Universität Stuttgart
Master of Science

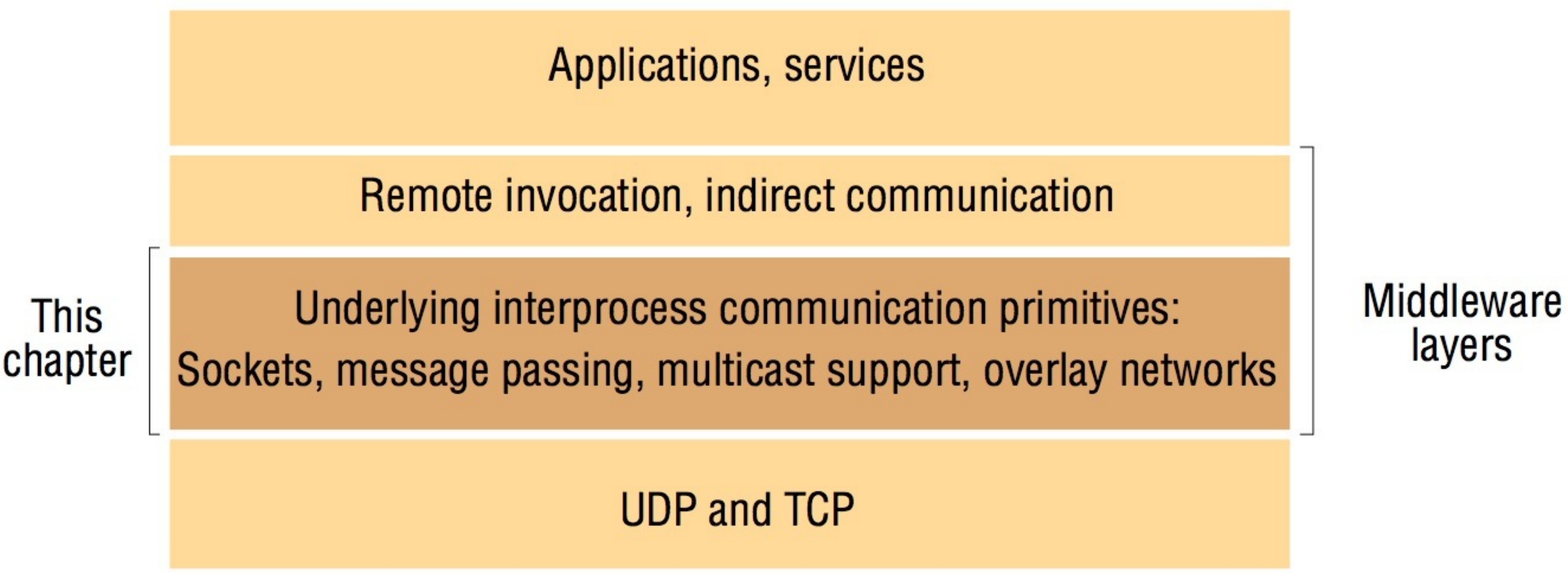
Interprocess Communication

Marco Aiello

Interprocess communication: middleware

- Middleware provides API to enable the communication of processes as a result of application requests relying on the transport layer

Figure 4.1
Middleware layers



Interprocess communication

- **Message passing:** *send* and *receive* of messages <destination, content>
- **Synchronous vs. asynchronous communication:** sync. implies blocking send and receive (multithreading allows the use of blocking operations, otherwise processes would hang)
- **Destination:** network address + port (a destination within a host which identifies a receiving process). Ports are unique.

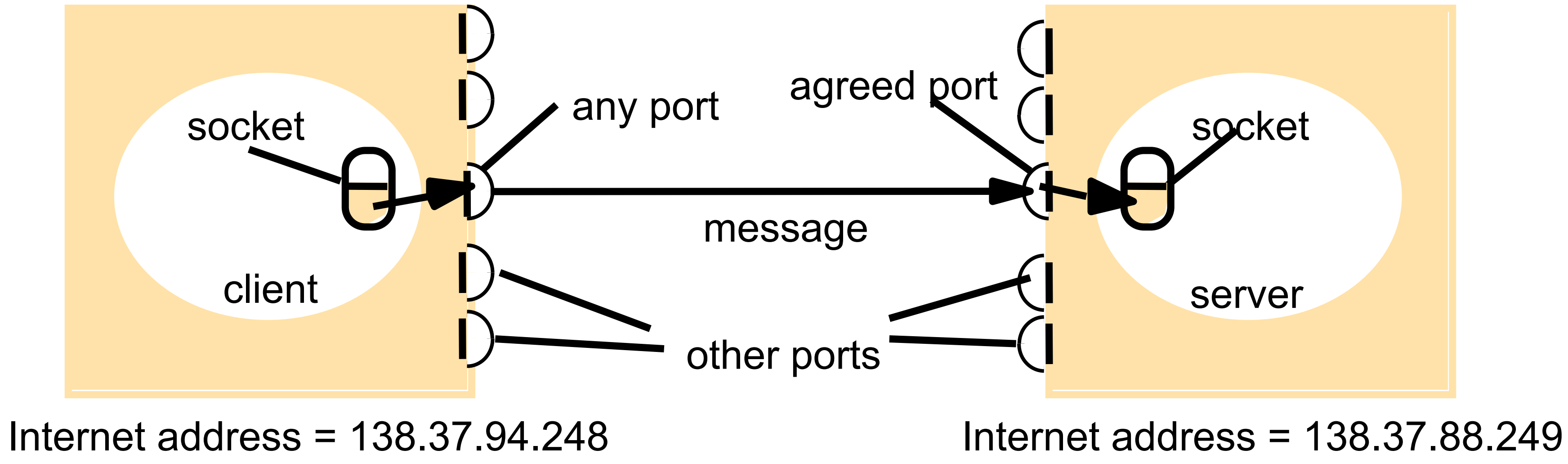
Sockets

- A *socket* is the abstraction of a connection from the process' perspective

- **E.g., on UNIX a pair of client-server processes each establish their own socket:**
- *client side:*
  1. Create a socket with the socket() system call
  2. Connect the socket to the address of the server using the connect() system call
  3. Send and receive data via UDP or TCP, for instance.

- *server side:*
  1. Create a socket with the socket() system call
  2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number and the host address.
  3. Listen for connections with the listen() system call
  4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
  5. Send and receive data via UDP or TCP, for instance.

Sockets and ports



User Datagram Protocol (UDP)

- Messages are sent without acknowledgement of receipt or retry
- **Message size:** the receiver decides on the size of the buffer for receiving purposes. Longer messages are truncated
- **Blocking:** usually, non-blocking send and blocking receive (eventually with timeout)
- **Receive from any:** this is the default, but a fixed sender socket is possible

User Datagram Protocol (UDP)

- **Failure model:**
  - *Omission failures:* datagrams may be dropped because there is no listening process on a port, or because the buffer is full
  - *Ordering failures:* there is no guarantee of order of delivery
- UDP is less reliable than TCP, but as it has less overhead it may be convenient to use. For example, DNS uses UDP

Java UDP API

- DatagramPacket:

  <message, length, internet address, port>

- DatagramSocket:

  To create and manages socket connections for sending UDP datagrams

Transfer Control Protocol (TCP)

- Gives the abstraction of a streaming connection. This includes hiding of:
  - message size
  - lost messages (if no ack, then resend)
  - full buffers (flow control to control consume speed)
  - unordered messages (numbering scheme of packets)
  - destination (connection oriented)
- Failure model:
  - Checksum for integrity checking
  - Timeouts for validity checking (problem: no way to differentiate between network problems and connection flush of one of the processes)

Java TCP API

- ServerSocket (for server listening on a port)
- Socket (for connecting to remote processes)

External data representation

- The transport layer is only concerned with the reliable transmission of sequences of bytes, but what about the type of information of transmitted data?

1. The communicating processes agree on an external format (e.g., ASCII characters)
2. The sender informs the receiver of the format together with the data

Marshalling

- **Marshalling:** the process of encoding a set of data items into a coherent form for the purpose of transmission
- **Unmarshmaling:** the corresponding decoding process

Three examples

- Corba common’s data representation
- Java’s serialization
- Extensible Markup Language (XML)

  - middleware or application marshalling/unmarshalling?
  - transformed into binary or textual?
  - self-describing or external references? (namespaces)

  - other examples: protocol buffers, JSON

Corba Common Data Representation

- data types for invocations
- 15 primitive types
- constructed types (ordered sequence)

Figure 4.7
CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Figure 4.8
CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	`5`	*length of string*
4–7	`"Smit"`	*'Smith'*
8–11	`"h___"`	
12–15	`6`	*length of string*
16–19	`"Lond"`	*'London'*
20-23	`"on__"`	
24–27	`1984`	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Figure 4.10 XML definition of the Person structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

Figure 4.11 Illustration of the use of a namespace in the *Person* structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
        <pers:name> Smith </pers:name>
        <pers:place> London </pers:place >
        <pers:year> 1984 </pers:year>
</person>
```

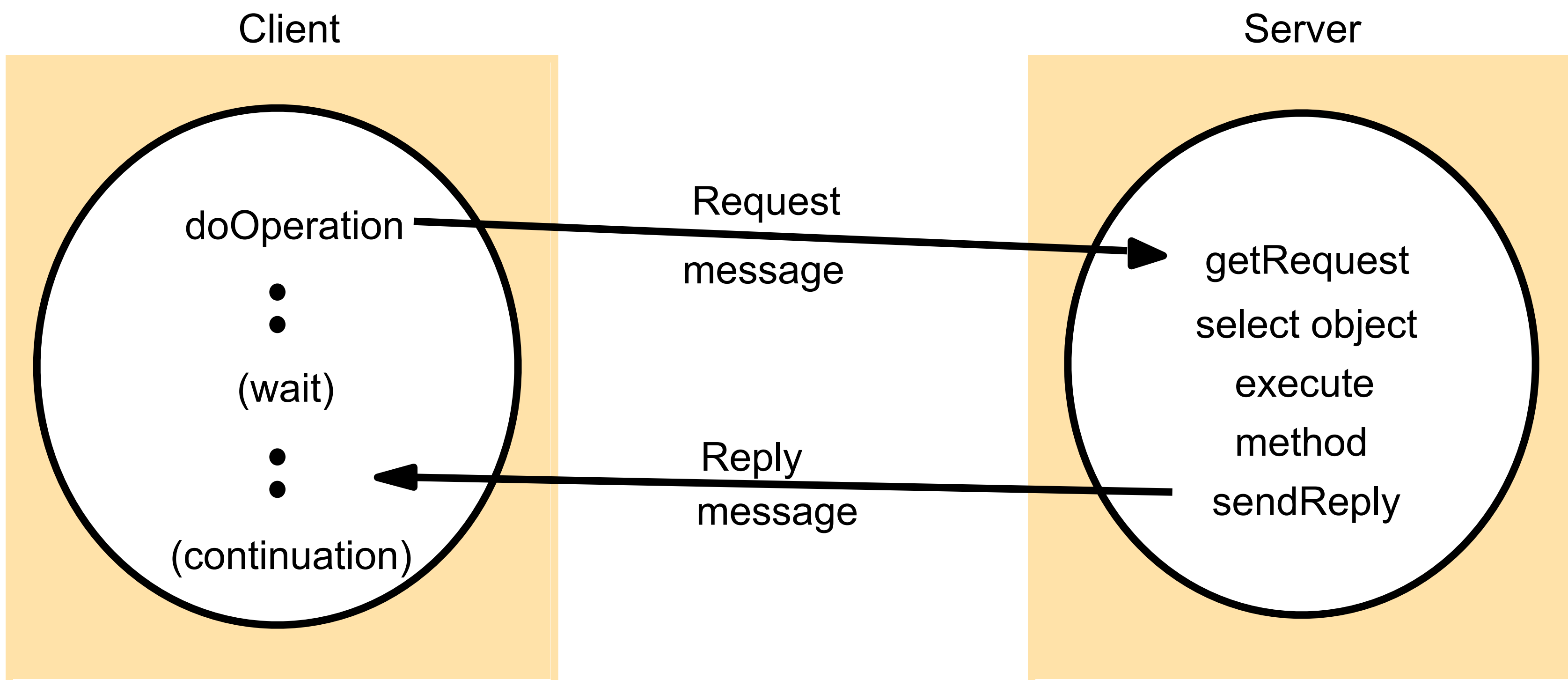
Figure 4.12 An XML schema for the *Person* structure

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions      >
        <xsd:element name= "person" type ="personType" />
                <xsd:complexType name="personType">
                        <xsd:sequence>
                                <xsd:element name = "name"  type="xs:string"/>
                                <xsd:element name = "place"  type="xs:string"/>
                                <xsd:element name = "year"  type="xs:positiveInteger"/>
                        </xsd:sequence>
                        <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
                </xsd:complexType>
</xsd:schema>
```

Client-Server communication

- Typical example of interprocess communication, based on the request-reply protocol
- Typical facts:
  - Acknowledgements are redundant (the reply is an ack)
  - Connection implies an extra pair of messages
  - Flow control is redundant

Request-reply communication



Operations of a request-reply protocol

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

A request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

Request-reply failure model

- Omission failure and ordering failure
  - Timeout of the *doOperation* or of the *send*
  - Discarding duplicate requests
  - Loosing reply messages
  - History to cope with duplicate requests (when the reply might have been lost)

RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

An example of request-reply: HTTP

- A protocol implemented over TCP
- Address of resource= URL
  - http://<host address>:<port>/resource
- Content negotiation (external data formats)
- Authentication is supported
- Persistent connections are supported (a request of a webpage may consist of several individual request-replies of resources)

HTTP request message

method	*http URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.nike.com:8080/a.htm	HTTP/ 1.1		

POST

netaddress,
port,
resource identifier

HEAD

PUT

DELETE

OPTIONS

TRACE

HTTP reply message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Group communication

- Multicast transmission: a message sent to a specified group of recipients
- Examples:
  - Fault tolerance based on replicated services (requests go to all servers)
  - Spontaneous networking (all members of the network receive messages)
  - Better performance through replicated data (the updated data goes to all storing the data)
  - Event notification

IP multicast

- Multicast group specified by a class D address
- Available only as UDP
- Hosts may join and leave multicast address dynamically
- Multicast routers allow for multicast messages over the whole Internet, not only local networks. Time To Live (TTL) may specify the max router hops
- Java: *multicastSocket*