Lecture 5 & 6

Wednesday, 20 September 2023 13:16

Interaction styles

↳ realized as middleware services by middleware frameworks

interaction styles pertain to communication aspects in the views

↓

typically: in the process view

OSI model: 7 layers

7: application
6: presentation
5: session
1-4: same as 5-layer stack (TCP/IP)

physical: send bits on medium
data link: detect/correct errors in frames
network: send packets from sender → receiver using multi-hop routing
transport: breaks messages into packets
session:
presentation:
application:

n^{th} level communication unit is payload of $(n-1)^{th}$ level unit of communication

data link layer has not only header, but also trailer

TCP: connection oriented, reliable, stream-oriented
UDP: unreliable (best effort)

referentially \ temporally	coupled	decoupled
coupled	direct	mailbox
decoupled	event-based	shared data space

the need for synchronizing data between different locations may make the system less reliable when referential coupling is present

---

middleware layer

↳ provide common services + protocols that can be used for many different applications

communication protocols

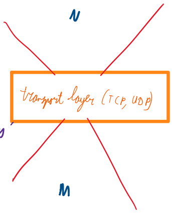
marshaling

naming

security

hourglass approach: connect N applications with M diverse physical layers

using a common transport layer, only N+M instead of N×M 'connectors' are necessary



middleware may merely provide message passing, or it may provide advanced features as well

the latter approach is useful for a thin client (since the middleware can handle most functions on a fat server)

adapted middleware layer

application
middleware
operating system
hardware
network

transient communication → discard undeliverable message, or at receiver
persistent communication → stores message as long as it takes to deliver it
asynchronous communication → continue after submitting for transmission
synchronous communication → sender is blocked until request is known to be accepted

SaS: synchronize at request submission
SaD: synchronize at request delivery
SaP: synchronize after processing by server

remote procedure calls

in process view
using a message sequence chart

↓

procedures are executed on separate machine

stubs are used to construct/destruct (build/unpack) messages for RPC over network

concerns for RPCs:

scalability

reliability → dependency between calls may cause inefficiency/ problems with reliability

performance

realization elements:

- marshalling
- protocol
- parameters
- how to include RPC in program
- bind client/server, server discovery
- partial failure handling → at most once, at least once, no guarantees, precisely once
- synchronization

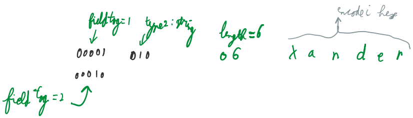
serialization is the activity by which the state of an object is converted into a byte stream in such a way that the byte stream can be converted back into a copy of the object

marshalling is the activity by which a stub converts local application data into network data XDR (external data representation) and packages the network data into packets for transmission (i.e. converting between formats)

protocol buffers are an XDR-format developed by google

```
messagePerson {
    required string user_name = 1;
    optional int64 favourite_number = 2;

}
```



RPCs generally cannot use pointers to pass results, as the pointer is not valid on other machines
→ address space is not the same, and remote referencing is complicated

passing results is not transparent, as the encoding cannot be hidden from clients

RPCs typically require remote resources to be useful

asynchronous RPC: if there is no result, there is no need to wait for it
e.g. when the RPC performs a state change

deferred synchronous RPC: let the client do something while waiting for the result. but the client should wait for the acceptance of its all/request

a one-way RPC caller continues immediately, without even waiting for acceptance

multicast RPC: client sends request to multiple servers

RPC failure models

1. request loss, request duplication
2. response loss, duplication
3. server crash
4. client crash

we assume non-malicious environment

responses to failures

1. timeout / explicit acknowledgement
2. define semantics explicitly at most / at least / exactly once
3. idempotent requests
4. prevent server from re-execution
   timestamping
5. divide time into epochs; recreate process if no response

remote method invocation → mostly the same, but involves classes / objects
used in Java

a proxy acts on behalf of another component, can implement filtering/caching, and is typically client side

(client) stub implements an interface for a remote object

server stub / skeleton: perform calls of client stub + handle messaging

a remote object is a distributed object whose state is not distributed

an adapter is a component that relays calls to an object interface and manages it

a broker is a component that handles and translates calls, and manages the binding between references and objects

RMIs are very similar to RPC, except that
1. an object represents a context/state together with operations on it
2. clear separation between interface and implementation
3. references to objects occur naturally in a program

RMIs pass local objects by value/result
& remote objects by reference

message-oriented middleware RPC & RMI are tightly coupled synchronous mechanisms
aims at high-level persistent asynchronous communication

TCP sockets give a point-to point byte-oriented transport service
roles : client , server
functions : socket, bind, listen, accept, connect, send, receive, close
TCP is reliable, UDP is not

a server typically runs in an infinite loop

sockets are low level , but often used the same way

example alternative (middleware): zeromq

socket patterns:
request - reply
publish - subscribe
pipeline

in queue - based messaging , the client and server
can be active or passive

message queuing system primitive operations : put , get, poll, notify