TCP/IP Sockets

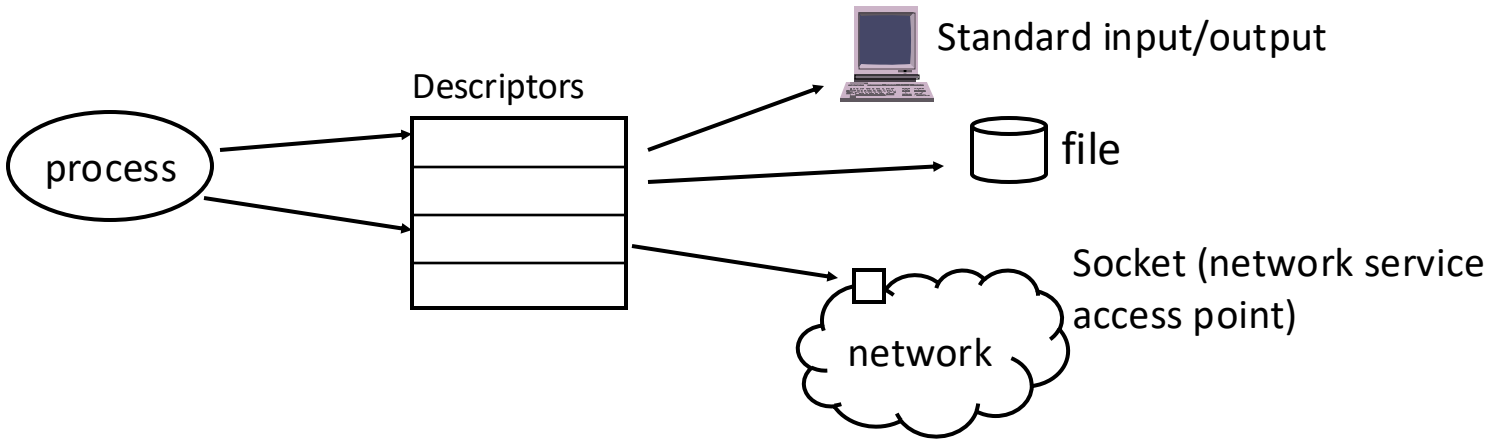
References for study: K.L. Calvert, M.J. Donahoo
TCP/IP Sockets in Java, 2nd Edition, O'Reilly 2011

The Socket API

- The de-facto standard API for accessing network services provided by Internet layers 4-3-2
- Born with **BSD Unix**, in the **C language**, but later implemented for all major OSs and languages
- The name comes from the main abstraction of the API: the **socket** (an endpoint of a layer 4-3-2 protocol)

General Features of the Socket API

- Procedural API in the C language
- Main idea: extend the conventional Unix I/O model:



- One model for several protocol stacks
- Later implementations have partially diverged from these original features

Sockets

- A **socket** is the abstraction of an inter-process communication channel endpoint or SAP
- Sockets "live" in **domains**, each one characterized by its **protocol family** and **address family**
- Communication between two different domains is impossible.
- Domain examples:

Domain	Protocol Family	Address Family
ARPA Internet	PF_INET	AF_INET
Internet with IPv6	PF_INET6	AF_INET6
ISO/OSI	PF_ISO	AF_ISO
Unix pipes	PF_UNIX	AF_UNIX

Socket Features: the TYPE

- The TYPE of a socket specifies the *service type* accessible through the socket. The main supported types are:
  - **SOCK_STREAM**

    continuous bidirectional byte stream (without delimiters), transmitted reliably (same byte order, no duplications) (**connection-oriented** service, offered by layer 4, e.g. TCP).
  - **SOCK_DGRAM**

    bidirectional message (datagram) **delivery** without reliability guarantee (messages can be delivered out of order and can be duplicated) (**connectionless** service, offered by layer 4, e.g. UDP).
  - **SOCK_RAW**

    direct access to the services provided by layer 2-3 protocols

Socket Features: the PROTOCOL

- In each domain, and for each socket type, a protocol to be used with the socket can be selected. For example, in the PF_INET domain the following choices are possible:

SOCK_STREAM type

- TCP Protocol (IPPROTO_TCP)

SOCK_DGRAM type

- UDP Protocol (IPPROTO_UDP)

SOCK_RAW type

- ICMP Protocol (IPPROTO_ICMP)
- IP Protocol (IPPROTO_RAW)

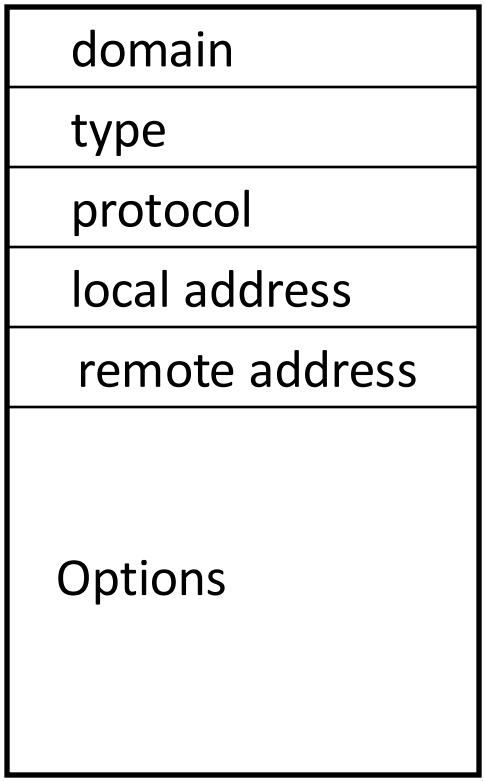
Socket Features: the OPTIONS

- Options specify various other socket features

Examples:

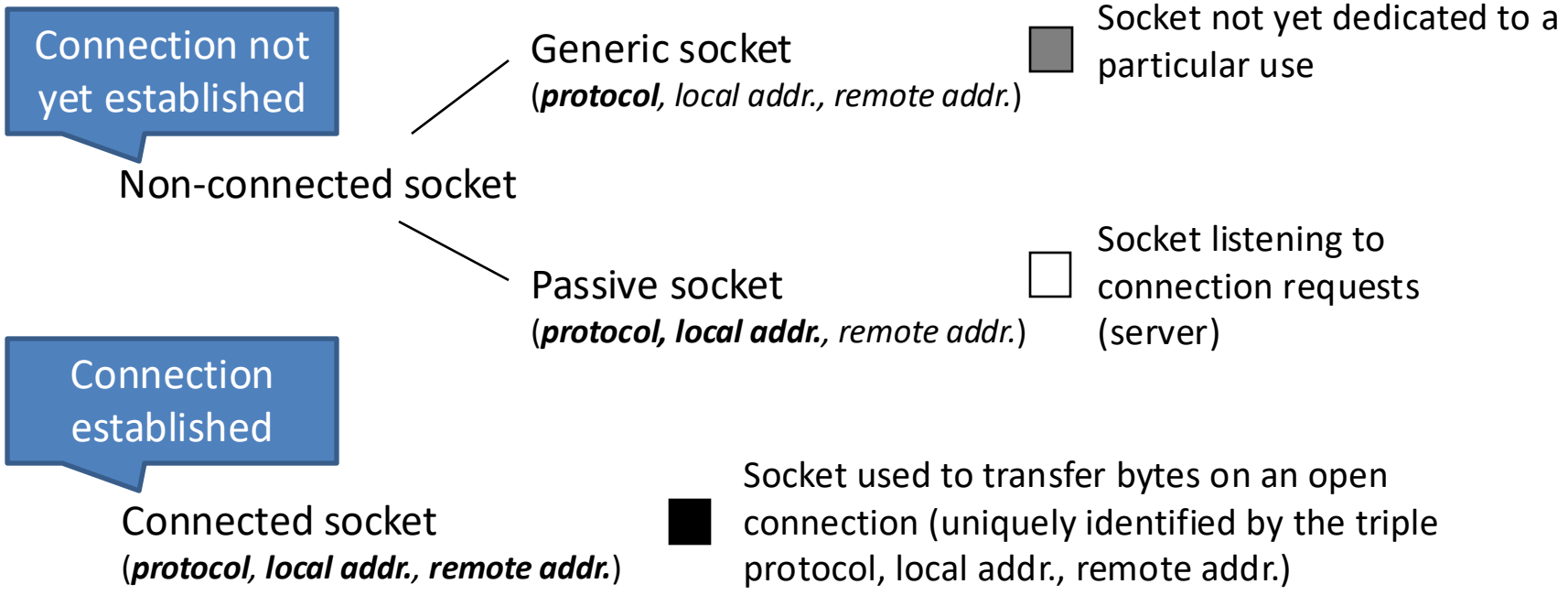
- SO_RCVBUF (size of receiver buffer)
- SO_SNDBUF (size of transmitter buffer)
- SO_LINGER (if enabled, delays connection closing when undelivered data are present in the socket buffer)
- SO_KEEPALIVE (if enabled activates transmission of periodic keepalive messages: if the response to a keepalive message is not received, the connection is closed)

Data Structure Associated with a Generic Socket

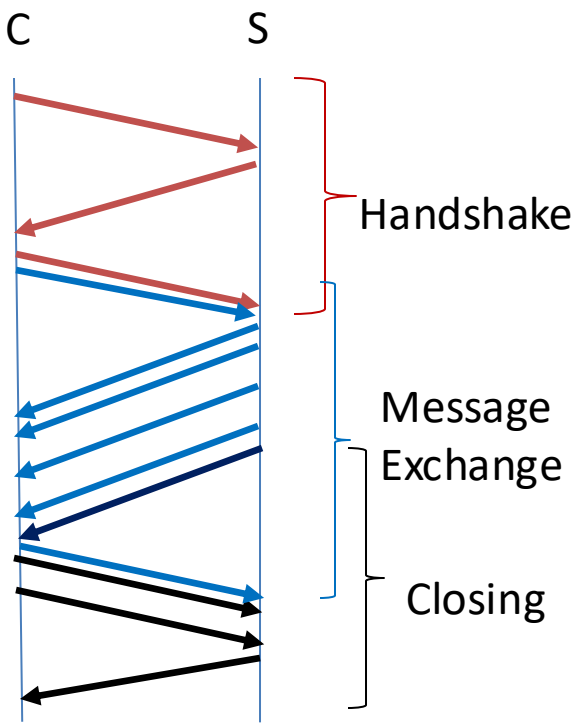


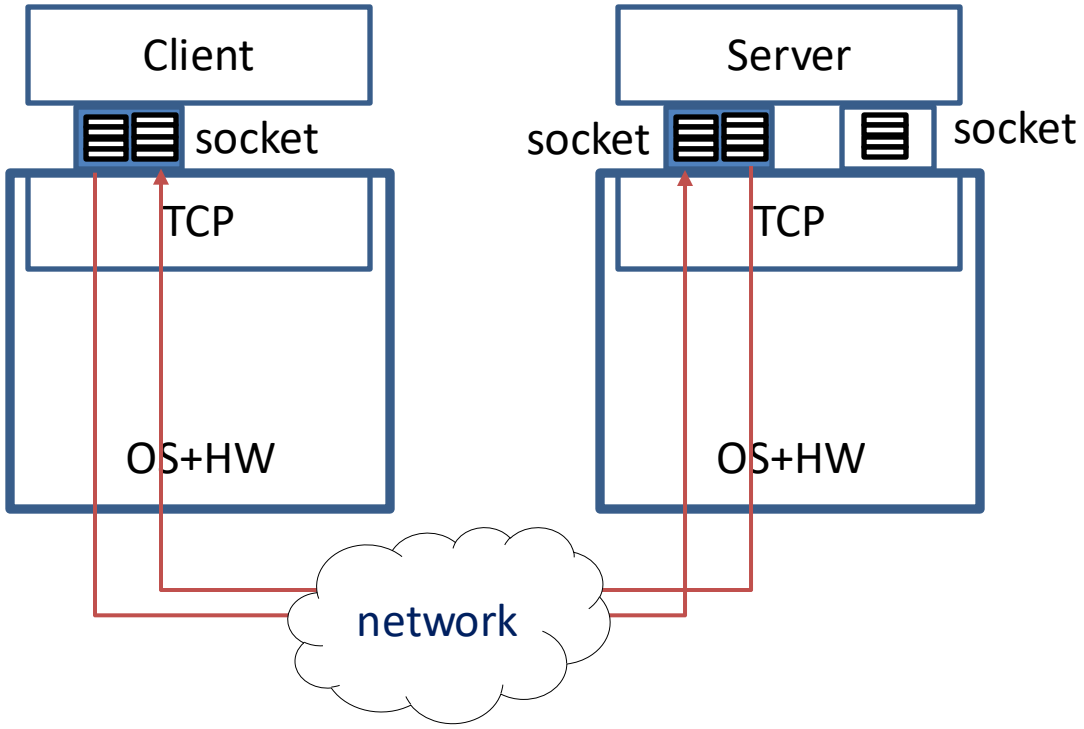
Using TCP (STREAM) sockets

- Endpoints can take different states/forms, according to the role they are playing in communication:



Connected and Passive Sockets



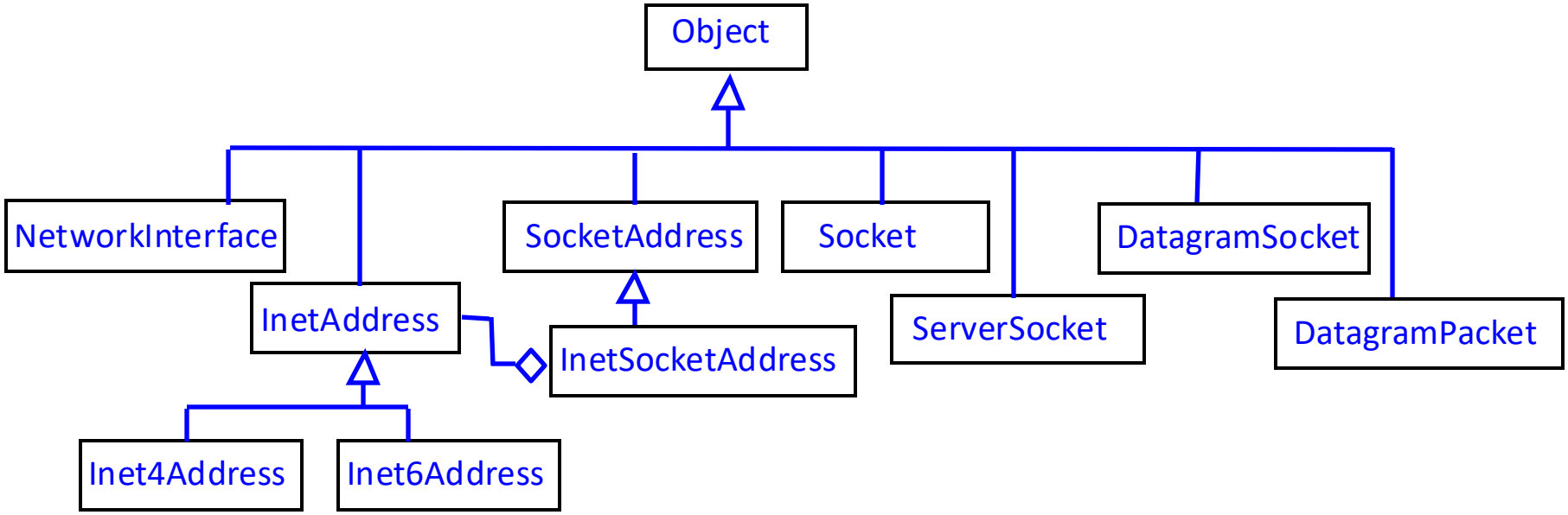


Operations Offered by the Socket API for Accessing Layer 4 (TCP)

- Allocate local resources for communication
- Specify endpoints
- Open a connection (client-side)
- Wait for the establishment of a connection (server-side)
- Send/receive data on a connection (including urgent data)
- Be notified when data arrive
- Gracefully terminate a connection or abort a connection
- Respond to graceful termination requests and abort conditions
- Release resources when a connection terminates

The Java Implementation of the Socket API

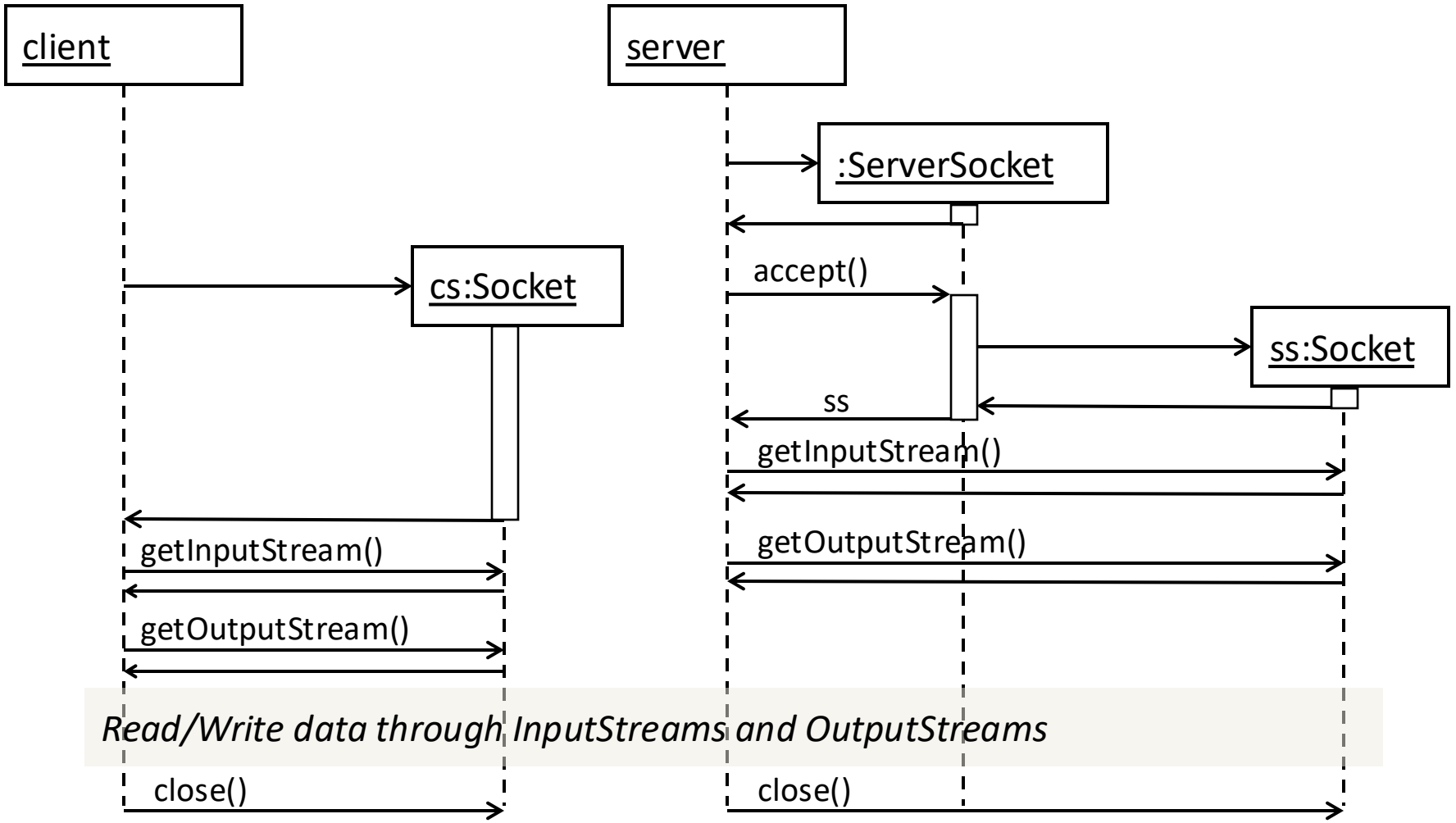
- The Java Socket API includes all the features of the original Socket API, with a special focus on the Internet domain (TCP/IP sockets)
- The core classes are in the **java.net** package



The java.net Package

Class	Meaning
NetworkInterface	interface information (name, IP addresses, etc)
InetAddress	Internet address (with methods for DNS services)
SocketAddress	Generic socket address (no address family)
InetSocketAddress	Internet socket address
Socket	Connected STREAM socket
ServerSocket	Passive STREAM socket
DatagramSocket	DGRAM socket
DatagramPacket	Datagram

Using Connected Sockets



Establishing a Connection

- Server-side
  - create ServerSocket

  ServerSocket(int port, int backlog, InetAddress bindAddr)
  - call accept() to wait for connection establishment

  => returns Socket when connection established

All parameters are optional

- Client-side
  - create Socket

  Socket(InetAddress address, int port, InetAddress localAddr, int localPort)

  Socket(String host, int port, InetAddress localAddr, int localPort)

Exceptions

java.io.IOException
|-- java.net.SocketException
| |-- BindException,
| |-- ConnectException,
| |-- NoRouteToHostException,
| |-- PortUnreachableException
|
|-- java.net.UnknownHostException

java.lang.SecurityException

Closing a Connection

- Socket.**close**() terminates *both* sides of the connection
  - a FIN is sent by TCP after all buffered data have been delivered
  - a FIN from the peer will be automatically acknowledged by TCP
  - no further possibility to read from or write to the socket
- Socket.**shutdownOutput**() terminates the output side
  - a FIN is sent by TCP after all buffered data have been delivered
  - no further possibility to write to the socket
- Socket.**shutdownInput**() terminates the input side
  - undelivered data are discarded
  - a FIN from the peer will be automatically acknowledged by TCP
  - no further possibility to read from the connection

How to Detect Closing

- A read operation on the socket input stream returns -1 if the incoming side of the connection has been closed by the peer
  - when this happens, the process may want to start the closing of the other side of the connection (after having sent any outstanding data)

Example: Echo Protocol on TCP

- Implement the simple echo service provided by Unix systems (on standard port 7):
  - the server continuously echoes back the bytes received on the connection
  - the connection is closed by the client when done

  => TcpEchoServer0.java
- Implement a client for the service
  - the client operates with line buffering

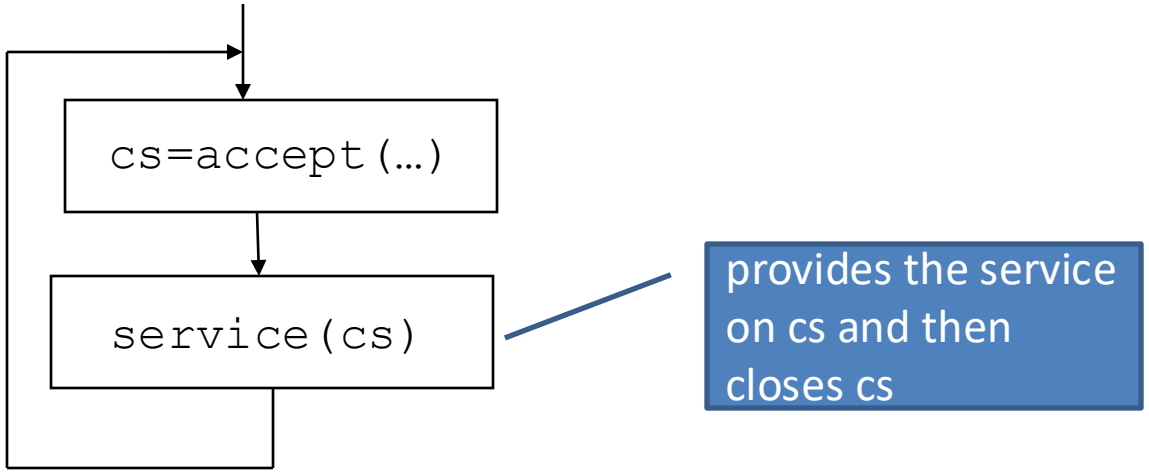
  => TcpEchoClient0.java

Blocking Operations and Timeouts

- Some operations are blocking
  - accept
  - new Socket (connect)
  - read operations on InputStream (block until 1 byte available)
  - write operations on OutputStream (block until space available)
- When blocking is undesired
  - the operation can be called by a dedicated thread
- Timeouts for limiting read blocking can be set by setting the SO_TIMEOUT socket option

Sequential Servers

- The **sequential** model is the simplest one: the server serves requests sequentially in FIFO order:
  - only when a request has been completely served, the next one is processed
  - for TCP servers, the algorithm has the following structure:



Problems of Sequential Servers

- Each request must wait until all previous requests have been served
- Waiting occurs even when the CPU is idle!
- When the request rate is high (time between requests and service time are comparable) it is likely that a request arrives while the server is busy.

$\Rightarrow$requests may be discarded, or clients may timeout.

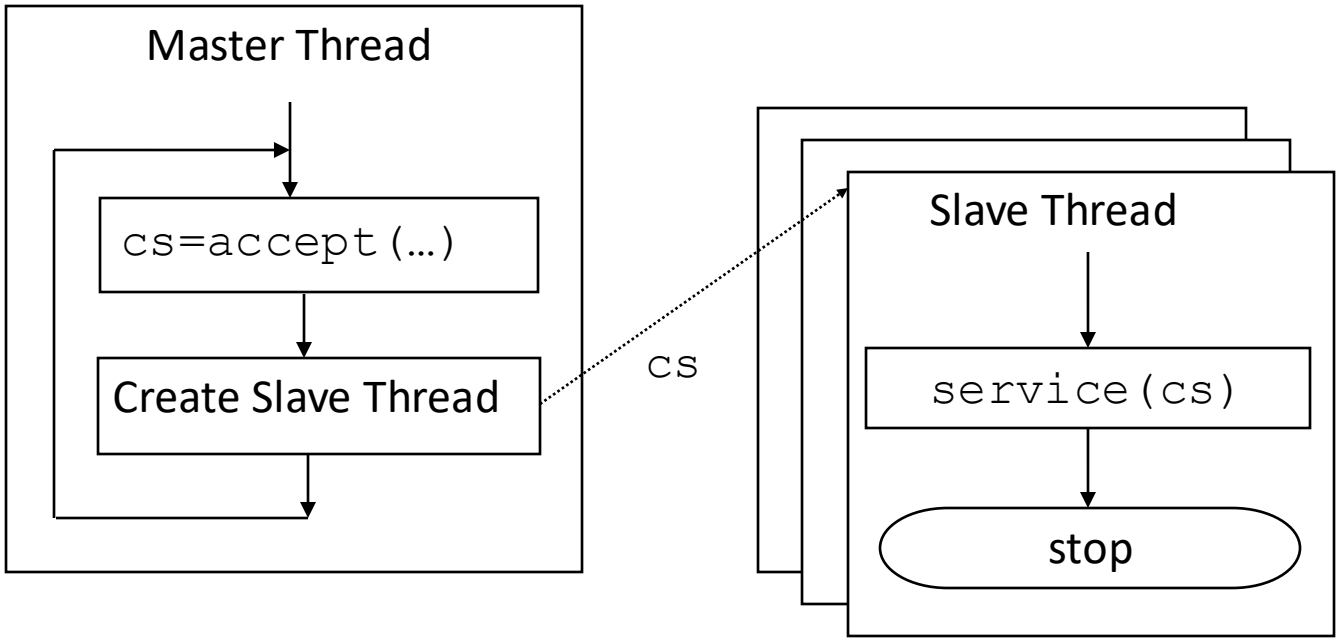
Possible solution: serve requests concurrently

Concurrent Servers

- May serve more requests concurrently:
  - as a request arrives its service is started and proceeds concurrently with the other ones
- The probability that a request arrives when the server cannot accept it is reduced
- There are different ways to implement a concurrent server:
  - by assigning each request to a different thread
  - by simulating concurrency within a single thread

TCP Concurrent Server with Thread Creation on Demand

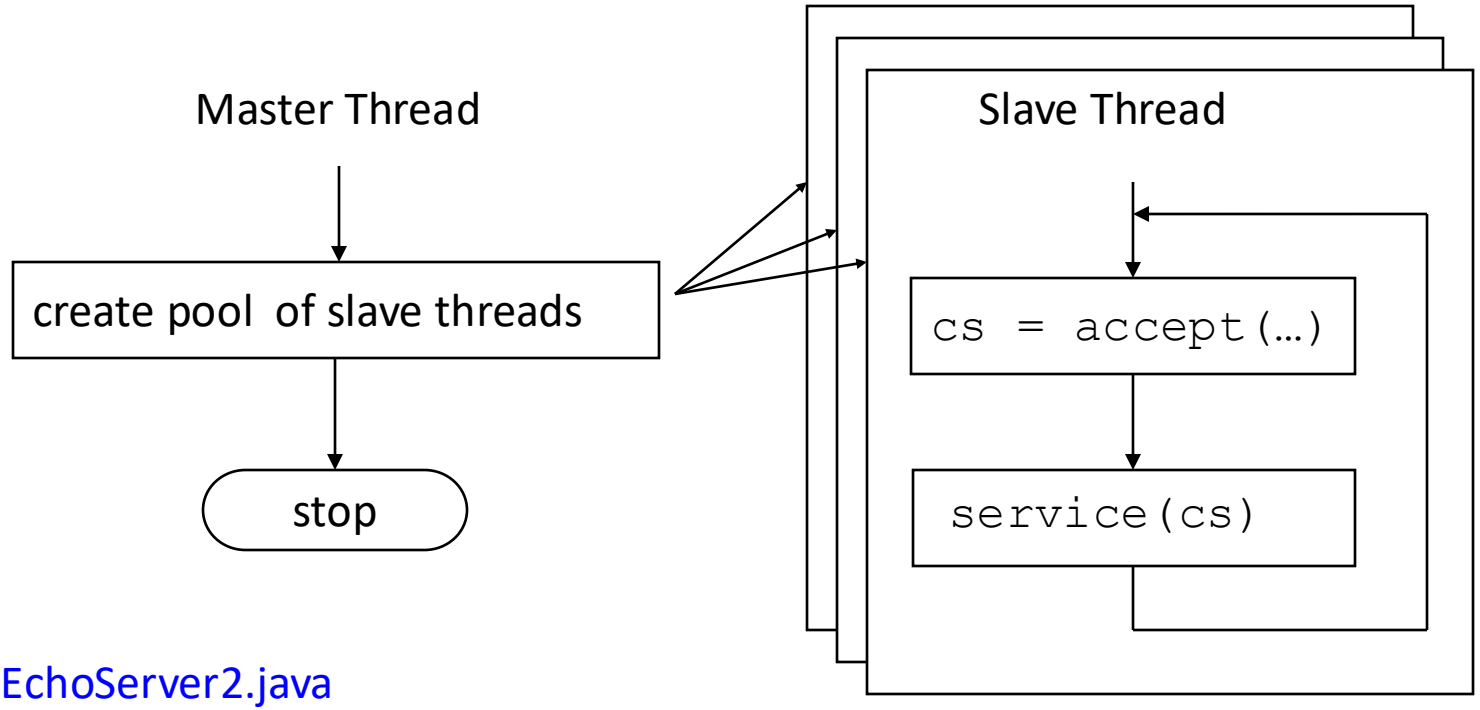
- A Master Thread continuously accepts connections
- As a new connection is accepted a slave thread is created to serve it



TcpEchoServer1.java

Solution Based on Pre-Created Thread Pool

- In order to reduce response time without giving up the advantages of concurrency, a pool of threads can be created at startup and allocated to requests when they arrive



TcpEchoServer2.java

Java Executors

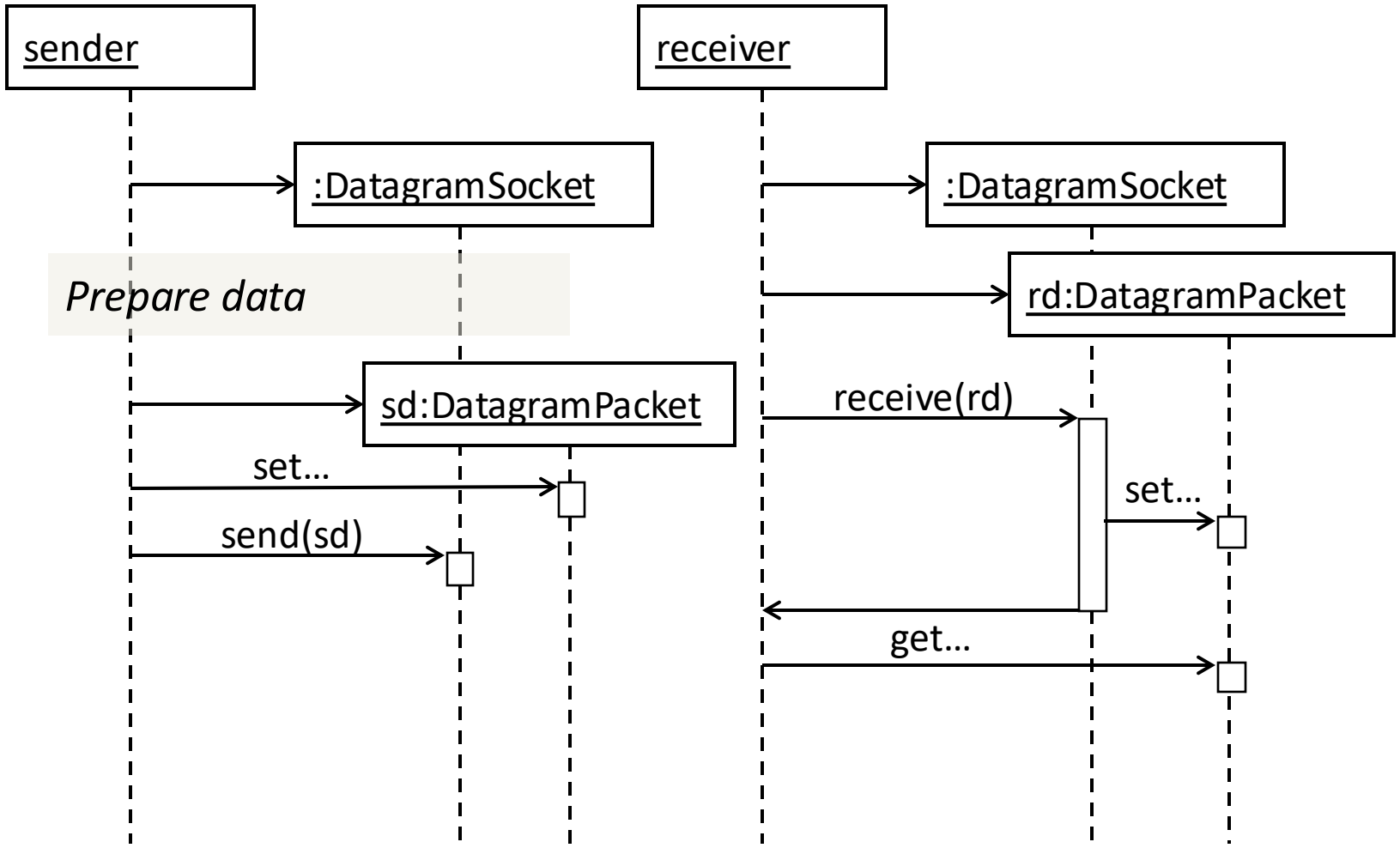
- Java supports thread pool management through Executors (in the java.util.concurrent library)
  - different executor types available
  - possibility to configure thread life-cycle (e.g. maximum/minimum number of threads), policies for re-launching failing threads,...

TcpEchoServer3.java

Operations Offered by the Socket API for Access to Layer 4 (UDP)

- Allocate local resources for communication
- Specify endpoints
- Send/Receive a datagram
- Release allocated resources

Using UDP Sockets in Java



Using UDP Sockets in Java

- DatagramSocket

  optional

  DatagramSocket(int localPort, InetAddress localAddr)

  – possibility to "connect" the socket to a remote address

  connect(InetAddress remoteAddr, int remotePort**)**

  – **close()** destroys socket and releases resources

- DatagramPacket

  optional

  DatagramPacket(byte[] buf, int length, InetAddress destAddr, int destPort)

Example: Echo Protocol on UDP

- Implement the simple echo service provided by Unix systems (on standard port 7):
  - the server continuously echoes back the datagrams received

  => UdpEchoServer0.java
- Implement a client for the service
  - the client operates with line buffering

  => UdpEchoClient0.java

Encoding/Decoding Data

- TCP and UDP provide transport of bytes
- Data encoding and decoding (marshalling/unmarshalling) is up to the programmer
  - possibility to use libraries for standard encoding formats (XML, JSON, Protocol buffers, ...)
  - in Java, other data encoding/decoding functions available in library classes (e.g., DataInputStream, DataOutputStream, String)
    - care necessary: e.g., integer endianness, charset