Middleware and Interprocess Communication

CS432: Distributed Systems
Spring 2017

Reading

- Coulouris (5th Edition): 4.1, 4.2, 4.6
- Tanenbaum (2nd Edition): 4.3

Outline

- **Introduction to Middleware**
- Introduction to Interprocess Communication
- External Data Representation
- Case Study: MPI

Middleware

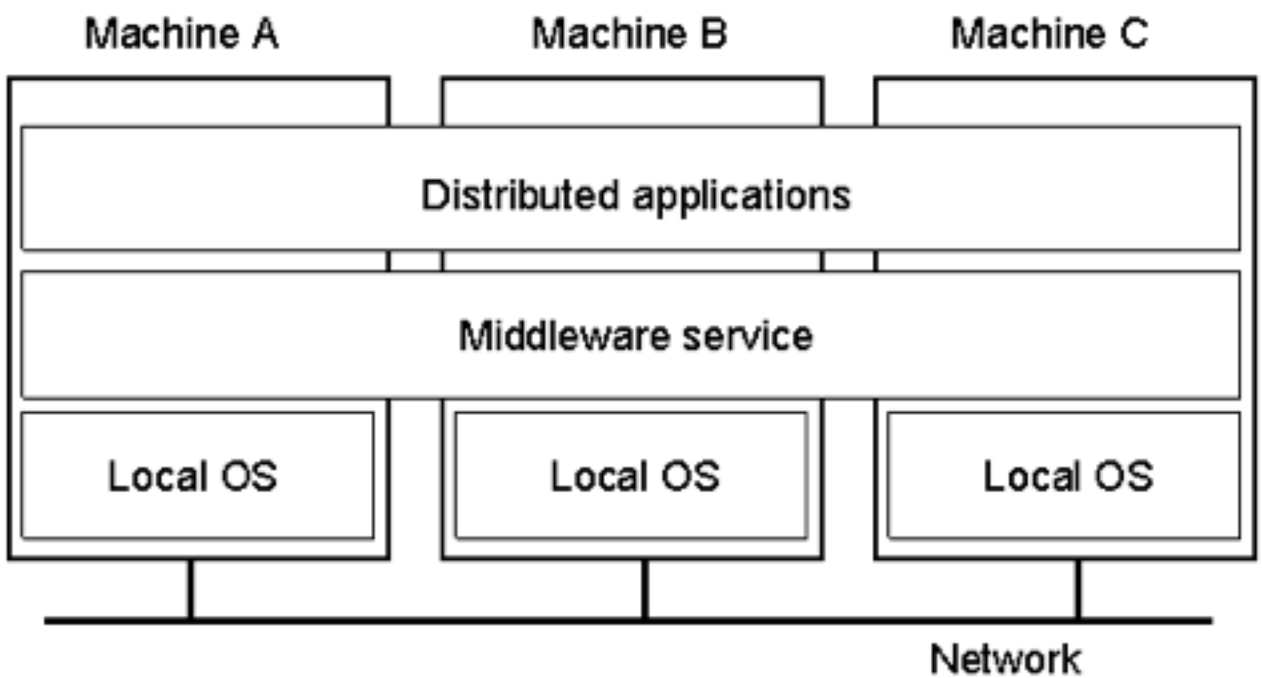
- It mostly, refers to the distributed system layer that enables communication between distributed systems
- Masks the heterogeneity of the operating system, hardware, and network layers
- Provides a uniform computational model for use by the programmers of servers and distributed applications

Middleware Layer

Distributed applications, services,..

Hides lower layers and provides a communication platform

Communications and other hardware infrastructure



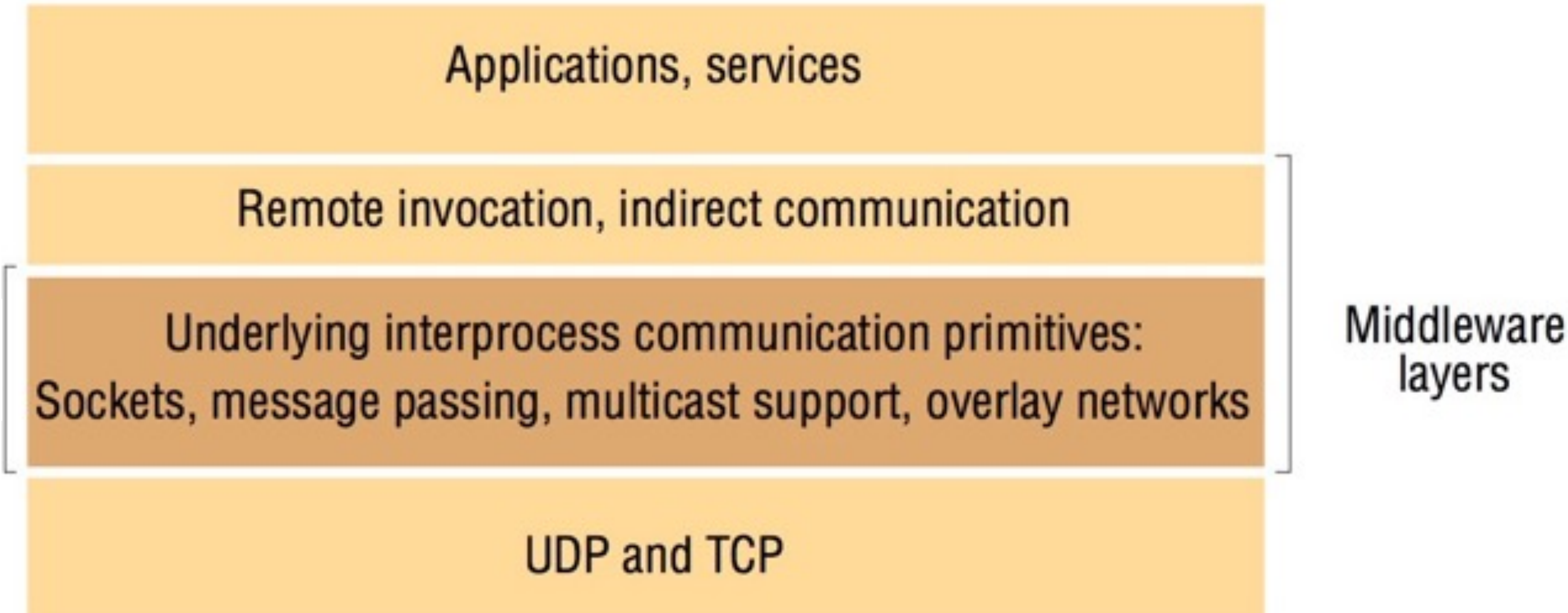
Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

Categories of Middleware

Major categories:	*Subcategory*	*Example systems*
Distributed objects (Chapters 5, 8)	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
Distributed components (Chapter 8)	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
Publish-subscribe systems (Chapter 6)	-	CORBA Event Service
	-	Scribe
	-	JMS
Message queues (Chapter 6)	-	Websphere MQ
	-	JMS
Web services (Chapter 9)	Web services	Apache Axis
	Grid services	The Globus Toolkit
Peer-to-peer (Chapter 10)	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

Middleware Layers

This Lecture



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012

Communication between Processes

- Shared storage:
  - Shared memory
  - Shared files
- Message passing:
  - Sockets
  - Pipes
  - MPI
  - ....
- Others
  - Overlay networks
  - Multicasting
  - ....

Outline

- Introduction to Middleware
- **Introduction to Interprocess Communication**
- External Data Representation
- Case Study: MPI

Interprocess Communication

- The ways that processes on different machines can exchange information
- Communication in distributed systems is always based on low-level message passing as offered by the underlying network
- Message passing between a pair of processes can be supported by two message communication operations: **send** and **receive**
  - Communicate data (sequence of bytes) from sending process to receiving process
  - Synchronization of the two processes

Characteristics of Interprocess Communication

- Synchronous and asynchronous
- Destination of a message
- Reliability
- Ordering

Synchronous and Asynchronous Communication

- A queue is associated with each message destination
  - Sending a message = adding message to remote queue
  - Receiving message = removing message from local queue
- Synchronous: the sending and receiving processes synchronize at every message (blocking send and receive)
  - A sending process (thread) blocks until the message is received
  - A receiving process (thread) blocks until a message arrive
- Asynchronous:
  - The sending operation is non-blocking. Sender proceeds while the message is being transmitted
  - The receiving process (thread) can be either blocking or non-blocking

Message Destinations

- Messages are sent to (*Internet address*, *local port) pairs*
- A port has exactly one receiver but can have many senders (multicast ports are exception)
- Fixed location: client uses a fixed Internet address to refer to a service, then the service has to always run on the same computer
- Location transparency: Client programs refer to services by name and use a name server to translate their names into server locations at runtime

Reliability

- Defines reliable communication in terms of validity and integrity
- **Validity**: messages are guaranteed to be delivered despite a ‘reasonable’ number of packets being dropped or lost
  - An unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost
- **Integrity**: messages must arrive uncorrupted and without duplication

Ordering

- Some applications require that messages be delivered in ***sender order***
- These applications will consider it as a failure if a sender messages are received out of order

Outline

- Introduction to Middleware
- Introduction to Interprocess Communication
- **External Data Representation**
- Case Study: MPI

External Data Representation

- Information is represented in a program as data structures (e.g set of interconnected objects)
- Information in messages consists of sequence of bytes
- Objective: data structures must be flattened before transmission and rebuilt at arrival
- Challenges: representation of basic types:
  - floating points
  - big-endian vs little-endian
  - character encoding

Marshalling/Unmarshalling

- Possible solutions:
  - Convert values to an external format before transmission, then convert it to local format when arrive at destination
  - Values are transmitted in the sender's format
- **Marshalling** is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message
- **Unmarshalling** is the process of disassembling messages on arrival to produce an equivalent collection of data items at the destination

Examples of External Data Representation

- CORBA's common data representation: external representation for the structured and primitive types. Uses and IDL (Interface Definition Language)
- Java's object serialization: flattening and external data representation of any single object or tree of objects (No IDL)
- XML or JSON (lightweight): defines a textual format for representing structured data
- Google protocol buffers (lightweight)

Outline

- Introduction to Middleware
- Introduction to Interprocess Communication
- External Data Representation
- **Case Study: MPI**

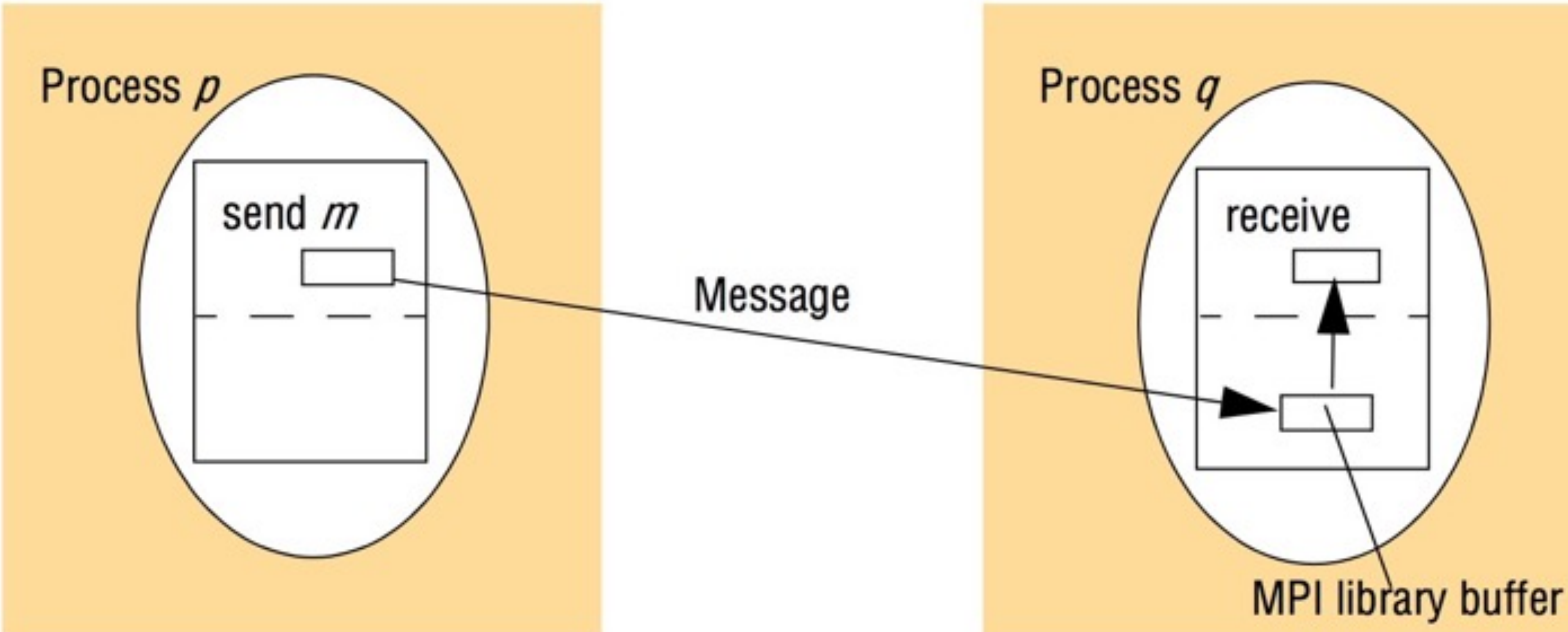
Message Passing Interface (MPI)

- Used when performance is paramount, for instance in high performance computing (HPC)
- Objective: portability through presenting a standardized interface independent of the operating system or programming language-specific socket interface
- MPI is flexible
- Interface is available as a message-passing library available for a variety of operating systems and programming languages, including C++ and Fortran

MPI vs. Sockets

Sockets	MPI
Support only simple send and receive primitives	Provide more variations of send and receive operations that handle advanced features such as buffering and synchronization
Designed for TCP/IP	Suitable for other protocols that are typically used for HPC clusters. Example: infiniband

Point-to-Point Communication in MPI



- An MPI library buffer in both the sender and the receiver is used to hold data in transit

Blocking in MPI

- Blocking = ‘blocked until it is safe to return’
  - application data has been copied into the MPI system and hence is in transit or delivered
  - application buffer can be reused (for example, for the next *send* operation)
- Various interpretation of ‘safe to return’ are used:
  - *MPI_Ssend* is the synchronous blocking send. Safety is interpreted as delivered
  - *MPI_Bsend* interprets safety as allocating and copying data to the library buffer (in transit)
  - *MPI_Rsend* interprets safety as the receiver is ready to accept the message and hence can be removed from library buffer (no handshake)

Send Operations in MPI

Send operations	**Blocking**	**Non-blocking**
Generic	*MPI_Send:* the sender blocks until it is safe to return –that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend:* the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via MPI_Wait or MPI_Test.
Synchronous	*MPI_Ssend:* the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend:* as with *MPI_Isend*, but with *MPI_Wait* and MPI_Test indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend:* the sender explicitly allocates an MPI buffer library (using a separate MPI_Buffer_attach call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend:* as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend:* the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend:* the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

Thank You