Communication

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

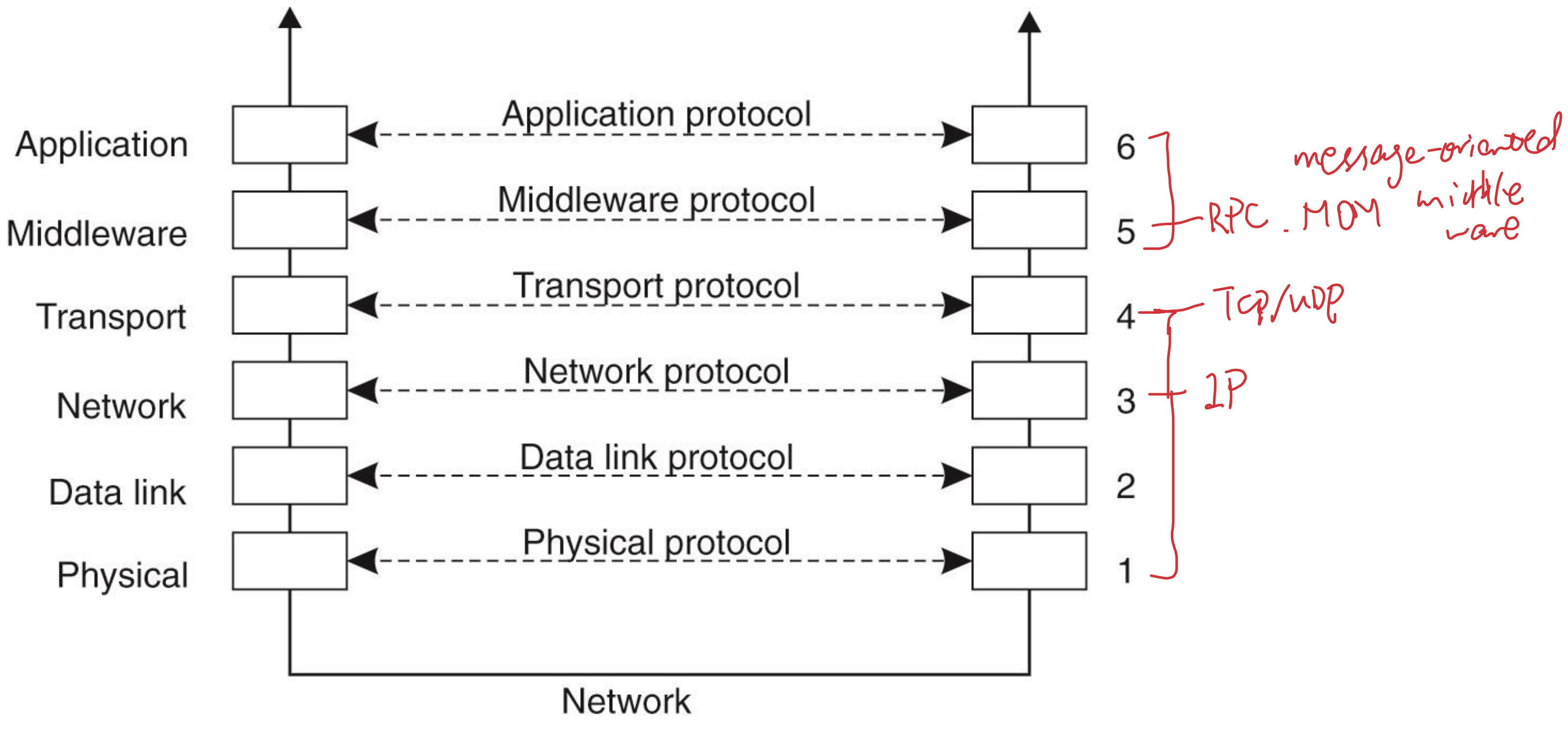
Slides are derived from A. S. Tanenbaum and M. Van Steen,
Distributed Systems: Principles and Paradigms, 2nd Edition, Pearson-Prentice Hall, 2006.
as well as
M. Van Steen and A. S. Tanenbaum, Distributed Systems, 3rd Edition, Pearson, 2017.

Learning objectives

- To understand the principles and implementation of remote procedure calls (RPCs).
- To understand the strengths and weaknesses of RPC middleware as a communication facilitator.
- To develop a conceptual understanding of the message-queuing model.

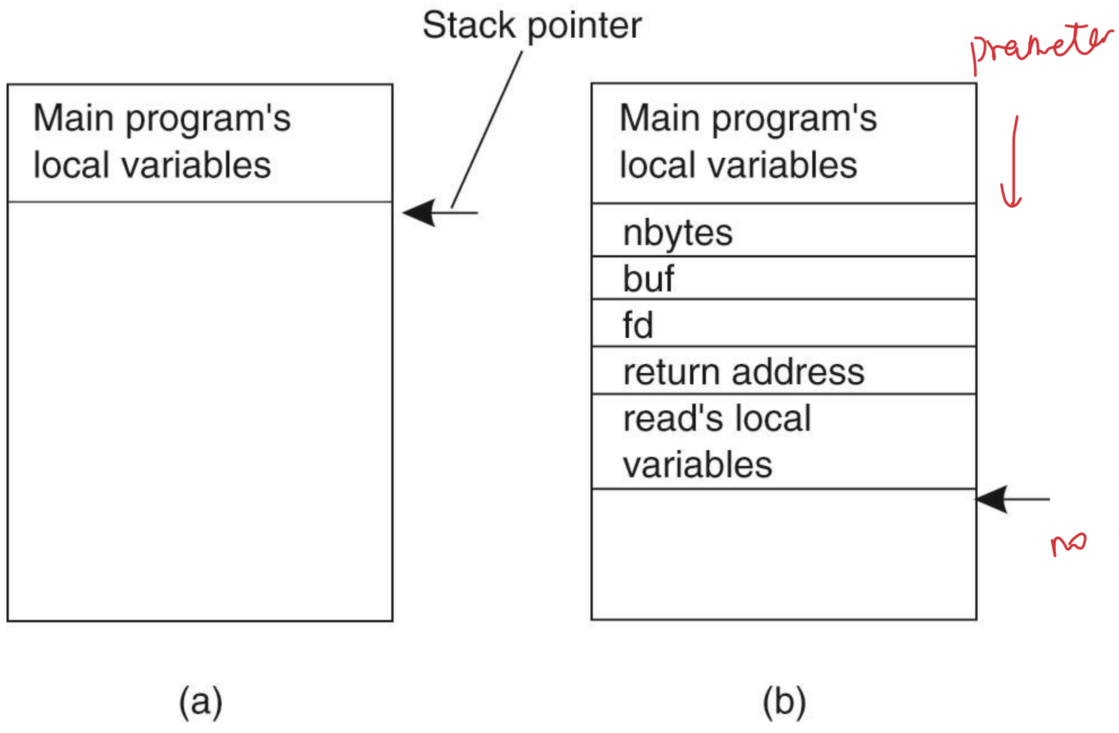
Layered network model
(from the textbook, layers 5+ different from OSI)

Middleware aims to provide access transparency by isolating the application from the transport layer (e.g., TCP or UDP).



Remote procedure calls

A **remote procedure call (RPC)** is a **transient communication** abstraction. It behaves similarly to a conventional procedure call (illustrated below) that passes parameters on the stack.



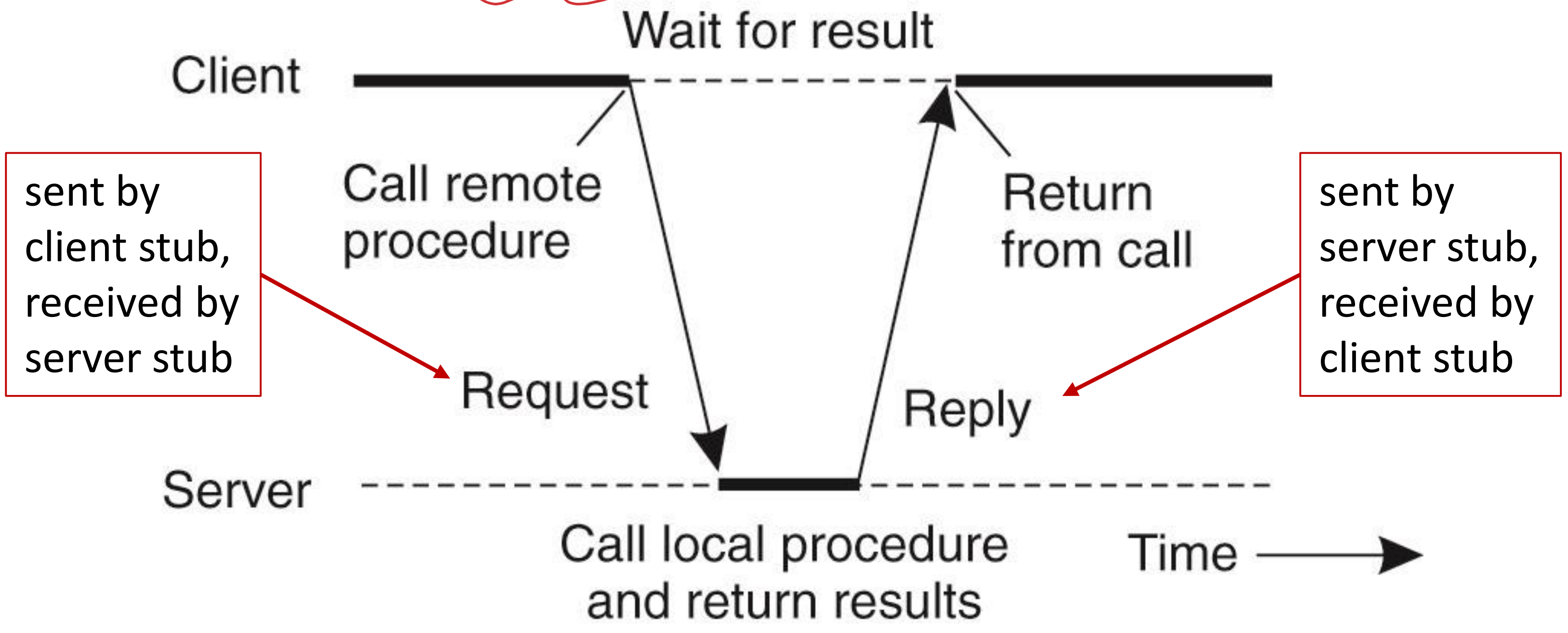
Part (a): stack before a conventional procedure call.

Part (b): stack while executing procedure.

no process share between stacks

Client and server stubs [illegible]

RPCs are implemented using a client-server protocol. The application calls an RPC using a **client stub** that translates the call to a protocol message that is received and processed at the server by a **server stub**.



sent by client stub, received by server stub

sent by server stub, received by client stub

Steps of an RPC

The execution of an RPC entails the following steps:

1. The client process invokes the client stub using an ordinary procedure call.
2. The client stub builds a message and passes it to the client's operating system (OS).
3. The client's OS sends the message to the server's OS.
4. The server's OS delivers the message to the server stub.
5. The server stub unpacks the parameters and invokes the appropriate service handler in the server process.

continued …

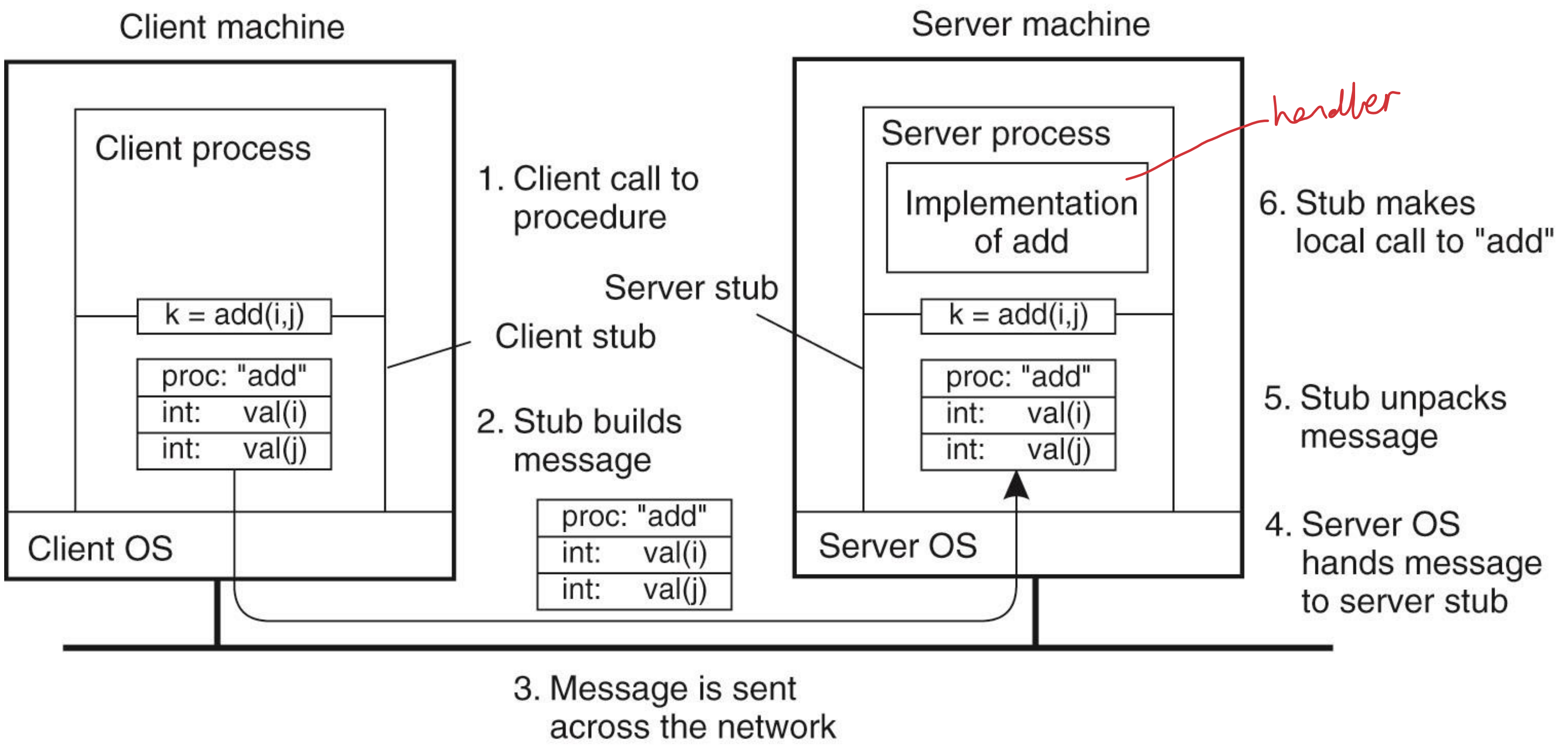
Steps of an RPC

6. The service handler does the work and returns the result to the server stub.
7. The server stub packs the result into a message and passes it to the server's OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS delivers the message to the client stub.
10. The client stub unpacks the result and returns it to the client process.

Food for thought: how many system calls in total are required to execute one RPC?

Steps 1-6 illustrated

Note: the act of packing parameter values into a message in step 2 is called **parameter marshalling**. 信号编集



Representation of values

- Marshalling (and unmarshalling) must take into account differences in the representation of values, such as numbers and strings, on different hardware platforms.
- Intel x86 and x86-64 machines use little-endian 字节序 encoding for numbers whereas Sun SPARC (up to v8) is big-endian. Java
- The Java Virtual Machine (JVM) stores multi-byte data items in big-endian order.
- Some architectures, such as ARM (v3 and up), SPARC v9, as well as Intel IA-64, are bi-endian. They support switchable endianness in data segments, code segments, or both.
- Endianness also concerns the implementation of network protocols, which transmit bits and bytes of data "over the wire." IP uses a big-endian network byte order.

Defining RPC interfaces

The signatures of RPCs are specified using an **Interface Definition Language (IDL)**, which is compiled into a client stub and a server stub. The IDL compiler determines the high-level format of protocol messages for a given RPC, including details such as which parameter goes first and how many bits or bytes it occupies.

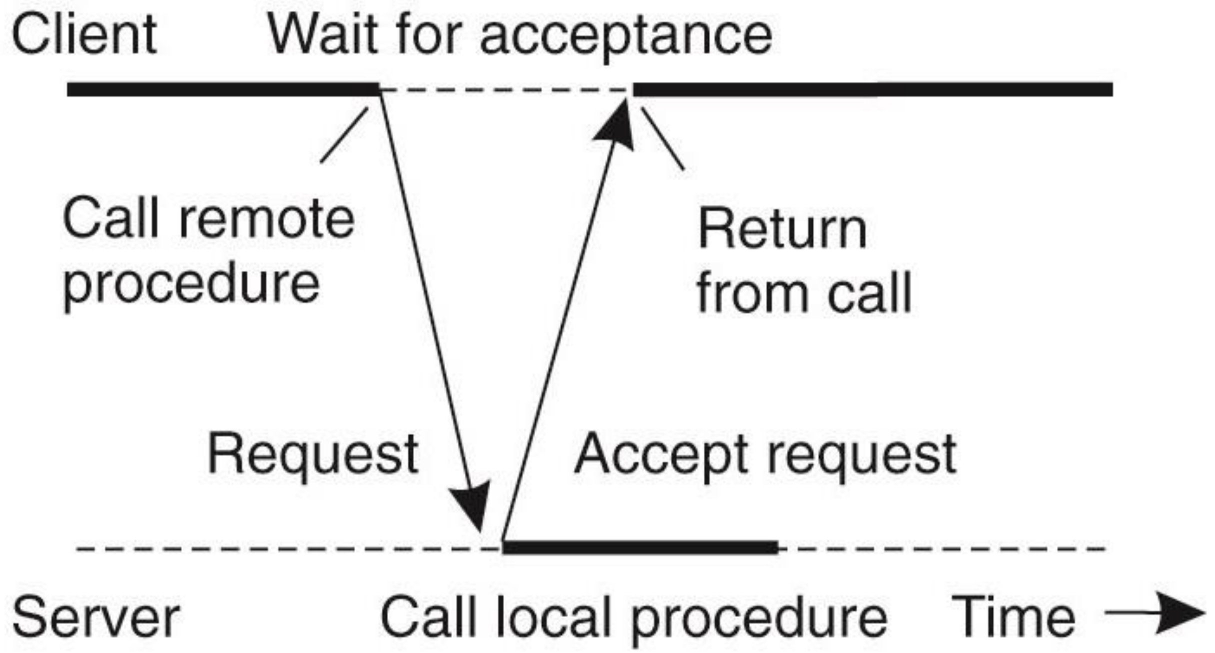
The client and server must agree not only on the endianness, but also on the binary format of primitive data types. For example, integers can be represented in two's complement, characters in 16-bit Unicode, and floating point values using IEEE 754, all in little-endian byte order.

Synchronous vs. asynchronous RPCs

- The client-server interaction shown in an earlier slide is an example of a **synchronous RPC**, meaning that the client waits for the return value while the server executes the procedure.
- In an **asynchronous RPC**, the client resumes executing as soon as the server acknowledges receipt of the request. A callback is initiated by the server (e.g., over the same TCP connection) to return the result back to the client.
- An asynchronous RPC in which a client does not wait for any acknowledgment from the server is a **one-way** RPC.

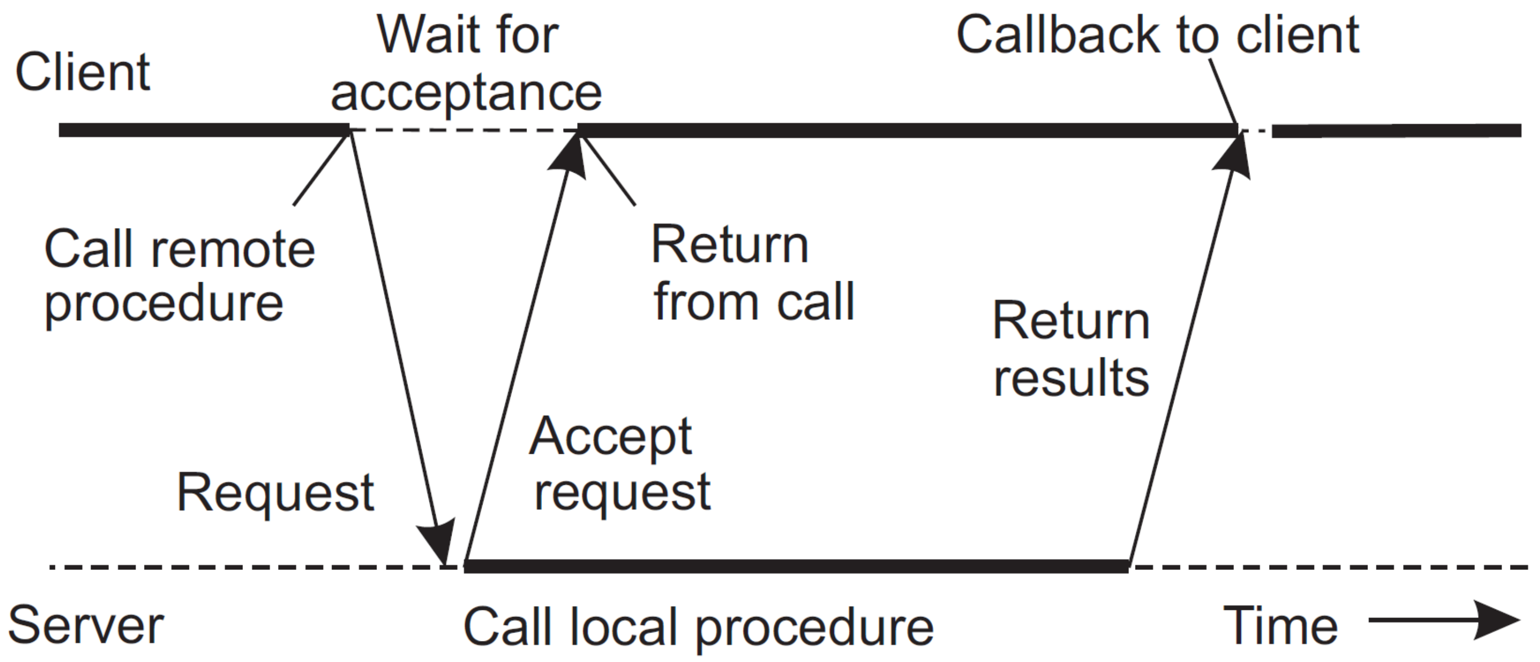
Asynchronous RPC

The client blocks until the server acknowledges receipt of the request. The result of the procedure call is not yet known.



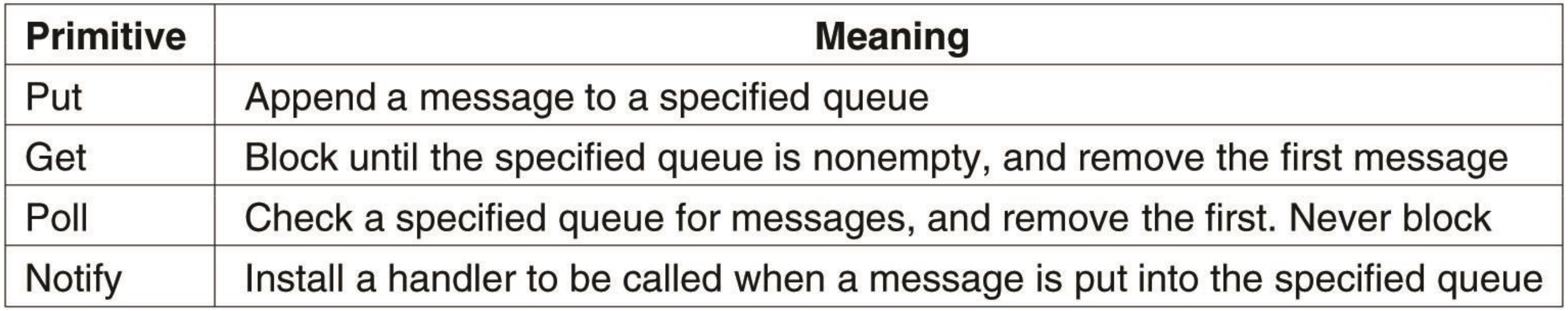
Asynchronous RPC (cont.)

Once the result is known, the server initiates a callback, which is handled by a callback function at the client. The callback function may execute in a dedicated thread.



Message queuing model

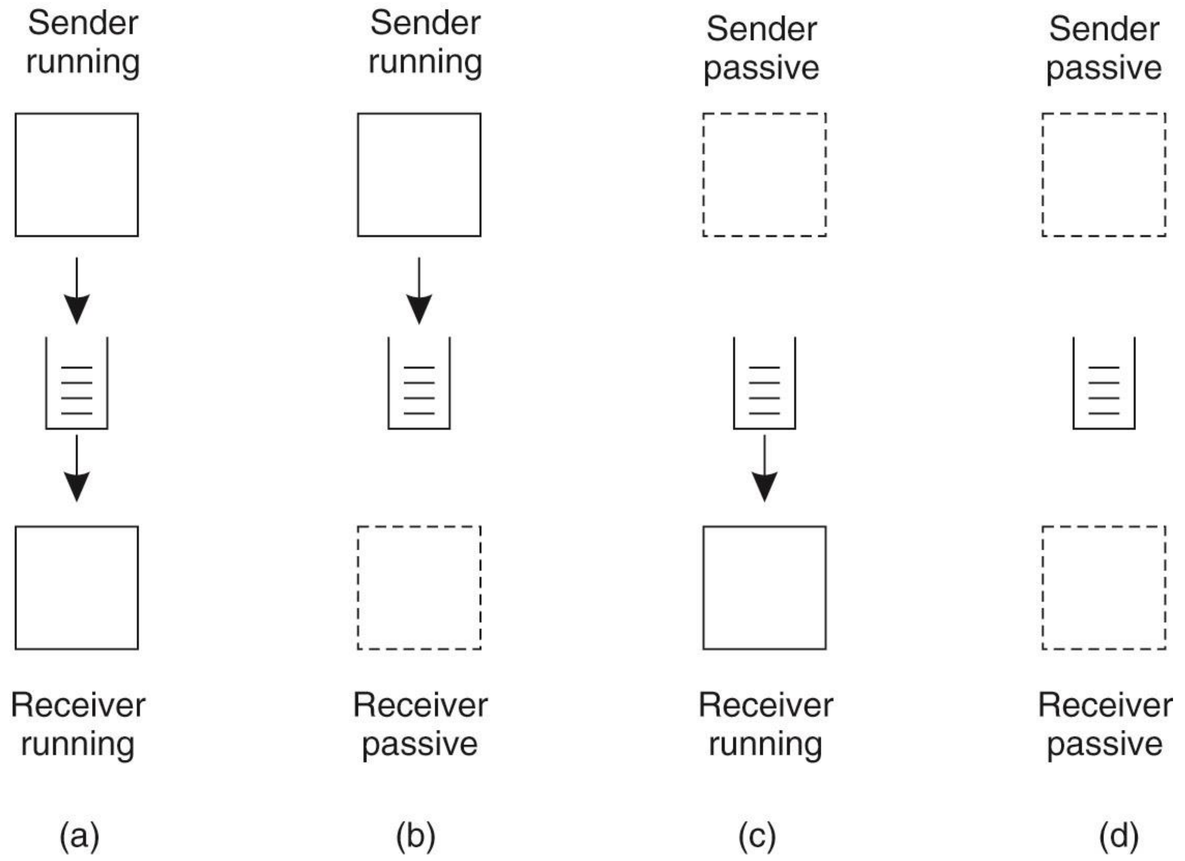
- As an alternative to RPCs, components may also communicate using a **message queue** that persists sent messages until they are consumed by a receiver.
- This enables **persistent communication** that is loosely-coupled in time, for example allowing the sender to send even when the receiver is not running (like Canada post).
- The basic interface of a message queue is shown below.



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

Message queuing model (cont.)

Sender and receiver interact with the queue in four modes.



Message queuing model (cont.)

- The loose coupling between the sender and receiver has some disadvantages. For example, the delivery of a message after it is sent depends on the receiver, and cannot be guaranteed.
- Message queueing is similar to publish-subscribe. Both are examples of **message-oriented middleware (MOM)**, which is characterized by asynchronous message passing.
- Some APIs, such as Java Message Service (JMS), support both message queues and pub-sub.

Coupling among processes

Referential coupling: when one process explicitly references another.

- Positive example: RPC client connects to server using an IP address and a port number.
- Negative example: publisher inserts a news item into a pub-sub system without knowing which subscriber will read it.

Temporal coupling: communicating processes must both be up and running.

- Positive example: a client cannot execute an RPC if the server is down.
- Negative example: a producer appends a job to a message queue today, and a consumer extracts the job tomorrow.

RPC vs. MOM

RPC:

- used mostly for two-way communication, particularly where client requires immediate response from server (e.g., querying a NoSQL storage system)
- the middleware is linked into the client and server processes, and no additional software infrastructure is required
- tighter coupling means that server failure can prevent client from making progress

MOM:

- used mostly for one-way communication where one party does not require an immediate response from another (e.g., disseminating news, batch processing)
- the middleware (e.g., event bus/message queue/tuple space) is a separate component sandwiched between the sender/publisher/producer and the receiver/subscriber/consumer
- looser coupling isolates one process (e.g., publisher) from another (e.g., subscriber), which contributes to flexibility and scalability