

---

---

# Communication: Layering, RPC, Sockets, MOM

10 November 2024  
Lecture 2

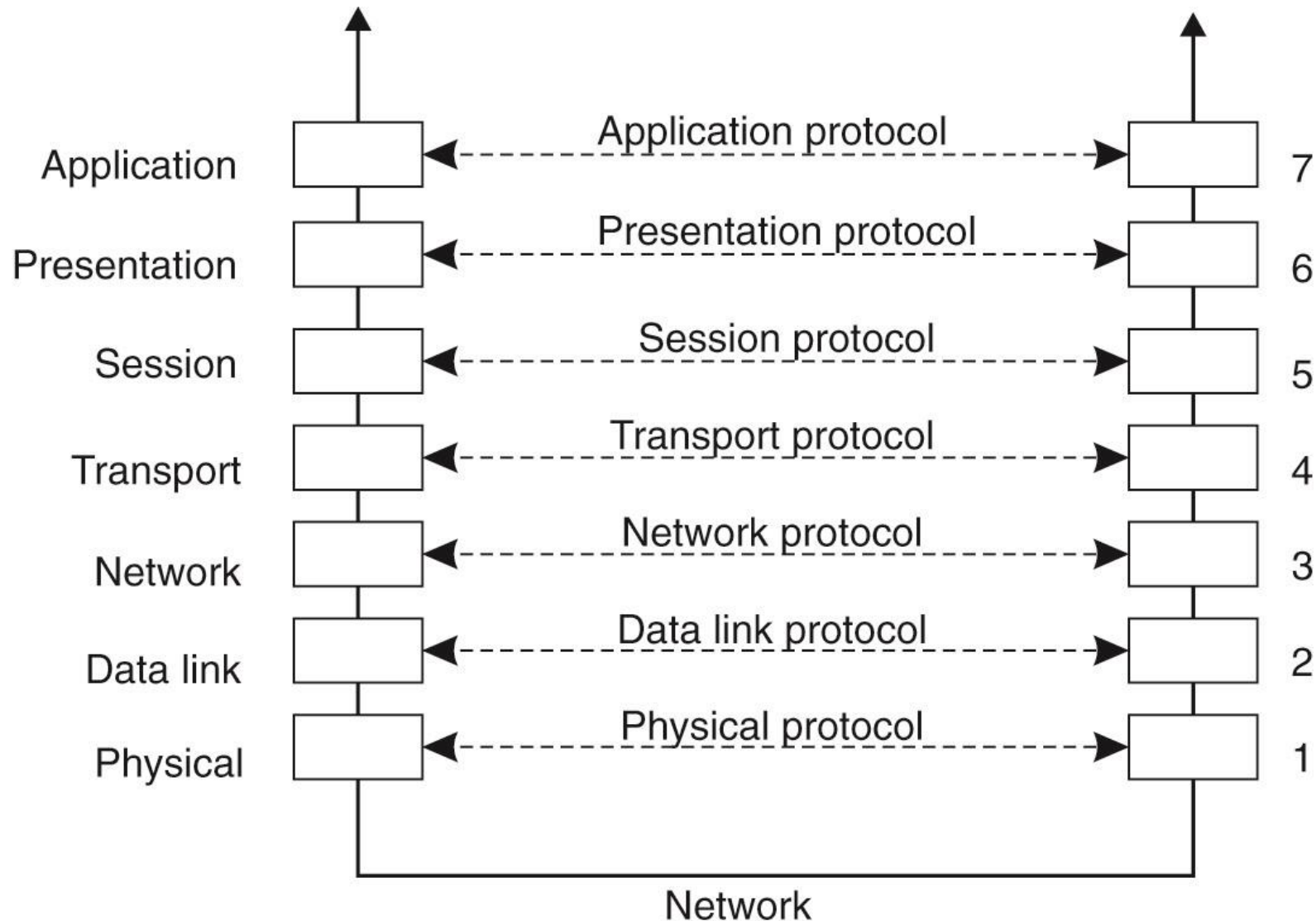
Slide Credits: Maarten van Steen

# Topics for Today

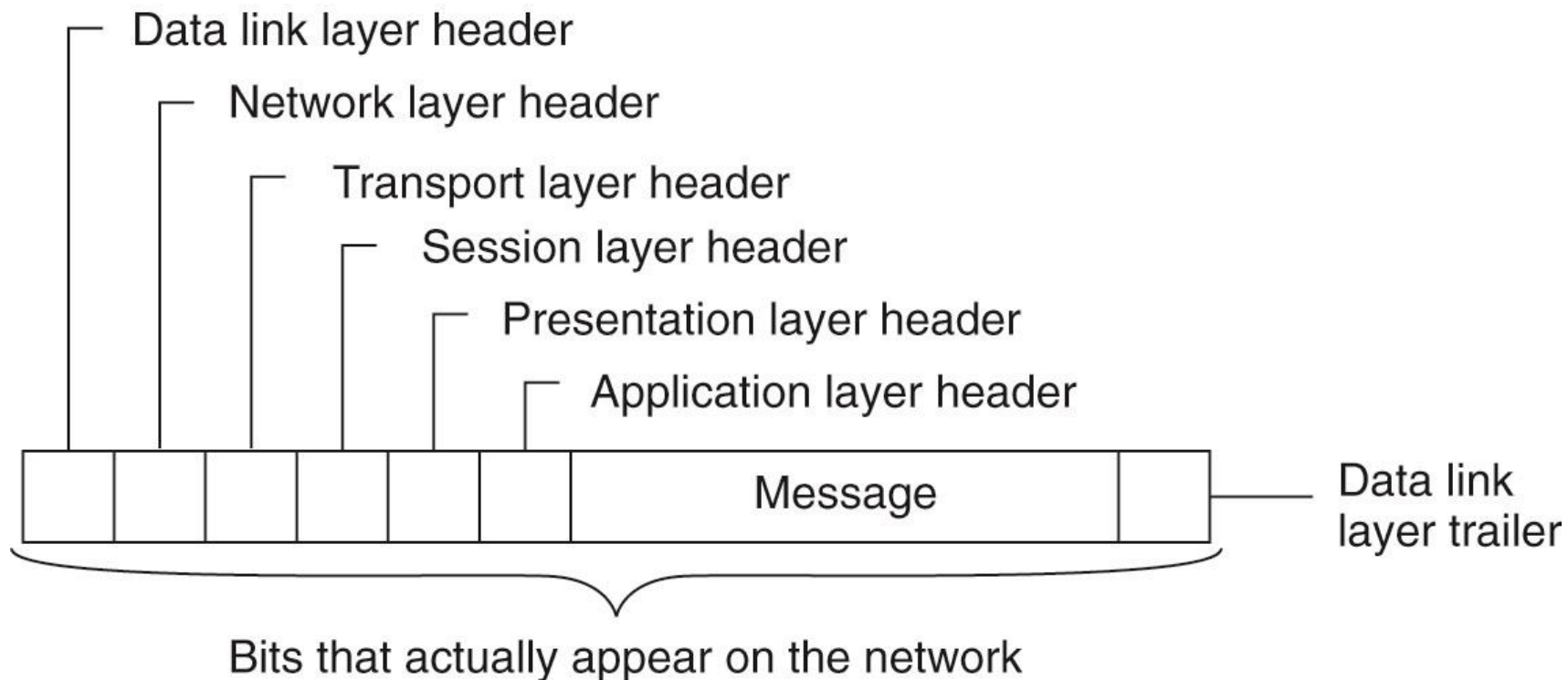
---

- Communication
  - Layered Communication
  - Types of Communication
- RPC and RMI
- Sockets
- Message Oriented Communication

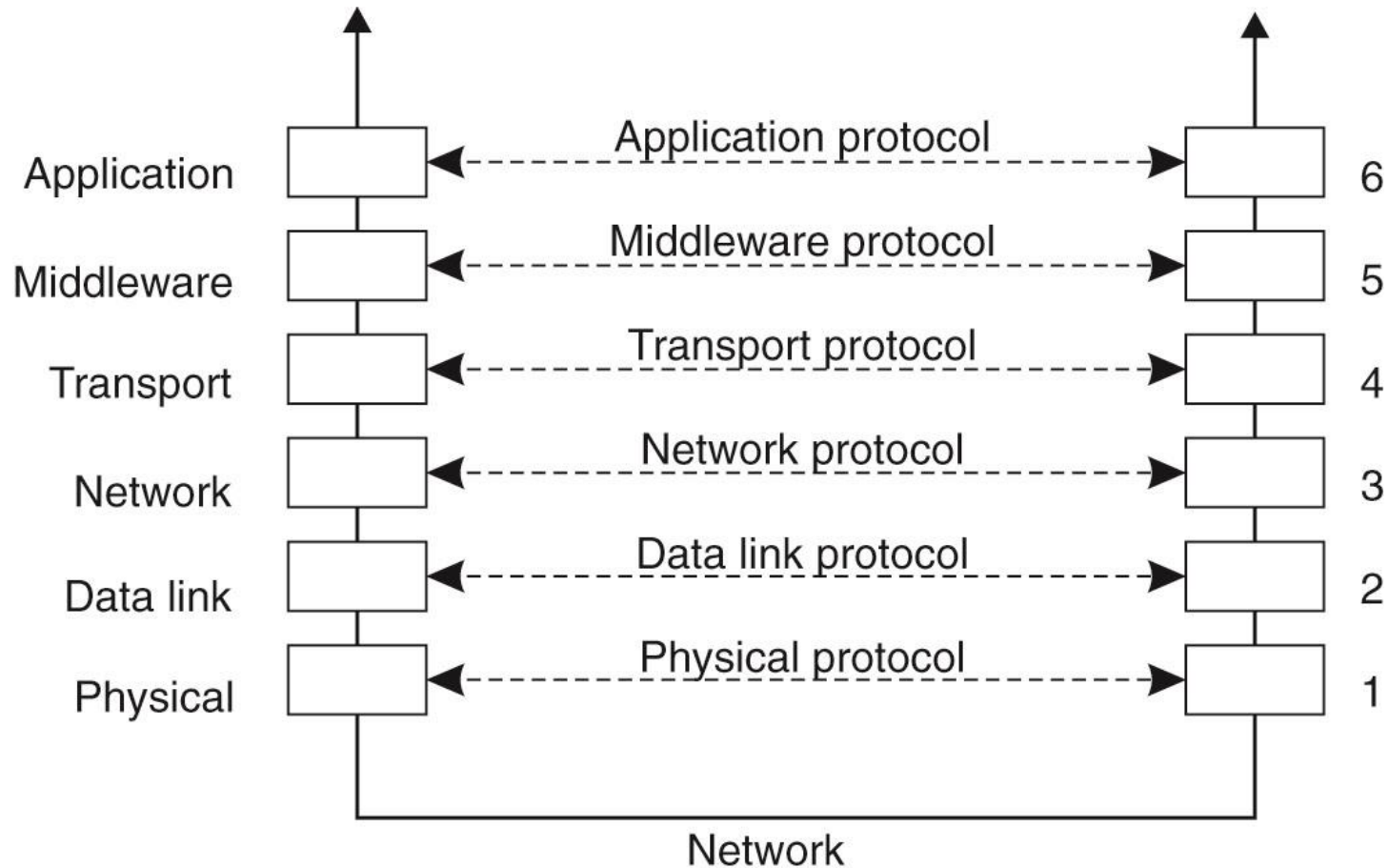
# Basic Networking Model



# Layered Protocols



# Adding Middleware



# Middleware Layer

---

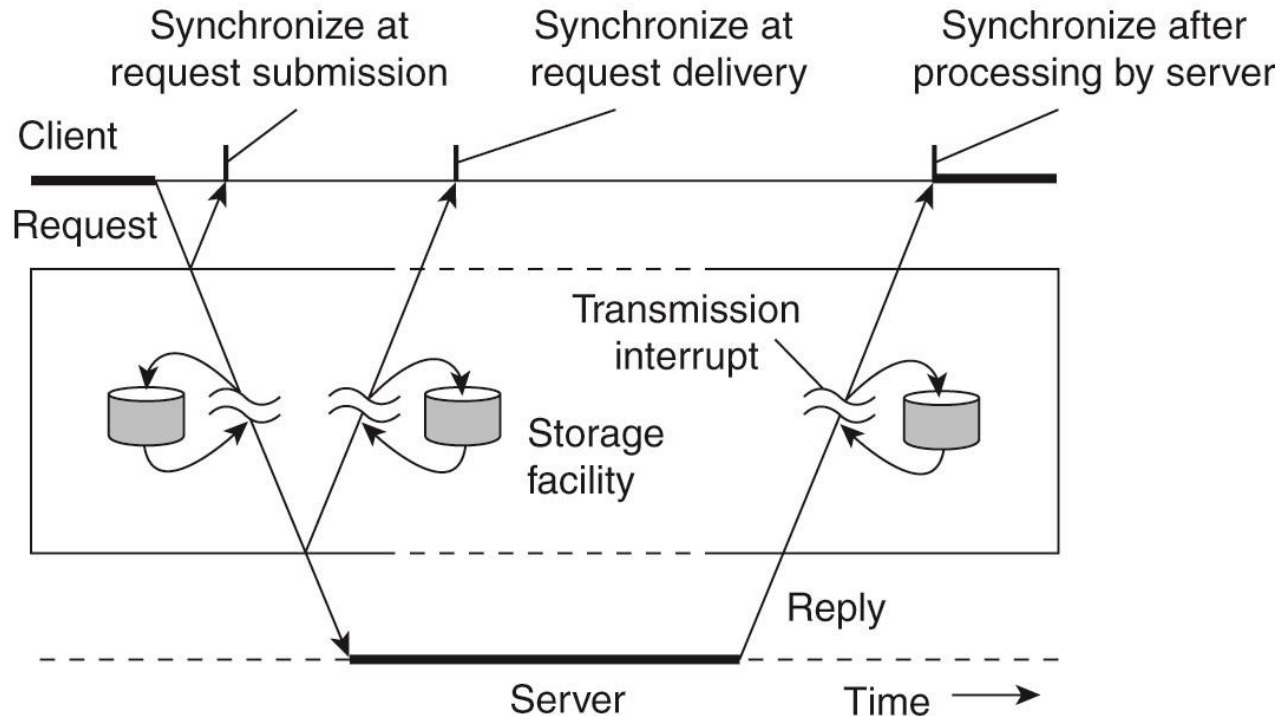
- **Observation:** Middleware is invented to provide **common** services and protocols that can be used by many **different** applications:
  - A rich set of **communication protocols**, but which allow different applications to communicate
  - **(Un)marshaling** of data, necessary for integrated systems
  - **Naming protocols**, so that different applications can easily share resources
  - **Security protocols**, to allow different applications to communicate in a secure way
  - **Scaling mechanisms**, such as support for replication and caching
- **Note:** what remains are truly **application-specific** protocols
- **Question:** Such as...?

# So Far

---

- Communication
  - Layered Communication
  - Types of Communication
- RPC and RMI
- Sockets
- Message Oriented Communication

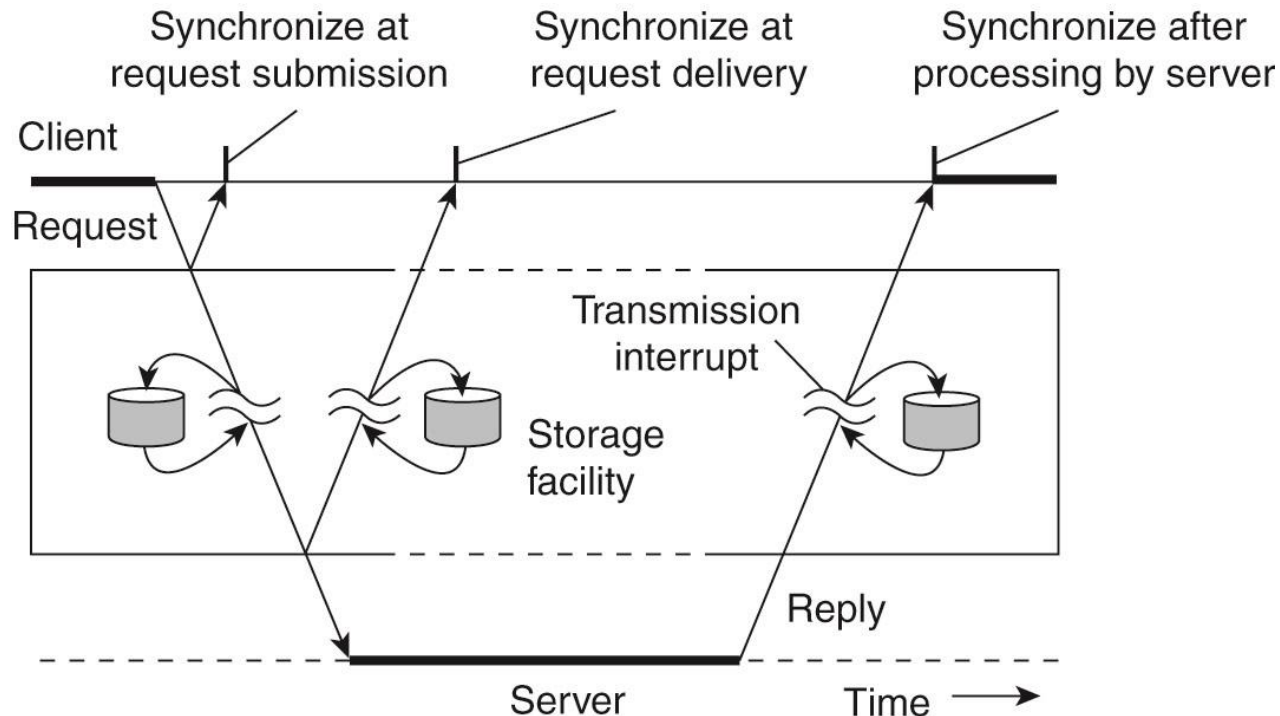
# Types of Communication (1/3)



- **Distinguish:**
  - **Transient** versus **persistent** communication
  - **Asynchronous** versus **synchronous** communication

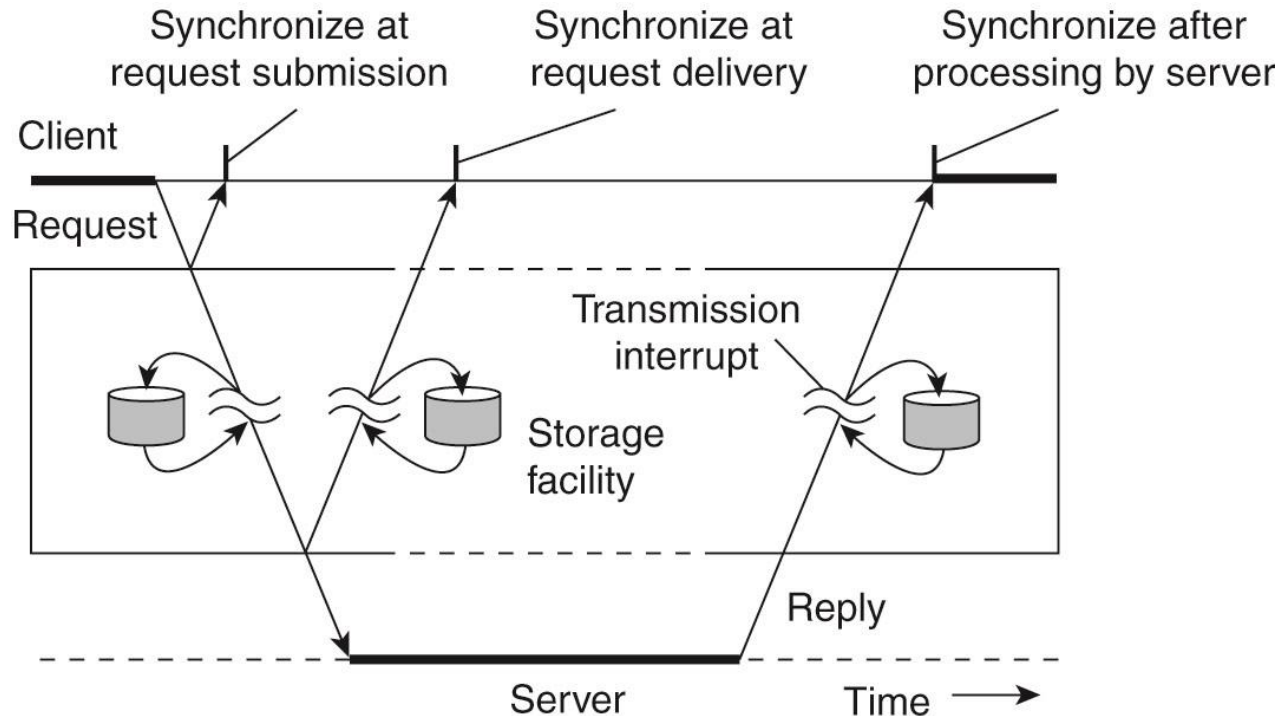


# Types of Communication (2/3)



- **Transient communication:** A message is discarded by a communication server as soon as it cannot be delivered at the next server, or at the receiver.
- **Persistent communication:** A message is stored at a communication server as long as it takes to deliver it at the receiver.

# Types of Communication (3/3)



- **Places for synchronization:**

- At request submission
- At request delivery
- After request processing

# Client/Server



- **Some observations:** Client/Server computing is generally based on a model of **transient synchronous communication**:

Client and server must be active at the time of communication

Client issues request and blocks until it receives reply

Server essentially waits only for incoming requests, and subsequently processes them

- **Drawbacks of synchronous communication:**

Client cannot do any other work while waiting for reply

Failures must be dealt with immediately (the client is waiting)

In many cases the model is simply not appropriate (mail, news)

# Messaging

- **Message-oriented middleware**: Aims at high-level persistent asynchronous communication:

Processes send each other messages, which are queued

Sender need not wait for immediate reply, but can do other things

Middleware often ensures fault tolerance



Image source: <http://tattoo-journal.com/40-lovely-mom-tattoo-designs/>

# So Far

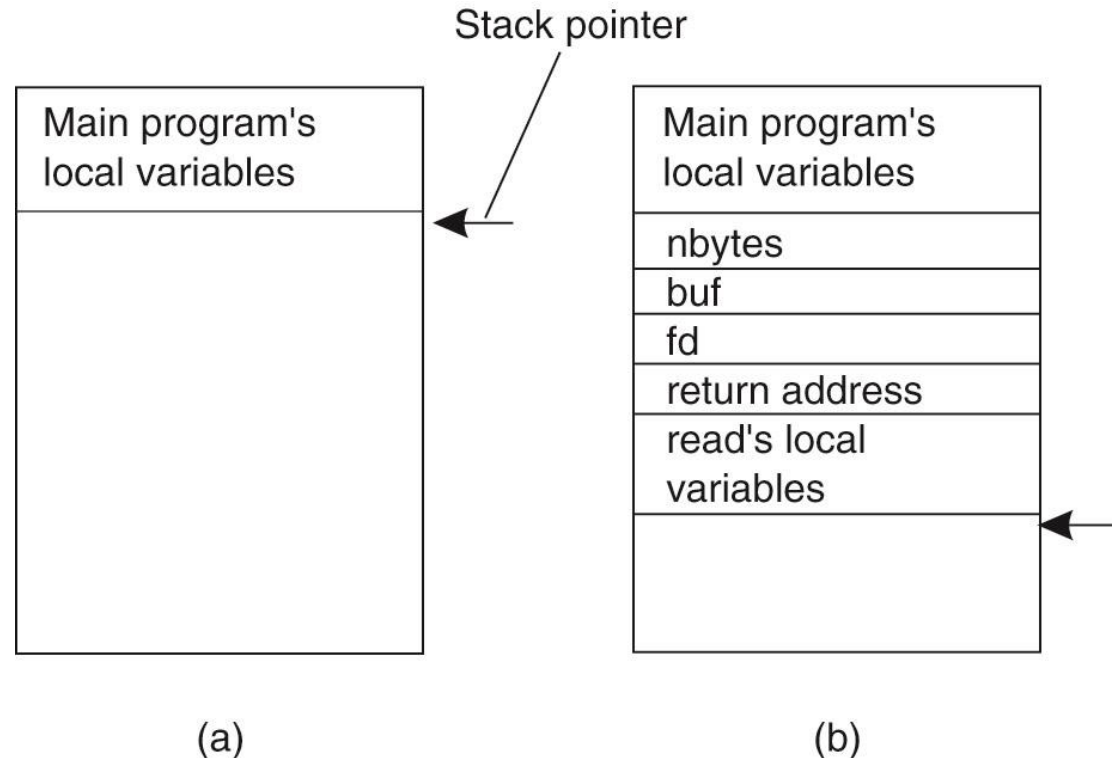
---

- Communication
  - Layered Communication
  - Types of Communication
- RPC and RMI
- Sockets
- Message Oriented Communication

# Conventional Procedure Call

Parameter passing in a local procedure call:

- (a) the stack before the call to read.
- (b) The stack while the called procedure is active.



# Remote Procedure Call (RPC)

---



Basic RPC  
operation

Parameter  
passing

Variations

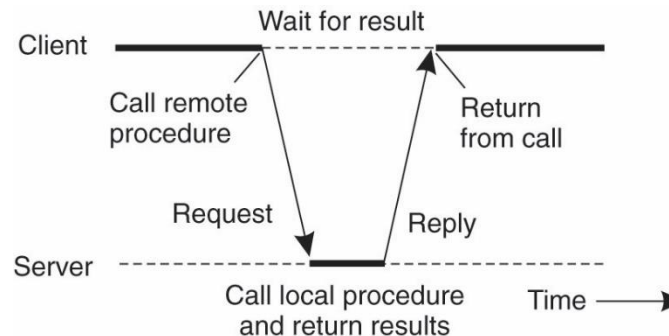
Java  
Remote  
Method  
Invocation  
(RMI)

# Basic RPC Operation (1/3)

## Observations:

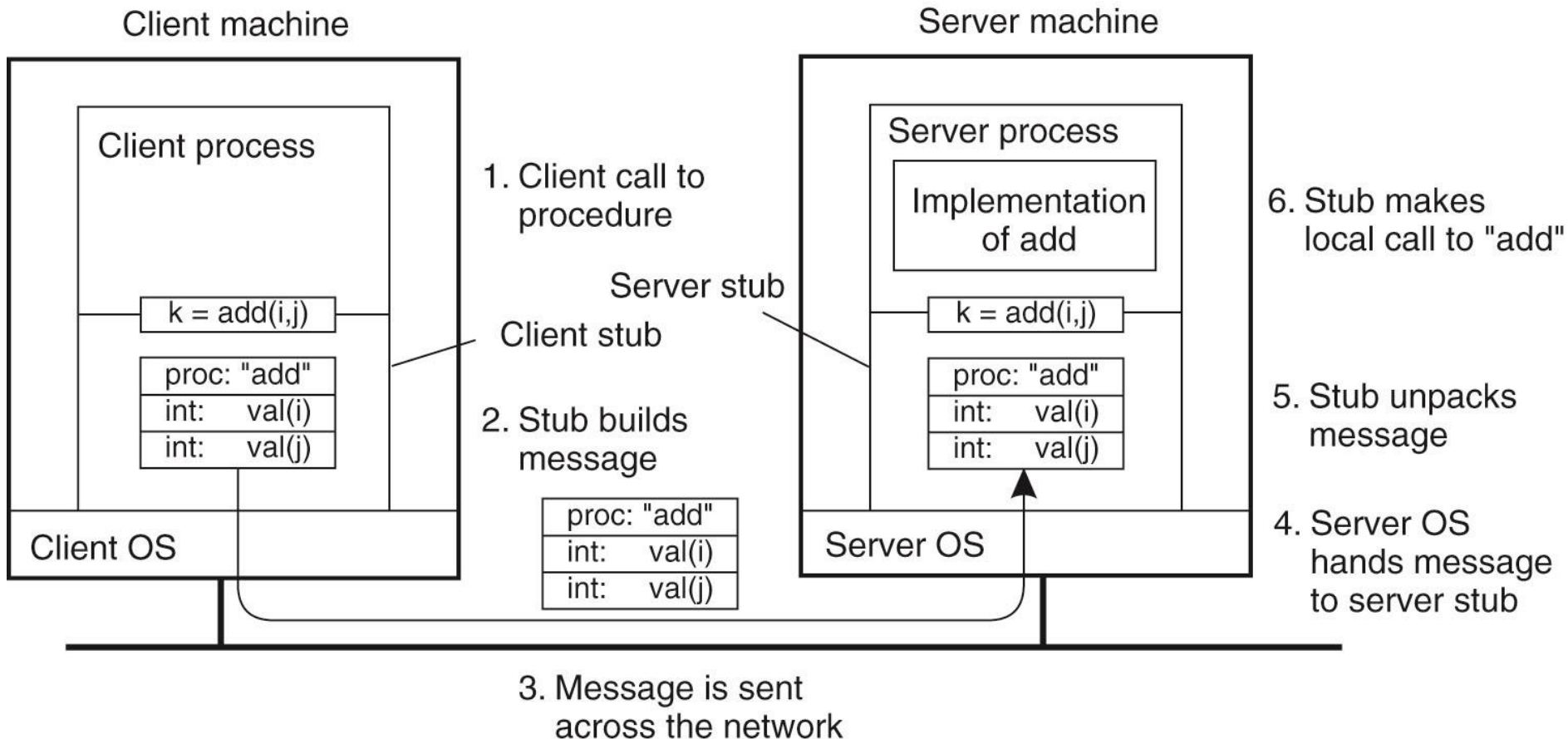
- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

**Conclusion:** communication between caller & callee can be hidden by using procedure-call mechanism.



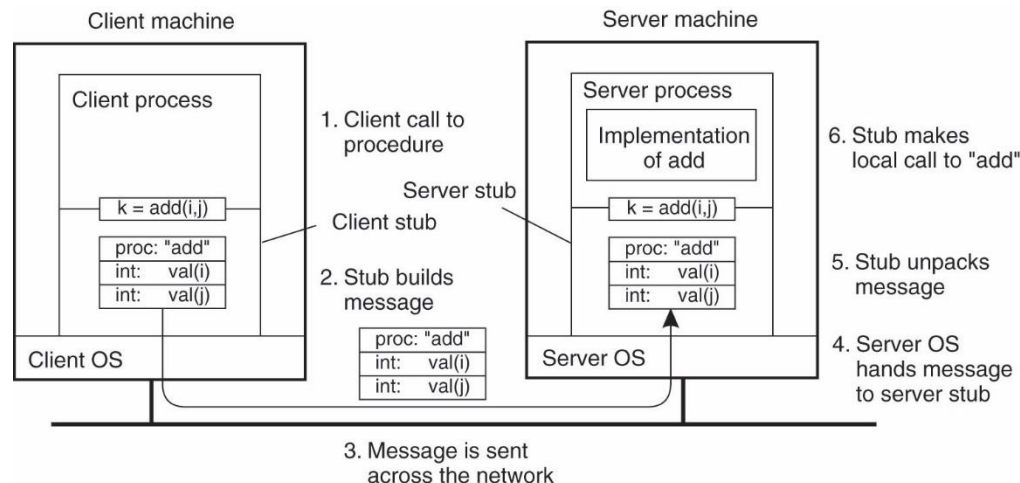


# Basic RPC Operation (2/3)



# Basic RPC Operation (3/3)

1. Client procedure calls client stub as usual.
2. Client stub builds message and calls local OS.
3. Client's OS sends message to remote OS.
4. Remote OS gives message to server stub.
5. Server stub unpacks parameters and calls server.
6. Server does work and returns result to the stub.
7. Server stub packs it in message and calls OS.
8. Server's OS sends message to client's OS.
9. Client's OS gives message to client stub.
10. Client stub unpacks result and returns to the client.



# RPC: Parameter Passing (1/2)

---

**Parameter marshaling:** There's more than just wrapping parameters into a message:

Client and server machines may have **different data representations** (think of byte ordering)

Wrapping a parameter means **transforming a value into a sequence of bytes**

Client and server must **agree on the encoding:**

- How are **basic data values** represented (integers, floats, characters)
- How are **complex data values** represented (arrays, unions)

Client and server need to **properly interpret messages** transforming them into machine-dependent representations.

# RPC: Parameter Passing (2/2)

---

## RPC parameter passing:

- RPC assumes **copy in/copy out** semantics: while procedure is executed, nothing can be assumed about parameter values (only Ada supports this model).
- RPC assumes **all** data that is to be operated on is passed by parameters. Excludes passing **references to (global) data**.

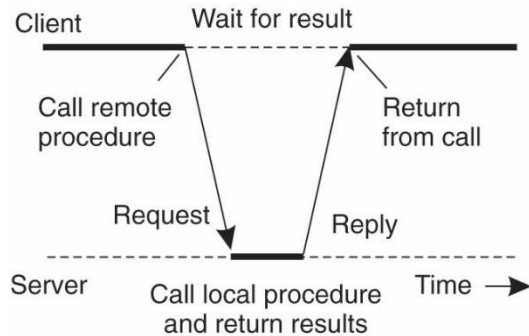
**Conclusion:** full access transparency cannot be realized.

**Observation:** If we introduce a **remote reference** mechanism, access transparency can be enhanced:

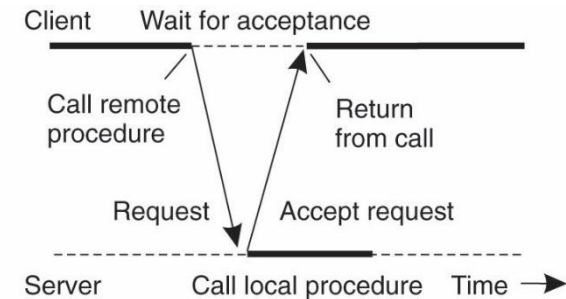
- Remote reference offers **unified access** to remote data
- Remote references can be **passed as parameter** in RPCs

# Asynchronous RPCs

**Essence:** Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.

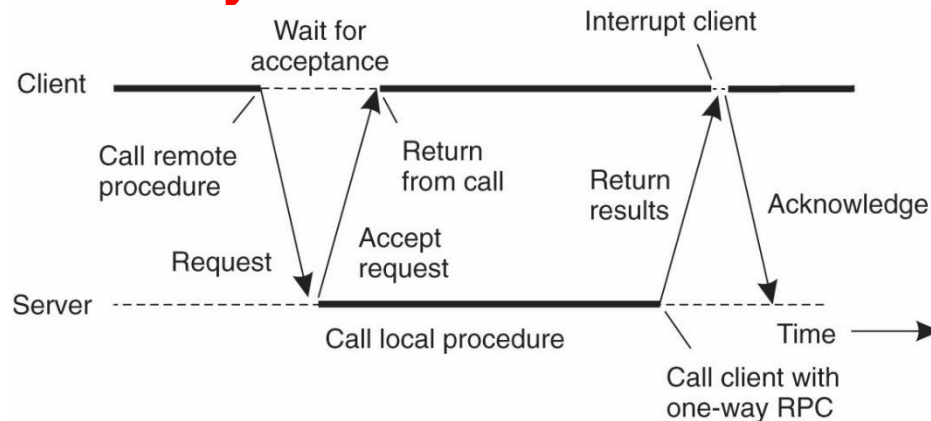


(a)



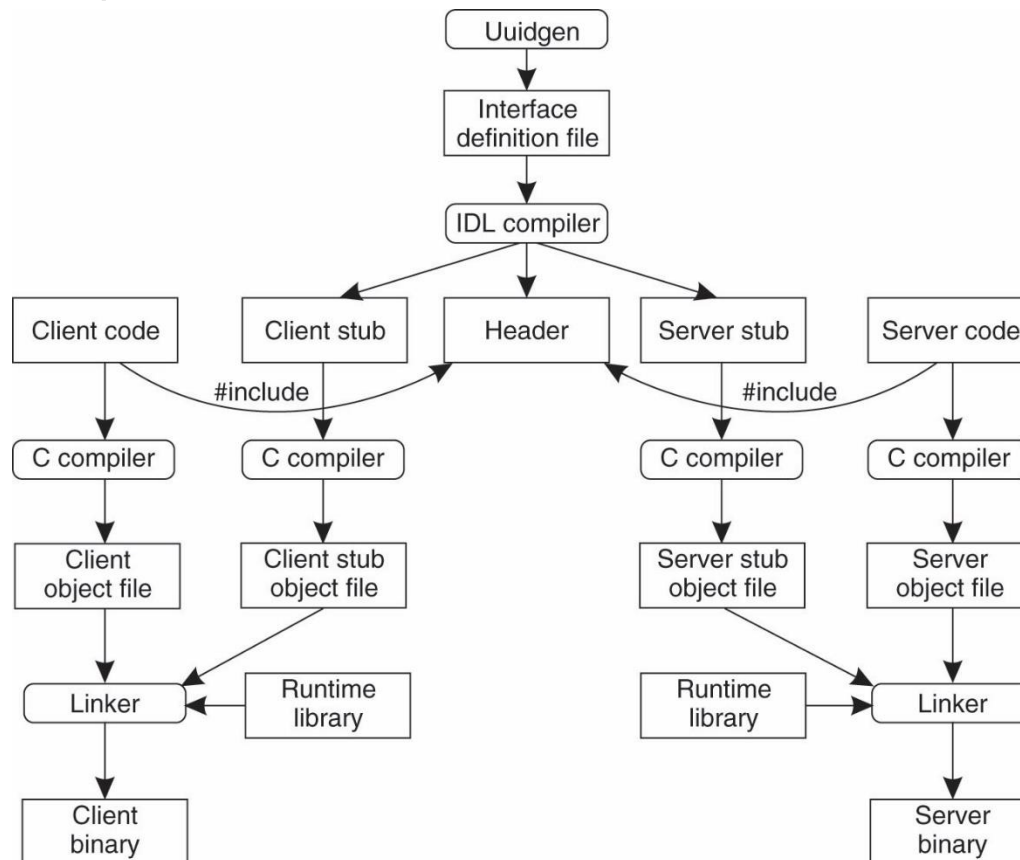
(b)

Variation: **deferred synchronous RPC:**



# RPC in Practice

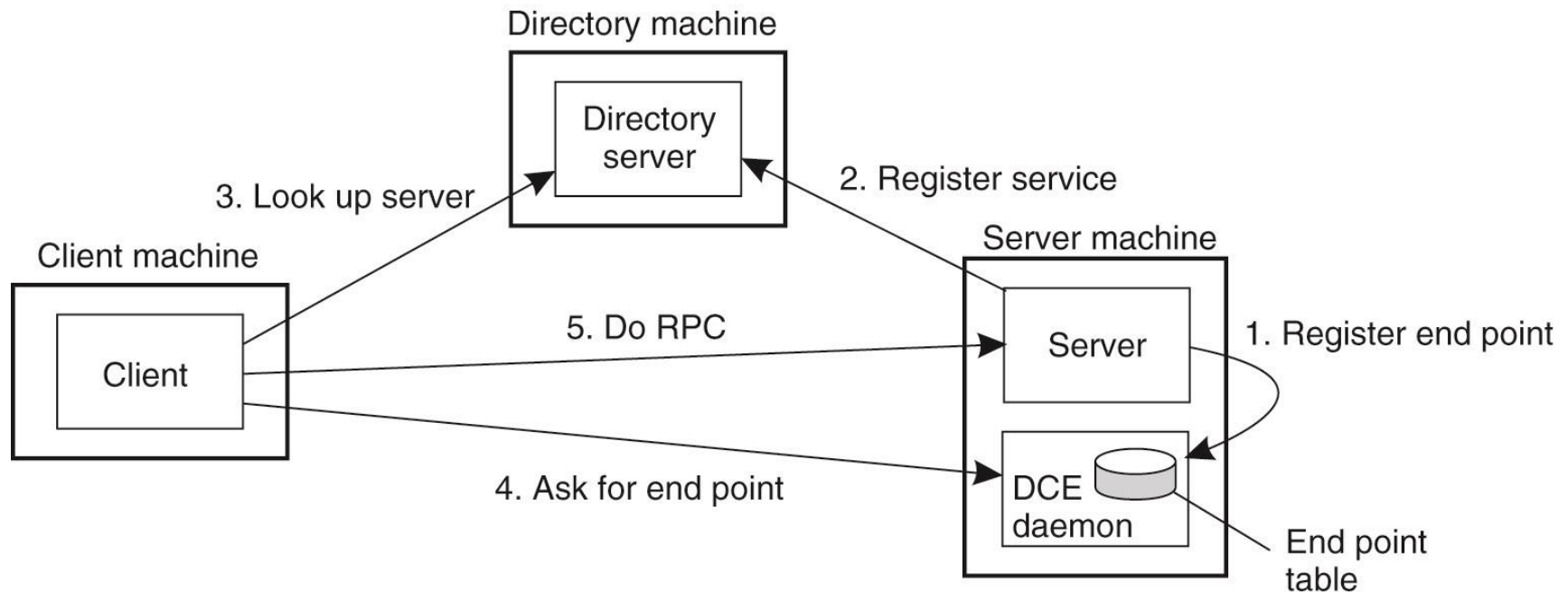
**Essence:** Let the developer concentrate on only the client- and server-specific code; let the RPC system (generators and libraries) do the rest.



# Client-to-Server Binding (DCE)

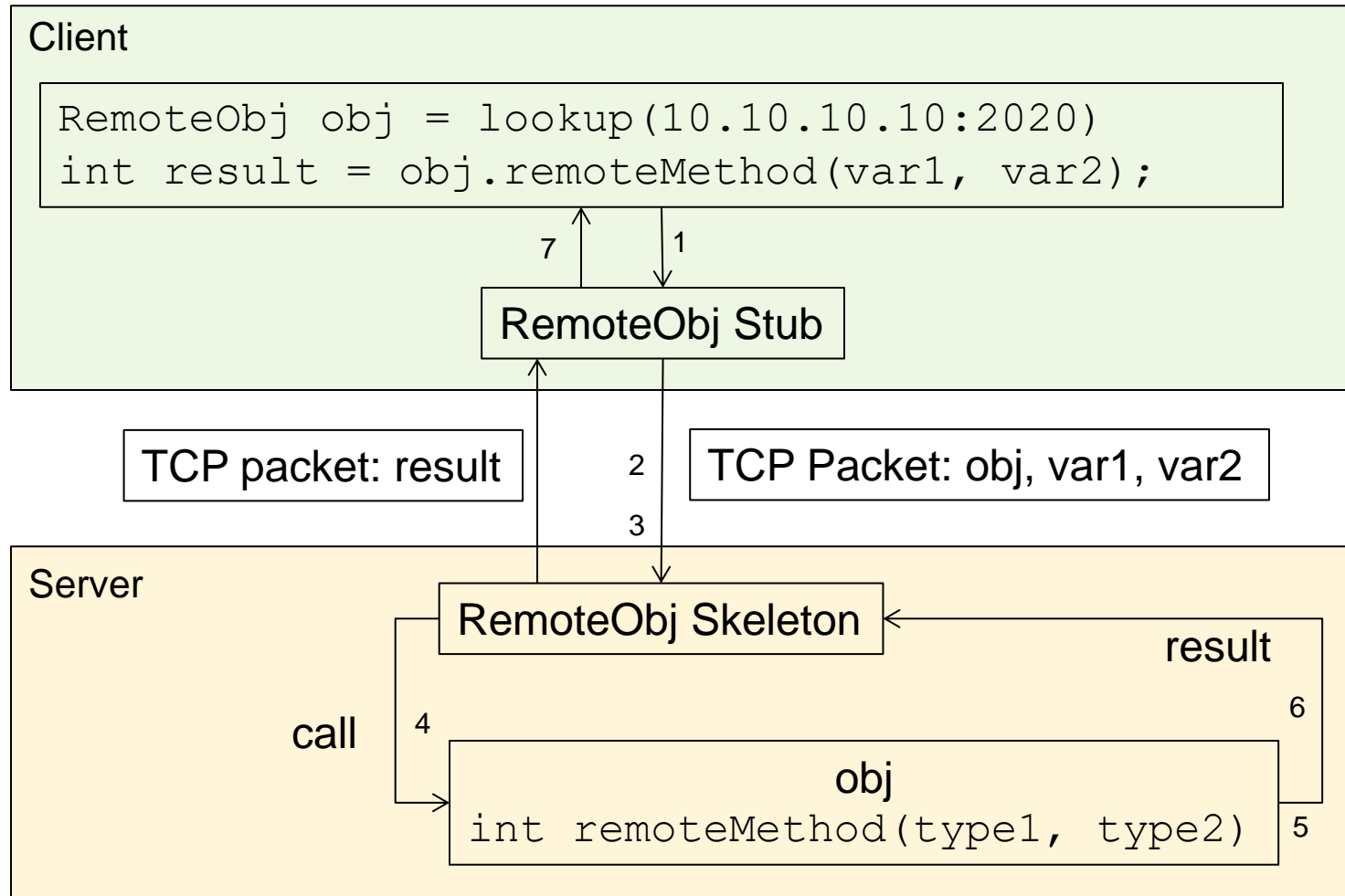
**Issues:** (1) Client must locate server machine, and (2) locate the server.

**Example:** DCE uses a separate daemon for each server machine.



# Remote Method Invocation Basics

Assume client stub and server skeleton are in place





# Remote Method Invocation Basics

---

1. Client invokes method at **stub**
2. **Stub** marshals request and sends it to **server**
3. **Server** ensures referenced **object** is active:
  1. Create separate process to hold object
  2. Load the object into server process
4. Request is unmarshaled by object's **skeleton**, and referenced **method** is invoked
5. If request contained an **object reference**, invocation is applied recursively (i.e., server acts as client)
6. **Result** is marshaled and passed back to **client**
7. **Client stub** unmarshals reply and passes result to client application

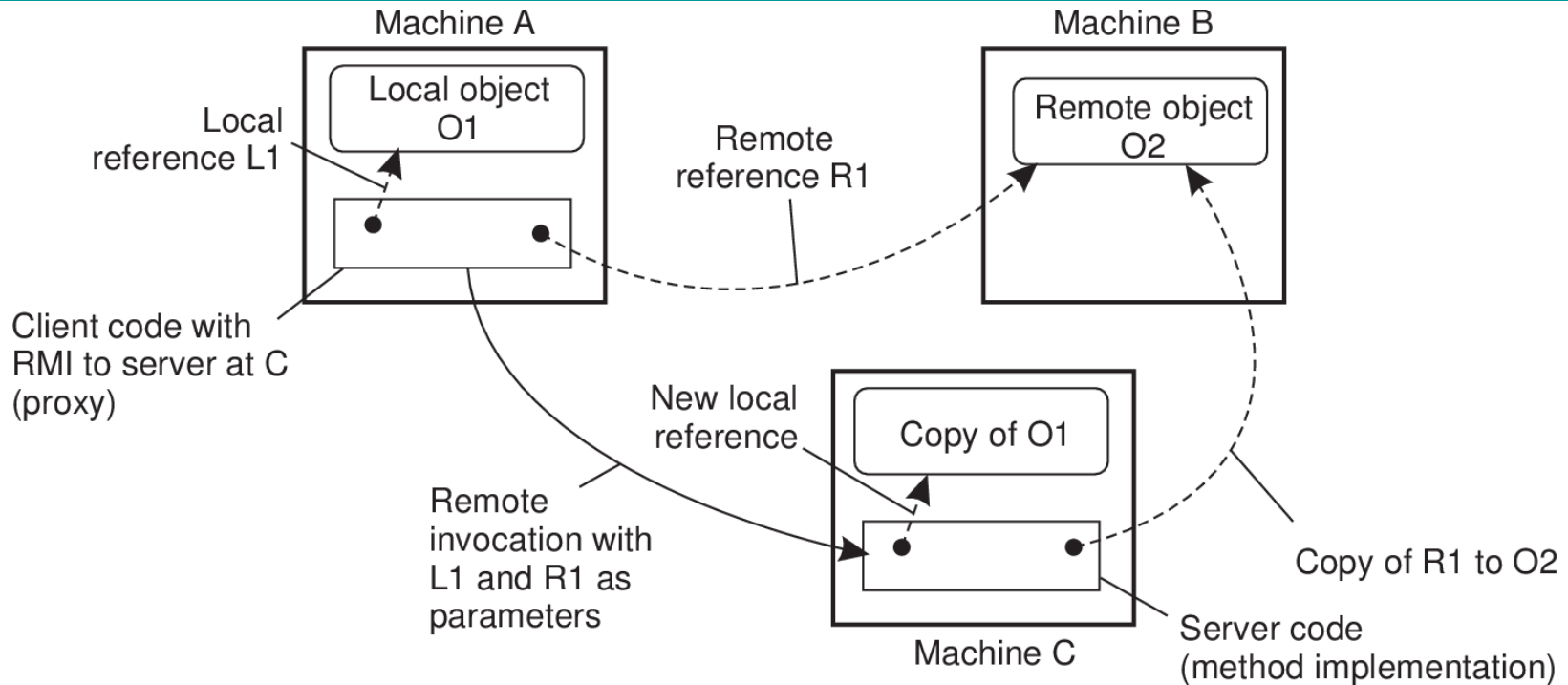
# RMI Parameter Passing

---

**Object-by-value:** A client may also pass a complete object as parameter value:

- An object has to be **marshaled**:
  - Marshall its state
  - Marshall its methods, or give a reference to where an implementation can be found
- Server **unmarshals object**. Note that we have now created a **copy** of the original object.
- Object-by-value passing tends to introduce nasty problems

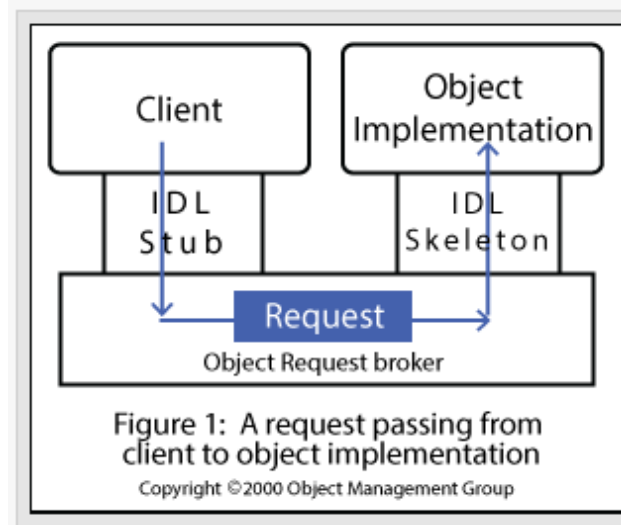
# RMI Parameter Passing



**Note:** System-wide object references generally contain **server address and port** to which adapters listens, and **local object ID**.

**Extra:** May also contain information on protocol between client and servers (TCP, UDP, SOAP, JSON, etc.)

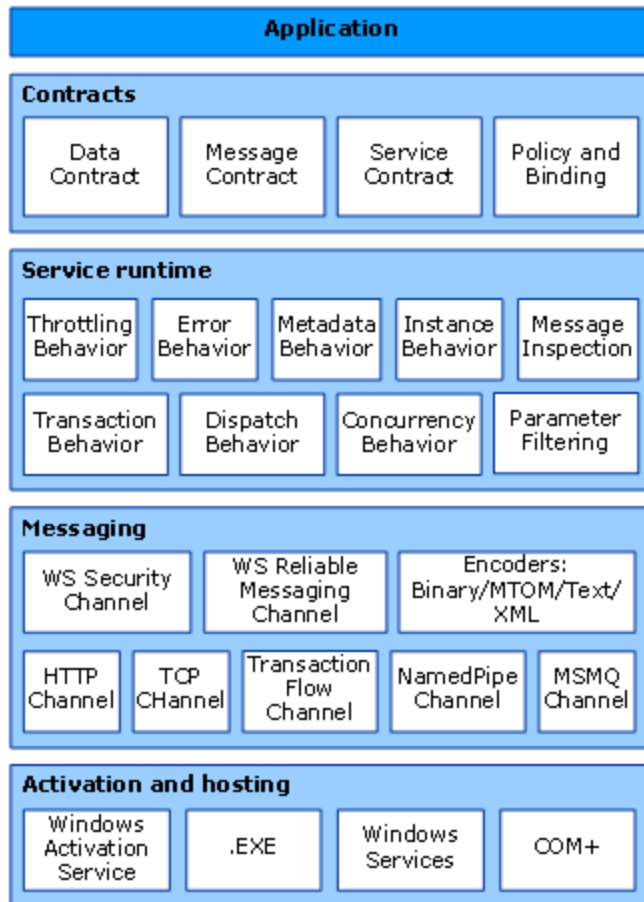
# Others: CORBA



- <https://corba.org>
- Multi-language support (COBOL, Ada, Lisp, Java, Python, etc.)
- Legacy technology, but still works

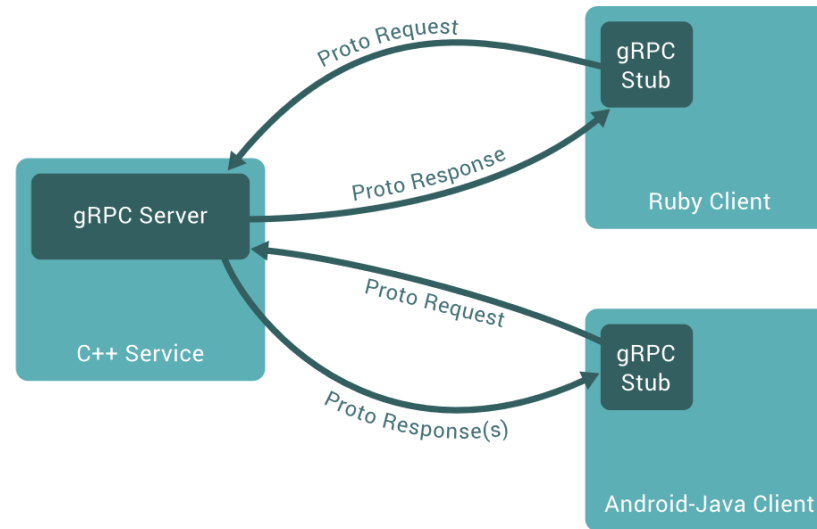
# Others: Windows Communication Foundation

## WCF Architecture



- <https://docs.microsoft.com/en-us/dotnet/framework/wcf>
- Microsoft based technology
- Web Services based
- Support for .NET languages and Windows, should work on Linux

# Others: gRPC



- <https://grpc.io/>
- Language independent (multi-platform)
- Uses binary underlying protocol (ProtoBuf from Google)

# So Far

---

- Communication
  - Layered Communication
  - Types of Communication
- RPC and RMI
- Sockets
- Message Oriented Communication

# Message-Oriented Communication

---

## Transient Messaging

- Sockets

## Message-Queuing System

- Message Brokers

## Examples

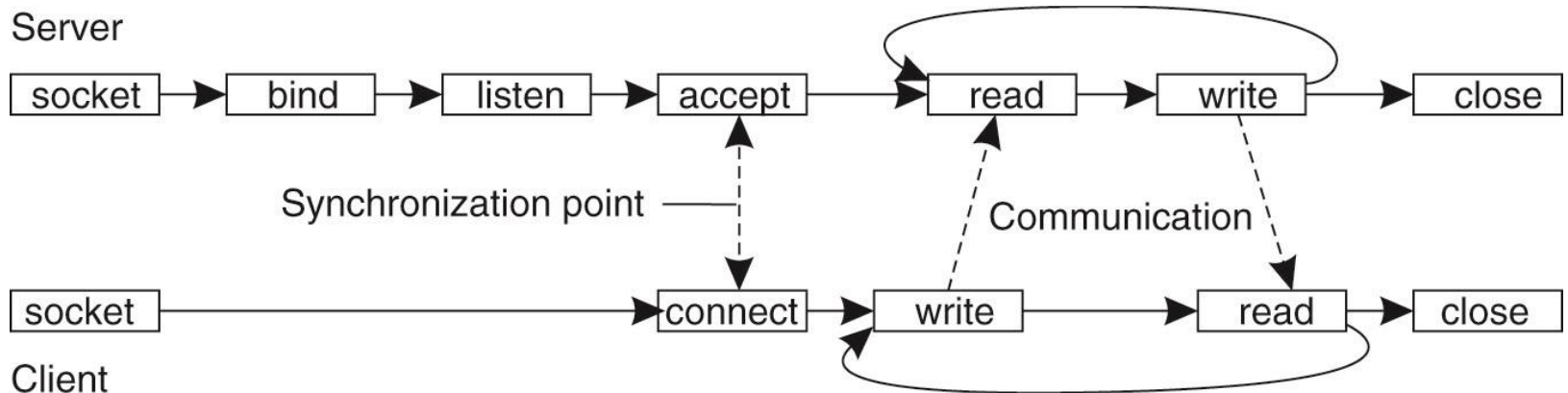
- IBM MQ (formerly WebSphere MQ)
- Others



# Transient Messaging: Sockets

**Example:** Consider the Berkeley **socket interface**, which has been adopted by all Posix systems, as well as Windows 95/NT/2000/XP/Vista:

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection



# Message-Oriented Middleware

---

**Essence:** Asynchronous persistent communication through support of middleware-level **queues**. Queues correspond to buffers at communication servers.

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

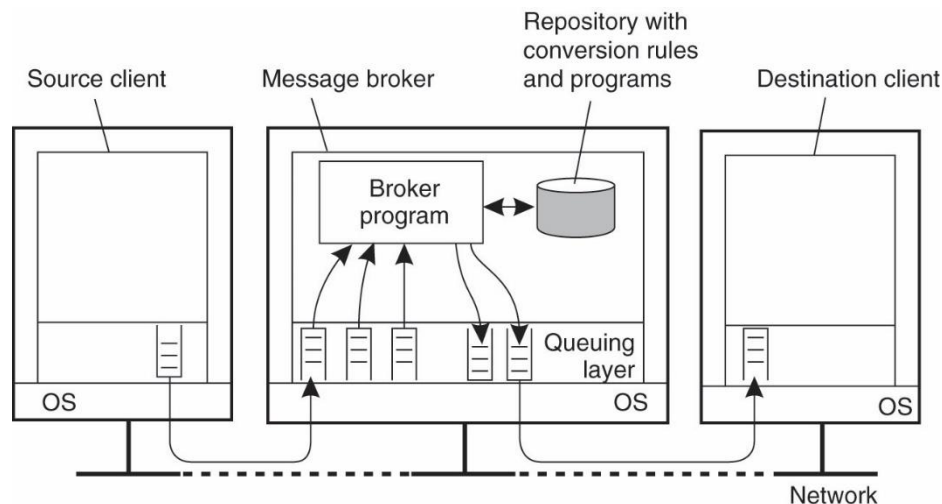
See also: [Java Queue Interface \(JDK 18\)](#)

# Message Broker

**Observation:** Message queuing systems assume a **common messaging protocol**: all applications agree on message format (i.e., structure and data representation)

**Message broker (Integration Bus):** Centralized component that takes care of application heterogeneity in an MQ system:

- Transforms incoming messages to target format
- Very often acts as an **application gateway**
- May provide **subject-based** routing capabilities ⇒ **Enterprise Application Integration**



# IBM MQ (1/3) (<https://www.youtube.com/watch?v=ynjc5GMQeRA>)

---

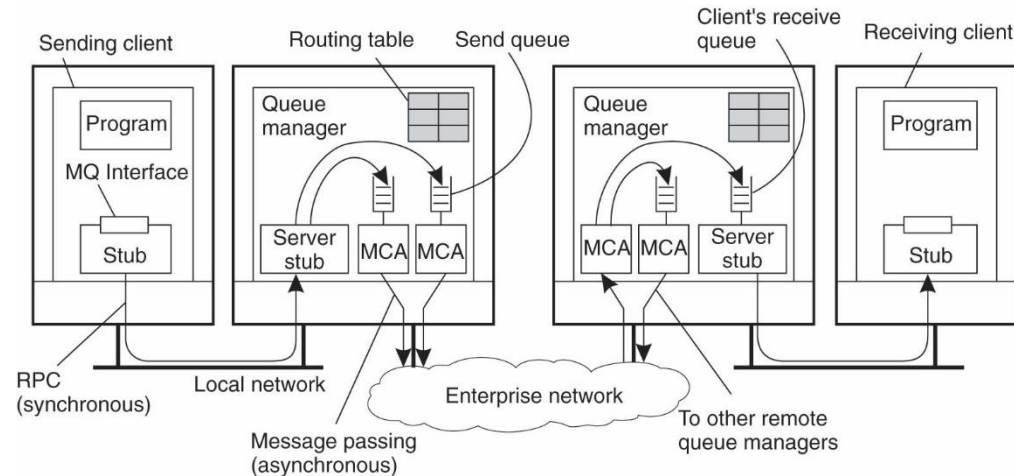
## Basic concepts:

- **Application-specific messages** are put into, and removed from **queues**
- Queues always reside under the regime of a **queue manager**
- Processes can put messages only in local queues, or through an RPC mechanism

## Message transfer:

- Messages are transferred between queues
- Message transfer between queues at different processes, requires a **channel**
- At each endpoint of channel is a **message channel agent**
- Message channel agents are responsible for:
  - Setting up channels using lower-level network communication facilities (e.g., TCP/IP)
  - (Un)wrapping messages from/in transport-level packets
  - Sending/receiving packets

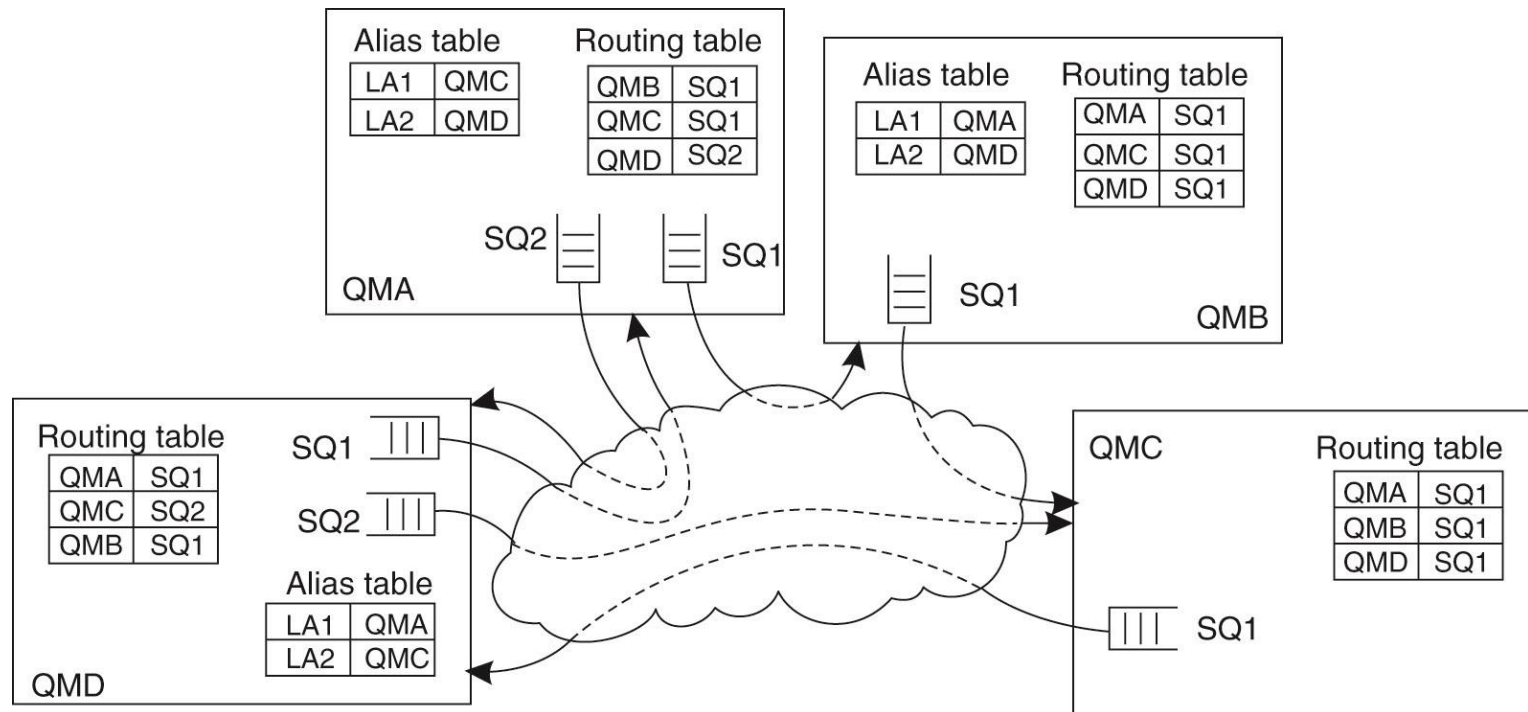
# IBM MQ (2/3)



- **Channels** are inherently unidirectional
- MQ provides mechanisms to **automatically start** MCAs when messages arrive, or to have a receiver set up a channel
- Any network of queue managers can be created; **routes are set up manually** (system administration)

# IBM MQ (3/3)

**Routing:** By using **logical names**, in combination with name resolution to local queues, it is possible to put a message in a **remote queue**

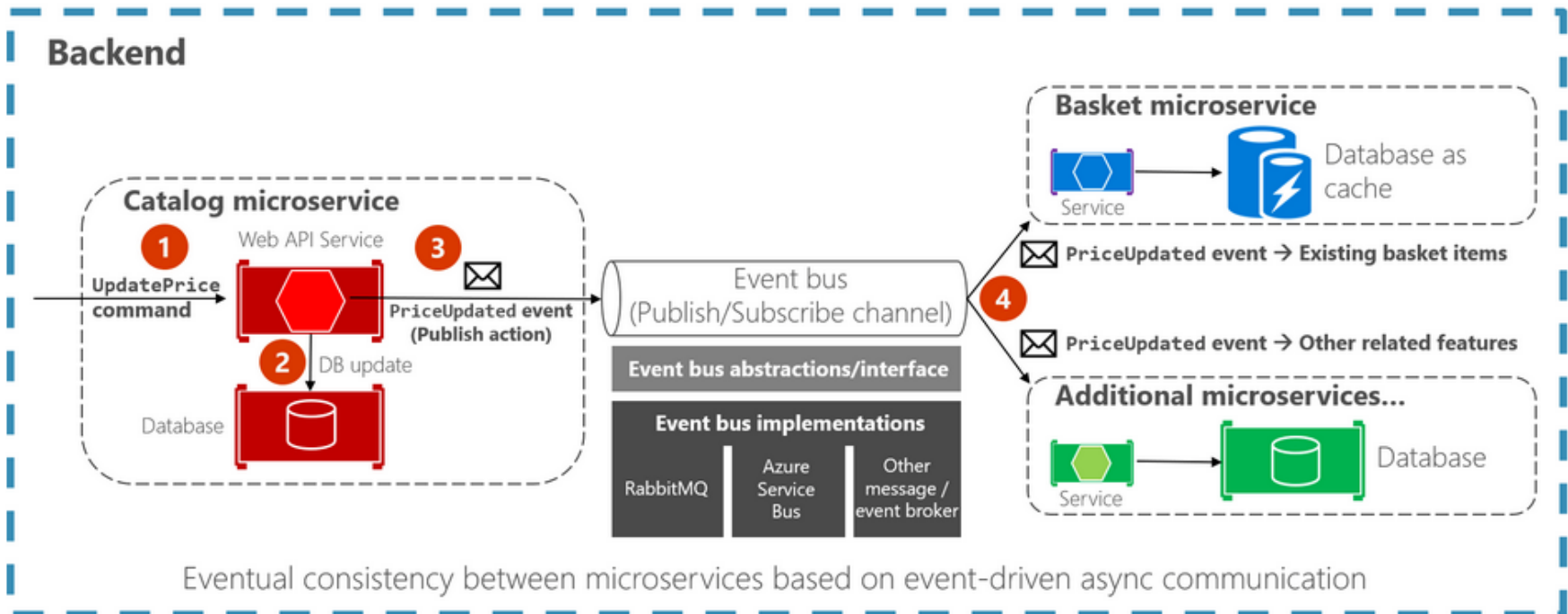


**Question:** What's a major problem here?

# Others: Microsoft


<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications>  
<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/subscribe-events>



## Implementing asynchronous event-driven communication with an event bus



# Others: Azure

<https://azure.microsoft.com/en-us/services/event-hubs/>

 **Microsoft Azure**

Contact Sales: 0201605845  Search  My account Portal S

Overview Solutions **Products** Documentation Pricing Training Marketplace Partners Support Blog More

Free acco

Explore Event Hubs: Pricing details Documentation

Event Hubs is a fully managed, real-time data ingestion service that's simple, trusted, and scalable. Stream millions of events per second from any source to build dynamic data pipelines and immediately respond to business challenges. Keep processing data during emergencies using the [geo-disaster recovery](#) and geo-replication features.

Integrate seamlessly with other Azure services to unlock valuable insights. Allow existing Apache Kafka clients and applications to talk to Event Hubs without any code changes—you get a managed Kafka experience without having to manage your own clusters. Experience real-time data ingestion and microbatching on the same stream.

[Link to video](#) >





# Other: Apache Camel

<https://camel.apache.org/>



## Integration you want

Camel is an open source integration framework that empowers you to quickly and easily integrate various systems consuming or producing data.

[Get started](#)

**PACKED WITH FUNCTIONALITY**

# Others: Amazon EBus (Event Bus)

<https://docs.aws.amazon.com/lumberyard/latest/userguide/ebus-intro.html>

The screenshot shows the AWS Lumberyard User Guide interface. The top navigation bar includes the AWS logo, GitHub and PDF download icons, and a language selector set to 'English'. The left sidebar, titled 'Lumberyard User Guide (Version 1.21)', contains a search bar and a list of links: 'What is Lumberyard?', 'Set Up', 'Create a Game Project', 'Build a Game Project', 'Migrate a Game Project', 'Lumberyard Editor', 'Sample Projects and Levels', 'Asset Pipeline', 'Component Entities', and 'Add Features and Assets with Gems'. The main content area displays the breadcrumb 'AWS Documentation » Amazon Lumberyard » User Guide » Working with the Event Bus (EBus) System' followed by the section title 'Working with the Event Bus (EBus) System'. The text explains that Event buses (EBuses) are a general-purpose communication system used by Lumberyard for dispatching notifications and receiving requests. It notes that EBus usage examples are found in 'Usage and Examples', and in-depth information is available in 'Event Buses in Depth'. For C++ API reference, it points to the 'EBus API Reference' in the 'Amazon Lumberyard C++ API Reference'. The section 'How Components Use EBuses' is also visible at the bottom.

**Working with the Event Bus (EBus) System**

Event buses (EBuses) are a general-purpose communication system that Lumberyard uses to dispatch notifications and receive requests. EBuses are configurable and support many different use cases.

To interact with the engine or other components in Lumberyard, include the component or system's EBus or API header in your code. Then make calls to the exposed EBuses. With this approach you can replace engine-level system APIs with implementations that you define in a gem. For example, you could replace Lumberyard's audio system with your own EBus handler. This would give you complete control over audio without having to recompile the engine.

For examples of EBus usage, see [Usage and Examples](#).

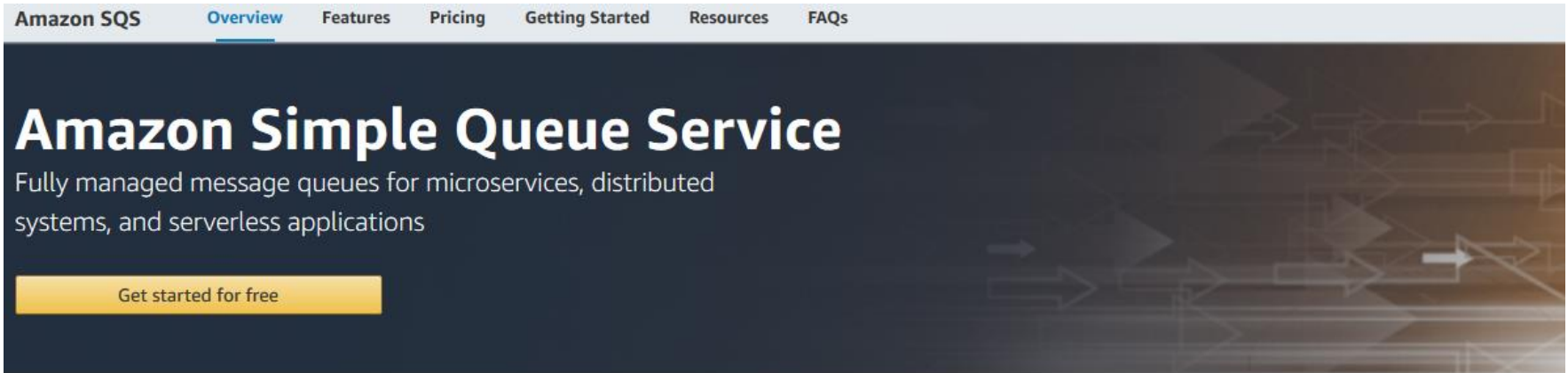
For in-depth information about EBuses, including conceptual diagrams, see [Event Buses in Depth](#).

For C++ API reference documentation on the core EBus code, see the [EBus API Reference](#) in the [Amazon Lumberyard C++ API Reference](#).

**How Components Use EBuses**

# Others: Amazon SQS

<https://aws.amazon.com/sqs/>

A screenshot of the Amazon SQS Overview page. The top navigation bar includes links for Overview, Features, Pricing, Getting Started, Resources, and FAQs. The main heading is "Amazon Simple Queue Service" in large white text. Below it, a subtitle reads "Fully managed message queues for microservices, distributed systems, and serverless applications". A yellow button with the text "Get started for free" is positioned below the subtitle. The background of the page features a dark blue gradient with faint, stylized white arrows pointing in various directions.

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. SQS eliminates the complexity and overhead associated with managing and operating message oriented middleware, and empowers developers to focus on differentiating work. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available. Get started with SQS in minutes using the AWS console, Command Line Interface or SDK of your choice, and three simple commands.

SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery. SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.



# Conclusion

---

- Communication
  - Layered Communication
  - Types of Communication
- RPC and RMI
- Sockets
- Message Oriented Communication