, the calling process (or thread) *must be blocked* while waiting for the procedure to return.
- On the server side, the main issue is that of parallel execution since the server may be used by *multiple clients*. Multiplexing the server resources calls for a *multithreaded* organization.



(a) direct execution

RPC- synchronization

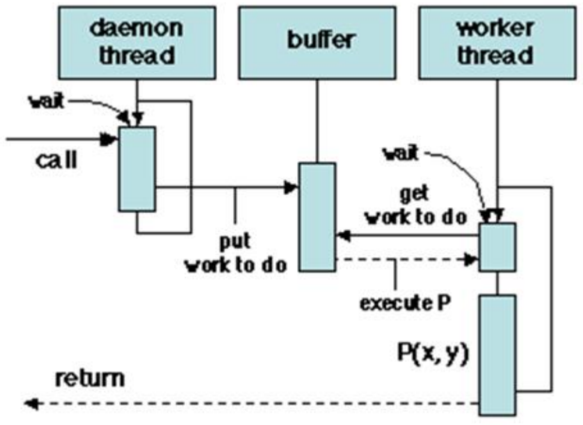
- On the server side , a *daemon thread* waits for incoming messages on a specified port. A new *worker thread* is created in order to execute the procedure, while the daemon returns to wait on the next call; the worker thread exits upon completion



(b) thread per call

RPC- process management

- In order to avoid the *overhead* due to thread creation, an alternate solution is to manage a fixed-size pool of worker threads .
- Worker threads communicate with the daemon through a shared buffer (queue of work orders) using the *producer-consumer scheme*.
- Worker threads are *waiting for new work* to arrive; after executing the procedure, a worker thread returns to the pool, i.e. tries to get new work to do.

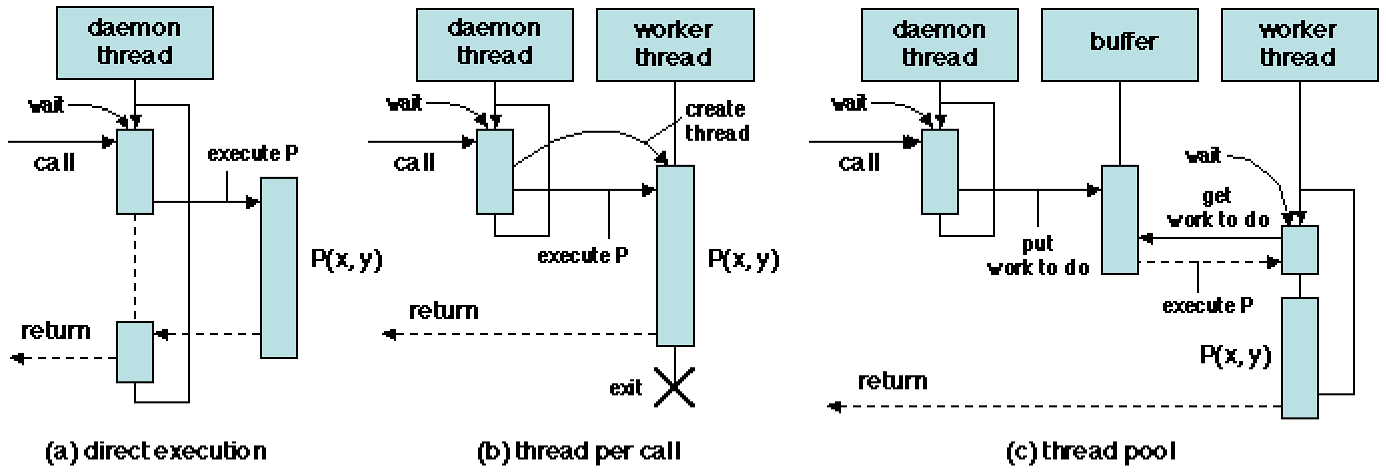


(c) thread pool

RPC- process management

- If all worker threads are *busy* when a call arrives, the execution of the call is *delayed* until a thread becomes free.
- All these synchronization and thread management operations are performed by the *stubs* and are invisible to the client and server main programs.

RPC- process management



Worker threads don’t know the details of what they call.
They execute callback functions (function pointers).

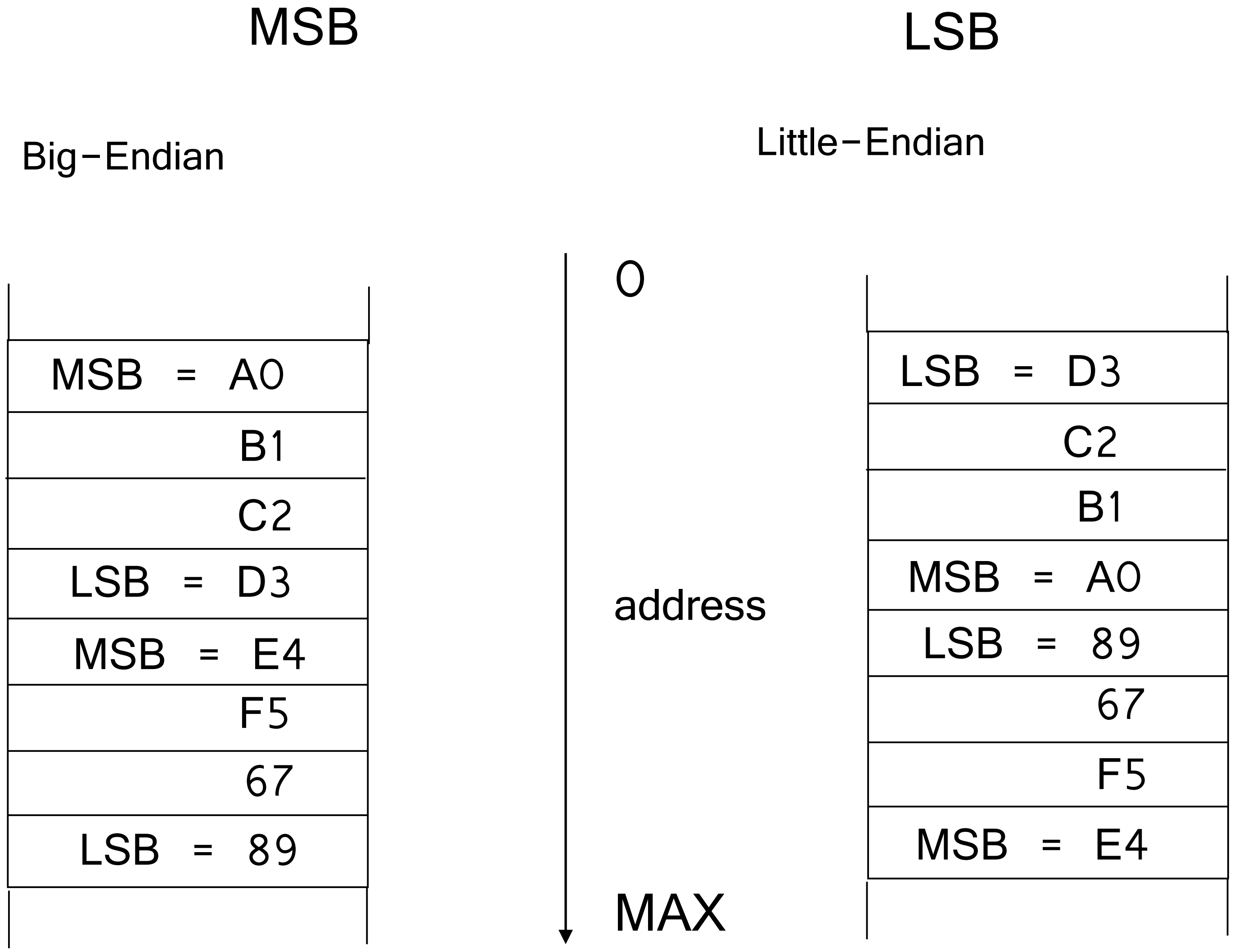
Midleware - goal

- We would like to hide the *heterogeneity* of the various hardware components, operating systems and communication protocols.

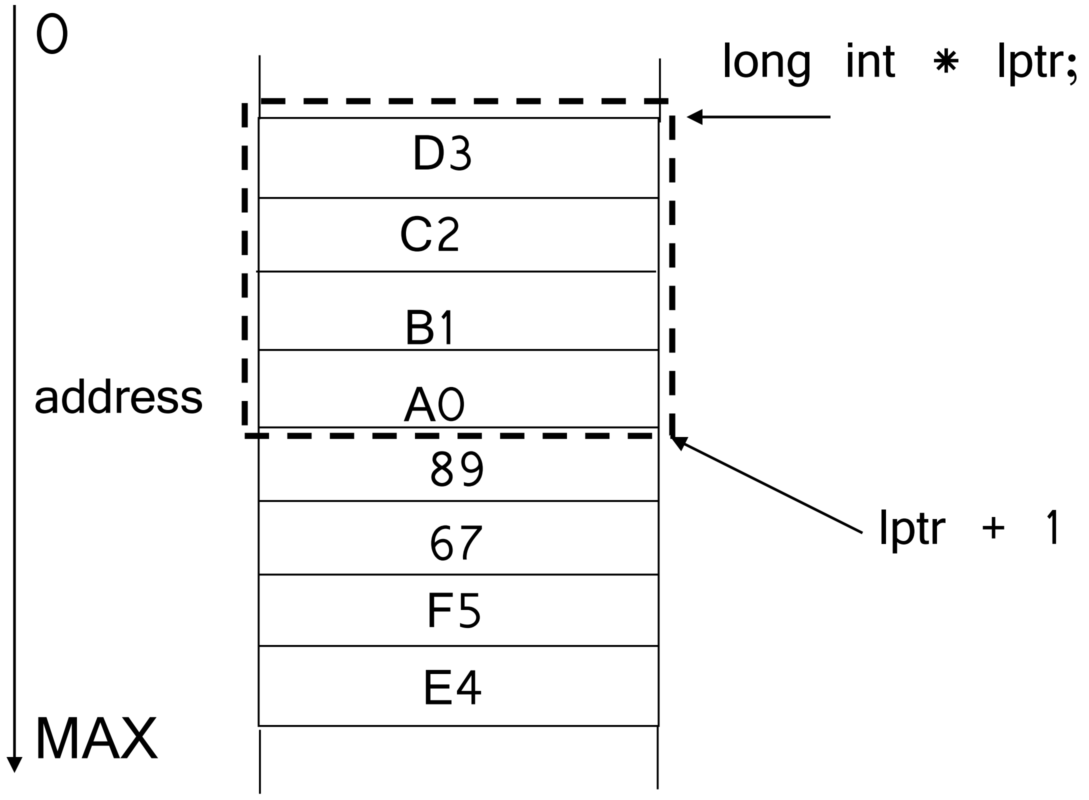
Little-Endian vs. Big-Endian Representation

0xA0B1C2D3

0xE4F56789



Pointers



Big-Endian

(* lptr) = 0xD3C2B1A0;
= 3552752032

Little-Endian

(* lptr) = 0xA0B1C2D3;
= 2696004307

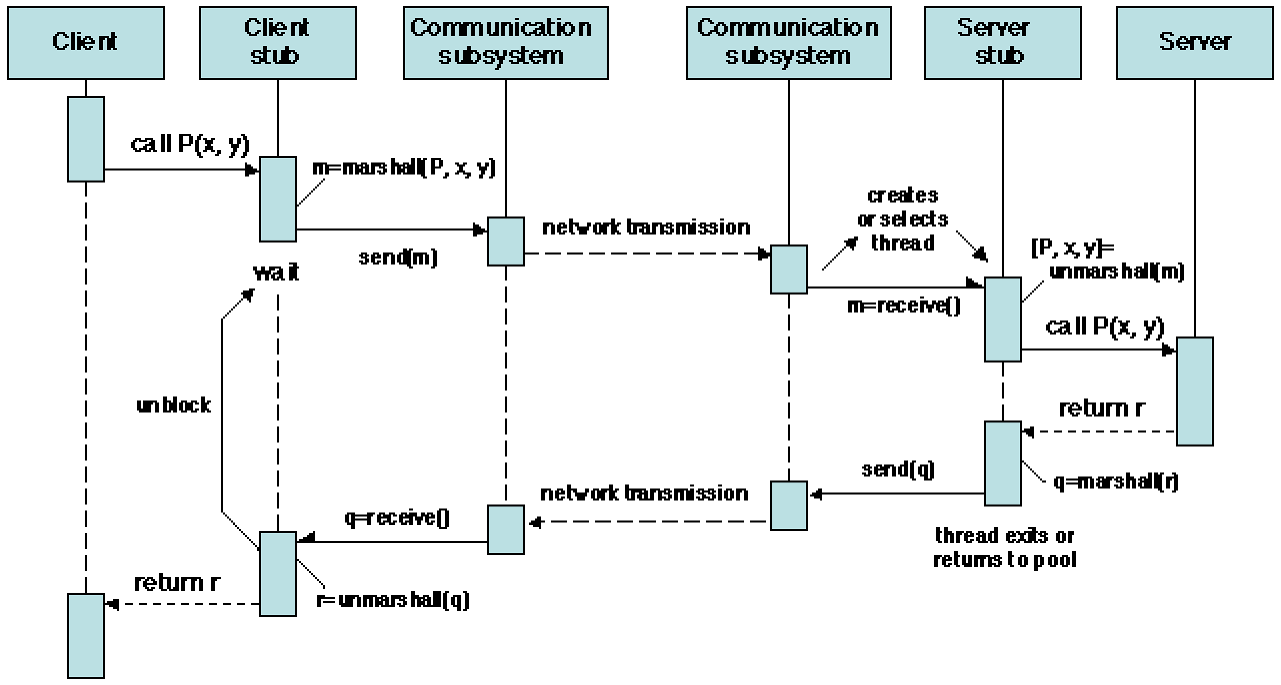
RPC- parameter marshaling

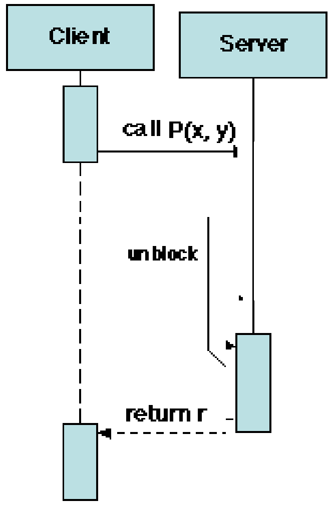
- Parameters and results need to be *transmitted over the network*. Therefore they need to be put in a *serialized* form, suitable for transmission.
    - In order to ensure portability, data should be independent of the underlying communication protocols as well as of the local data representation conventions (e.g. byte ordering) on the client and server machines.
- Converting data from a local representation to the standard serialized form is called *marshaling*; the reverse conversion is called *unmarshaling*.
    - A marshaller is a set of routines, one for each data type (e.g. writeInt, writeString, etc.), that write data of the specified type to a sequential data stream.
    - An unmarshaller performs the reverse function and provides routines (e.g. readInt, readString, etc.) that extract data of a specified type from a sequential data stream. These routines are called by the *stubs* when conversion is needed.

RPC- Reacting to failures

- As already mentioned, failures may occur on the client site, on the server site, and in the communication network. Taking potential failures into account is a three step process:
  1. Formulating failure hypotheses
     - The usual failure hypotheses are fail-stop for nodes, and message loss for communication (i.e. either the node operates correctly, and message arrives uncorrupted or it stops)
  2. Detecting failures
     - Failure detection mechanisms are based on timeouts. When a message is sent, a timeout is set at an estimated upper bound of the expected time for receipt of a reply. If the timeout is triggered, a recovery action is taken
  3. Reacting to failure detection
     - Such timeouts are set both on the client site (after sending the call message) and on the server site (after sending the reply message). In both cases, the message is resent after timeout
- It is usually impossible to guarantee the so-called "exactly once" semantics.
  - i.e. after all failures have been repaired, the call has been executed exactly one time.

RPC- flow control





RPC- Application Devlopment

- In order to actually develop an application using RPC, a number of practical issues need to be settled:
  - how are the client and the server linked together?
  - how are the client and server stubs constructed?
  - how are the programs installed and started?

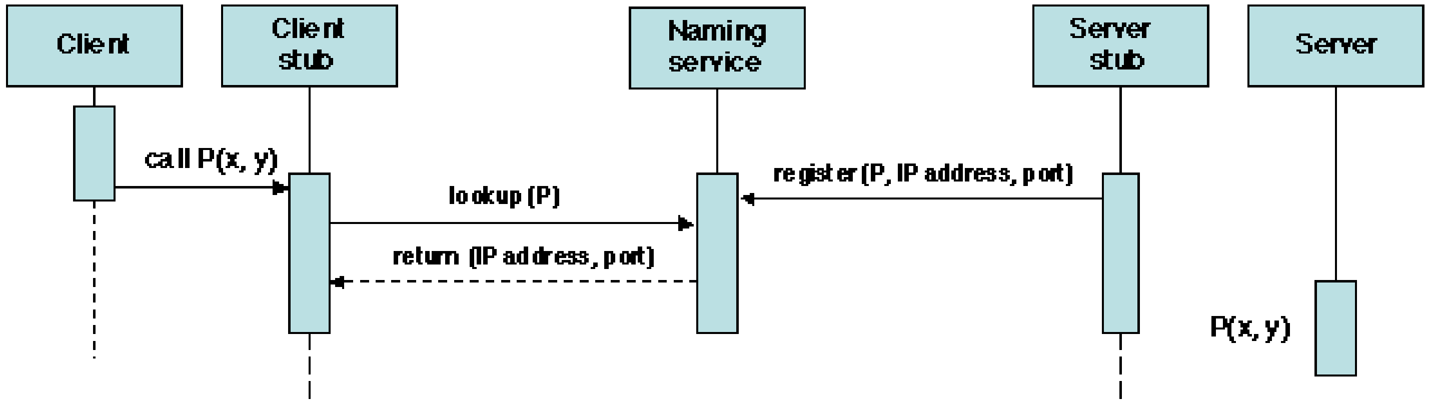
RPC- Client-Server Binding

- The client needs to know the server address and port number to which the call should be directed.
- To preserve abstraction and ensure increase availability, it is preferable to allow for a late binding of the remote execution site.
- The problem of binding the server to the client is solved by including the address and port number of the client in the call message.

RPC- Client-Server Binding

- The client locates the server site prior to the call using name servers (as opposed to static binding).
  - A server registers a procedure name together with its IP address and the port number of the daemon process which is waiting for calls on that procedure.
  - A client consults the naming service to get the IP address and port number for a given procedure.
  - The naming service is usually implemented as a server of its own, whose address and port number are known to all participant nodes.

RPC- Client-Server Binding



Remote procedure call: locating the server

RPC- Stub generation

- The stubs fulfill a set of well-defined functions and thus are obvious candidates for automatic generation
- The call-specific parameters needed for stub generation are specified in a special notation known as an Interface Definition Language (IDL).
  - An interface description written in IDL contains all the information that defines the interface of the procedure call
  - It acts as a contract between the caller and the callee.
  - For each parameter, the description specifies its type and mode of passing (e.g. by value, by copy-restore, etc.).

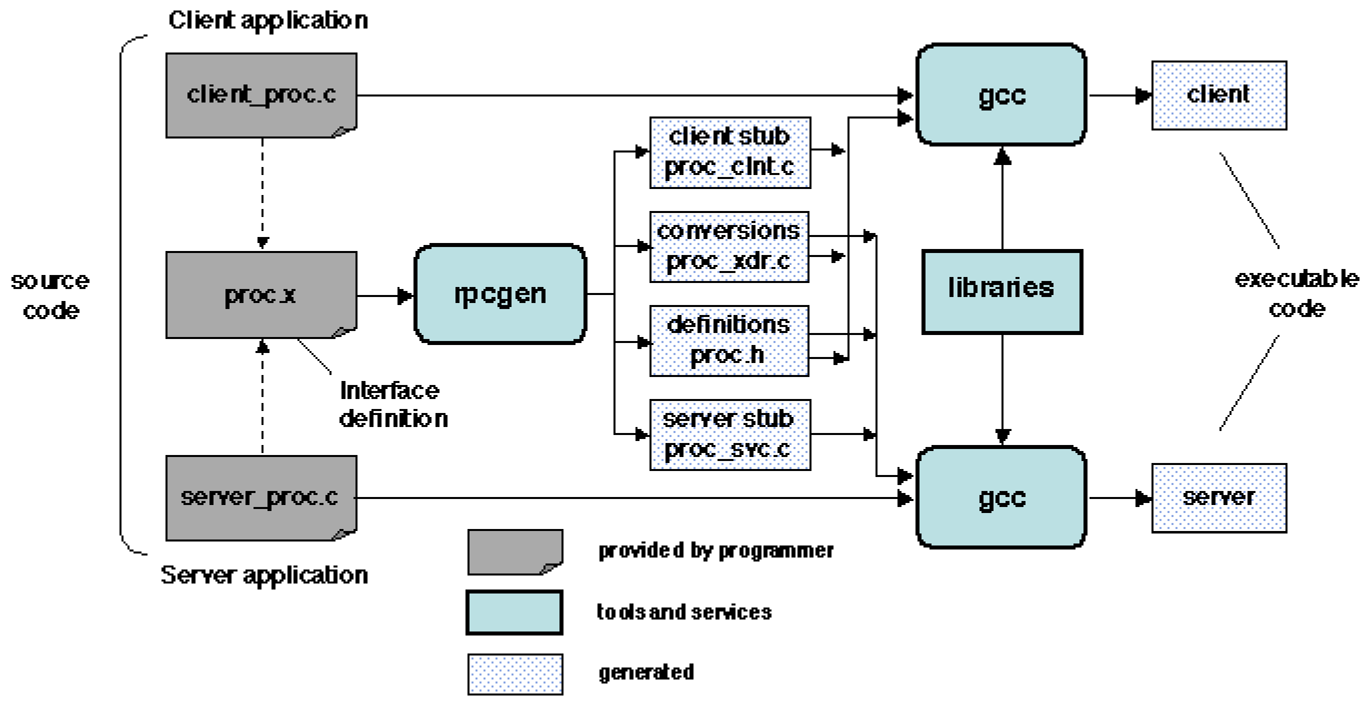
RPC- Stub generation

- The stub generator is associated with a specific common data representation format -- it inserts the conversion routines provided by the corresponding marshallers and unmarshallers.

- Example: A method Add which returns the sum of two long integers
  - **Boolean** add (**in long** addend1, **in long** addend2, **out long** sum);

- For more details on IDL see:
  - http://www.eecs.berkeley.edu/~messer/netappc/Supplements/10-idl.pdf

RPC- Stub generation



RPC – Synchronous Model

- **Coupling**
  - Designed for tightly coupled systems -- any changes to the interfaces will need to be propagated through the codebase of both systems.
  - This makes RPC a very invasive mechanism of distribution. As the number of changes to source or target systems increase, the cost will increase too.
- **Reliability**
  - Most RPC implementations provide little or no guaranteed reliable communication capability; they are very vulnerable to service outages.

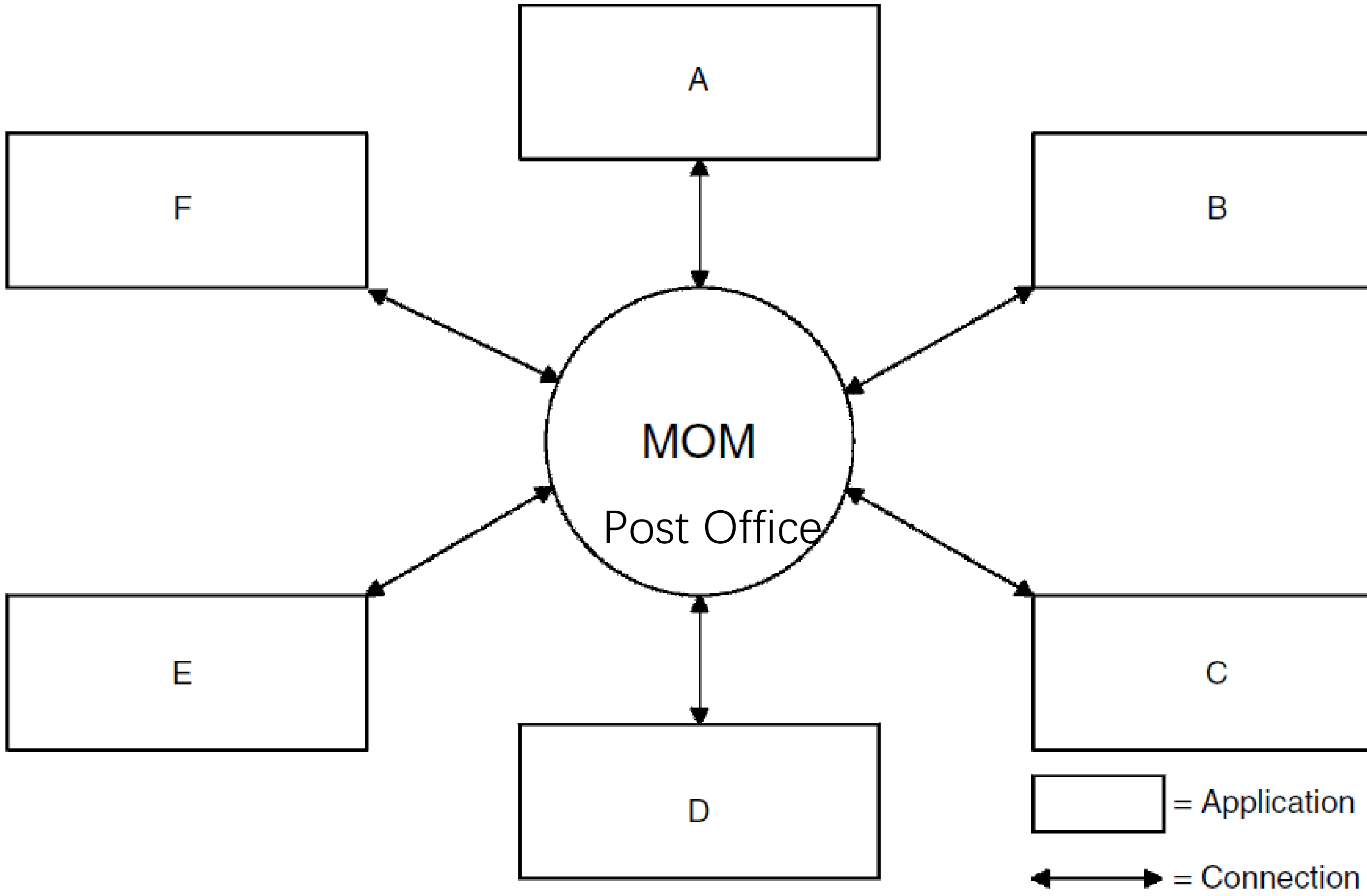
RPC – Synchronous Model

- **Scalability**
  - Subsystems do not scale equally, so it slows the whole system down to the maximum speed of its slowest participant.
  - Subsystems have trouble coping when elements of the system are subjected to a high-volume burst in traffic.
  - Interactions use more bandwidth because several calls must be made across the network in order to support a synchronous function call.
- **Availability**
  - Systems built using the RPC model are interdependent, requiring the simultaneous availability of all subsystems; a failure in a subsystem could cause the entire system to fail.

Message-Oriented Middleware (MOM)

- MOM systems provide distributed communication on the basis of the asynchronous interaction model
  - Non-blocking
  - allows the delivery of messages when the sender or receiver is not available to respond
  - Message deliver may take longer

Message-Oriented Middleware

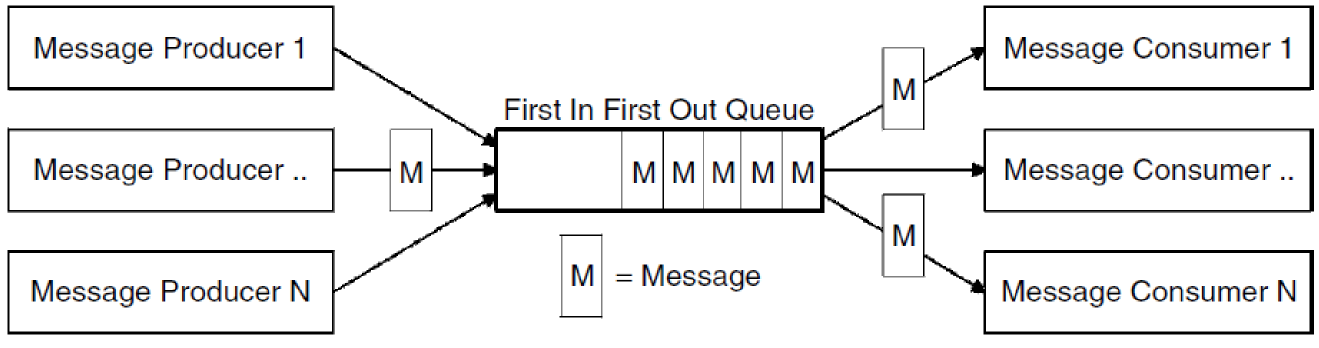


Message-Oriented Middleware (MOM)

- Loose coupling
  - independent layer acts as an intermediary to exchange messages between message senders and receivers
- Reliability
  - message loss through network or system failure is prevented by using a store and forward mechanism for message persistence
- Scalability
  - effective load balancing, by allowing a subsystem to choose to accept a message when it is ready to do
- Availability
  - MOM does not require simultaneous or “same-time” availability of all subsystems.
  - Failure in one of the subsystems will not cause failures to ripple throughout the entire system.

Message Queues

- The message queue is a fundamental concept within MOM.
- Queues provide the ability to store messages on a MOM platform.
- The standard queue found in a messaging system is the First-In First-Out (FIFO)



Messaging Models

- Two main message models are commonly available:
  - Point-to-point model
  - Publish/subscribe model
- Both of these models are based on the exchange of messages through a queue.

Messaging Models- *Point-to-Point*

- Messages from producing clients are routed to consuming clients via a queue.
- Usually only a single consuming client
- Each message is delivered only once to only one receiver.
- The model allows multiple receivers to connect to the queue, but only one of the receivers will consume the message.
  - Multiple consuming clients to read from a queue can be used to introduce load balancing into a system.
- Messages are always delivered and will be stored in the queue until a consumer is ready to retrieve them.

Messaging Models- *Publish/Subscribe*

- A mechanism used to disseminate information between anonymous message consumers and producers.
- Allows a single producer to send a message to one or many consumers
- The sending and receiving application is free from the need to understand anything about the target application.
- Clients producing messages ‘publish’ to a specific topic or channel, these channels are then ‘subscribed’ to by clients wishing to consume messages.
- The service routes the messages to consumers on the basis of the topics to which they have subscribed as being interested in

References

- Middleware Architecture with Patterns and Frameworks, Sacha Krakowiak
- Designing Embedded Communications Software, by T. Sridhar, ISBN: 157820125x, CMP Books
- IT Architectures and Middleware – 2nd edition. Chris Britton, Peter Bye. Addison-Wesley
- Tanenbaum & van Steen Distributed Systems: Principles and Paradigms, 2nd ed. ISBN: 0-132-39227-5.
- Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. ACM Transactions on Computer Systems, 2(1):39-59.
- Fox, A., Gribble, S. D., Chawathe, Y., and Brewer, E. A. (1998). Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, pages 10-19.
- White, J. E. (1976). A high-level framework for network-based resource sharing. In *National Computer Conference*, pages 561-570.
- A Perspective on the Future of Middleware-based Software Engineering. V. Issarny, M. Caporuscio, N. Georgantas. In Future of Software Engineering 2007 (FOSE) at ICSE (International Conference on Software Engineering). L. Briand and A. Wolf editors, IEEE-CS Press. 2007.
- Balter, R., Bernadat, J., Decouchant, D., Duda, A., Freyssinet, A., Krakowiak, S., Meysembourg, M., Le Dot, P., Nguyen Van, H., Paire, E., Riveill, M., Roisin, C., Rousset de Pina, X., Scioville, R., and Vandôme, G. (1991). Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31-67.