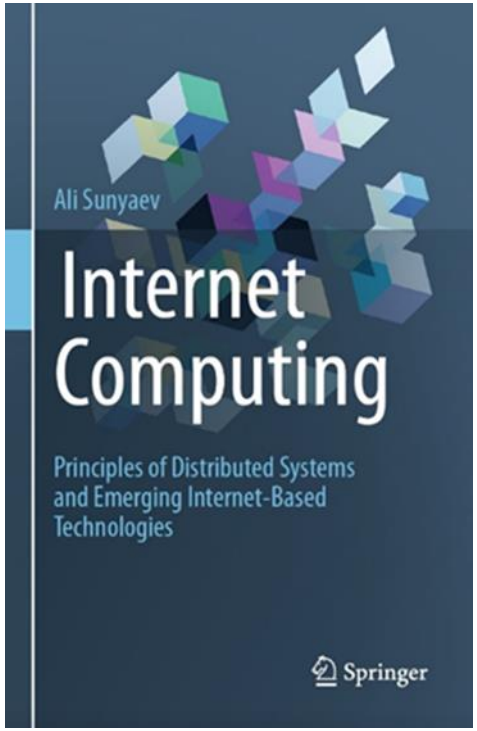
AI: Internet Computing

Lecture 5 — Middleware

Learning Goals of the Lecture

- Understand the concept of middleware
- Learn about the history and current use of middleware
- Understand remote procedure calls
- Understand the differences between the message-oriented, transaction-oriented, and object-oriented middleware

Reference to the Teaching Material Provided



Chapter 5
Middleware

Abstract
In the context of IT applications and especially in large organizations, integration of existing information systems into new IT environments poses many challenges. One of the biggest issue in this regard is dealing with the systems' heterogenity in terms of used programming languages, operating systems, or even data formats. In order to ensure communication between different information systems, developers must establish common interfaces. This chapter introduces middleware as a type of software which manages and facilitates interactions between applications across computing platforms. Besides a brief definition and overview of middleware, several of its characteristics are described. Furthermore, the differences between the three middleware categories (message-oriented, transaction-oriented and object-oriented middleware) are defined. In addition to these theoretical foundations, some practical implementations are presented.

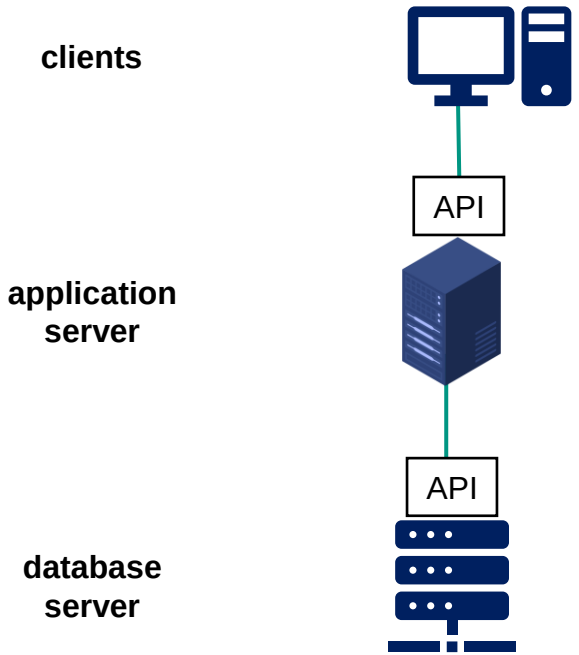
Learning Objectives of this Chapter
The primary learning objective of this chapter is to flesh out the concept, history, and current use of middleware. The chapter's main goal is to point out the difference between transaction-oriented middleware, message-oriented middleware, and object-oriented middleware. The subchapters present the key characteristics of each of these categories and their commercial implementation. Students will also

Introduction to Middleware

Applied Informatics and Formal Description Methods (AIFB)
Critical Information Infrastructures (cii)

Distributed Information Systems

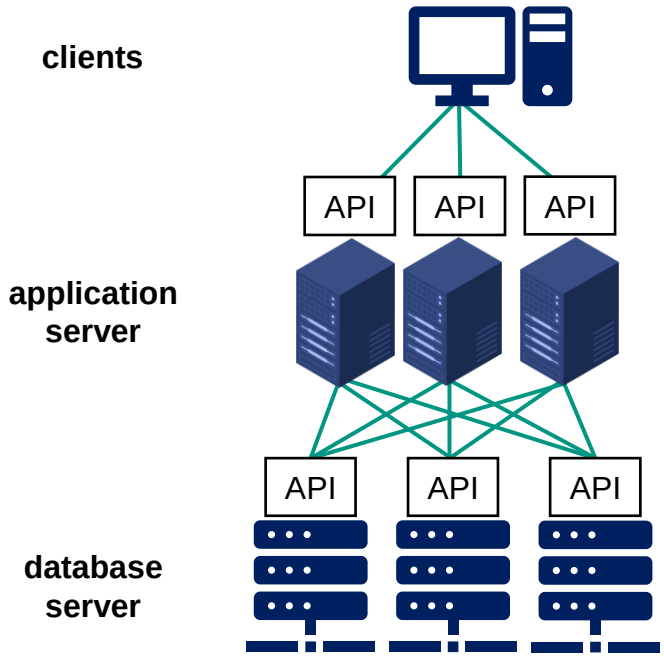
- To develop **distributed information systems** we have to deal with
  1. Data management, application logic, and presentation



Icons from Iconfinder.com

Distributed Information Systems

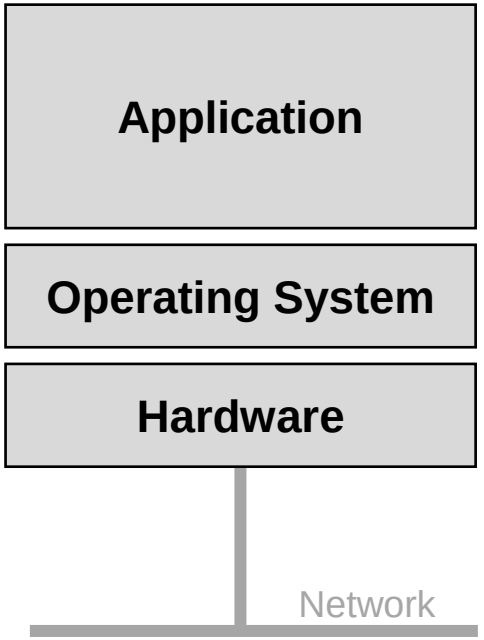
- To develop **distributed information systems** we have to deal with
  1. Data management, application logic, and presentation
  2. Network communication, resource management, fault handling, data consistency, security, etc.



Icons from Iconfinder.com

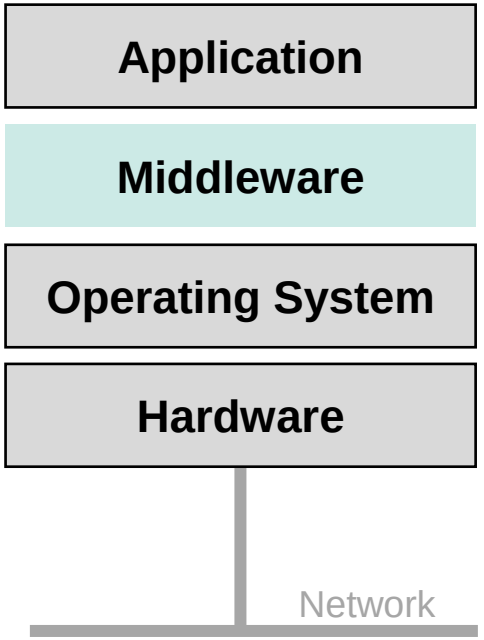
Distributed Information Systems

- To develop **distributed information systems** we have to deal with
  1. Data management, application logic, and presentation
  2. Network communication, resource management, fault handling, data consistency, security, etc.
- **Huge gap** between system to be developed and services provided by the operating system (OS)



Distributed Information Systems

- To develop **distributed information systems** we have to deal with
  1. Data management, application logic, and presentation
  2. Network communication, resource management, fault handling, data consistency, security, etc.
- **Huge gap** between system to be developed and services provided by the operating system (OS)
- **Middleware closes gap** between OS and (distributed) application



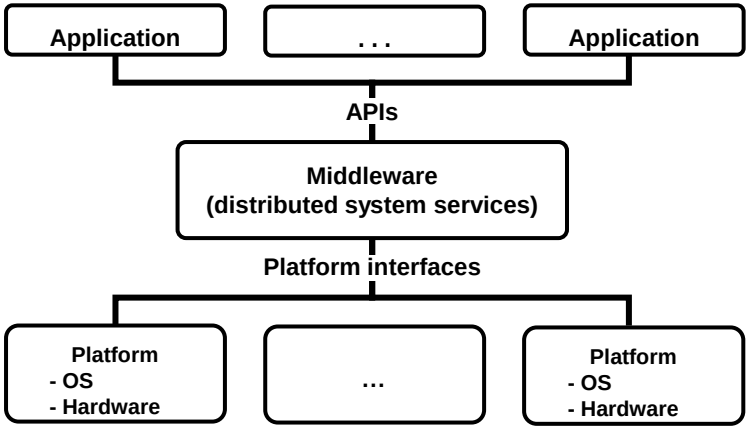
What is “Middleware”?

Image source: [Wood glue work] by Mai Ulrike, January 11th 2013. Pixabay License.

What is “Middleware”?

Definition

Middleware is a type of software used to manage and facilitate interactions between applications across computing platforms.



Middleware in a distributed system

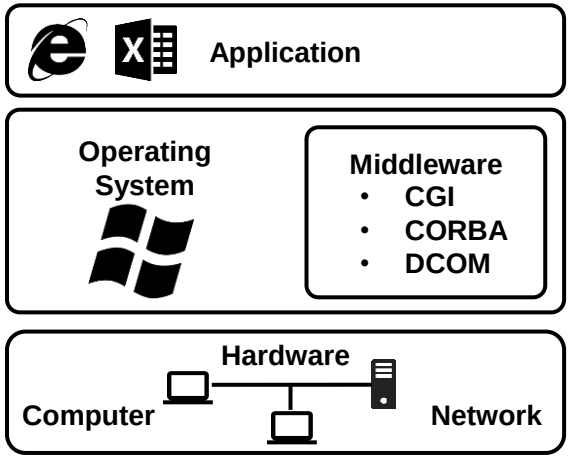
Image source: Adapted from Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Communications of the ACM*, *39*(2), 86-98.

What is “Middleware”?

Definition

Middleware is a type of software used to manage and facilitate interactions between applications across computing platforms.

- Within a layer model, "middleware" is an open (incomplete) software layer between (network) operating system and application level

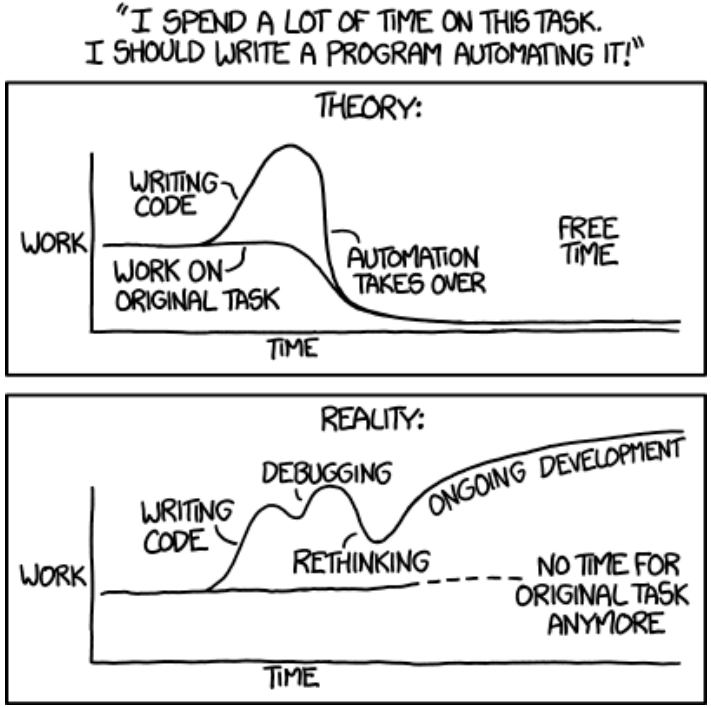


Middleware in a layer model

What is “Middleware”?

- The term middleware tends to be associated with a specific service area and the competing technologies:
  - Common Object Request Broker Architecture (CORBA)
  - Distributed Component Object Model (DCOM)
  - Remote Method Invocation (RMI)
- Middleware enables the distribution of applications to multiple computers in the network
- Special type of middleware (or extension of the middleware concept): Web services (next lecture, today we focus on conventional types of middleware systems)
- Middleware has a different meaning in other contexts
  →Example: Video game development: Middleware may refer to subsystems for areas like game physics
- Different middleware systems provide different abstractions (remote procedure call, distributed objects, message queues, or Web services)

Principle of Parsimony



Remote Procedure Call

Applied Informatics and Formal Description Methods (AIFB)
Critical Information Infrastructures (cii)

Remote Procedure Call

Definition

A remote procedure call is the synchronous *language-level transfer of control* between programs in *disjoint address spaces* whose primary communication medium is a narrow channel.

Nelson, 1981

Remote Procedure Call

- The Remote Procedure Call (RPC) is the most basic ‘type‘ of middleware
- RPC allows functions to be called from other address spaces (frequently on another computer on a shared network)
- Usually, the called functions and the calling program are not executed on the same computer
- The general idea of a procedure call fits the request–response pattern

  →The client makes a request to some external application, sleeps, and finds the result after control is returned to the procedure

Remote Procedure Call: Functional Overview

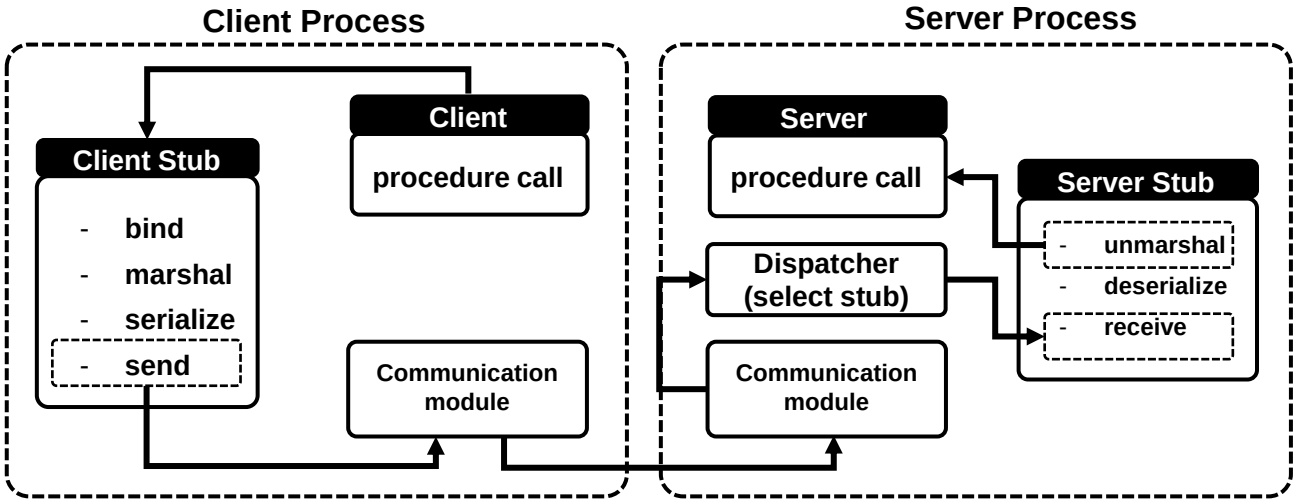


Image source: Adapted from Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). Web services. In *Web Services* (pp. 123-149). Springer, Berlin, Heidelberg.

How Procedure Calls Work

- To **call a procedure** …

application code

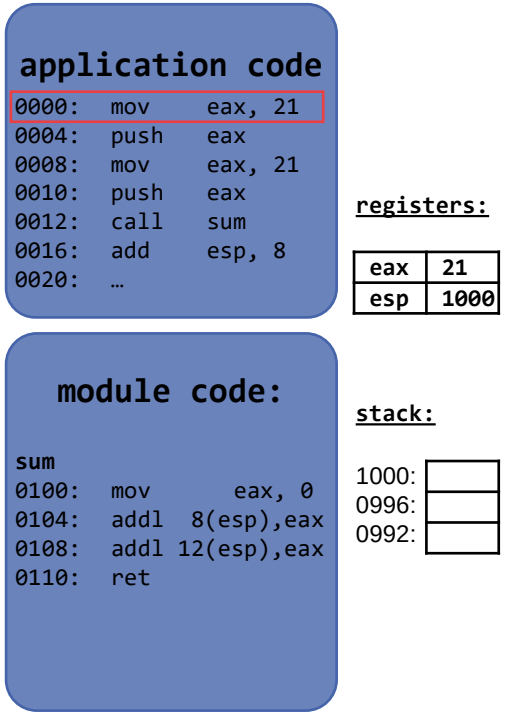
```
0000:  mov    eax, 21
0004:  push   eax
0008:  mov    eax, 21
0010:  push   eax
0012:  call   sum
0016:  add    esp, 8
0020:  …
```

module code:

```
sum
0100:  mov      eax, 0
0104:  addl  8(esp),eax
0108:  addl 12(esp),eax
0110:  ret
```

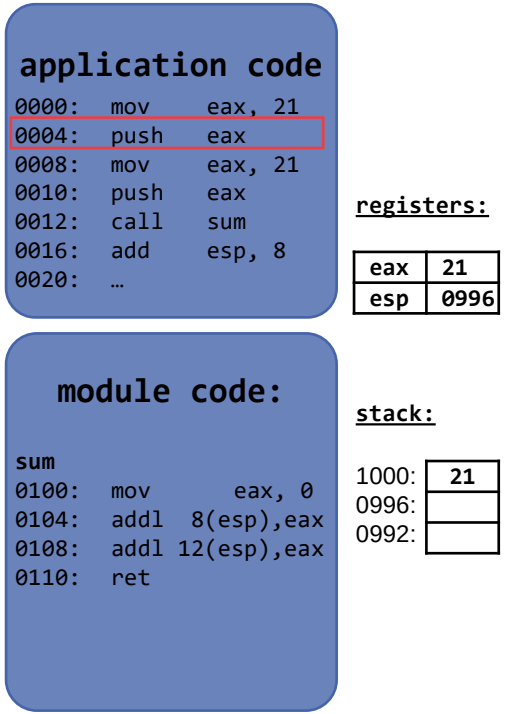
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack



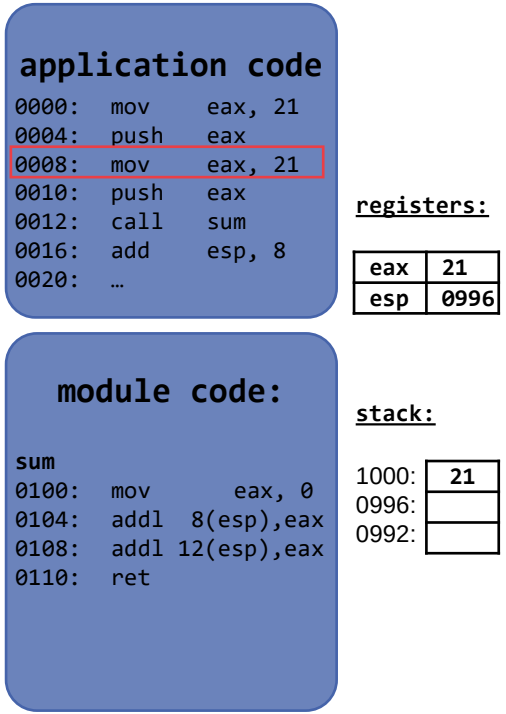
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack



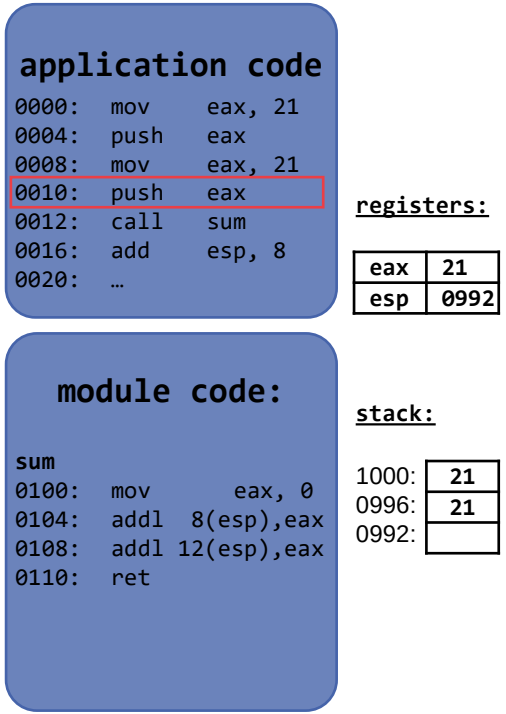
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack



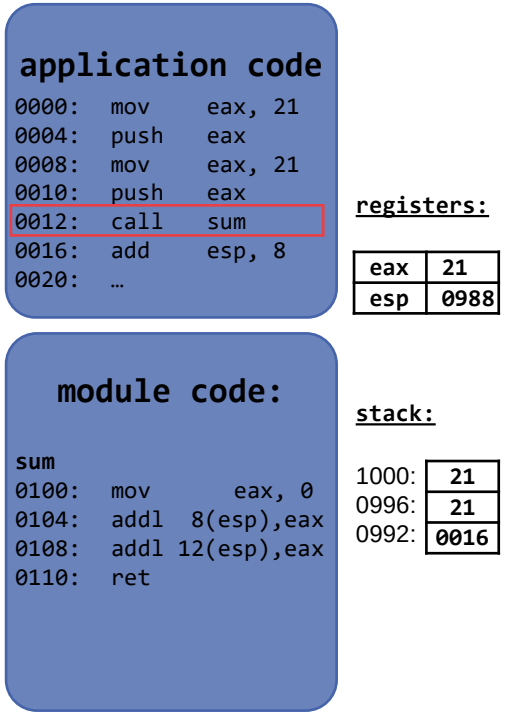
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack



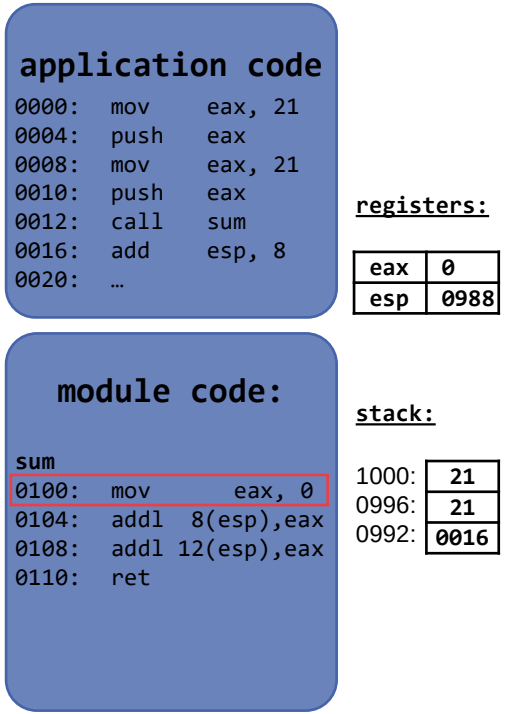
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee



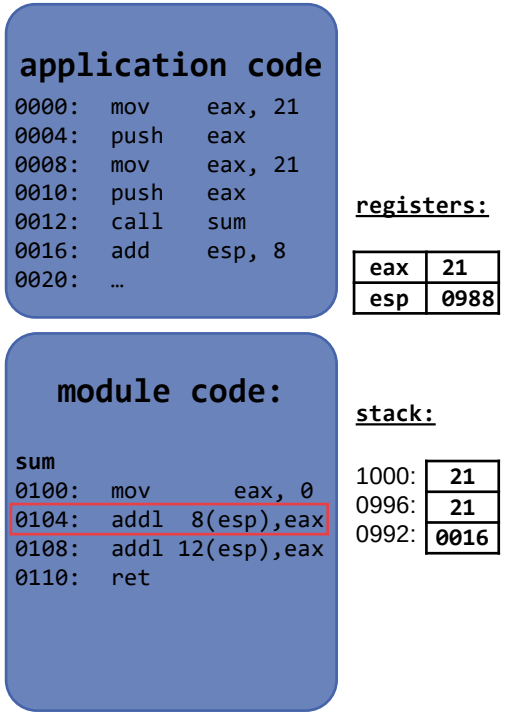
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers



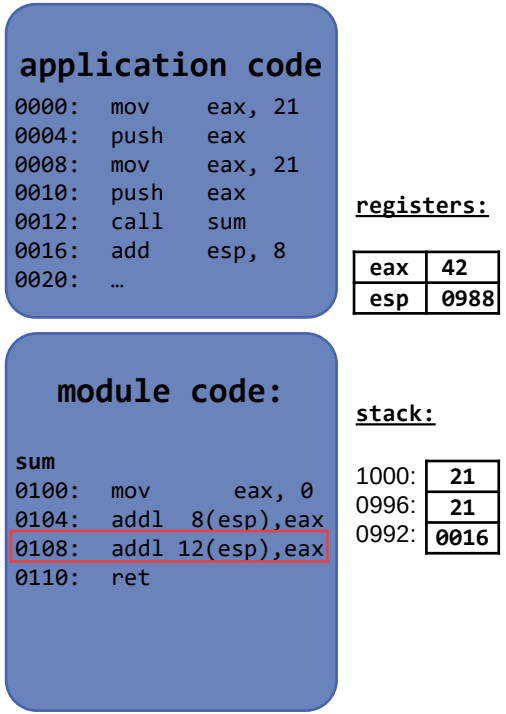
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers



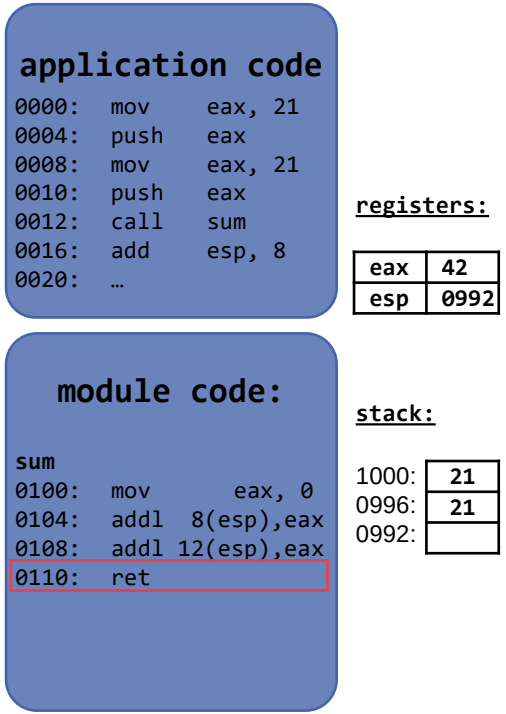
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers



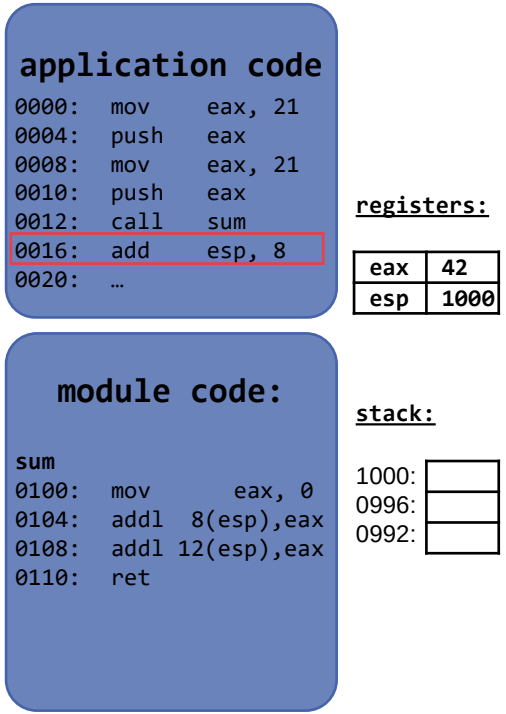
How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers
  4. Callee returns control to caller



How Procedure Calls Work

- To **call a procedure …**
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers
  4. Callee returns control to caller
- **Technical details** of procedure calls are managed by **compilers**



How Procedure Calls Work

- To **call a procedure** …
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers
  4. Callee returns control to caller
- **Technical details** of procedure calls are managed by **compilers**
- Fits **request-response pattern**
  1. Call procedure and sleep (request)
  2. Wake up and find result (response)

application code

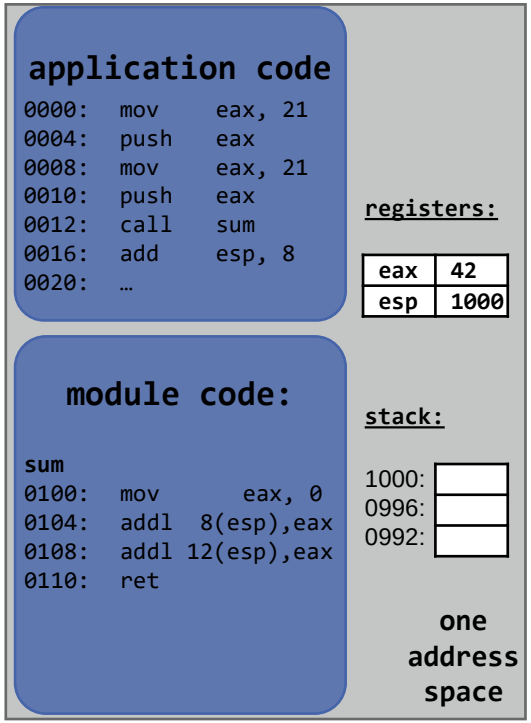
```
0000:  mov    eax, 21
0004:  push   eax
0008:  mov    eax, 21
0010:  push   eax
0012:  call   sum
0016:  add    esp, 8
0020:  …
```

module code:

```
sum
0100:  mov      eax, 0
0104:  addl  8(esp),eax
0108:  addl 12(esp),eax
0110:  ret
```

How Procedure Calls Work

- To **call a procedure** …
  1. Caller pushes arguments onto stack
  2. Caller pushes return address onto stack and transfers control to callee
  3. Callee performs task and stores result in registers
  4. Callee returns control to caller
- **Technical details** of procedure calls are managed by **compilers**
- Fits **request-response pattern**
  1. Call procedure and sleep (request)
  2. Wake up and find result (response)
- **But**: Limited to single address space



Remote Procedure Call: Local vs. Remote

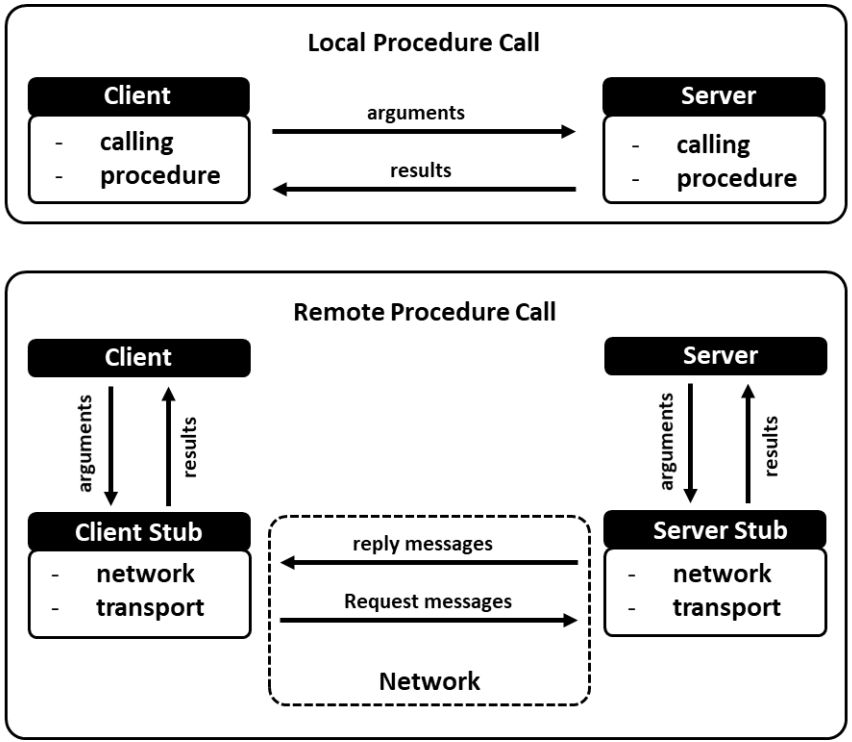
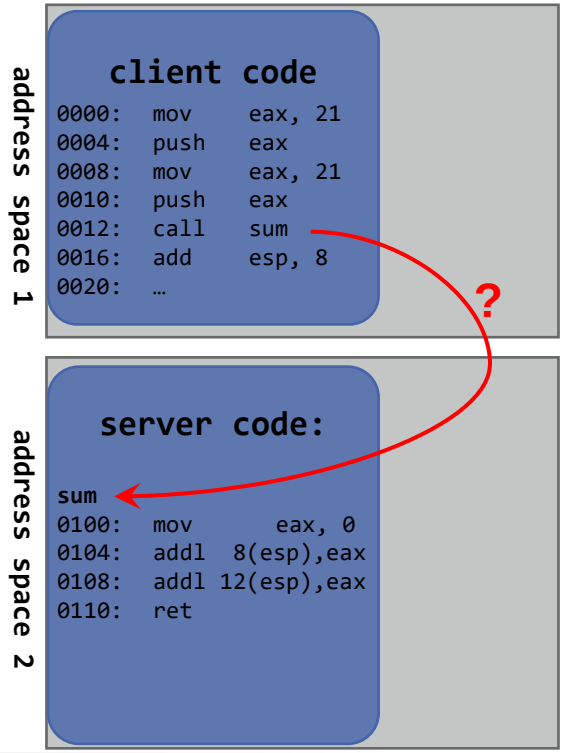


Image source: Adapted from Muppidi, S., Krawetz, N., Beedubail, G., Marti, W., & Pooch, U. (1996). Distributed computing environment (DCE) porting tool. In *Distributed Platforms* (pp. 115-129). Springer, Boston, MA.

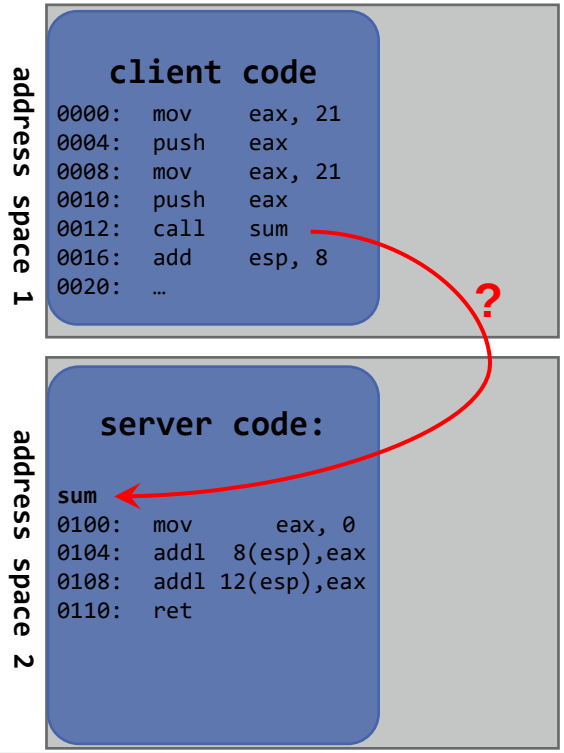
Remote Procedure Call (RPC)

- Idea: Generalize procedure calls to **remote procedures**



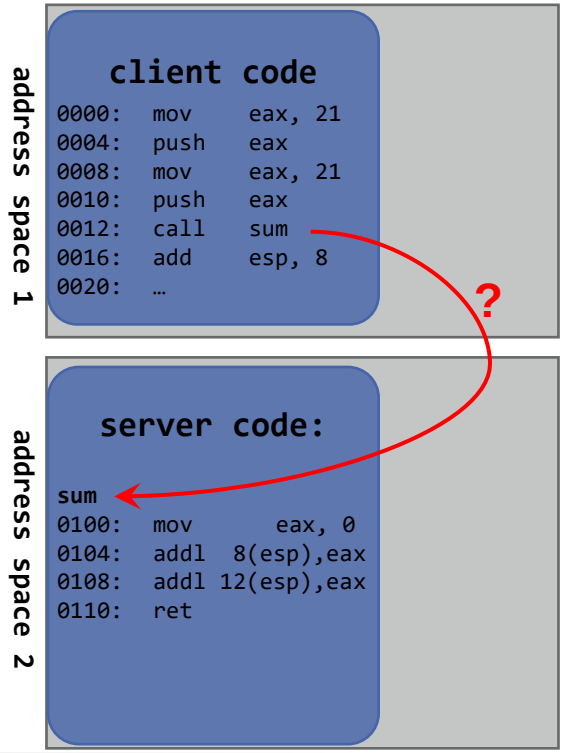
Remote Procedure Call (RPC)

- Idea: Generalize procedure calls to **remote procedures**
- Remote = different address space, i.e.
  - **different process** on same machine
  - or different process on **different machine**
    - → Inter Process Communication



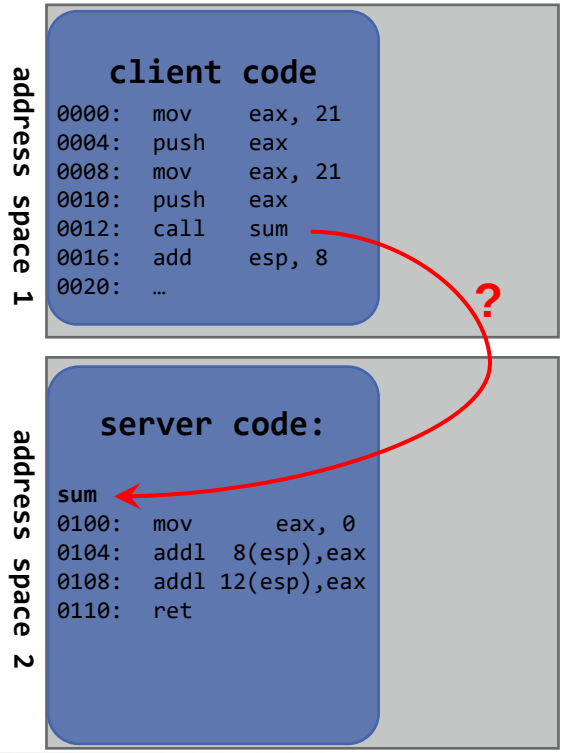
Remote Procedure Call (RPC)

- Idea: Generalize procedure calls to **remote procedures**
- Remote = different address space, i.e.
  - **different process** on same machine
  - or different process on **different machine**
    - → Inter Process Communication
- Idea dates back to 1970s, first implementations in early 1980s



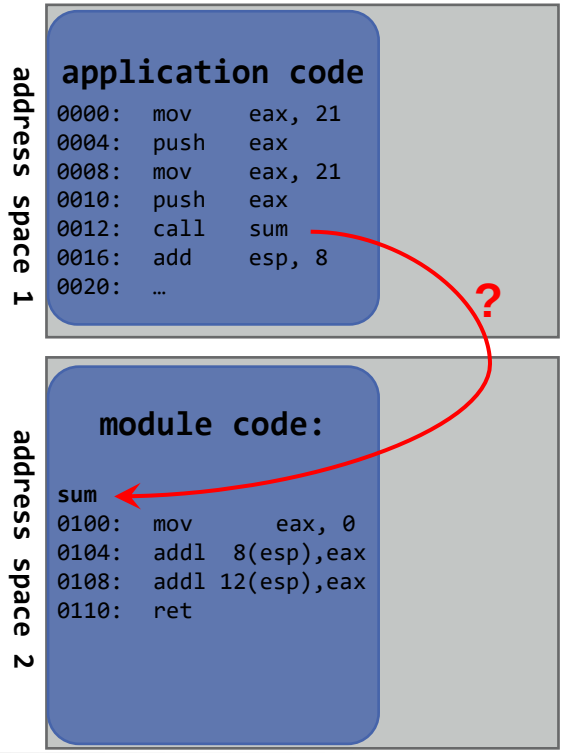
Remote Procedure Call (RPC)

- Idea: Generalize procedure calls to **remote procedures**
- Remote = different address space, i.e.
  - **different process** on same machine
  - or different process on **different machine**
    - → Inter Process Communication
- Idea dates back to 1970s, first implementations in early 1980s
- Used to be **de-facto standard** for the development of distributed systems



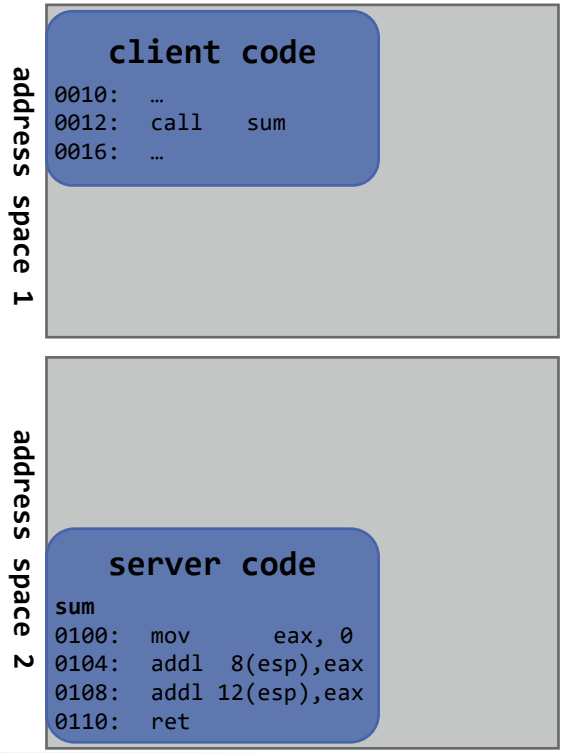
Remote Procedure Call (RPC)

- Idea: Generalize procedure calls to **remote procedures**
- Remote = different address space, i.e.
  - **different process** on same machine
  - or different process on **different machine**
    - → Inter Process Communication
- Idea dates back to 1970s, first implementations in early 1980s
- Used to be **de-facto standard** for the development of distributed systems
- Examples: SUN RPC, DCE/RPC, Java RMI, .NET Remoting, DCOM



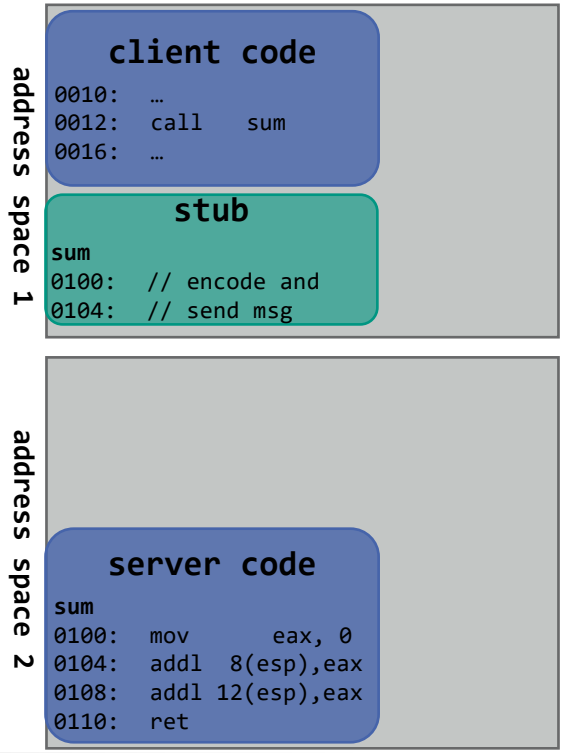
How Remote Procedure Calls Work

- Idea: Embed additional code in address space of client and server
- **Client stub**
  1. Can be called via regular procedure call
  2. **Marshals arguments** and sends request message to server
- **Server stub**
  1. Receives request message and unmarshals arguments
  2. Calls server procedure
  3. **Marshals results** and sends response message to client
- Stub code can be **generated automatically**



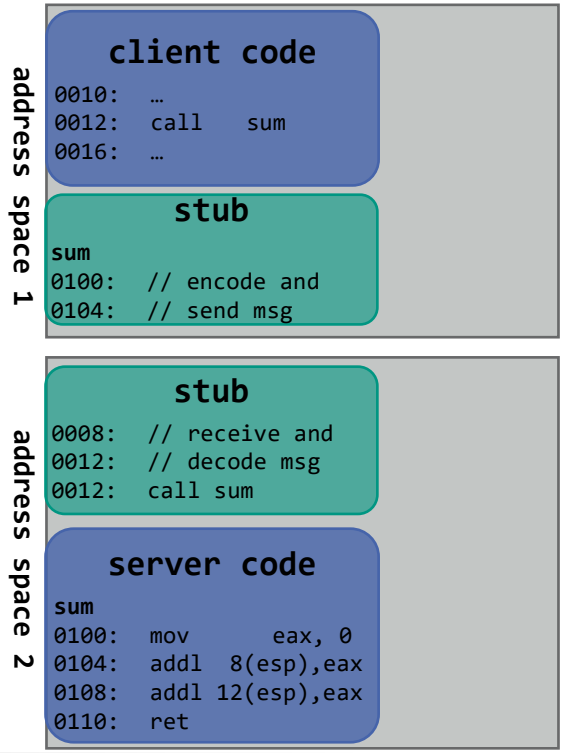
How Remote Procedure Calls Work

- Idea: Embed additional code in address space of client and server
- **Client stub**
  1. Can be called via regular procedure call
  2. **Marshals arguments** and sends request message to server
- **Server stub**
  1. Receives request message and unmarshals arguments
  2. Calls server procedure
  3. **Marshals results** and sends response message to client
- Stub code can be **generated automatically**



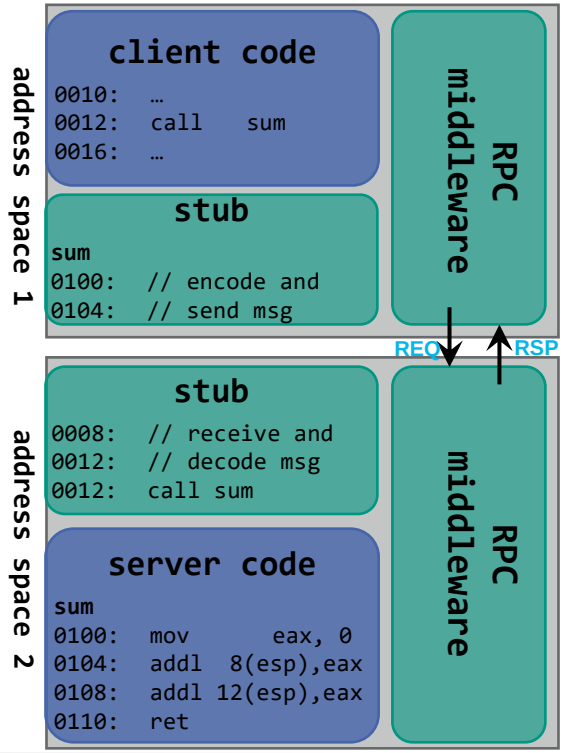
How Remote Procedure Calls Work

- Idea: Embed additional code in address space of client and server
- **Client stub**
  1. Can be called via regular procedure call
  2. **Marshals arguments** and sends request message to server
- **Server stub**
  1. Receives request message and unmarshals arguments
  2. Calls server procedure
  3. **Marshals results** and sends response message to client
- Stub code can be **generated automatically**



How Remote Procedure Calls Work

- Idea: Embed additional code in address space of client and server
- **Client stub**
  1. Can be called via regular procedure call
  2. **Marshals arguments** and sends request message to server
- **Server stub**
  1. Receives request message and unmarshals arguments
  2. Calls server procedure
  3. **Marshals results** and sends response message to client
- Stub code can be **generated automatically**



Remote Procedure Call: Functional Overview

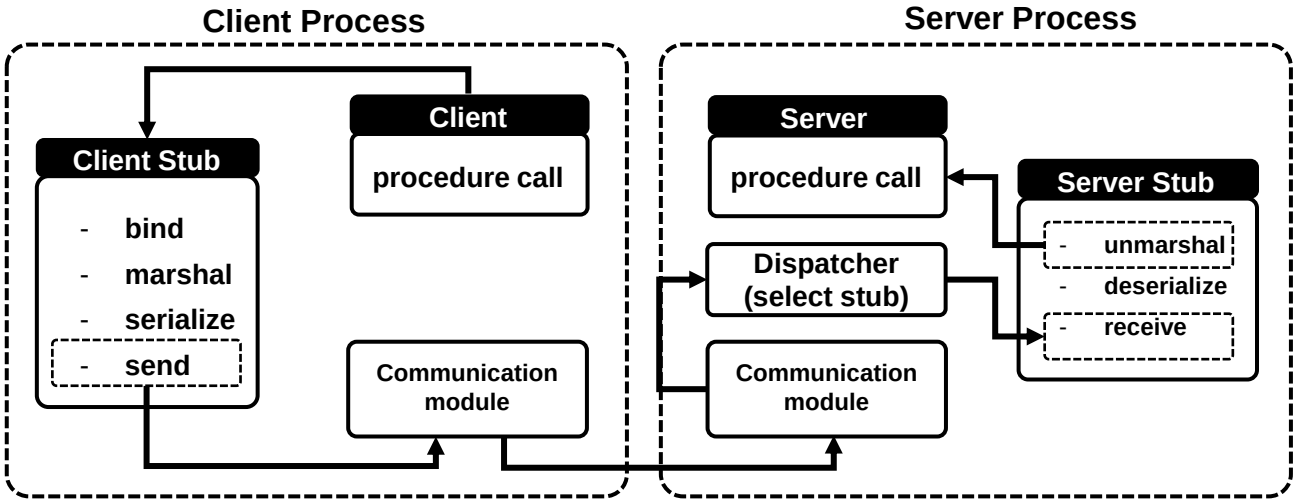


Image source: Adapted from Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). Web services. In *Web Services* (pp. 123-149). Springer, Berlin, Heidelberg.

Marshalling

Problem:

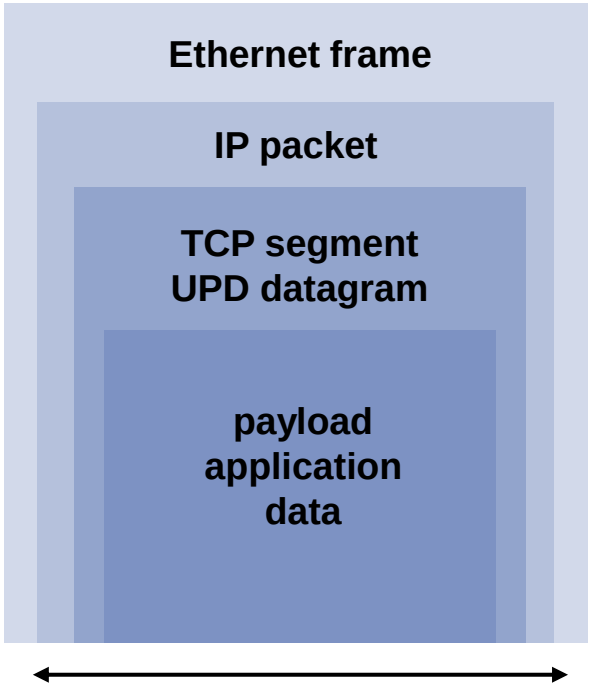
- Arguments and return values of procedures need to be **serialized**

Image source: [Rangierbahnhof Hagen-Vorhalle] by Dr. G. Schmitz., August 30th 2008. Licensed under CC BY-SA 3.0.

Marshalling

Problem: Arguments and return values of procedures need to be **serialized**.

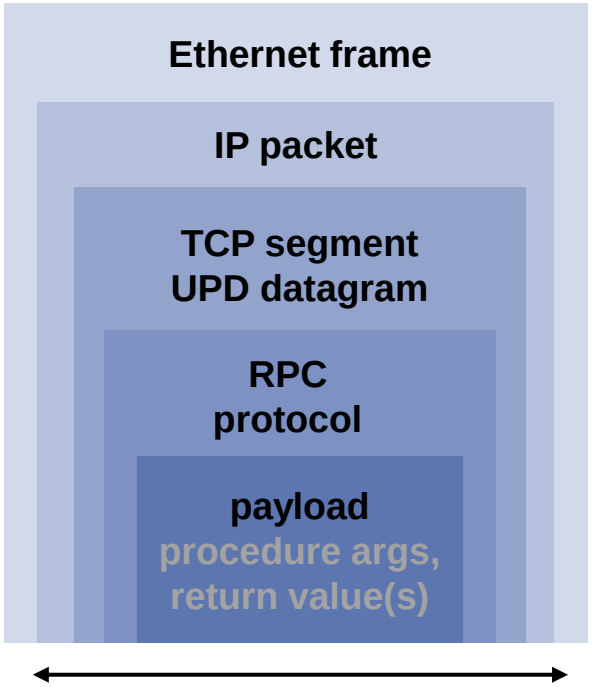
- RPC defines **application-layer protocol** above transport layer
  1. **Format and sequence** of messages (requests, response, binding, etc.)
  2. Mapping to **transport protocol(s)**



Marshalling

Problem: Arguments and return values of procedures need to be **serialized**.

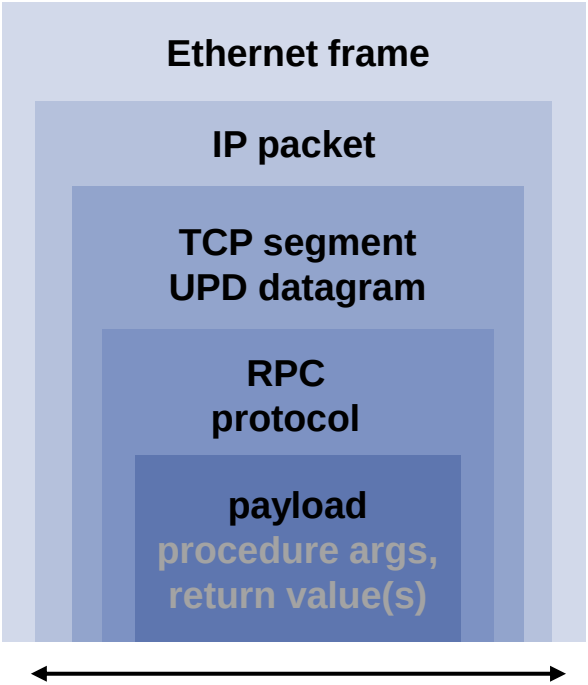
- RPC defines **application-layer protocol** above transport layer
  1. **Format and sequence** of messages (requests, response, binding, etc.)
  2. Mapping to **transport protocol(s)**
- Additionally, specifies **encoding of language types** to message payload
- Examples: XDR, XML, …



Marshalling

Problem: Arguments and return values of procedures need to be **serialized**.

- RPC defines **application-layer protocol** above transport layer
  1. **Format and sequence** of messages (requests, response, binding, etc.)
  2. Mapping to **transport protocol(s)**
- Additionally, specifies **encoding of language types** to message payload
- Examples: XDR, XML, …
- Standard protocols and encodings are key for **interoperability** between different middleware systems



Remote Procedure Call: Functional Overview

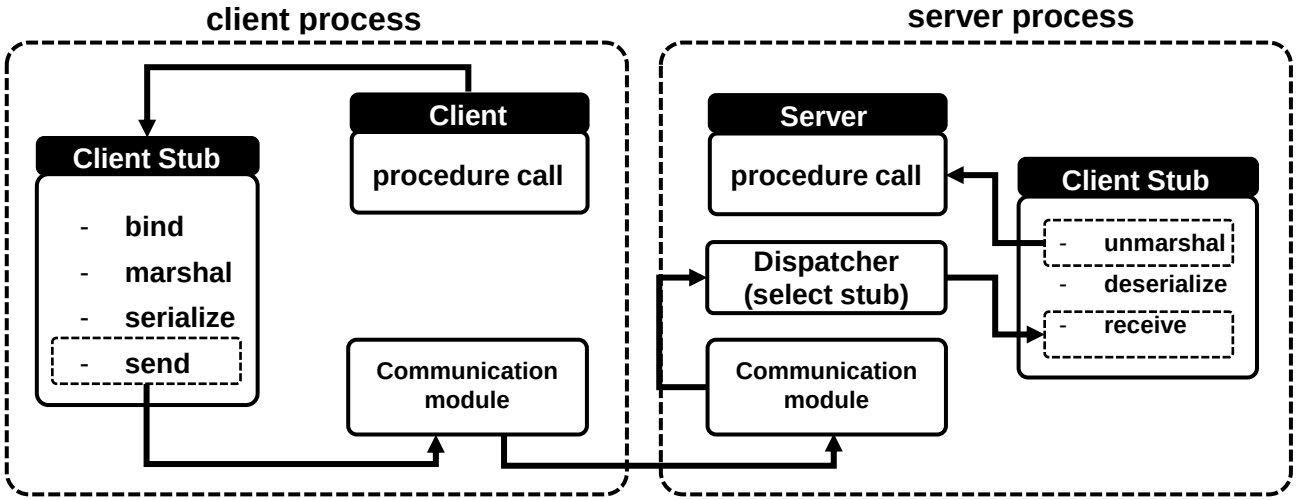


Image source: Adapted from Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). Web services. In *Web Services* (pp. 123-149). Springer, Berlin, Heidelberg.

Binding

Problem:

- Client must specify **to which server** it wants to bind

Image source: [Old Ski Set] Gerhard G., October 13th 2013. Pixabay License.

Binding

Problem: Client must specify **to which server** it wants to bind.

Static Binding

- Client **statically bound** to address of server

Dynamic Binding

Binding

Problem: Client must specify **to which server** it wants to bind.

Static Binding

- Client **statically bound** to address of server
- **Advantages**
  1. Simple
  2. No overhead

Dynamic Binding

Binding

Problem: Client must specify **to which server** it wants to bind.

Static Binding

- Client **statically bound** to address of server
- **Advantages**
  1. Simple
  2. No overhead
- **Disadvantages**
  1. No load balancing
  2. No failover

Dynamic Binding

Binding

Problem: Client must specify **to which server** it wants to bind.

Static Binding

- Client **statically bound** to address of server
- **Advantages**
  1. Simple
  2. No overhead
- **Disadvantages**
  1. No load balancing
  2. No failover

Dynamic Binding

- Client **dynamically locates** server before (first) call

Binding

Problem: Client must specify **to which server** it wants to bind.

Static Binding

- Client **statically bound** to address of server
- **Advantages**
  1. Simple
  2. No overhead
- **Disadvantages**
  1. No load balancing
  2. No failover

Dynamic Binding

- Client **dynamically locates** server before (first) call
- **Advantages**
  1. Enables load balancing
  2. Redundant servers provide failover

Binding

Problem: Client must specify **to which server** it wants to bind.

Static Binding

- Client **statically bound** to address of server
- **Advantages**
  1. Simple
  2. No overhead
- **Disadvantages**
  1. No load balancing
  2. No failover

Dynamic Binding

- Client **dynamically locates** server before (first) call
- **Advantages**
  1. Enables load balancing
  2. Redundant servers provide failover
- **Disadvantages**
  1. Requires additional service
  2. Overhead of lookup

Portmapping and Dispatching

Problem:

- Requests need to be **dispatched to correct process and procedure**

Image source: Junge, Peter Heinz (1953). *Berlin, Postamt O 17, Päckchenverteilung* [Photograph]. Allgemeiner Deutscher Nachrichtendienst.

Portmapping and Dispatching

Problem: Requests need to be **dispatched to correct process and procedure.**

- Q: How does the client know the **port of the server process**?
- **Static binding to server port**?

```
$ rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    4   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100003    4   tcp   2049  nfs
    100024    1   udp  32770  status
    100021    1   udp  32770  nlockmgr
    100021    3   udp  32770  nlockmgr
    100021    4   udp  32770  nlockmgr
    100024    1   tcp  32769  status
    100021    1   tcp  32769  nlockmgr
    100021    3   tcp  32769  nlockmgr
    100021    4   tcp  32769  nlockmgr
    100005    1   udp    644  mountd
    100005    1   tcp    645  mountd
    100005    2   udp    644  mountd
    100005    2   tcp    645  mountd
    100005    3   udp    644  mountd
    100005    3   tcp    645  mountd
```

Portmapping and Dispatching

Problem: Requests need to be **dispatched to correct process and procedure.**

- Q: How does the client know the **port of the server process**?
- **Static binding to server port**?
- Example: **Portmapping** in SUN RPC
  - Client specifies **address, program and version number** of server
  - Portmapper returns port of server process

```
$ rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    4   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100003    4   tcp   2049  nfs
    100024    1   udp  32770  status
    100021    1   udp  32770  nlockmgr
    100021    3   udp  32770  nlockmgr
    100021    4   udp  32770  nlockmgr
    100024    1   tcp  32769  status
    100021    1   tcp  32769  nlockmgr
    100021    3   tcp  32769  nlockmgr
    100021    4   tcp  32769  nlockmgr
    100005    1   udp    644  mountd
    100005    1   tcp    645  mountd
    100005    2   udp    644  mountd
    100005    2   tcp    645  mountd
    100005    3   udp    644  mountd
    100005    3   tcp    645  mountd
```

Portmapping and Dispatching

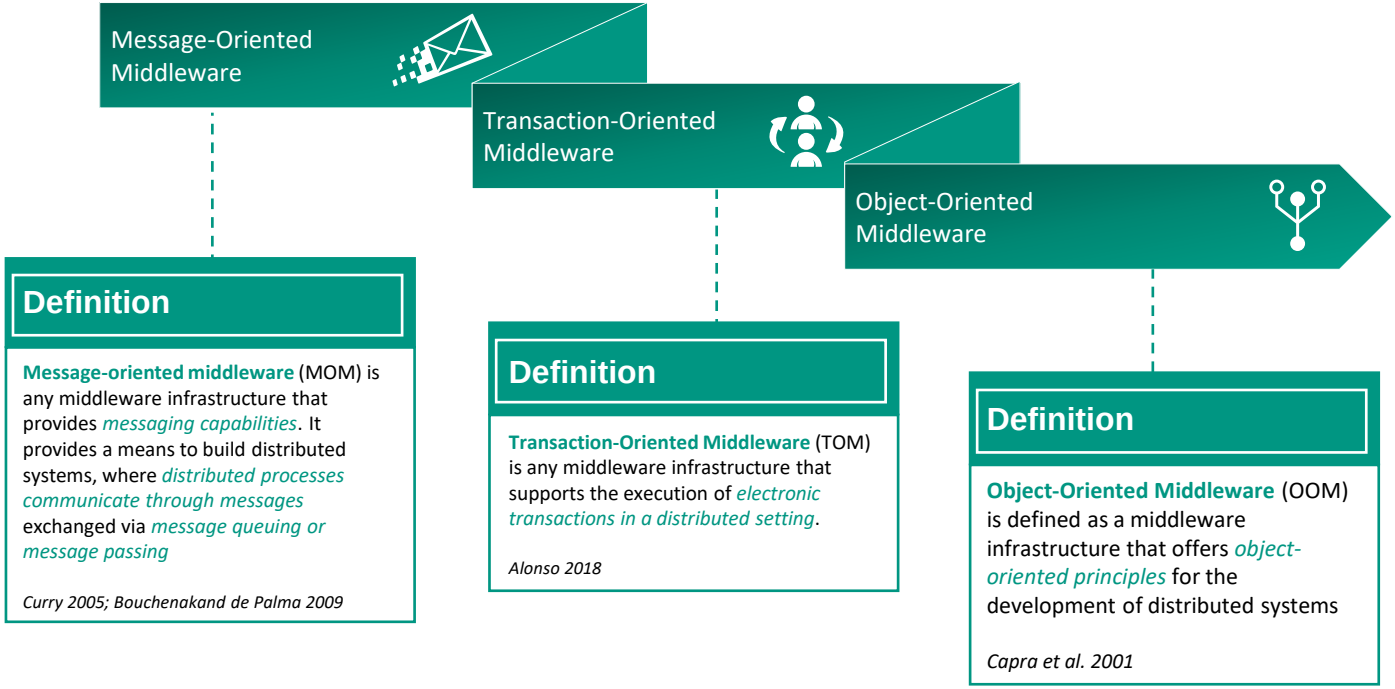
Problem: Requests need to be **dispatched to correct process and procedure.**

- Q: How does the client know the **port of the server process**?
- **Static binding to server port**?
- Example: **Portmapping** in SUN RPC
  - Client specifies **address, program, and version number** of server
  - Portmapper returns port of server process
- Server stub **dispatches request message** to correct method implementation

```
$ rpcinfo -p
 program vers proto   port
  100000    2   tcp    111  portmapper
  100000    2   udp    111  portmapper
  100003    2   udp   2049  nfs
  100003    3   udp   2049  nfs
  100003    4   udp   2049  nfs
  100003    2   tcp   2049  nfs
  100003    3   tcp   2049  nfs
  100003    4   tcp   2049  nfs
  100024    1   udp  32770  status
  100021    1   udp  32770  nlockmgr
  100021    3   udp  32770  nlockmgr
  100021    4   udp  32770  nlockmgr
  100024    1   tcp  32769  status
  100021    1   tcp  32769  nlockmgr
  100021    3   tcp  32769  nlockmgr
  100021    4   tcp  32769  nlockmgr
  100005    1   udp    644  mountd
  100005    1   tcp    645  mountd
  100005    2   udp    644  mountd
  100005    2   tcp    645  mountd
  100005    3   udp    644  mountd
  100005    3   tcp    645  mountd
```

Middleware Categories

Types of Middleware



Message-Oriented Middleware (MOM)

Applied Informatics and Formal Description Methods (AIFB)
Critical Information Infrastructures (cii)

Message-Oriented Middleware (MOM)

Definition

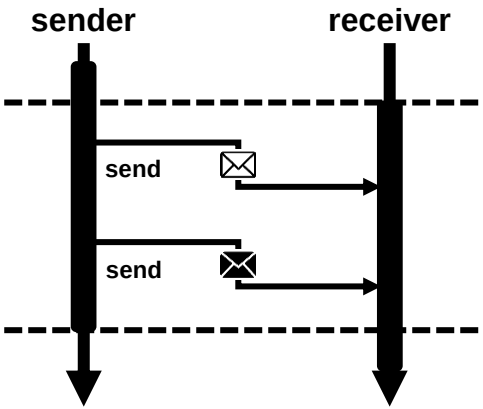
Message-oriented middleware (MOM) is any middleware infrastructure that provides *messaging capabilities*. It provides a means to build distributed systems, where *distributed processes communicate through messages* exchanged via *message queuing or message passing*.

Curry; Bouchenak and de Palma

- Messages are exchanged **peer to peer** (P2P)—mostly in XML format
- Based on queuing software
- Message queue as an intermediary
  - Message queue accepts messages to different programs; addressed programs are activated as needed, answers can be fetched later from the queue
  - Queue manager can optimize the performance (e.g., prioritization, load allocation)
  - Queue manager utilizes persistence and transaction support to protect against loss

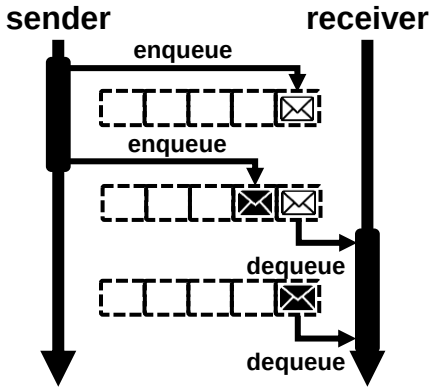
Source: Curry, E. (2005). Message-oriented middleware. In: Mahmoud Q (ed) Middleware for communications. Wiley, Chichester, pp 1–29
Bouchenak, S. & de Palma, N. (2009). Message queuing systems. In: Liu, L. & Özsu, M.T. (eds). Encyclopedia of database systems. Springer, Boston, MA, pp 1716–1717

Message Passing vs. Message Queuing



- Message passing refers to transient communication between two processes that are active at the same time
- Example: asynchronous RPC

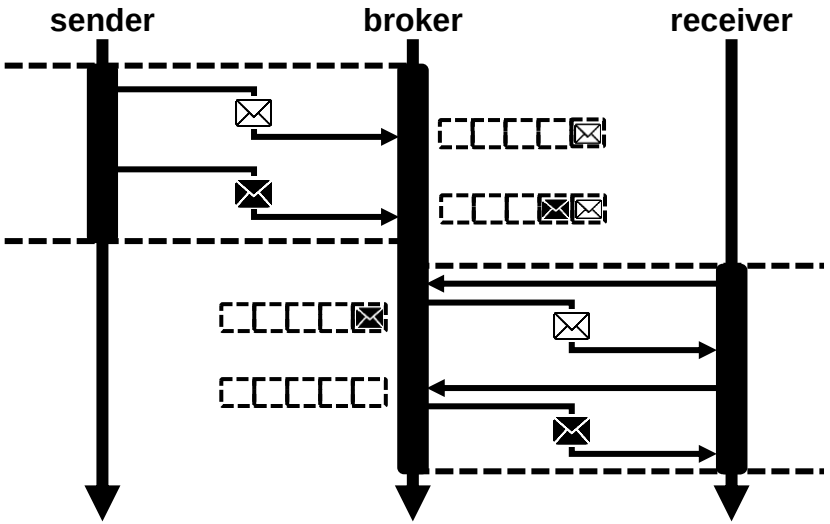
Both models eventually pass the message, the **difference is the obligation for the processes to be active at the same time for message passing**



- The message-queuing model requires an additional intermediary component, which is called the message queue (essentially a mailbox independent of the receiver)

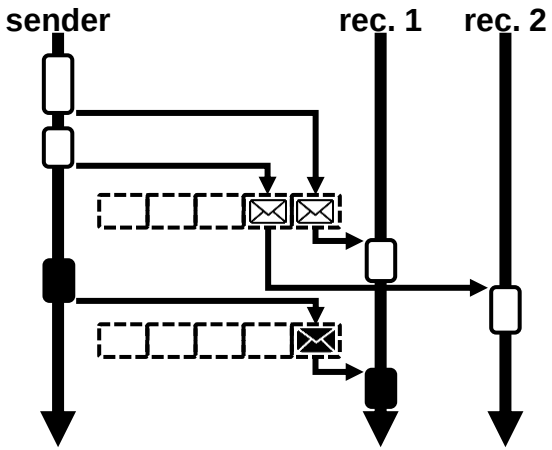
Message Broker

- For handling the routing requests of messages, MOM uses a **message broker** which implements message queues:

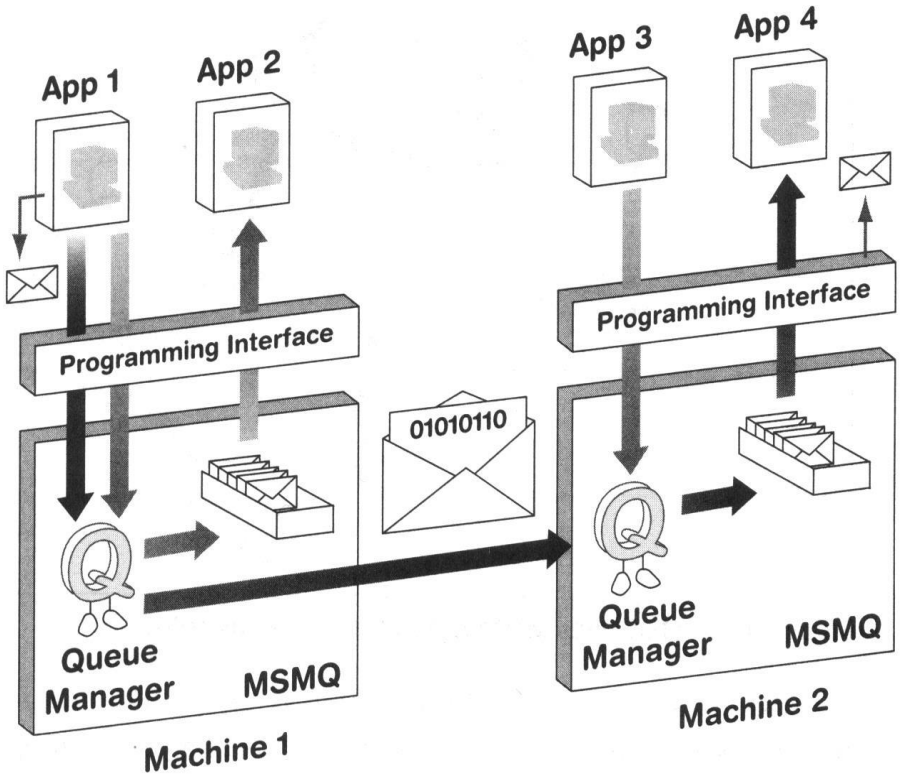


Message Broker

- **Sender** does not need to be aware on which system the **server** is running
- Receiver **can change location** during the runtime—location dependency is reduced
- Option to change to multicast or broadcast system (not deleting message after first deque operation)
  - So there can be **multiple receivers**:



Message Queuing



Linthicum, Fig. 9.5

Image source: Linthicum, D.S. (2000). Enterprise Application Integration. Addison-Wesley, Boston. pp. 171

Message-Oriented Middleware (MOM)

- Two different models possible:
  - **Process-to-process**: Processes must be active for communication
  - **Message queuing**: Message is managed in a queue, target application and network (additional queue on client platform) do not have to be currently available
  - Can guarantee that messages reach their target application, even if the target application is not active when receiving the message
  - Requesting application can be informed when the answer is available
- Support for heterogeneity through standard API on different platforms
- Log files can be generated (communication documentation)
- Normally, the message (or the data in it) **can not be transformed**, no intelligent routing, no event-driven processing (events and the associated data must always be defined in applications)

Implementation: MSMQ (Microsoft Message Queue)

- The MSMQ server software runs on **Windows** platforms
- Dominates **common protocols** like TCP/IP (others too, is currently changing)
- Storage of the message to disk and log-based recovery (optional, since it costs performance)
- **Transaction support** (full rollback in case of failure)
- Support MSMQ administration from a workstation through MSMQ Explorer
- Can access through a **standard API**:
  - Typical Windows applications like Excel or Word
  - MTS (Microsoft Transaction Server), IIS (Internet Information Server), SQL server
  - All applications implemented with appropriate development environments that provide them with the standard API

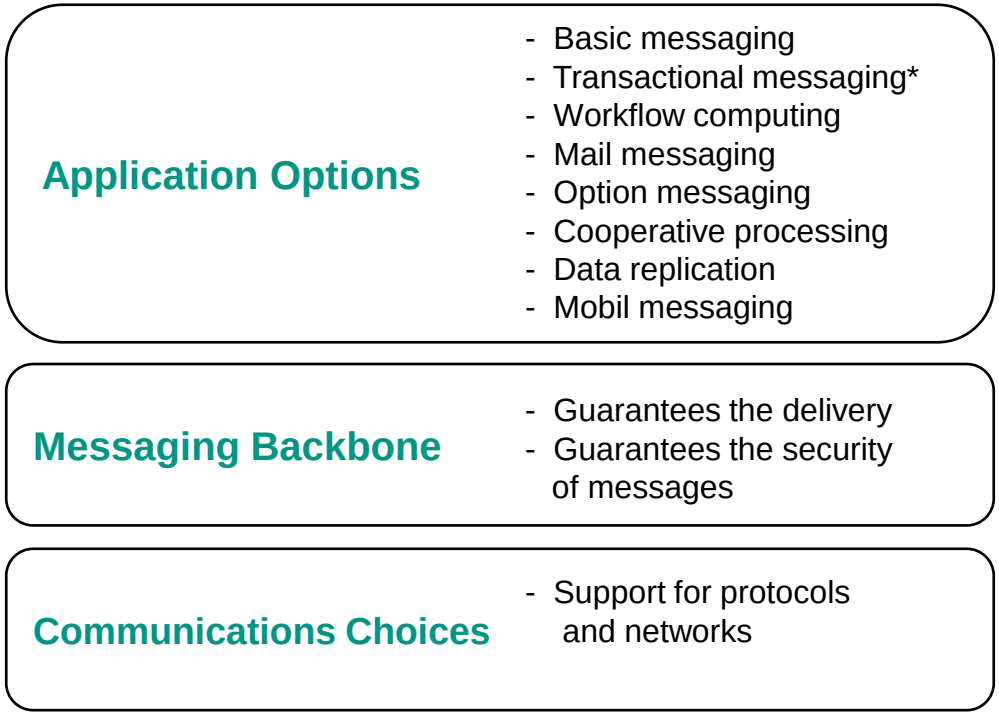
MSMQ (Microsoft Message Queue)

- **Features**
  - One-time, in-order delivery
  - Message-routing services (last cost routing)
  - Notification services (incidents: message received, message processed, time out)
  - Message priorities
  - Journaling (journal is a list of copies of messages; journals can selectively contain specific messages or all messages or all messages of a queue, developers use journals for recovery in case of failure)
- **Commitment**
  - On each participating computer a queue manager
  - If the target application is on a different computer, the queue managers communicate with each other via the queues and the messages
- **Interface**
  - Initially generated with MQOpenQueue queue handle for a specific target
  - Then messages are put into the queue with MQSendMessage
  - Finally, with MQCloseQueue, the application can close the queue

IBM WebSphere MQ (MQSeries)

- **Supports many platforms** (OS/390, Pyramid, Open VMS, IBM AIX, NCR, UNIX, Solaris, OS/2, Windows, etc.)
  - Messages can, e.g., be sent from a Windows workstation routed through UNIX or VMS machines, eventually reaching an application under OS/ 390
- Provides a **standard API** for the various development environments and platforms (third-party vendors)
- Supports **all services of MSMQ and more**
- E.g., publish/subscribe service (advantage: an application does not need to know anything about the target applications, the pub/sub engine can provide information to all interested applications)
- Previously, known as: MQSeries

IBM WebSphere MQ



E.g., ability to update multiple databases simultaneously on different platforms

Comparison

- **MSMQ**
  - Used for simple integration of Windows applications
- **WebSphere MQ**
  - Available on almost all platforms
  - Better scalable because available on more powerful platforms
  - Higher fault tolerance, greater reliability of delivery of messages, higher security, better database access
  - Broadcasting, publish/subscribe service
  - Mature, tested
  - Disadvantage: Installation consuming
- **Guess**:
  - WebSphere MQ will continue to dominate the market for heterogeneous high-end platforms in the future
  - MSMQ will dominate the integration of Windows platforms

Transaction-Oriented Middleware (TOM)

Applied Informatics and Formal Description Methods (AIFB)
Critical Information Infrastructures (cii)

Transaction-Oriented Middleware (TOM)

Definition

Transaction-Oriented Middleware (TOM) is any middleware infrastructure that supports the **execution of electronic transactions** in a distributed setting

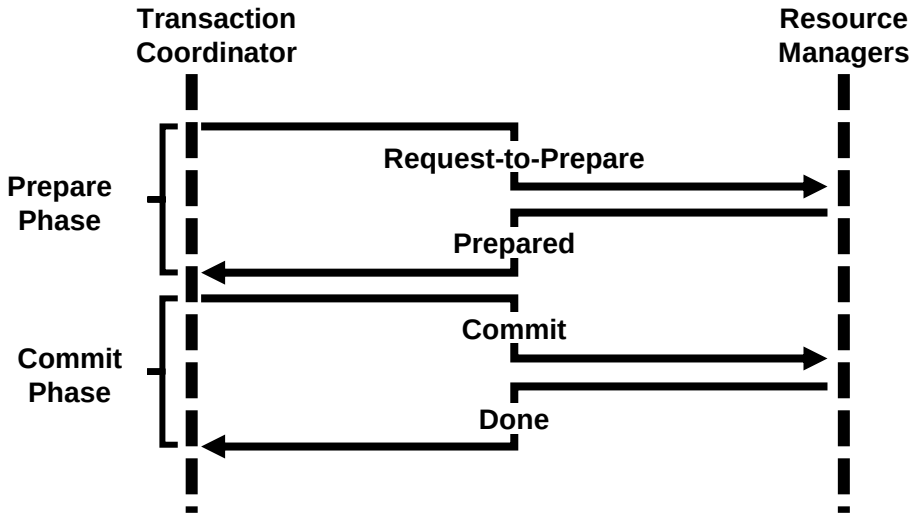
Alonso

- Supports synchronous and asynchronous communication among heterogeneous hosts
- Facilitates integration between servers and database management systems
- Concepts for transactions:
  - **Two-phase commit**
  - Classical transactions → ACID properties
    - **atomicity, consistency, isolation, durability**

Source: Alonso, G. (2018). Transactional middleware. In: Liu, L. & Özsu, M.T. (eds). Encyclopedia of databasesystems. Springer, New York, NY, pp 4201–4204

Two-Phase Commit

- Guarantees Atomicity
- Transaction coordinator can only commit once all participants approved the transaction (“prepared”)



ACID — Transactions

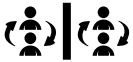
Atomicity
Each transaction is either fully executed or not at all (rollback if something fails)

A

C

Consistency
Transaction leaves database in a consistent state given it was consistent before the transaction

Isolation
Transactions cannot influence other transactions when executed parallel



I

D

Durability
Permanent storage in DB is guaranteed once a transaction is completed successfully

Components of a TP Monitor

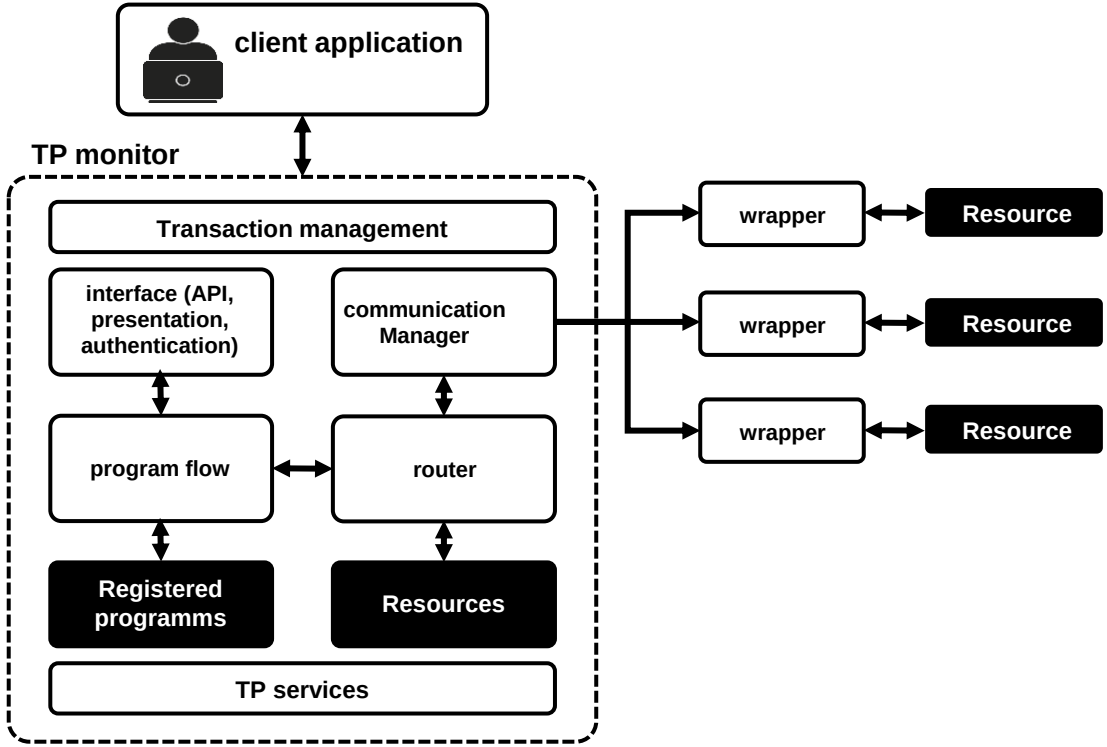


Image source: Adapted from Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). Web services. In *Web Services* (pp. 123-149). Springer, Berlin, Heidelberg.

TP Monitor

- **Products**
  - CICS (IBM, 1968)
  - Tuxedo (BEA Systems)
  - MTS (Microsoft)
- Connection via **low-level connectors** requires sophisticated development work
- Transactions can processed in parallel by several TP monitors or TP monitors processes (performance and scaling)
- Coping with peak loads by queued input buffer
- Processing priorities in the input queue
- TP products offer queuing, routing, messaging, and support the distribution of applications (these features can also be used without transaction support)

TP Lite and Heavy

TP-Lite Monitor	vs	TP-Heavy Monitor
Integrated in DB System		Supports client-server architecture
Stored procedure invoked and executed according to ACID principles		Allows for very complex transactions
Better for periodic data synchronisation		Standalone product
Oracle Tuxedo Oracle RDBMS		Oracle Tuxedo

Transaction-Oriented Middleware

Application Server

- Many similarities with TP Monitor
- But modern languages (e.g., Java, instead of C or COBOL)
- Platform for processing business logic (application)
- Connectors to backend resources (like TP Monitor)
- Support for the development of the user interface

Standards

- Transaction Processing Reference Model (1991)
- Distributed Transaction Reference Model (1994)
  - Transaction system features and their communication with other systems
  - By X/Open: http://www.opengroup.org
- XA interface
  - Set of function calls for cooperation of the Transaction Manager with the resource managers (e.g., „resting“ or „ready“ as a call for the resource manager)
  - Defines the communication between TP manager and resource manager (e.g., database)

Middleware as „Intermediate“

- Connection of **heterogeneous systems** (client and server side)
- Database Multiplexing
- Load distribution
- Prioritization
- Fault tolerance
  - Dynamic switching
  - Two-phase commit protocol
  - Repeat

Object-Oriented Middleware (OOM)

Applied Informatics and Formal Description Methods (AIFB)
Critical Information Infrastructures (cii)

Object-Oriented Middleware (OOM)

Definition

Object-Oriented Middleware (OOM) is defined as a middleware infrastructure that offers **object-oriented principles** for the development of distributed systems.

Capra et al.

- Based on **RPC mechanism**
- Also referred to as: request broker, object broker, object request broker, or distributed object middleware (Gokhale, 2009; Mahmoud, 2005; Alonso et al., 2004)
- Enables communication between objects within distributed systems
- Every OOM uses a **Interface Description Language** (IDL)
  - Declarative formal language containing a language syntax to describe a software component's interfaces
  - The IDL serves purely to describe the interface but not to formulate algorithms

Source: Capra, L., Emmerich, W., Mascolo, C. (2001). Middleware for mobile computing: awarenesss vs. transparency. Paper presented at the 8th workshop on hot topics in operating systems, Elmau, 20–22 May 2001

ORB: Implementation

- **Standard**:
  - **CORBA** (Common Object Request Broker Architecture) of the OMG (Object Management Group)
  - Implementation of the standard by the platform providers

  → Implementation of the heterogeneous variant
- **Product**:
  - **COM**/DCOM/COM+ (Distributed) **Component Object Model** by Microsoft
  - Windows Communication Foundation (WCF)—COM successor by Microsoft

  → Implementation of the homogeneous variant
  - Internet Communications Engine (ICE)

  → Implementation of the heterogeneous variant

CORBA Architecture

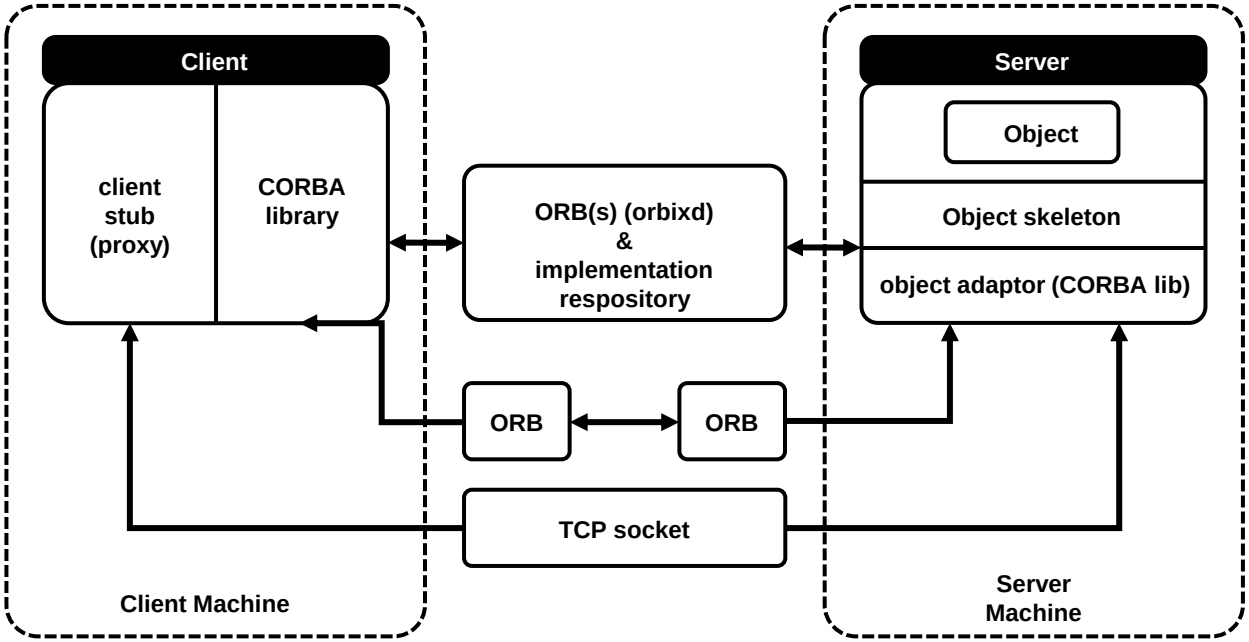


Image source: Adapted from Chung, P. E., Huang, Y., Yajnik, S., Liang, D., Shih, J. C., Wang, C. Y., & Wang, Y. M. (1998). DCOM and CORBA side by side, step by step, and layer by layer. *C++ Report*, *10*(1), 18-29.

CORBA API and Protocols

- **Dynamic Invocation Interface** (DII)
  - Allows clients to discover objects and invoke methods without a client stub
- **Dynamic Skeleton Interface** (DSI)
  - Server counterpart to DII: Services at runtime without server skeleton
    (Gokhale and Schmidt 1996)
- **Object Adapter** / Portable Object Adapter (POA)
  - Interface used by servant object (register implementation, count, references, create instances)
  - POA standardizes interface used by object implementation (“servant”)
- **General Inter-ORB Protocol** (GIOP)
  - Specification containing, e.g., how IDL data is mapped

Commercial Implementations

Component Object Request Broker Architecture

CORBA is implemented in many solutions:

- The ACE ORB (TAO) (scalable QoS)
- Visibroker (Java based)
- ORBacus (used in critical systems)

Component Object Model

Inter-process communication on **Windows** using a **component Model**.
OLE (“Object Linking and Embedding) and ActiceX are based on COM.
Replaced by .NET Remoting, which is now replaced by **WCF**

Windows Communication Foundation

Service-oriented **communication platform for distributed applications in Windows**
Combines many functions and offers a uniform Programming Interface

Distributed Component Object Model

DCOM is a **RPC System by Windows** (former OLE)
Enables COM Software to communicate in a distributed setting

Remote Method Invocation

Java’s implementation of RPC.
Objects from another JVM can be handled like local Objects.
Objects are located by an identifier (RMI URL) which can be transformed into the reference by a name service

References

- Nelson BJ (1981) Remote procedure call. Carnegie-Mellon University, Pittsburgh, PA, May 1981
- Curry E (2005) Message-oriented middleware. In: Mahmoud Q (ed) Middleware for communica-tions. Wiley, Chichester, pp 1–29
- Bouchenak S, de Palma N (2009) Message queuing systems. In: Liu L, Özsu MT (eds) Encyclo-pedia of database systems. Springer, Boston, MA, pp 1716–1717
- Alonso G (2018) Transactional middleware. In: Liu L, Özsu MT (eds) Encyclopedia of databasesystems. Springer, New York, NY, pp 4201–4204
- Capra L, Emmerich W, Mascolo C (2001) Middleware for mobile computing: awarenesss vs. transparency. Paper presented at the 8th workshop on hot topics in operating systems, Elmau, 20–22 May 2001
- Gokhale A (2009) Request broker. In: Liu L, Özsu MT (eds) Encyclopedia of database systems.Springer, Boston, MA, pp 2415–2418
- Mahmoud Q (2005) Middleware for communications. Wiley, Chichester
- Alonso G, Casati F, Kuno H, Machiraju V (2004) Web services. In: Alonso G, Casati F, Kuno H,Machiraju V (eds) Web services: concepts, architectures and applications. Data-centric systemsand applications, 1st edn. Springer, Berlin

Images

- Bernstein PA (1996) Middleware: a model for distributed system services. Commun ACM 39(2):86–98
- Muppidi S, Krawetz N, Beedubail G, Marti W, Pooch U (1996) Distributed computing environment(DCE) porting tool. In: Schill A, Mittasch C, Spaniol O, Popien C (eds) Distributed platforms.Springer, Boston, MA, pp 115–129
- Alonso G, Casati F, Kuno H, Machiraju V (2004) Web services. In: Alonso G, Casati F, Kuno H,Machiraju V (eds) Web services: concepts, architectures and applications. Data-centric systemsand applications, 1st edn. Springer, Berlin
- Emerald P, Yennun C, Yajnik S, Liang D, Shih JC, Wang C-Y, Wang Y-M (1998) DCOM andCORBA side by side, step by step, and layer by layer.http://course.ece.cmu.edu/~ece749/docs/DCOMvsCORBA.pdf. Accessed 17 Sept 2019

Questions

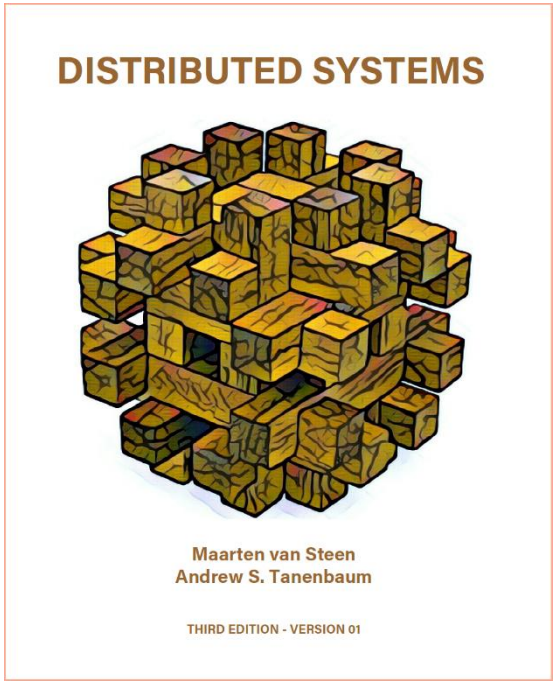
Questions

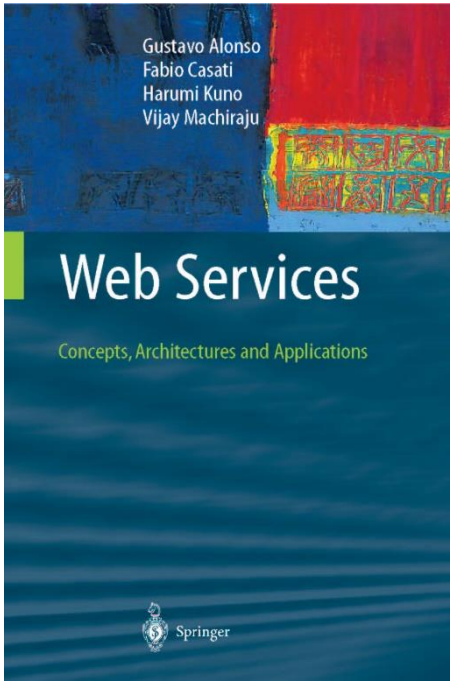
1. What is location transparency in the context of RPCs?
2. Explain step by step what happens when a client makes a remote procedure call.
3. How do existing middleware categories differ from each other?
4. What are the components of a TP monitor?
5. What is the difference between a local and a remote procedure call?
6. Why do we need middleware when various software components already offer APIs?
7. What are the most common commercial implementations of middleware?
8. What is the difference between a Web service and middleware?

Further Reading

- Alonso G, Casati F, Kuno H, Machiraju V (2004) Web services. In: Alonso G, Casati F, Kuno H, Machiraju V (eds) Web services: concepts, architectures and applications. Data-centric systems and applications, 1st edn. Springer, Berlin
- Bernstein PA (1996) Middleware: a model for distributed system services. Commun ACM 39 (2):86–98
- Curry E (2005) Message-oriented middleware. In: Mahmoud Q (ed) Middleware for communications. Wiley, Chichester, pp 1–29
- Siegel J (2000) CORBA 3: fundamentals and programming, 2nd edn. Wiley, New York, NY
- Umar A (2004) Third generation distributed computing environments. NGE Solutions, Fort Lauderdale, FL

Highlighted Literature





Source: van Steen, M. & Tanenbaum, A.S. (2018). Distibuted Systems (3.02). Van Steen M. Freely available at https://www.distributed-systems.net/index.php/books/ds3/
Source: Alonso G, Casati F, Kuno H,Machiraju V (eds) (2004). Web services: concepts, architectures and applications (1). Springer, Berlin. Available at https://link.springer.com/book/10.1007/978-3-662-10876-5