合肥學院
HEFEI UNIVERSITY

Distributed Computing

Lesson 5: Sockets

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://www.it-weise.de

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

website

Overview

- What are sockets?
- Which protocols can they offer?
- How is the API for sockets designed in languages such as `Java` and `C`?
- What to consider when exchanging data between hosts in a heterogeneous system?

- What are sockets in networking?

- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]

Introduction

- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]
- Available for all major programming languages

Introduction

- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]
- Available for all major programming languages:
  - Java [2–11]

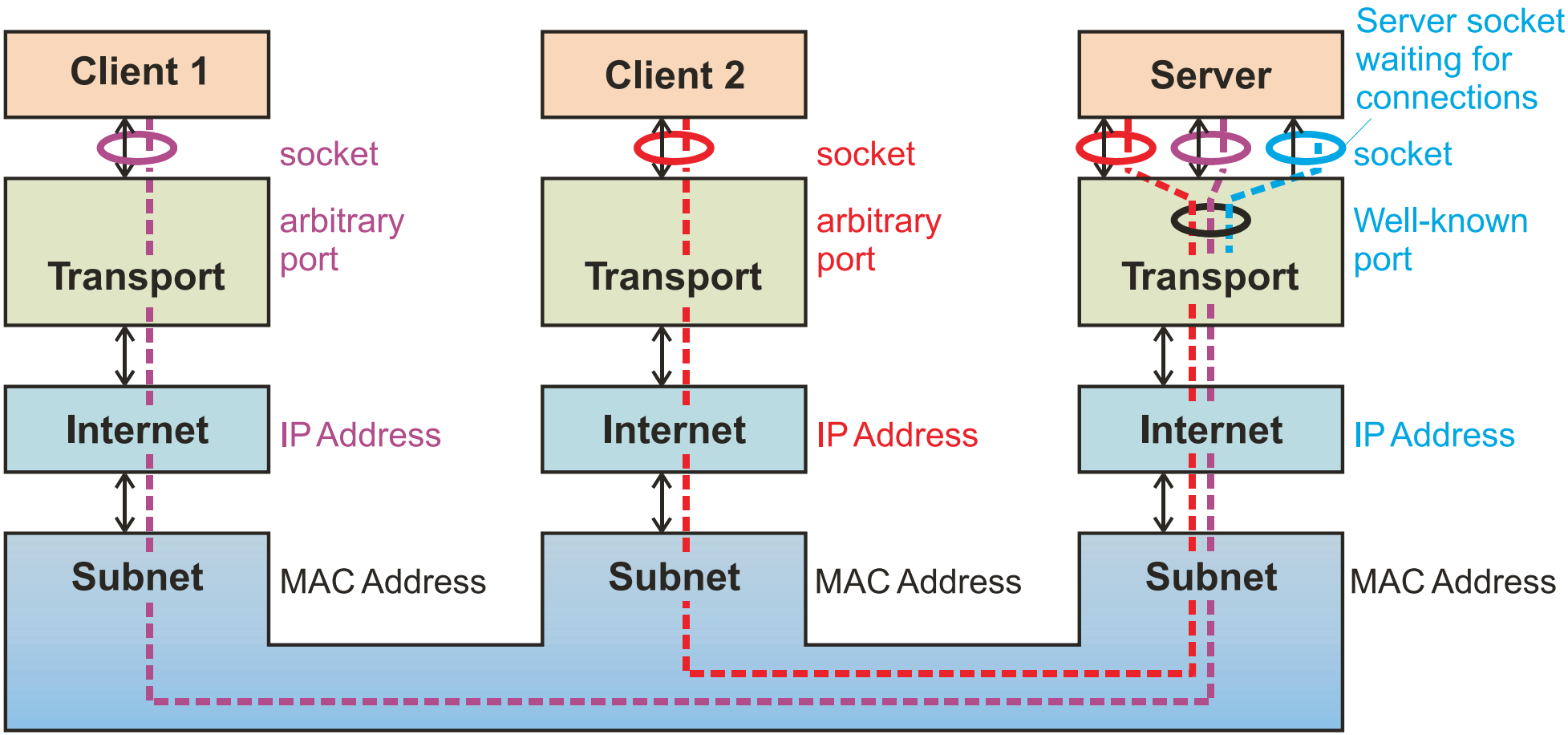
- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]
- Available for all major programming languages:
  - Java [2–11]
  - C/C++ [12–15]

- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]
- Available for all major programming languages:
  - Java [2–11]
  - C/C++ [12–15]
  - C# [16, 17]

- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]
- Available for all major programming languages:
  - Java [2–11]
  - C/C++ [12–15]
  - C# [16, 17]
  - Python [18]

- What are sockets in networking?
- API provided by OS for accessing protocols of OSI Layer 4 and below [1]
- Available for all major programming languages:
  - Java [2–11]
  - C/C++ [12–15]
  - C# [16, 17]
  - Python [18]
- Allow data exchange via IP [19], UDP [19], and TCP [19, 20] protocols

Sockets


Client 1
Client 2
Server
Server socket waiting for connections
socket
socket
socket
arbitrary port
arbitrary port
Well-known port
Transport
Transport
Transport
Internet
Internet
Internet
IP Address
IP Address
IP Address
Subnet
Subnet
Subnet
MAC Address
MAC Address
MAC Address

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols
  - Each protocol can be used by multiple processes for each network adapter

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols
  - Each protocol can be used by multiple processes for each network adapter
- A socket uniquely identifies a communication channel used by one process using one transport layer protocol to communicate via one network adapter

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols
  - Each protocol can be used by multiple processes for each network adapter
- A socket uniquely identifies a communication channel used by one process using one transport layer protocol to communicate via one network adapter
- (client) sockets can be uniquely identified by

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols
  - Each protocol can be used by multiple processes for each network adapter
- A socket uniquely identifies a communication channel used by one process using one transport layer protocol to communicate via one network adapter
- (client) sockets can be uniquely identified by:
  1. IP Address (Internet Layer)

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols
  - Each protocol can be used by multiple processes for each network adapter
- A socket uniquely identifies a communication channel used by one process using one transport layer protocol to communicate via one network adapter
- (client) sockets can be uniquely identified by:
  1. IP Address (Internet Layer) +
  2. Transport Layer protocol name (e.g., TCP or UDP)

Identification & Addressing

- Assumption: Communication via TCP/IP or UDP/IP
- Internet Layer:
  - Identify *network interface* of a host: IP-Address (v4, v6)
  - There are many computers in the internet
  - Each host may have more than one network adater, each with a different IP address (Example: routers)
- Transport Layer:
  - For each network adapter, OS provides transport layer protocols
  - Each protocol can be used by multiple processes for each network adapter
- A socket uniquely identifies a communication channel used by one process using one transport layer protocol to communicate via one network adapter
- (client) sockets can be uniquely identified by:
  1. IP Address (Internet Layer) +
  2. Transport Layer protocol name (e.g., TCP or UDP) +
  3. Transport Layer port (0...65535)

Identification & Addressing

- Domain Name System (DNS): Translates textual names to IP Addresses

Identification & Addressing

- Domain Name System (DNS): Translates textual names to IP Addresses
  - e.g., `www.baidu.com` $\equiv$ `119.75.218.70`

Identification & Addressing

- Domain Name System (DNS): Translates textual names to IP Addresses
  - e.g., `www.baidu.com` $\equiv$ `119.75.218.70`
  - `localhost` $\equiv$ `127.0.0.1`

Sockets

- Sockets are network-wide unique resources owned by processes

Sockets

- Sockets are network-wide unique resources owned by processes
- Division between **client** and **server** sockets

Sockets

- Sockets are network-wide unique resources owned by processes
- Division between **client** and **server** sockets
- **Server** sockets accept incoming client connections or data

Sockets

- Sockets are network-wide unique resources owned by processes
- Division between **client** and **server** sockets
- **Server** sockets accept incoming client connections or data
- **Client** sockets initiate communication with a server

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)
  5) close server socket

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)
  5) close server socket
- **Client socket**

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)
  5) close server socket
- **Client socket**
  1) client socket is bound to random free port

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)
  5) close server socket
- **Client socket**
  1) client socket is bound to random free port
  2) connect to a server socket

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)
  5) close server socket
- **Client socket**
  1) client socket is bound to random free port
  2) connect to a server socket
  3) communicates by sending und receiving data

TCP Sockets

- TCP [19, 20] is a connection-oriented protocol
- Server sockets wait for incomming connection requests
- Client sockets try to establish a connection
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) listens at that port for new clients
  3) creates a new socket for each client it accepts
  4) new socket: continue at "**client socket**" 3)
  5) close server socket
- **Client socket**
  1) client socket is bound to random free port
  2) connect to a server socket
  3) communicates by sending und receiving data
  4) close the socket

TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy

TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`

TCP Sockets in Java

- In `Java` [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port

TCP Sockets in Java

- In `Java` [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out **1)** and **2)** as one step

TCP Sockets in Java

- In `Java` [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections

TCP Sockets in Java

- In `Java` [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`

TCP Sockets in Java

- In `Java` [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)

TCP Sockets in Java

- In `Java` [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)
  - For communication (4 + 2), each (client) socket has

TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)
  - For communication (4 + 2), each (client) socket has
    - an `InputStream` for reading incoming data (get with `getInputStream`)

TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)
  - For communication (4 + 2), each (client) socket has
    - an `InputStream` for reading incoming data (get with `getInputStream`)
    - an `OutputStream` for sending data (get with `getOutputStream`)

TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)
  - For communication (4 + 2), each (client) socket has
    - an `InputStream` for reading incoming data (get with `getInputStream`)
    - an `OutputStream` for sending data (get with `getOutputStream`)
    - reads are blocking, writes are non-blocking

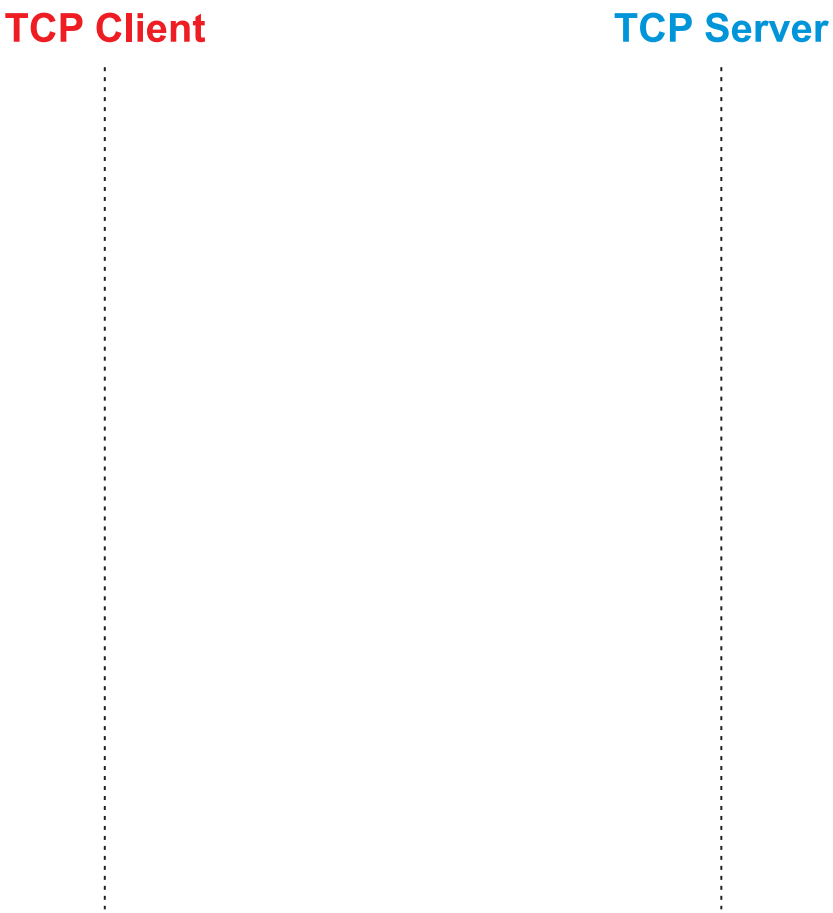
TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)
  - For communication (4 + 2), each (client) socket has
    - an `InputStream` for reading incoming data (get with `getInputStream`)
    - an `OutputStream` for sending data (get with `getOutputStream`)
    - reads are blocking, writes are non-blocking
- Sockets are closed with their `close` method 3) + 5)

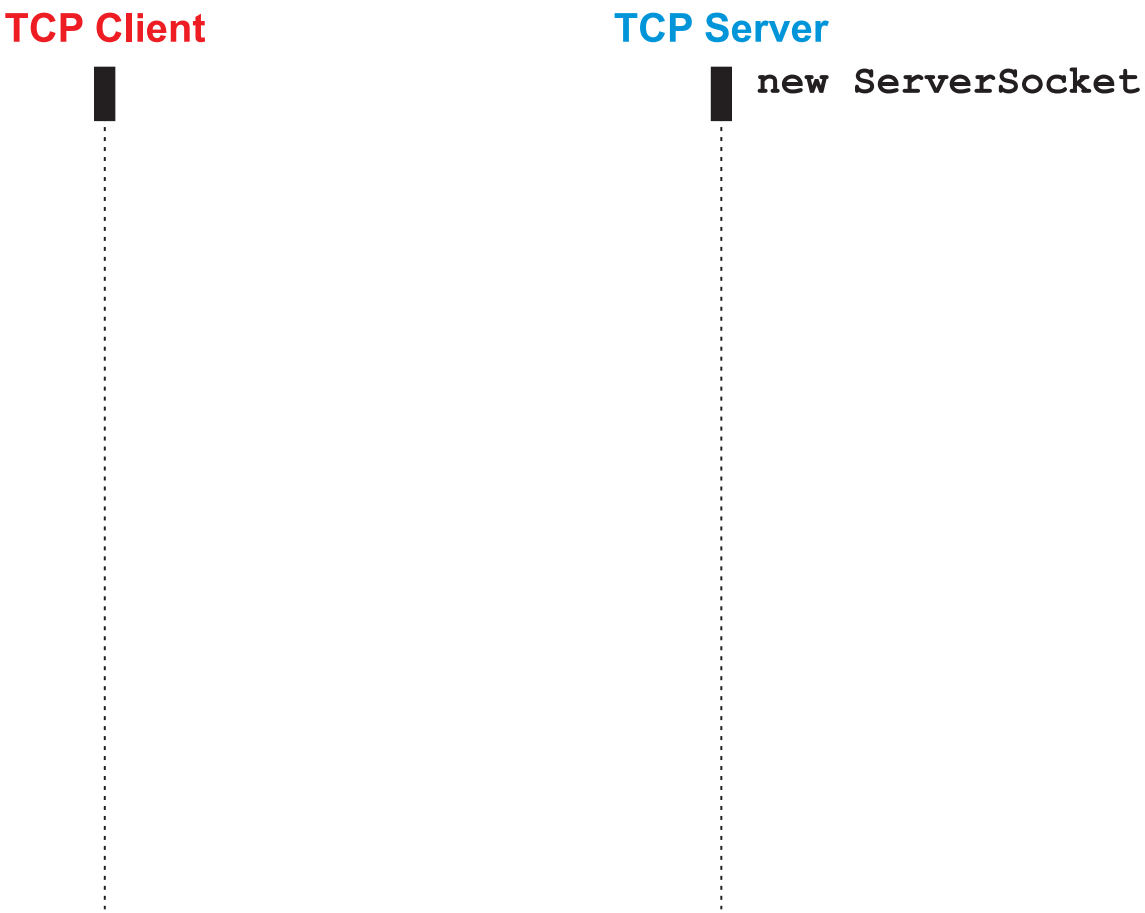
TCP Sockets in Java

- In Java [21, 22], using TCP/IP sockets is fairly easy
- The **server** socket that accepts connections is an instance of the class `ServerSocket`
  - In its constructor, we pass in the port
  - The constructor carries out 1) and 2) as one step
  - The method `accept` performs a *blocking* wait for incoming connections, which are returned as `Socket`s, i.e., client sockets
- A **client** is an instance of the class `Socket`
  - it can be connected to a server by passing the server's address (`InetAddress`) and port into the constructor 1)
  - For communication (4 + 2), each (client) socket has
    - an `InputStream` for reading incoming data (get with `getInputStream`)
    - an `OutputStream` for sending data (get with `getOutputStream`)
    - reads are blocking, writes are non-blocking
- Sockets are closed with their `close` method 3) + 5)
- All the above may cause errors, thrown as `IOException`s

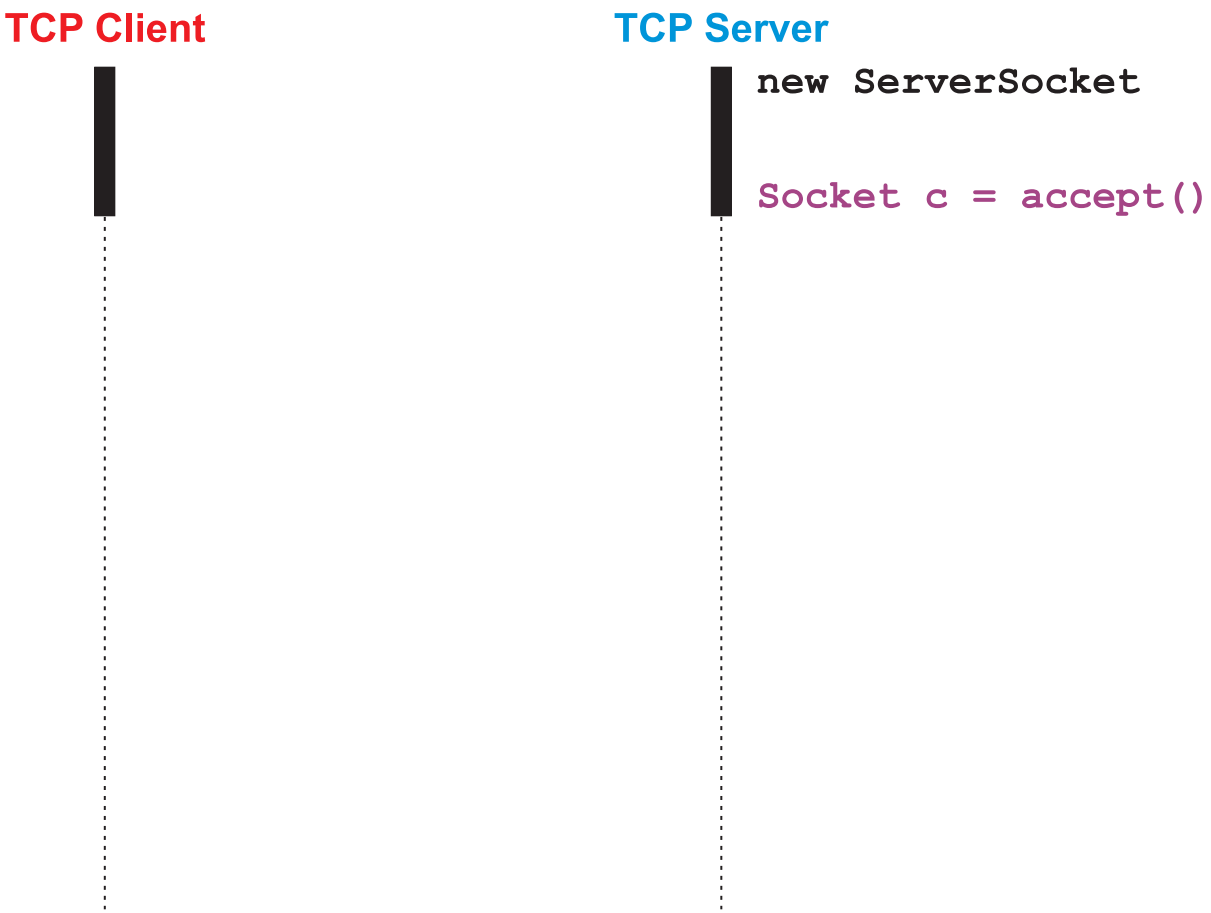
TCP: 3-way Handshake



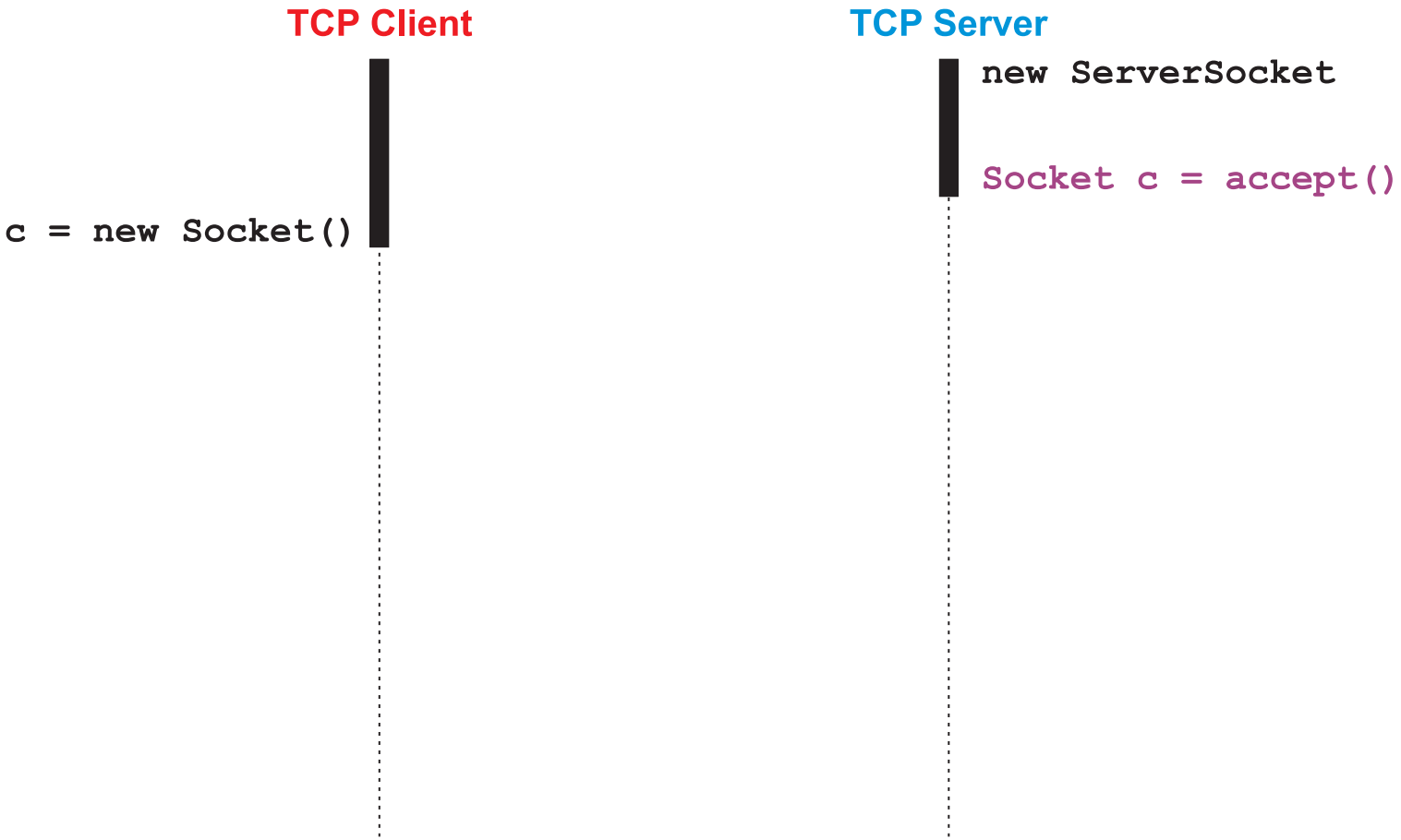
TCP: 3-way Handshake



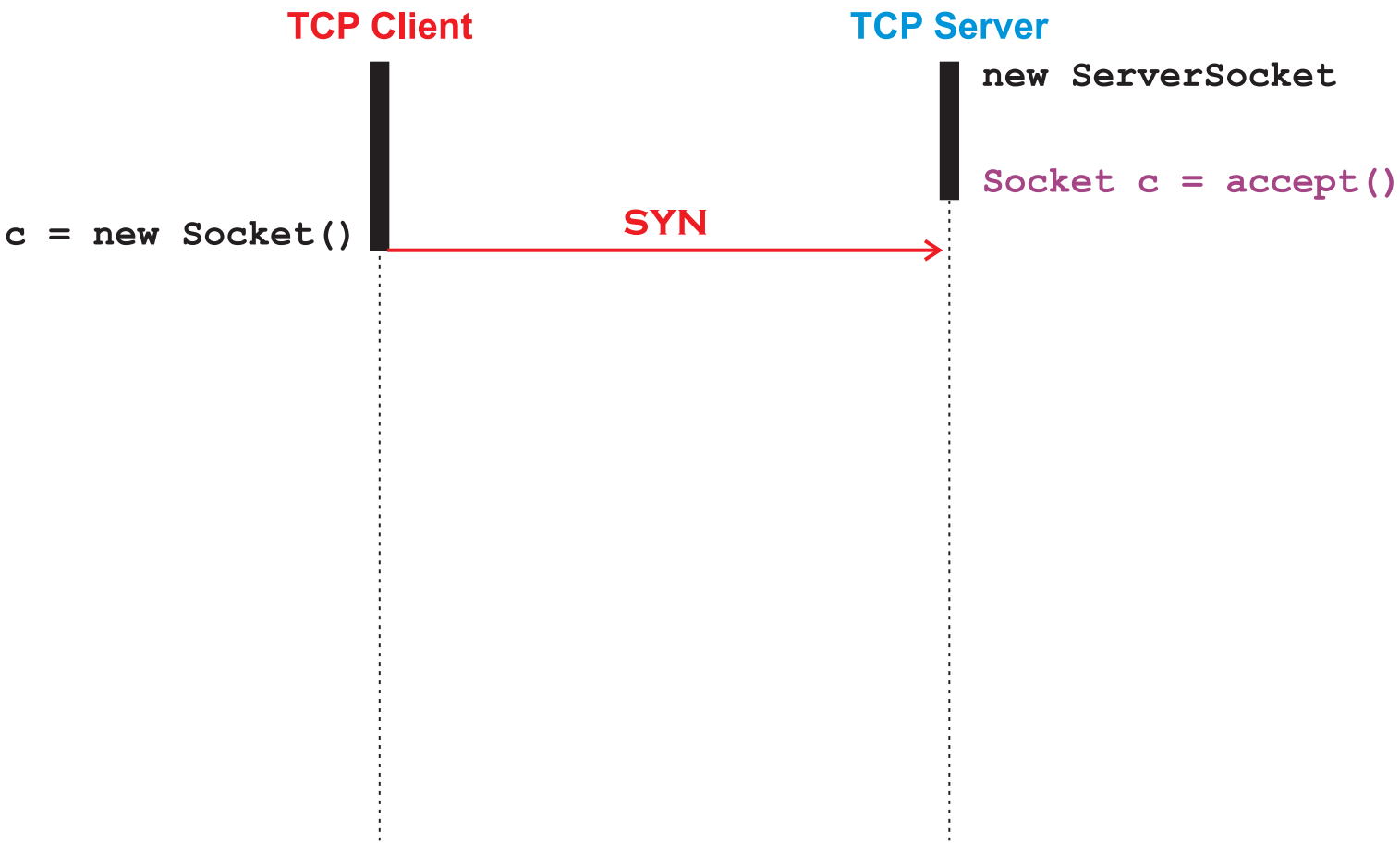
TCP: 3-way Handshake



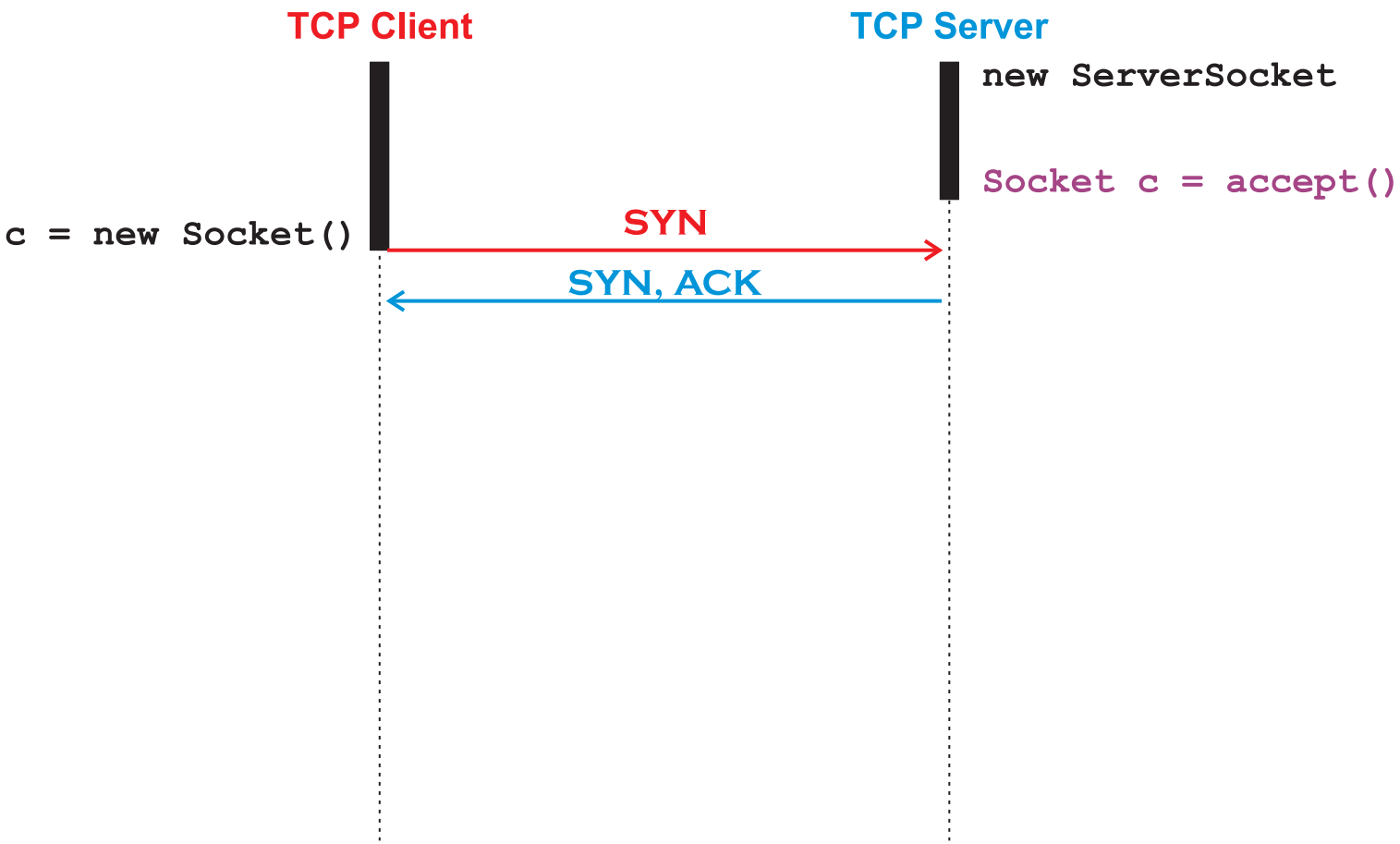
TCP: 3-way Handshake



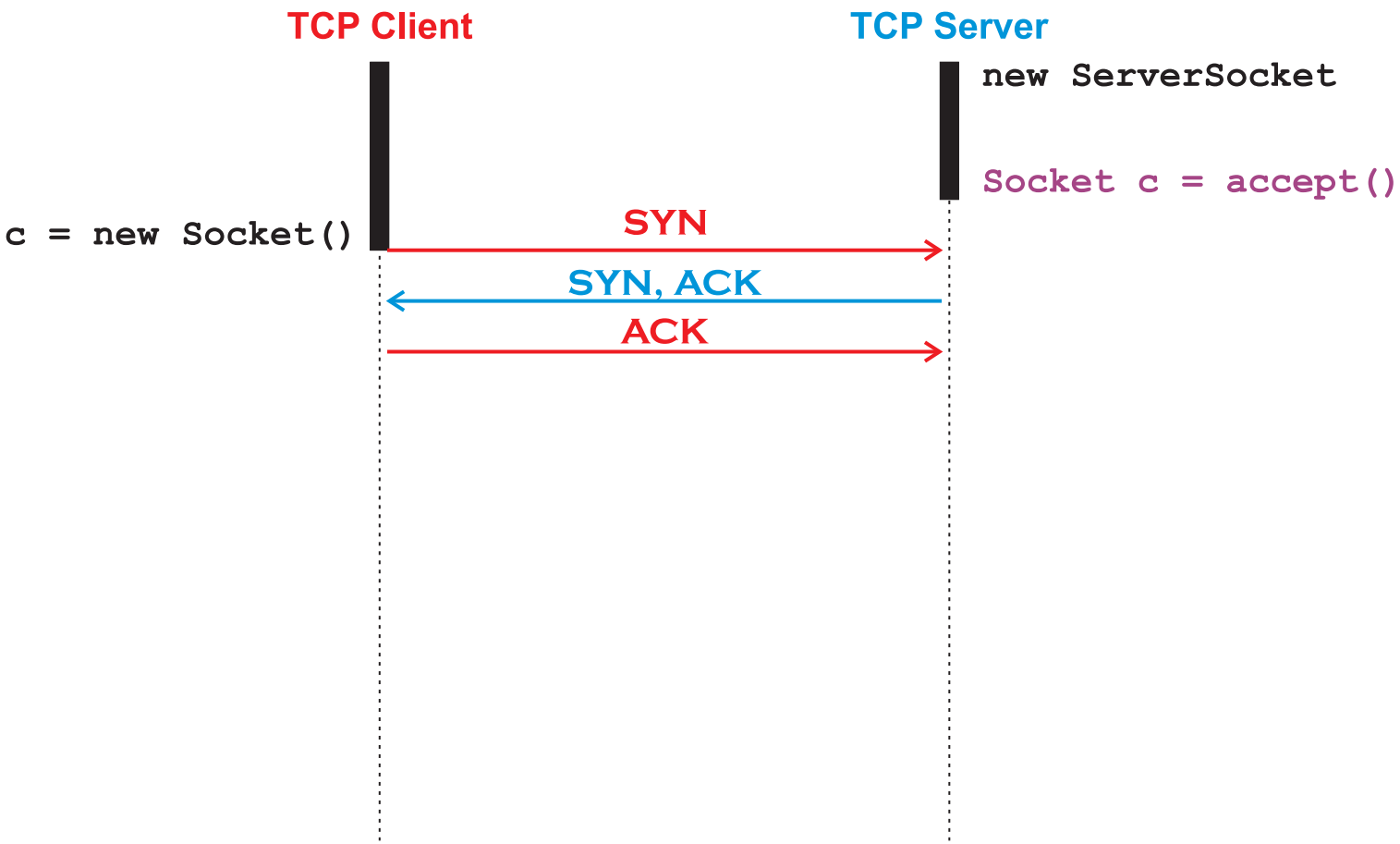
TCP: 3-way Handshake



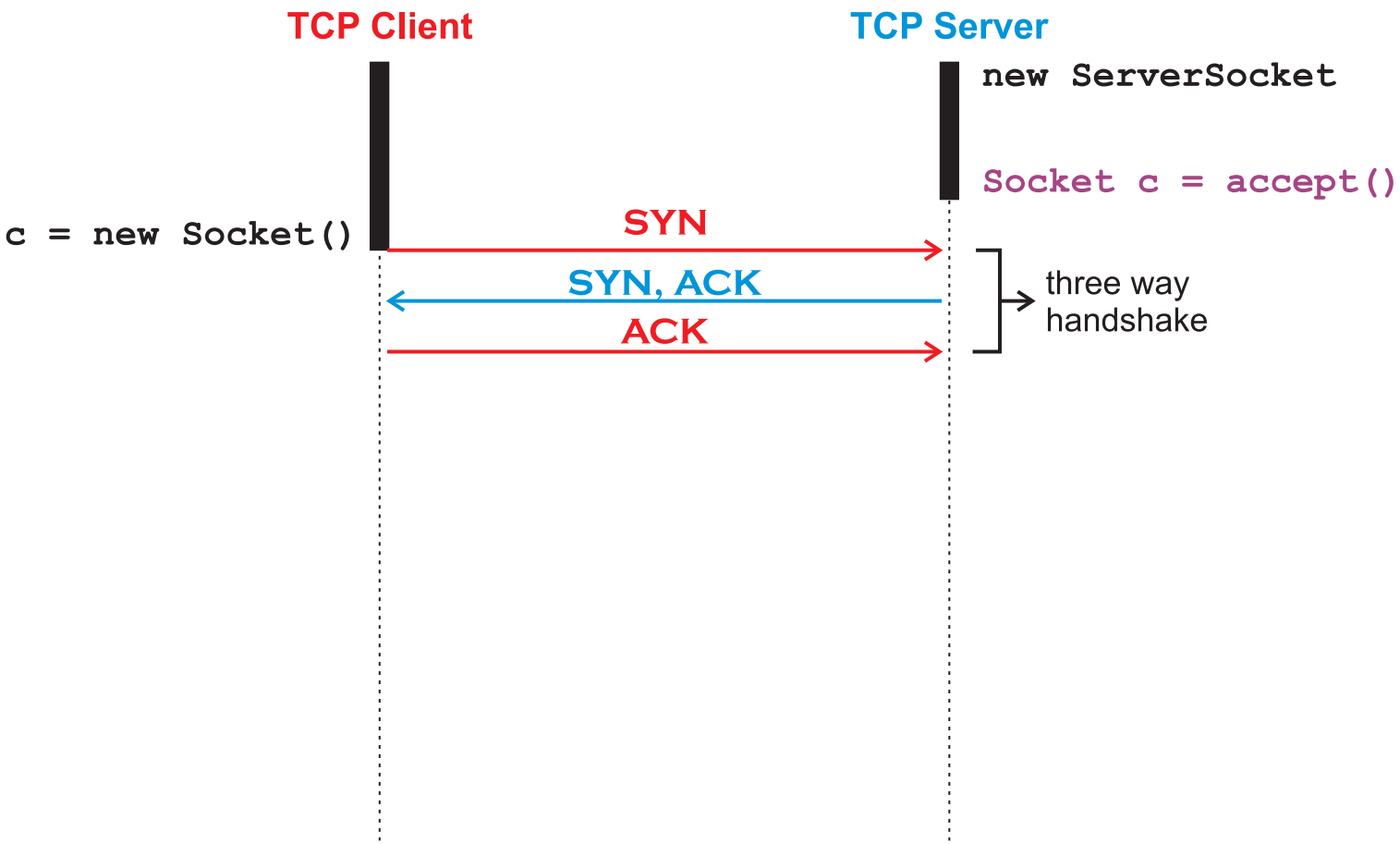
TCP: 3-way Handshake



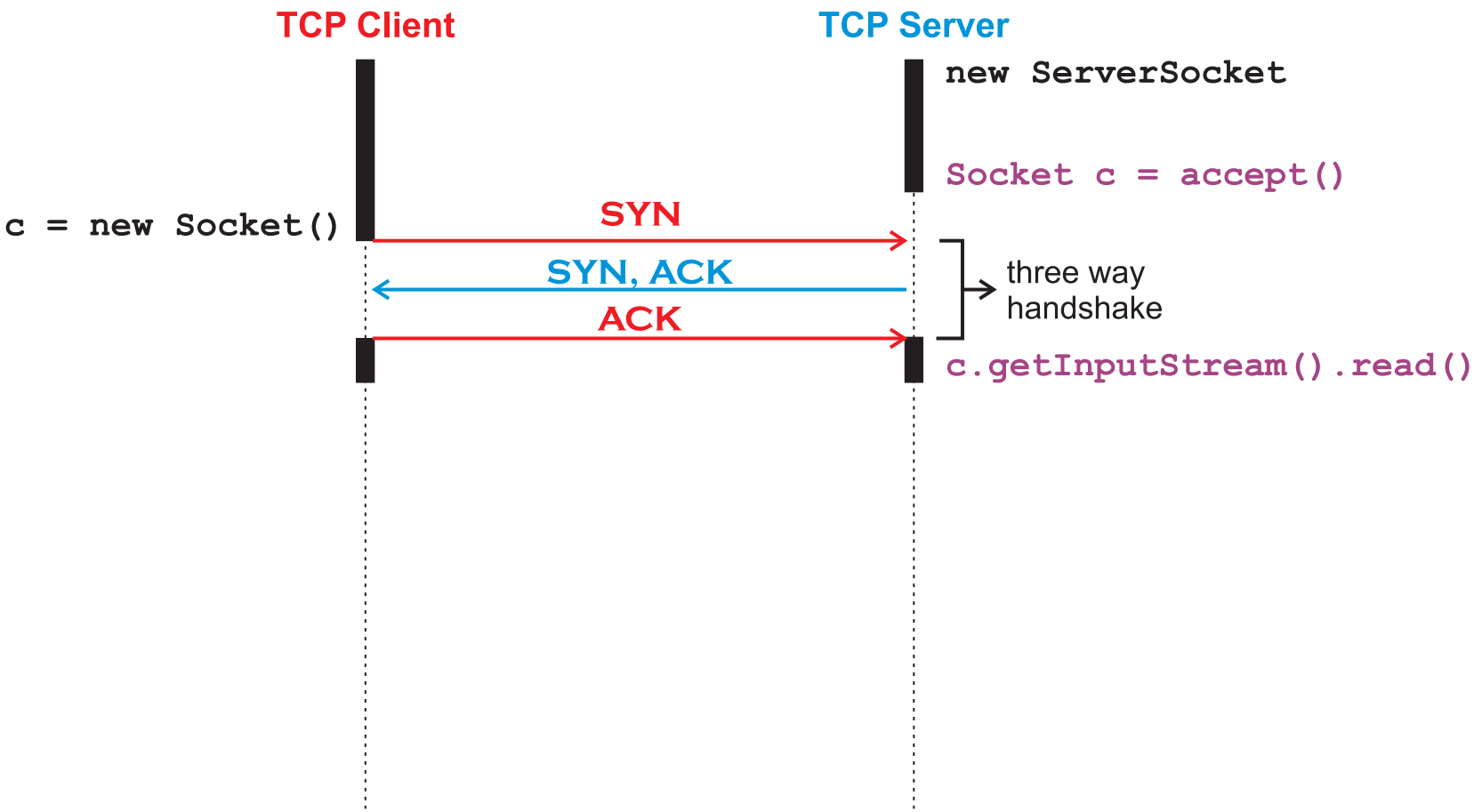
TCP: 3-way Handshake



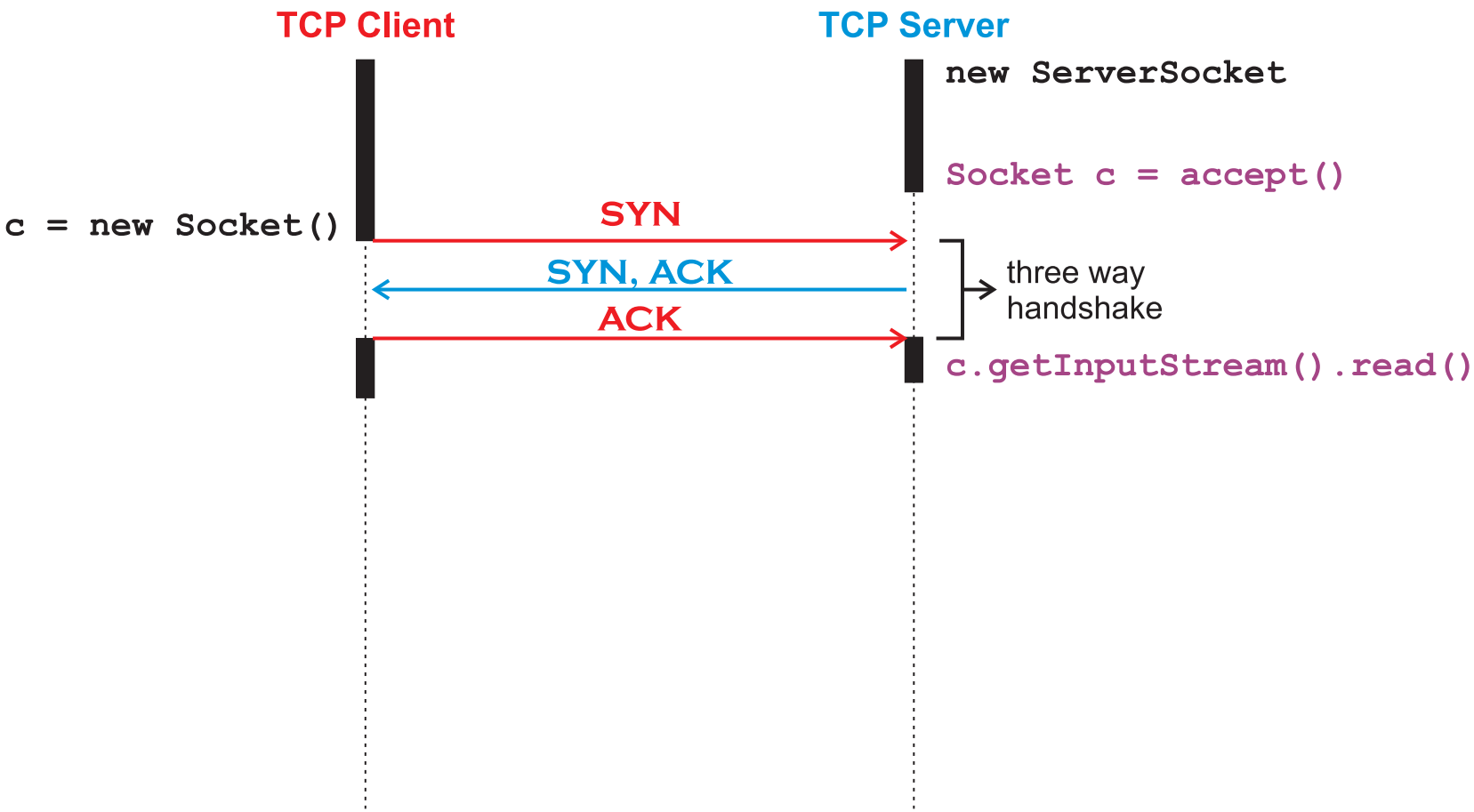
TCP: 3-way Handshake



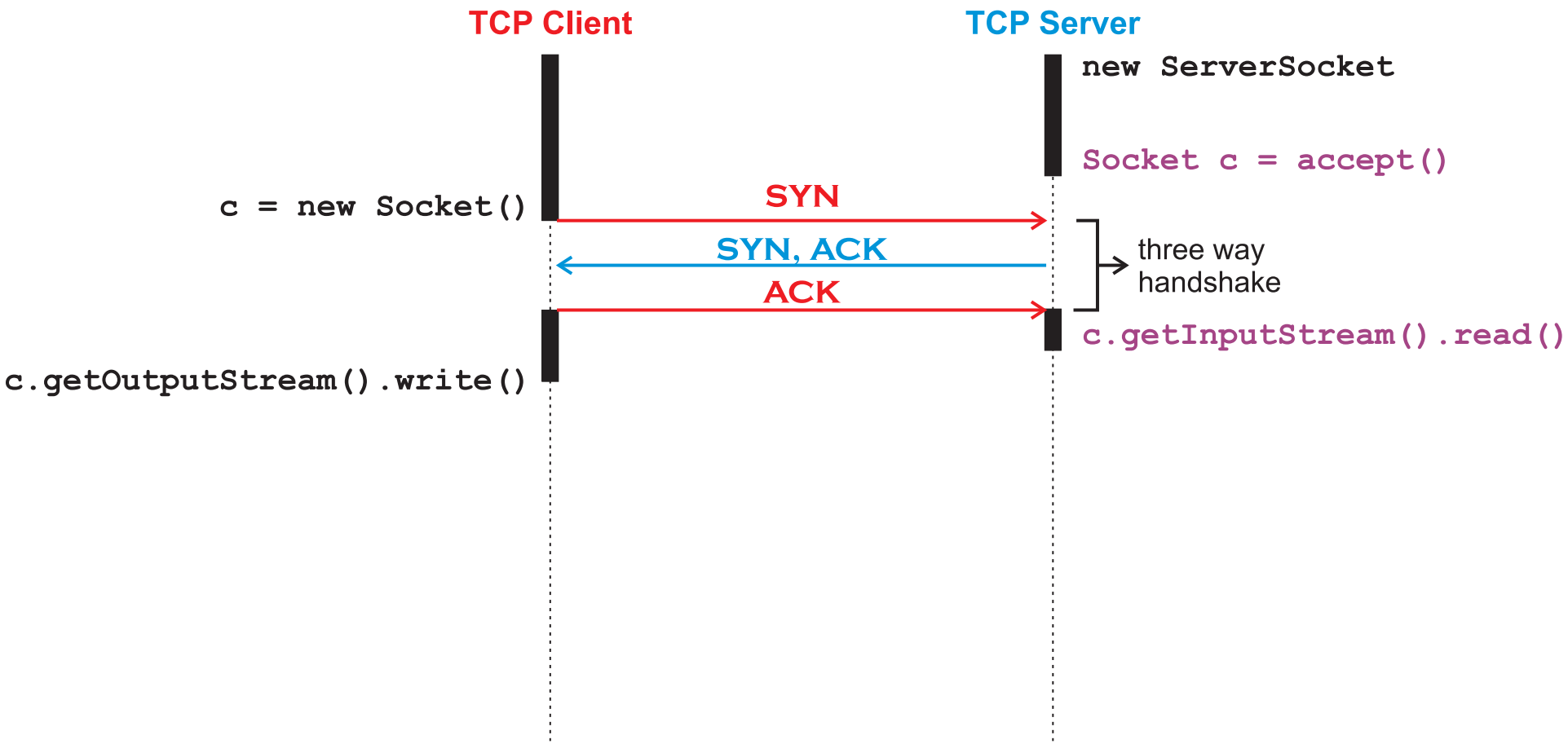
TCP: 3-way Handshake



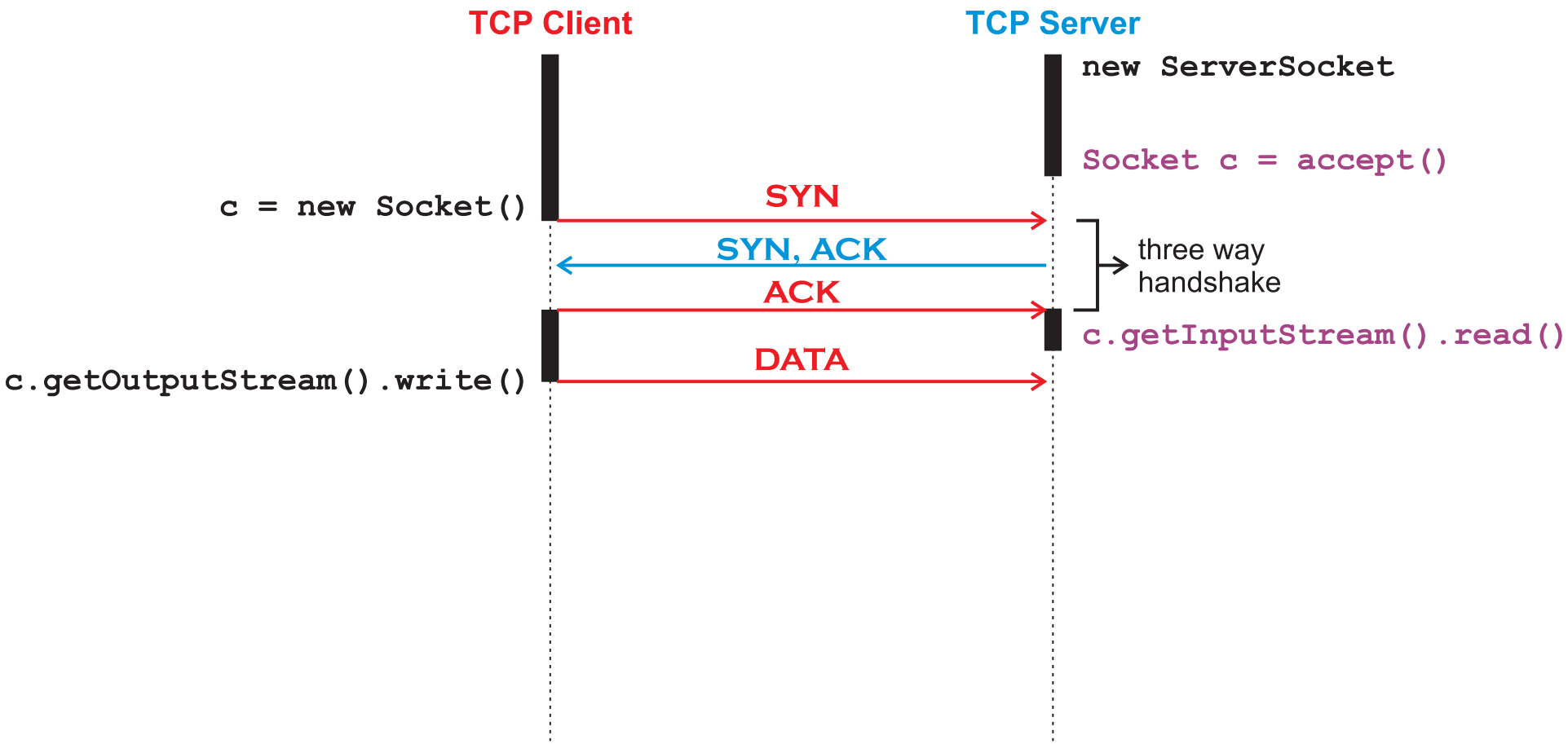
TCP: 3-way Handshake



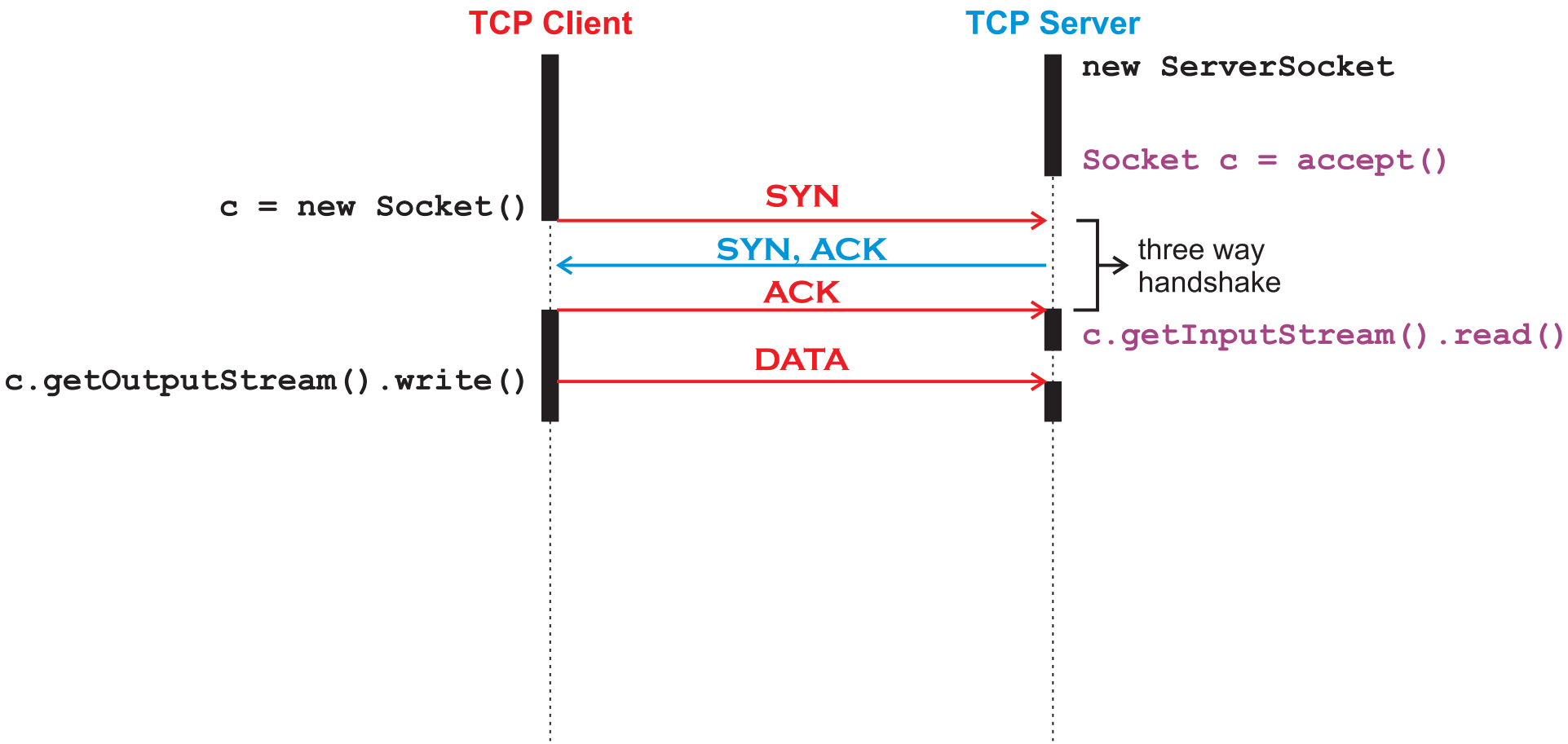
TCP: 3-way Handshake



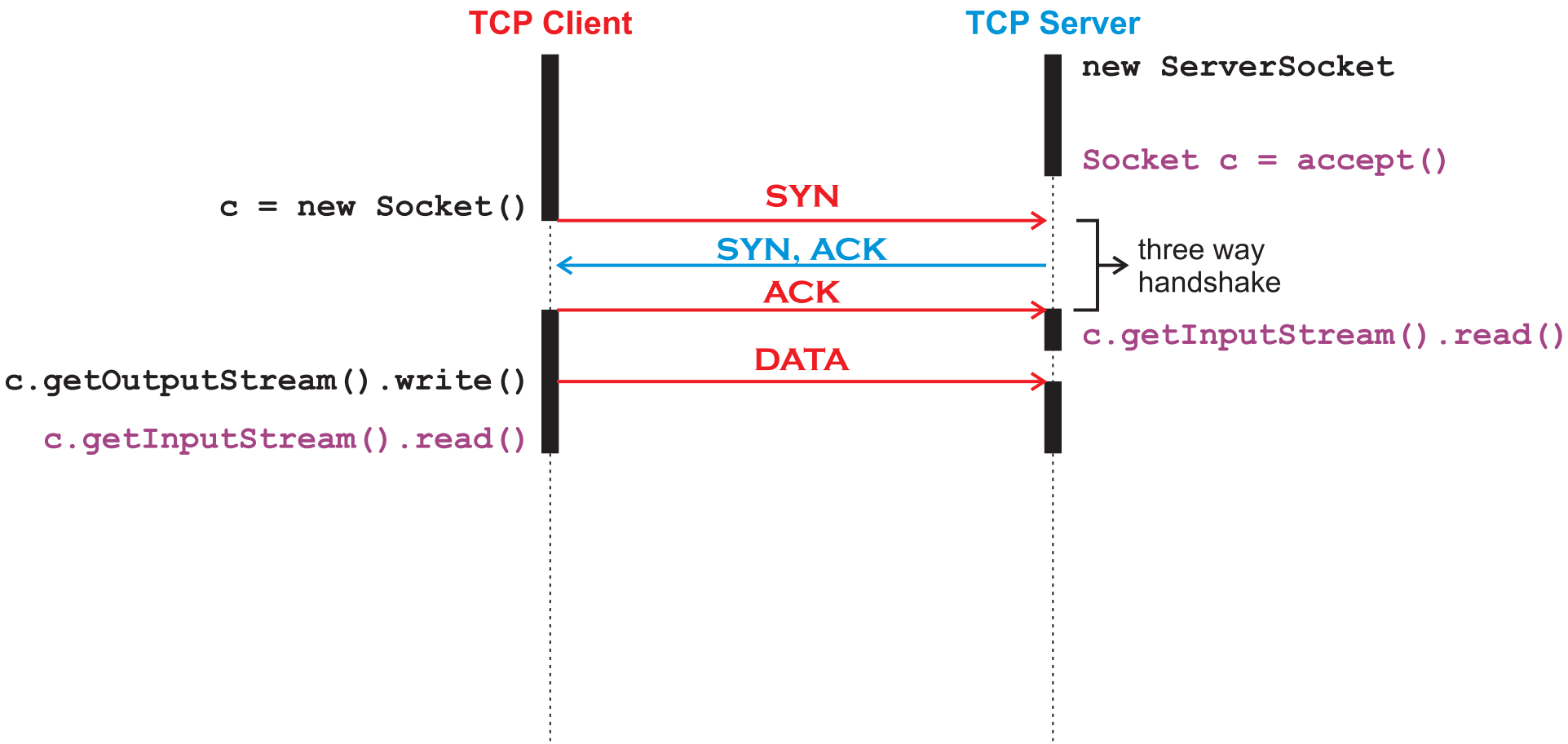
TCP: 3-way Handshake



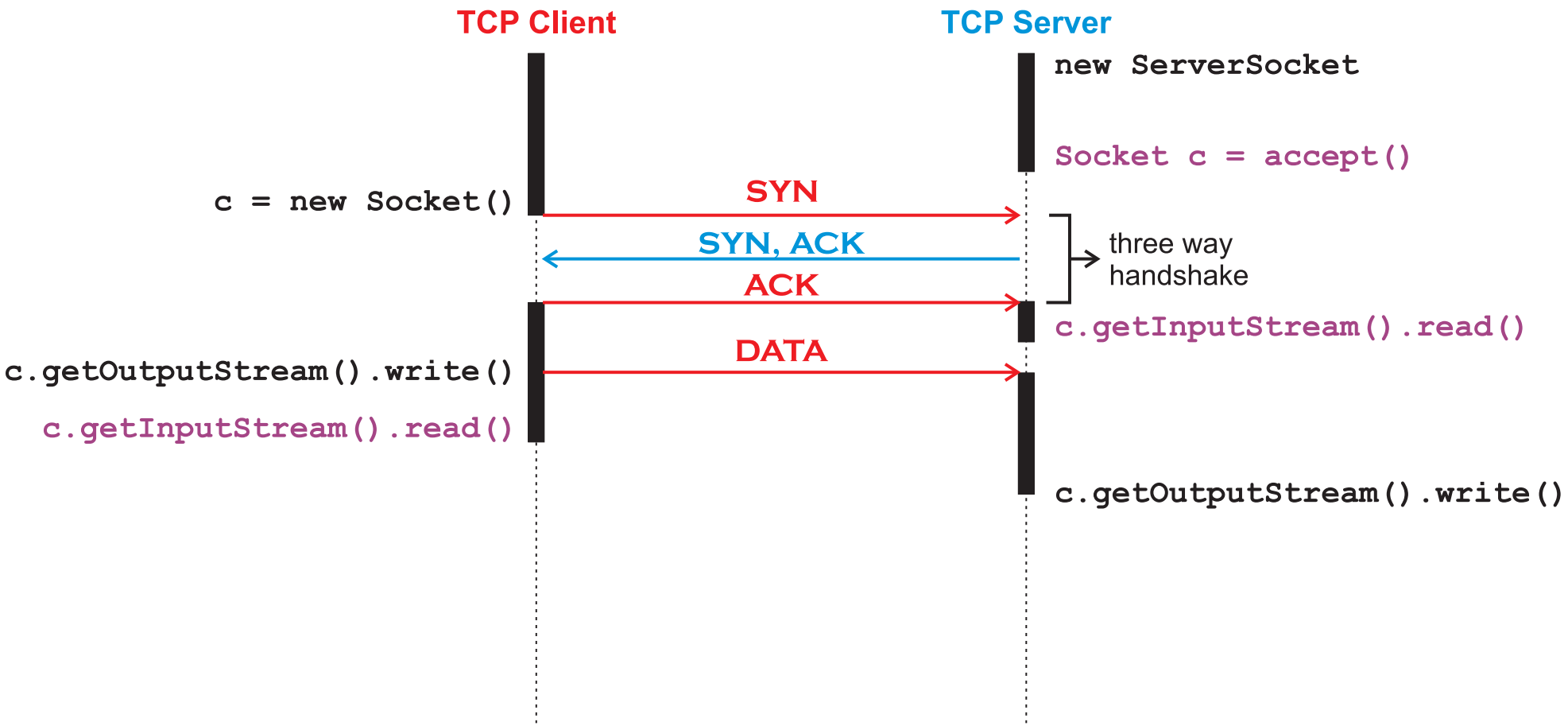
TCP: 3-way Handshake



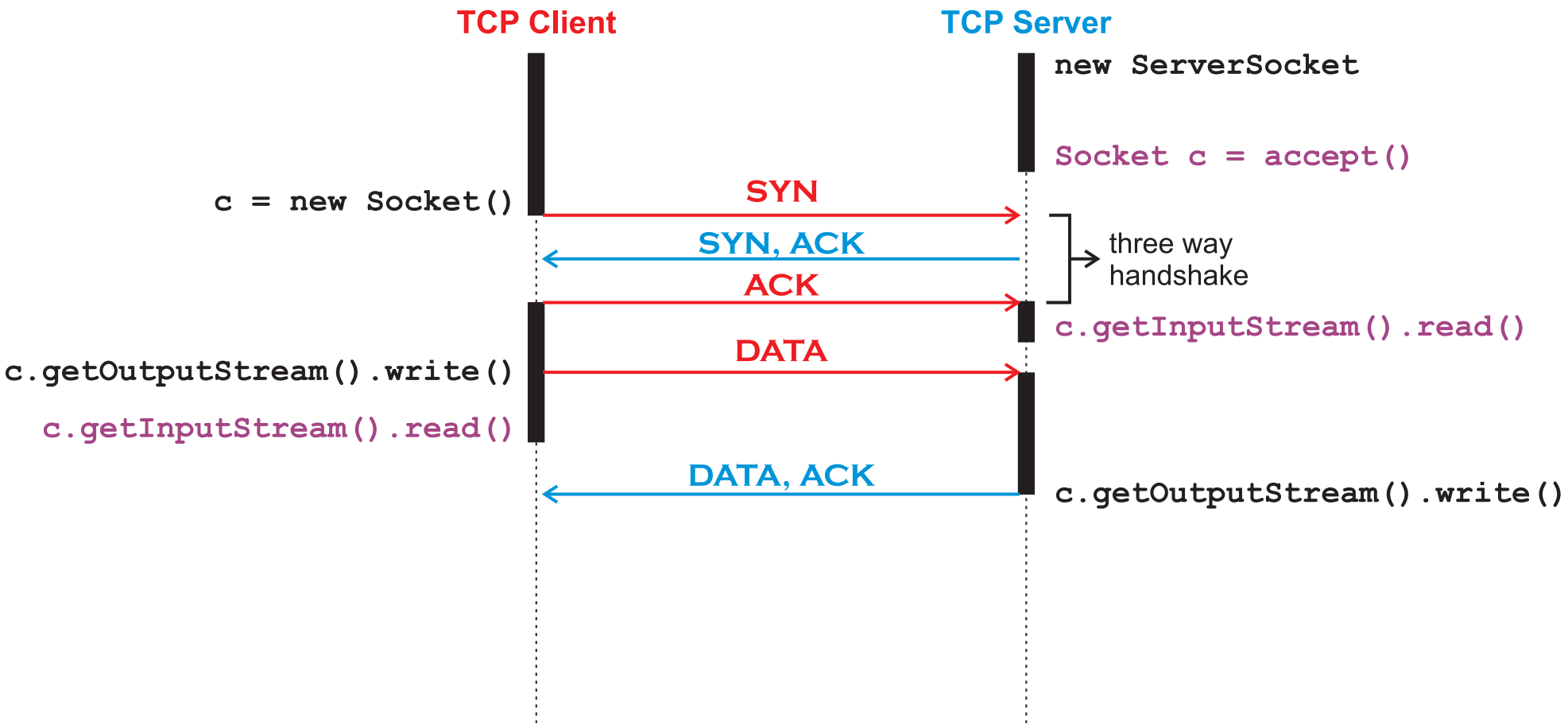
TCP: 3-way Handshake



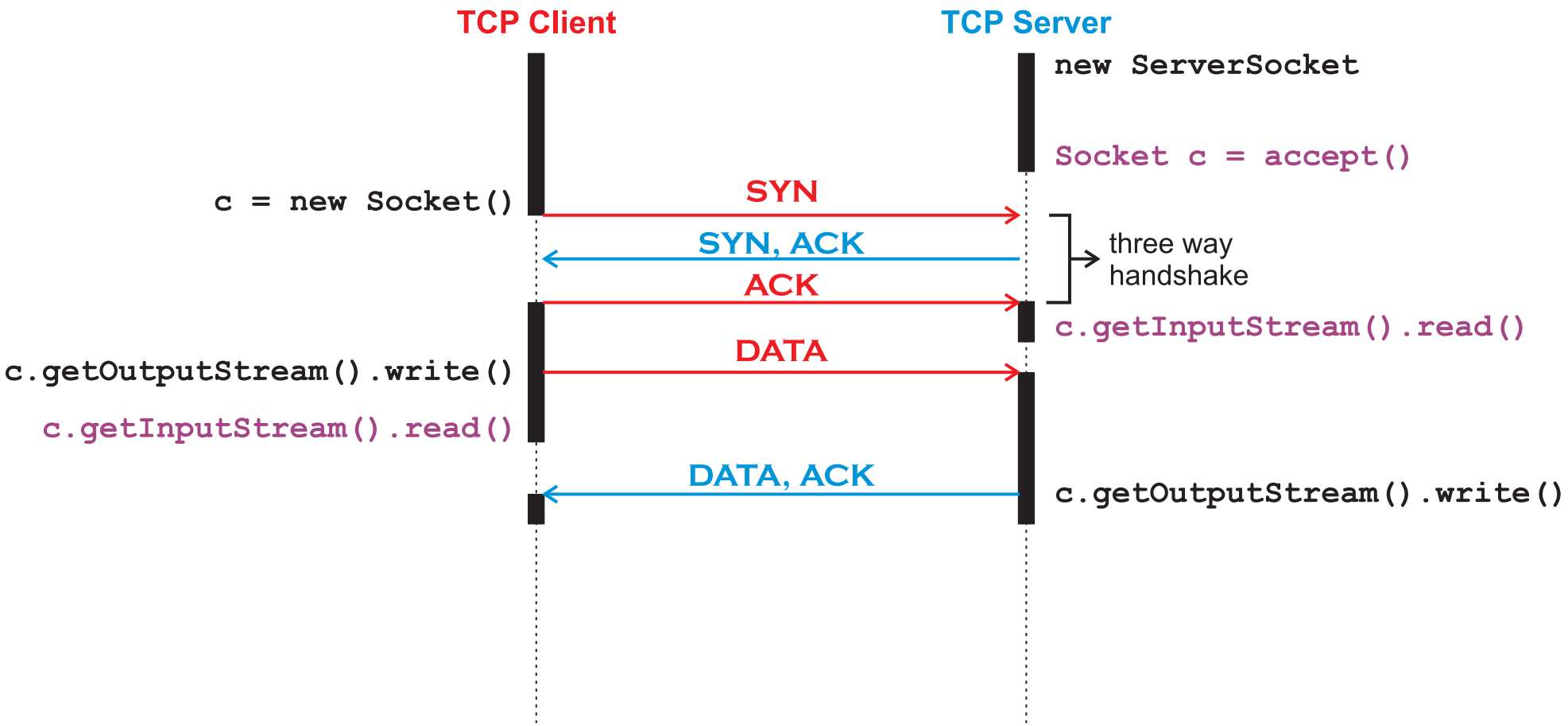
TCP: 3-way Handshake



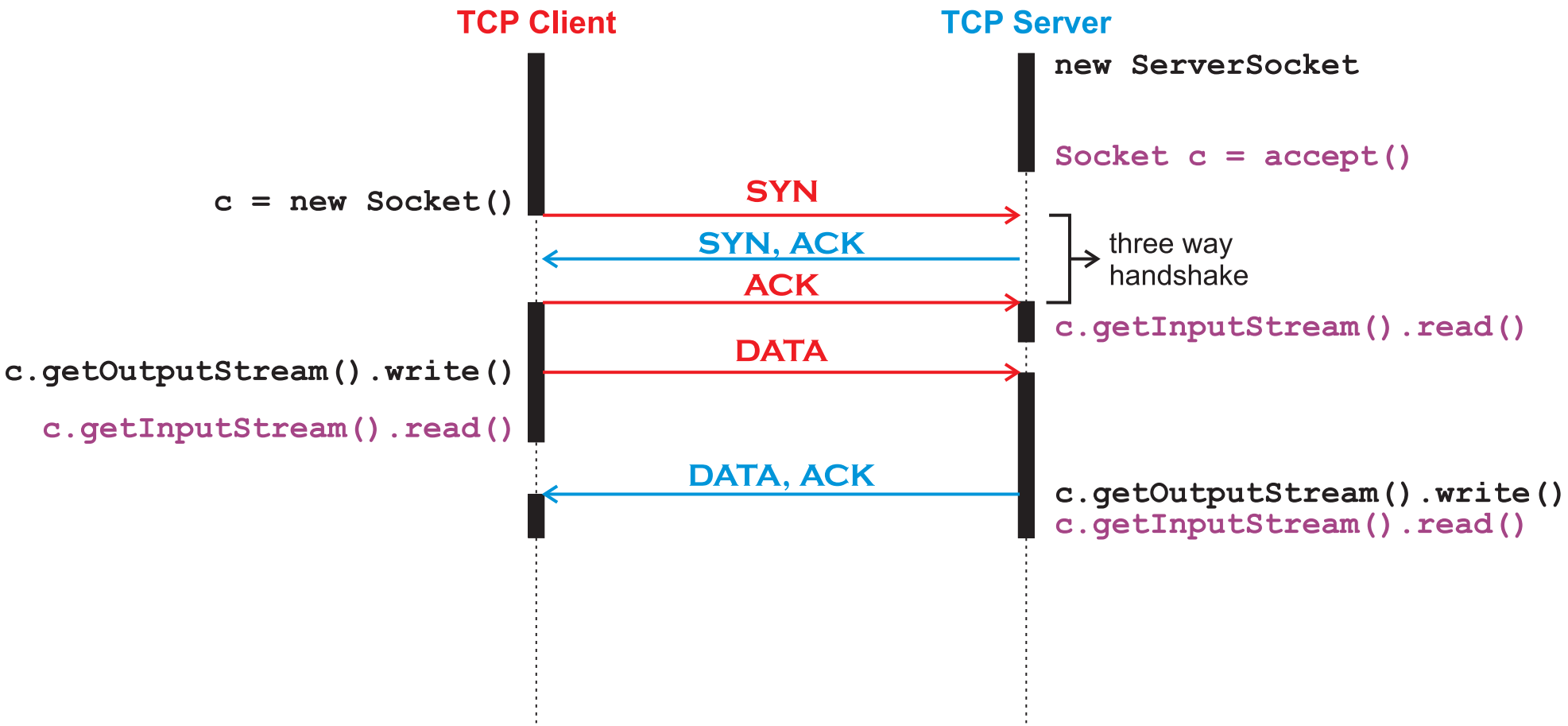
TCP: 3-way Handshake



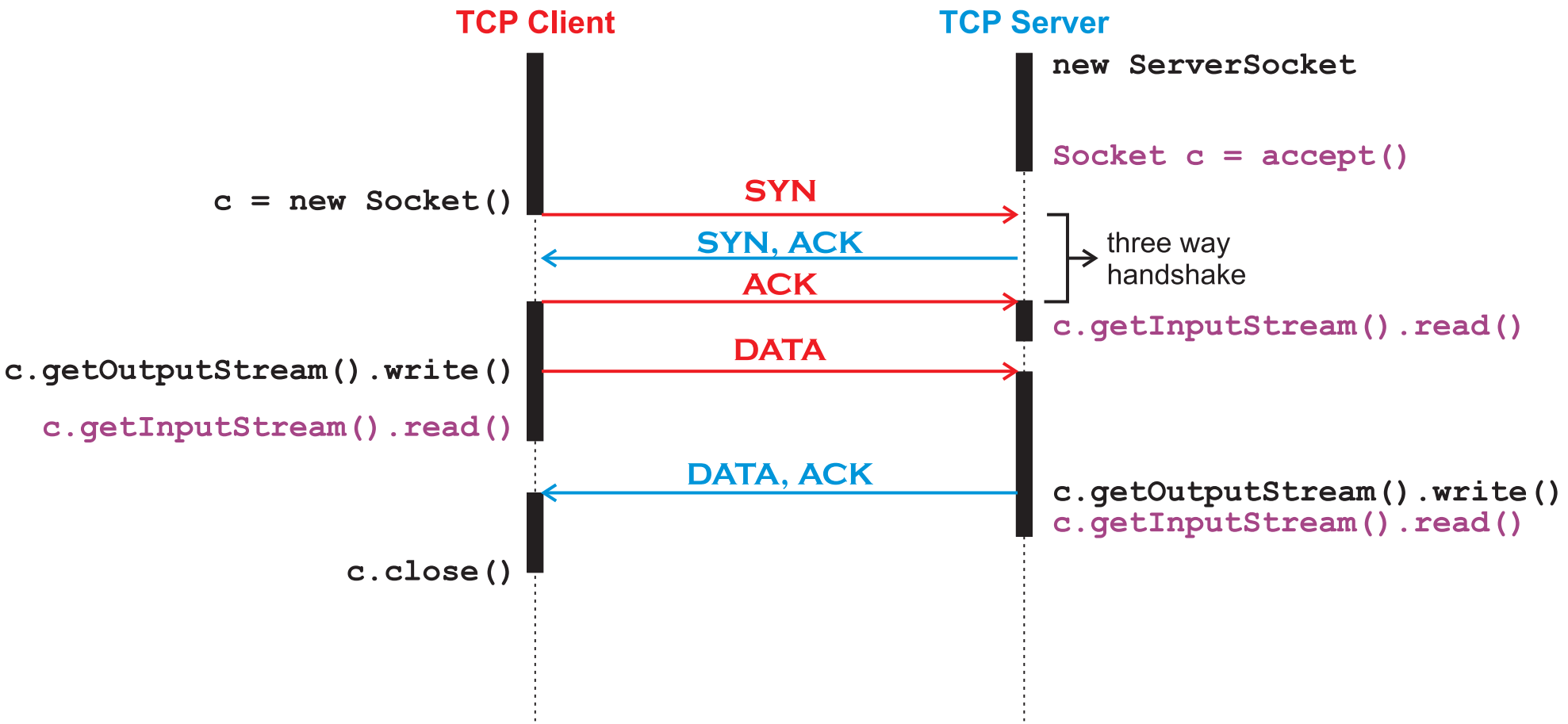
TCP: 3-way Handshake



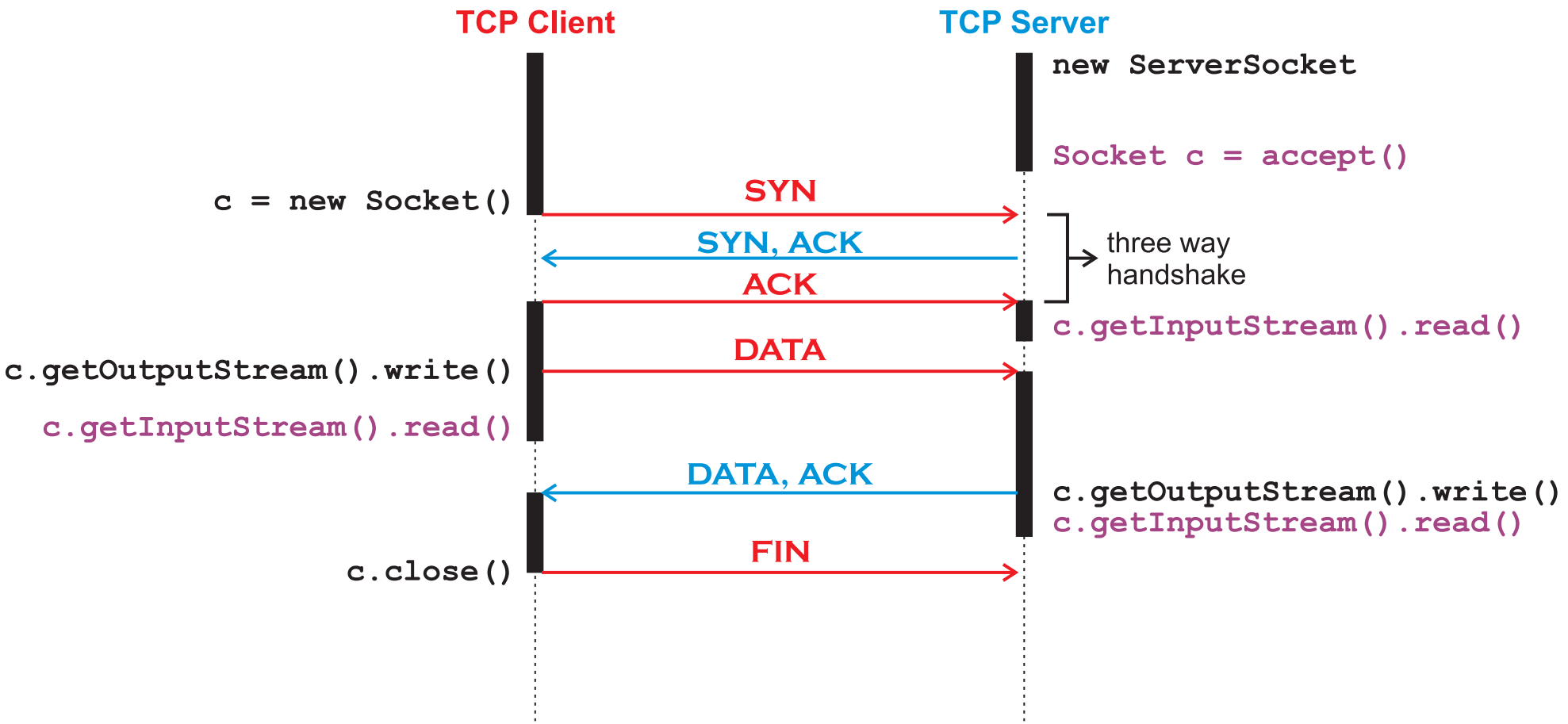
TCP: 3-way Handshake



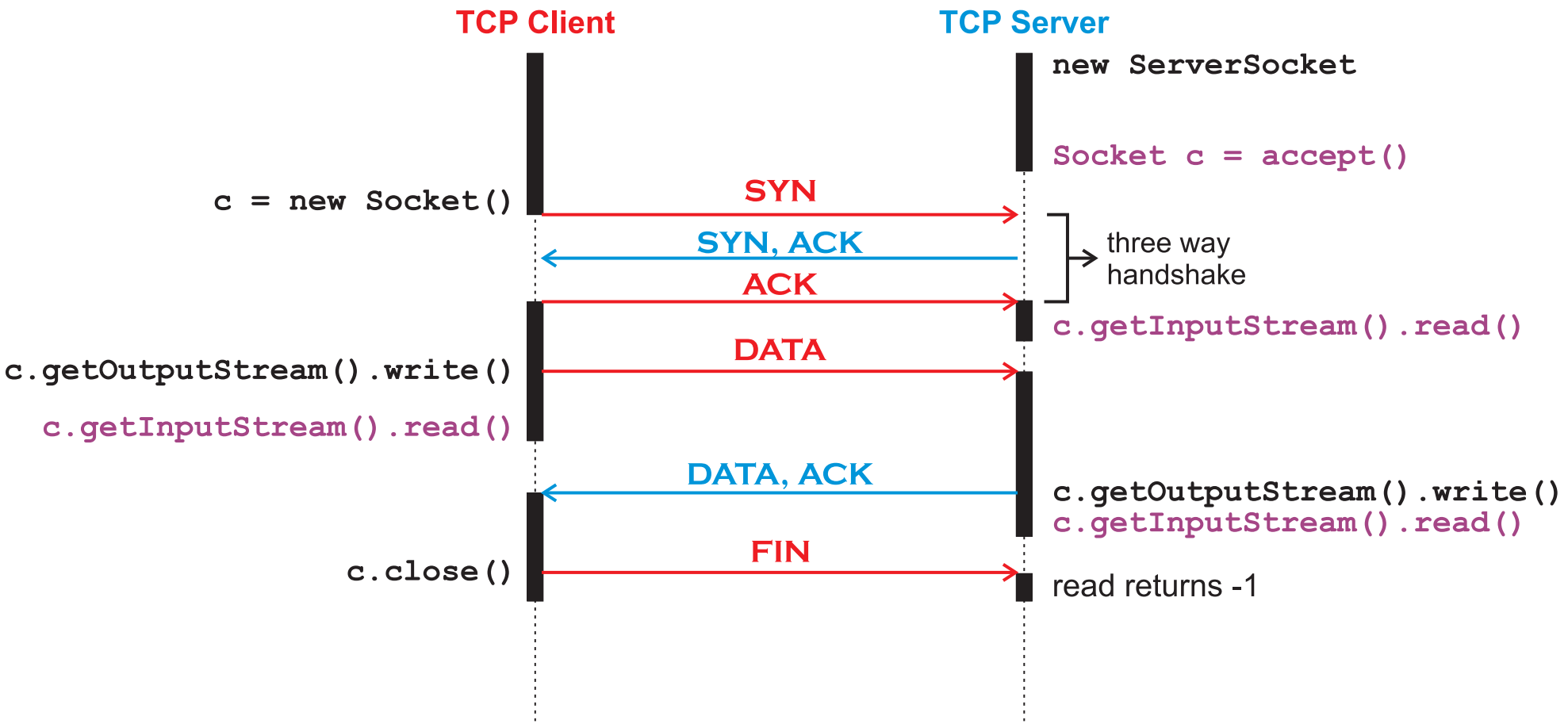
TCP: 3-way Handshake



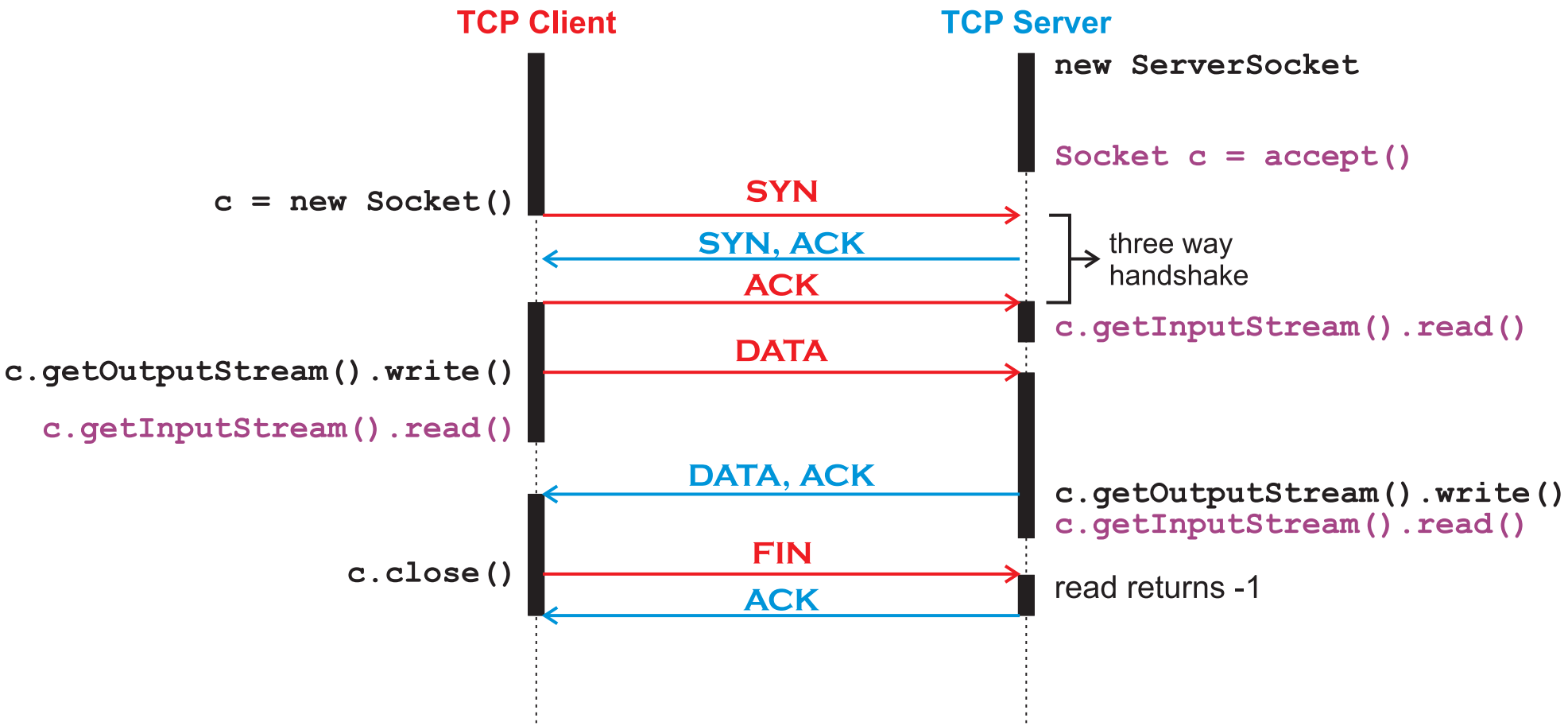
TCP: 3-way Handshake



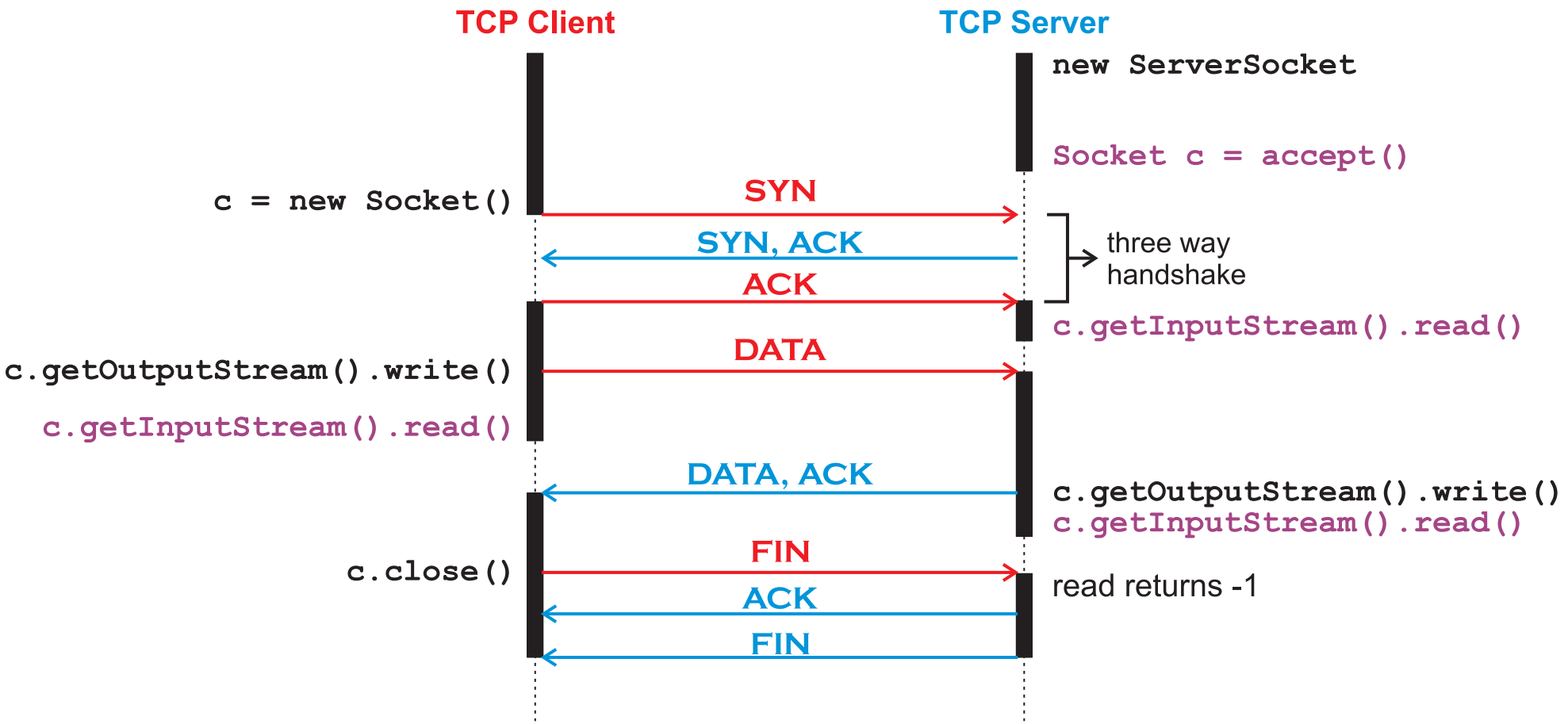
TCP: 3-way Handshake



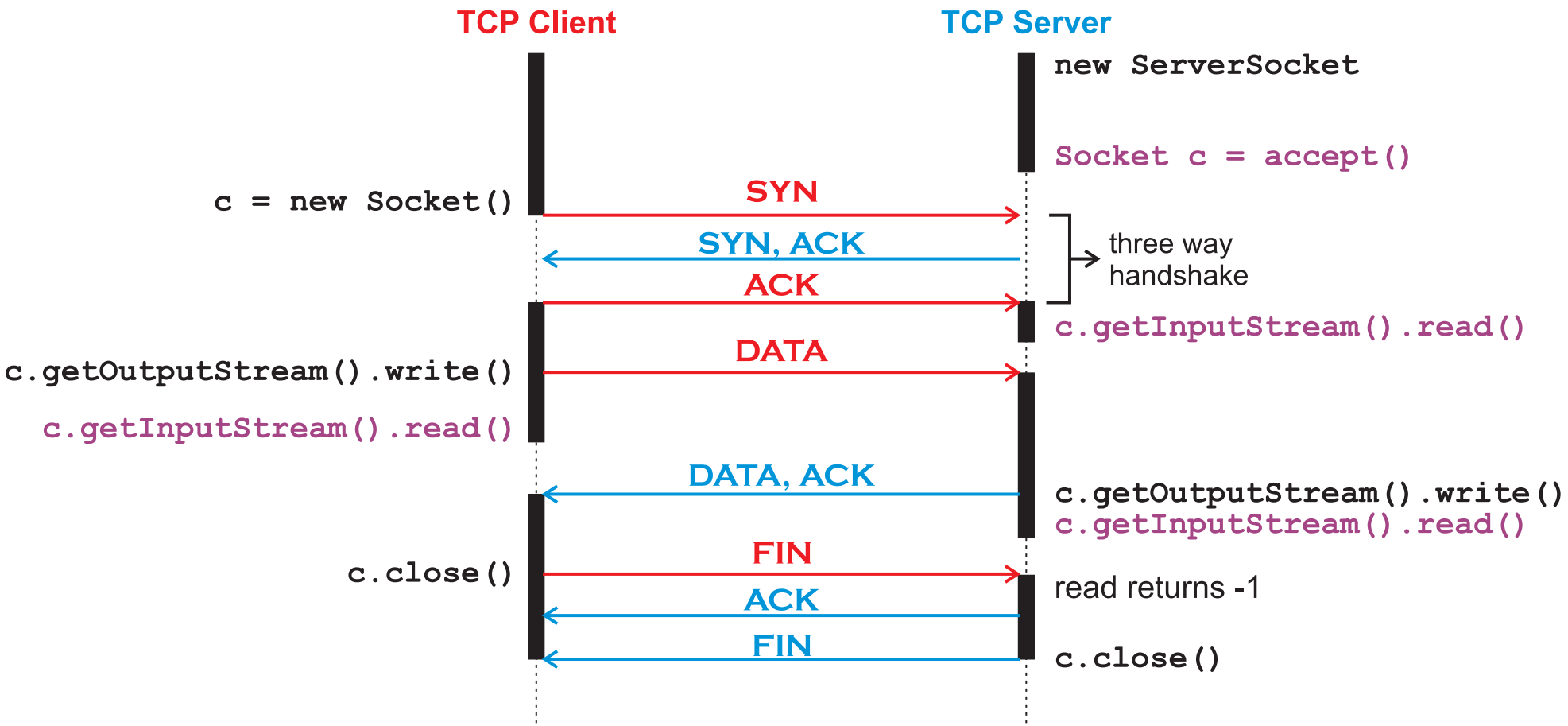
TCP: 3-way Handshake



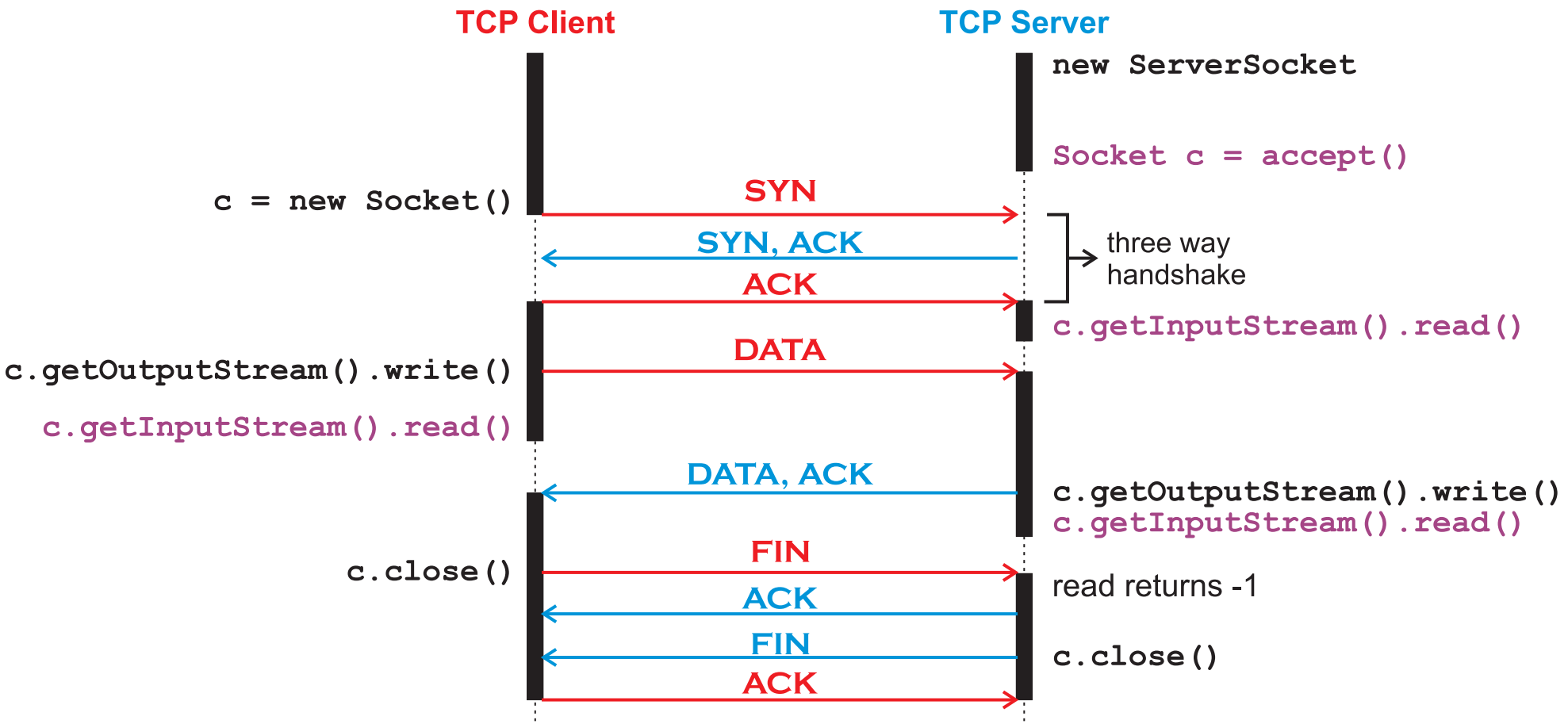
TCP: 3-way Handshake



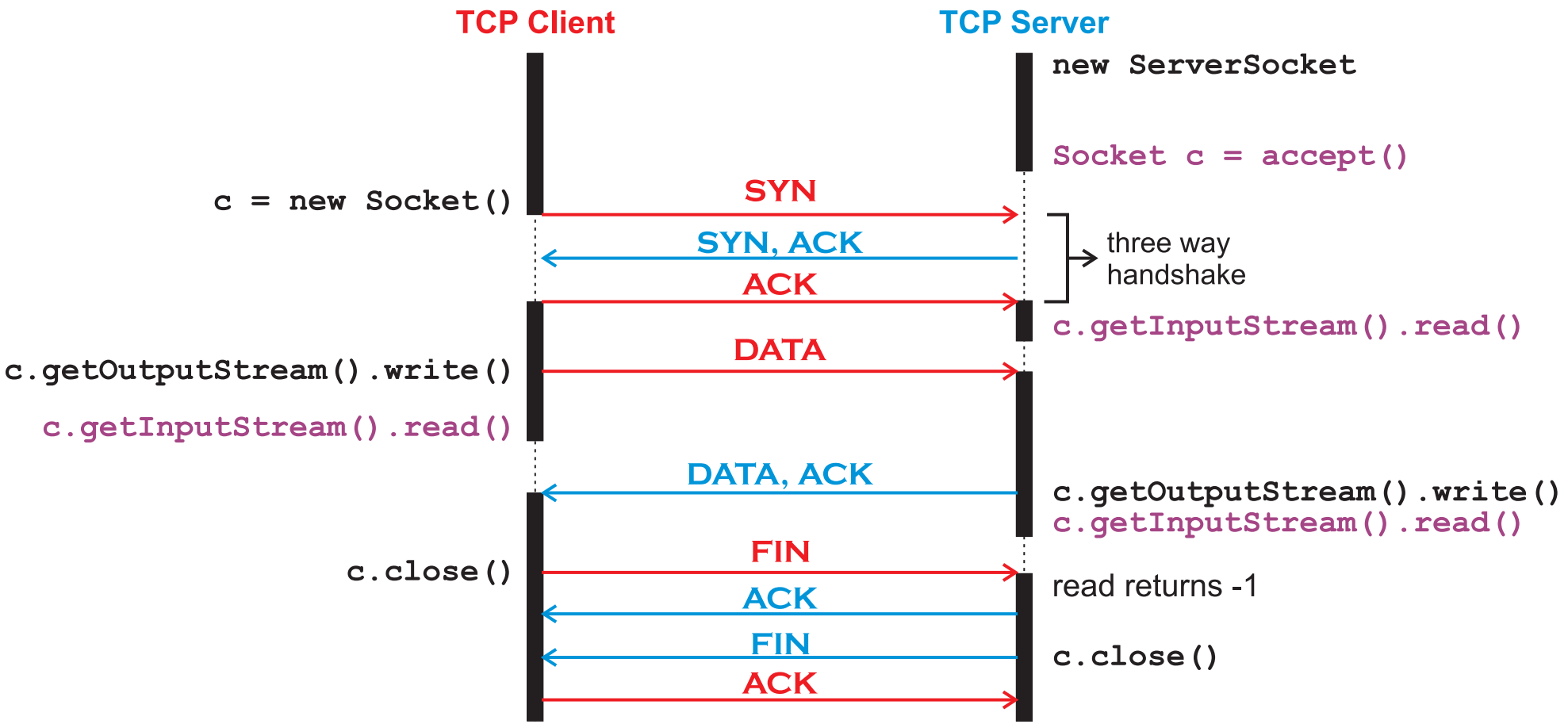
TCP: 3-way Handshake



TCP: 3-way Handshake



TCP: 3-way Handshake



TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data

TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]

TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]

TCP Sockets in Java

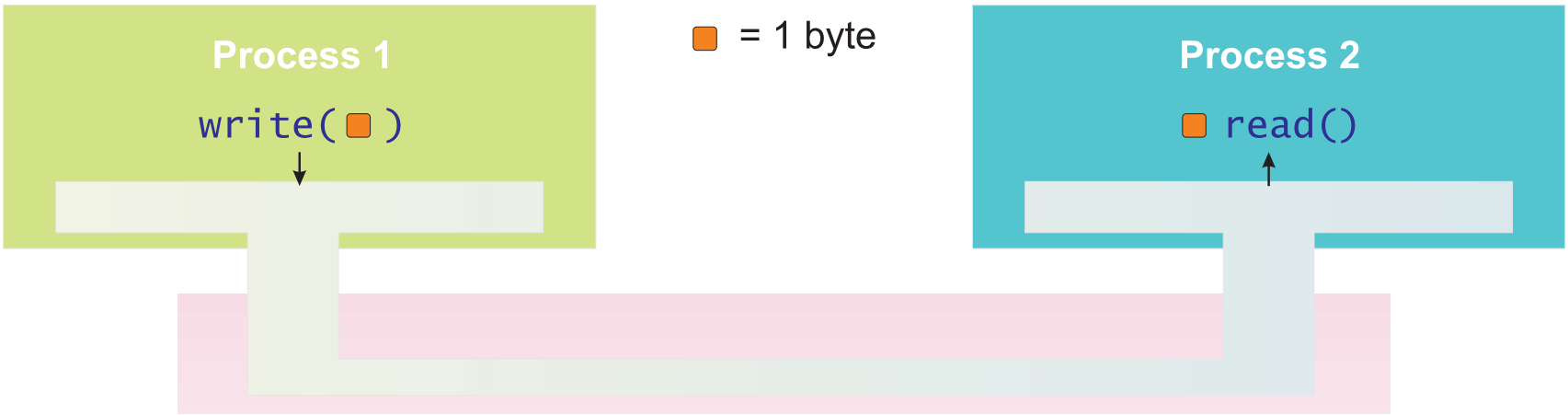
- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]:
  - `OutputStream`: write bytes sequentially

TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]:
  - `OutputStream`: write bytes sequentially
  - `InputStream`: read bytes sequentially

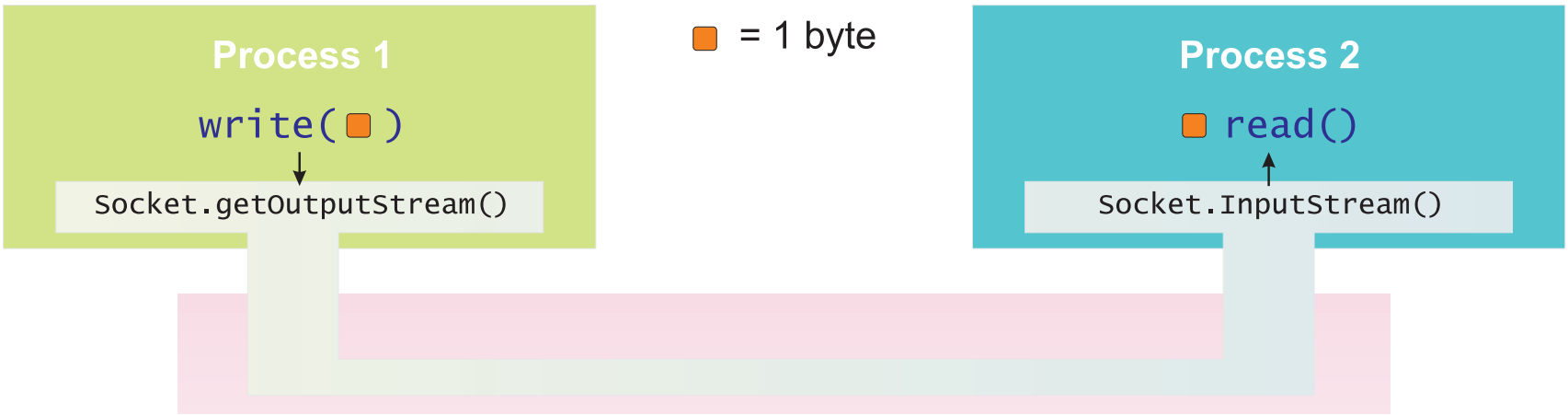
TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]:
  - `OutputStream`: write bytes sequentially
  - `InputStream`: read bytes sequentially
- `Socket` offers `getInputStream()` and `getOutputStream()` to access streams for reading and writing of data



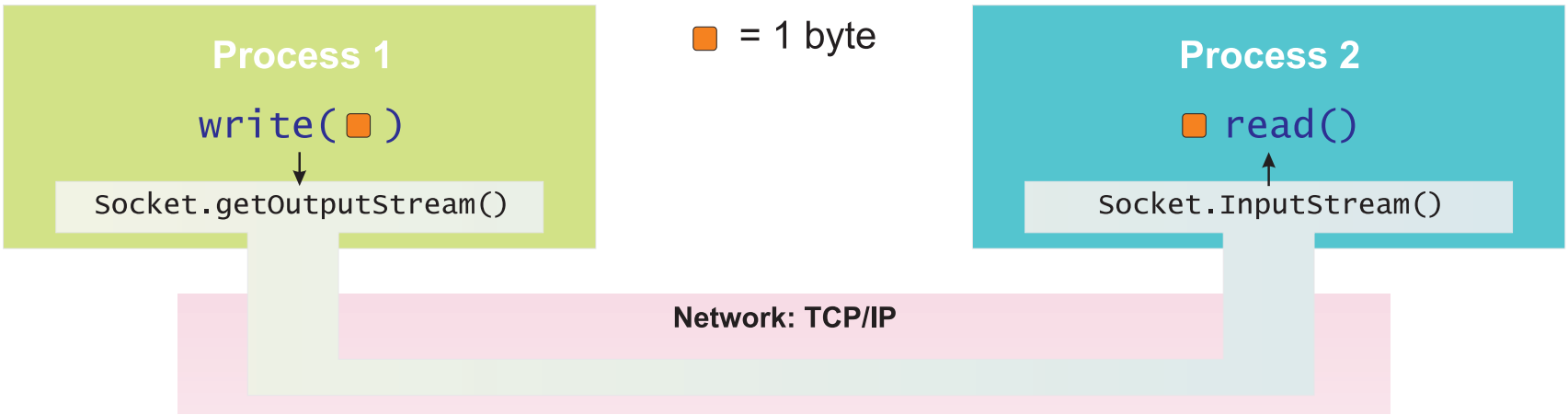
TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]:
  - `OutputStream`: write bytes sequentially
  - `InputStream`: read bytes sequentially
- `Socket` offers `getInputStream()` and `getOutputStream()` to access streams for reading and writing of data



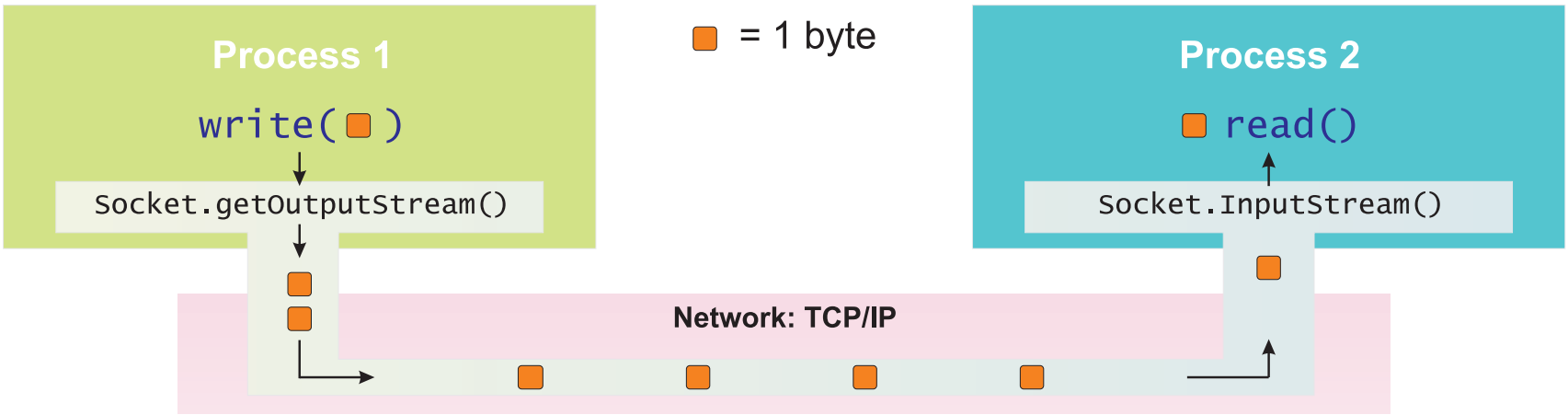
TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]:
  - `OutputStream`: write bytes sequentially
  - `InputStream`: read bytes sequentially
- `Socket` offers `getInputStream()` and `getOutputStream()` to access streams for reading and writing of data



TCP Sockets in Java

- Connection-oriented sockets offer sequential sending and receiving of data
- For sequential input and output of bytes, there is the stream API in Java [2, 23]:
  - `OutputStream`: write bytes sequentially
  - `InputStream`: read bytes sequentially
- `Socket` offers `getInputStream()` and `getOutputStream()` to access streams for reading and writing of data



TCP Sockets in Java: Server

Listing: TCPServer.java TCP Server in Java

```
import java.io.InputStream;   import java.net.ServerSocket;   import java.net.Socket;

public class TCPServer {
  public static final void main(final String[] args) {
    ServerSocket server;          InputStream  is;
    Socket       client;

    try {
      server = new ServerSocket(9999);//1 + 2)

      for (int j = 5; (--j) >= 0;) {  //process only 5 clients, so I can show 5) below
        client = server.accept(); //wait for incoming connection 3)
        System.out.println("New connection from " + client.getRemoteSocketAddress());

        is     = client.getInputStream(); //get stream to read from
        System.out.println(is.read());//4 + 3)

        client.close(); //close connection to client
      }
      server.close();   //5)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

TCP Sockets in Java: Client

Listing: TCPClient.java TCP Client in Java

```
import java.io.OutputStream;  import java.net.InetAddress;  import java.net.Socket;

public class TCPClient {

  public static final void main(final String[] args) {
    Socket        client;
    OutputStream  os;
    InetAddress   ia;

    try {
      ia = InetAddress.getByName("localhost");//get local host address

      client = new Socket(ia, 9999); //create socket 1+2)

      os = client.getOutputStream(); //get stream to write to
      os.write(1);  //write one byte of value 1 3)

      client.close(); //close 4)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

TCP Server in Java: Printing Text

Listing: TCPServerPrintingRawChars.java TCP Server in Java

```
import java.io.InputStream; import java.net.ServerSocket; import java.net.Socket;

public class TCPServerPrintingRawChars {

  public static final void main(final String[] args) {
    ServerSocket serv;      Socket client;      InputStream is;    int i;

    try {
      serv = new ServerSocket(9999); //start server 1 + 2)

      for (;;) {
        client = serv.accept();        //wait for incoming connection 3)
        is = client.getInputStream(); //get stream to read from connection

        while ((i = is.read()) >= 0) {//read bytes until connection closed 4 + 3)
          System.out.print((char) i); //cast byte to char: dangerous!
        }
        System.out.println(); //print newline

        is.close(); //close reading stream of connection
        client.close(); //close connection 4)
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

TCP Client in Java: Sending Text

Listing: TCPClientSendingRawChars.java TCP Client in Java

```
import java.io.OutputStream;    import java.net.InetAddress;    import
    java.net.Socket;

public class TCPClientSendingRawChars {

  public static final void main(final String[] args) {
    Socket        client;       OutputStream  os;
    InetAddress   ia;           int           ch;

    try {
      ia = InetAddress.getByName("localhost");

      client = new Socket(ia, 9999); //1+2)

      os = client.getOutputStream();
      while ( (ch = System.in.read()) != '\n' ){ //read 1 char (until newline)
        os.write(ch); //write char to connection, may be buffered and not yet sent 3)
      }

      client.close(); //flush and close connection 4)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]
- This statement makes sure that resources (implementing `AutoCloseable`) are automatically closed

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]
- This statement makes sure that resources (implementing `AutoCloseable`) are automatically closed, even if `Exceptions` (errors) occurs

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]
- This statement makes sure that resources (implementing `AutoCloseable`) are automatically closed, even if `Exceptions` (errors) occurs
- `try-with-resource` is similar to a special `try-final` statement

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]
- This statement makes sure that resources (implementing `AutoCloseable`) are automatically closed, even if `Exceptions` (errors) occurs
- `try-with-resource` is similar to a special `try-final` statement
- This makes code more compact and less error prone

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]
- This statement makes sure that resources (implementing `AutoCloseable`) are automatically closed, even if `Exceptions` (errors) occurs
- `try-with-resource` is similar to a special `try-final` statement
- This makes code more compact and less error prone
- Resources that can automatically closed are all types of sockets and streams

Java 1.7 Try-With-Resource Statement

- Since Java 1.7, there is the so-called `try-with-resource` statement [24]
- This statement makes sure that resources (implementing `AutoCloseable`) are automatically closed, even if `Exceptions` (errors) occurs
- `try-with-resource` is similar to a special `try-final` statement
- This makes code more compact and less error prone
- Resources that can automatically closed are all types of sockets and streams
- This makes our socket code much smaller

General form of the Try-With-Resource Statement

Listing: General form of the Try-With-Resource Statement

```
...
 try(ResourceClass resource = new ResourceClass(...)){  //
     create/open resource
       ... //do something with resource
 }  // resource is automatically closed when end of block is
     reached
...
```

TCP Server Socket (Java Try-With-Resource)

Listing: TCPServerJava17.java TCP Server in Java

```
import java.io.InputStream; import java.net.ServerSocket; import java.net.Socket;

public class TCPServerJava17 {
  public static final void main(final String[] args) {

    try(ServerSocket server = new ServerSocket(9999)){  //1 + 2)

      for (int j = 5; (--j) >= 0;) { //process only 5 clients, so I can show 5) below
        try(Socket client = server.accept()) { //wait for incoming connection 3)
          System.out.println("New connection from " +
              client.getRemoteSocketAddress());

          try(InputStream is = client.getInputStream()){//get stream to read
            System.out.println(is.read()); //4 + 3)
            } //close reading end of connection
          } //close connection 5)
        }
    //5)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

TCP Client Socket (Java Try-With-Resource))

Listing: TCPClientJava17.java TCP Client in Java

```
import java.io.OutputStream; import java.net.InetAddress; import java.net.Socket;

public class TCPClientJava17 {

  public static final void main(final String[] args) {
    InetAddress   ia;

    try {
      ia = InetAddress.getByName("localhost");

      try(Socket client = new Socket(ia, 9999)){    //1+2)

       try(OutputStream os = client.getOutputStream()) {
           os.write(1);  //write one byte with value 1 3)
         } //close writing end of connection
      }  //4)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

TCP Sockets in C

- In C [25], using TCP/IP sockets is a bit more complicated

TCP Sockets in C

- In C [25], using TCP/IP sockets is a bit more complicated
- Windows and Unix/Linux have different headers and a slightly different API

TCP Sockets in C

- In C [25], using TCP/IP sockets is a bit more complicated
- Windows and Unix/Linux have different headers and a slightly different API
- Code is not a priori portable, can maybe made portable with lots of `#define`s

TCP Sockets in C

- In C [25], using TCP/IP sockets is a bit more complicated
- Windows and Unix/Linux have different headers and a slightly different API
- Code is not a priori portable, can maybe made portable with lots of `#define`s
- Windows: Compile as

  `gcc` *`fileName`*`_windows.c -o` *`fileName`*`_windows.exe -lws2_32`

TCP Sockets in C

- In C [25], using TCP/IP sockets is a bit more complicated
- Windows and Unix/Linux have different headers and a slightly different API
- Code is not a priori portable, can maybe made portable with lots of `#define`s
- Windows: Compile as

  `gcc` *`fileName`*`_windows.c -o` *`fileName`*`_windows.exe -lws2_32`

  where `-lws2_32` says "link against Winsock"

TCP Sockets in C

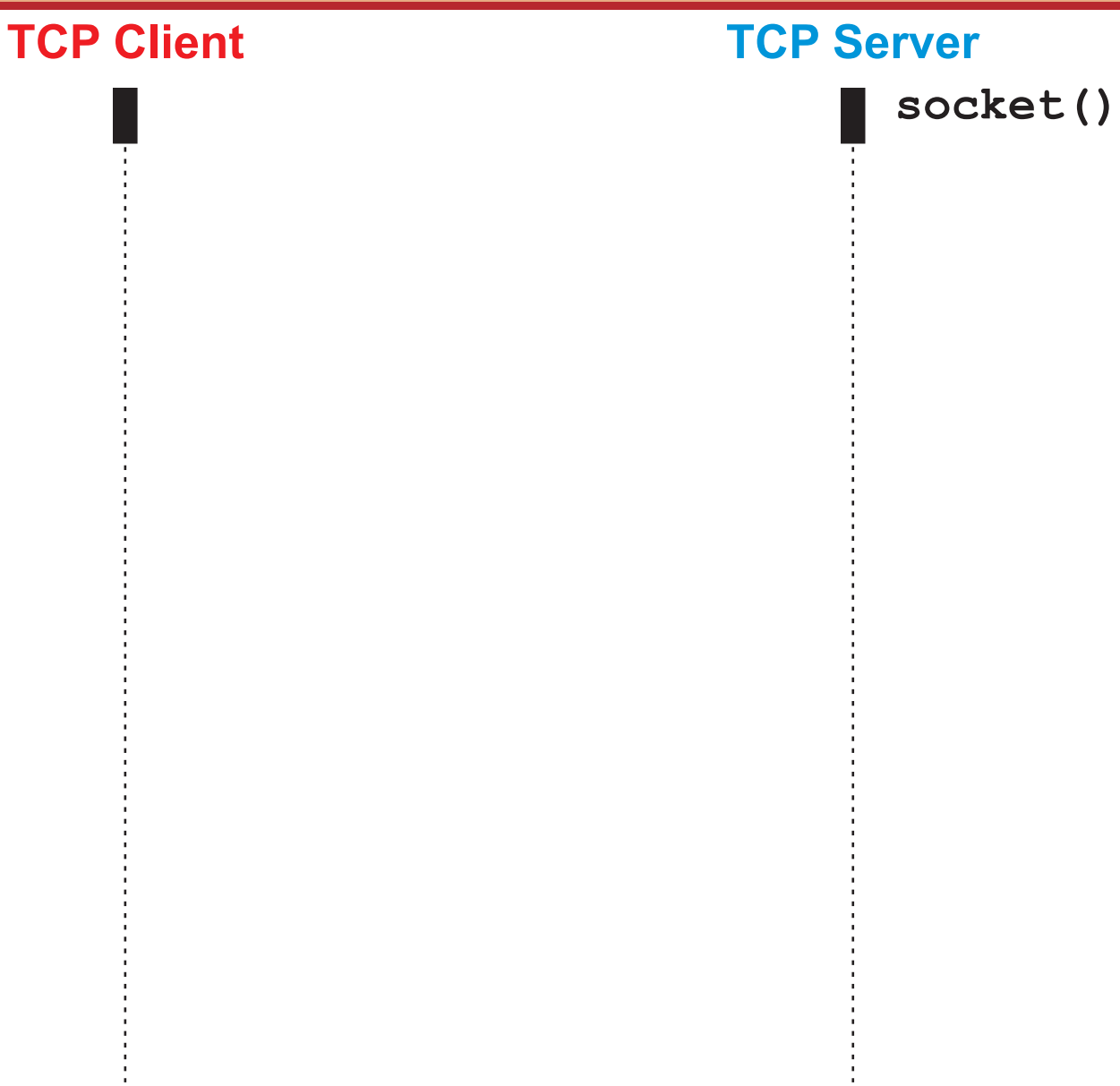
- In C [25], using TCP/IP sockets is a bit more complicated
- Windows and Unix/Linux have different headers and a slightly different API
- Code is not a priori portable, can maybe made portable with lots of `#define`s
- Windows: Compile as

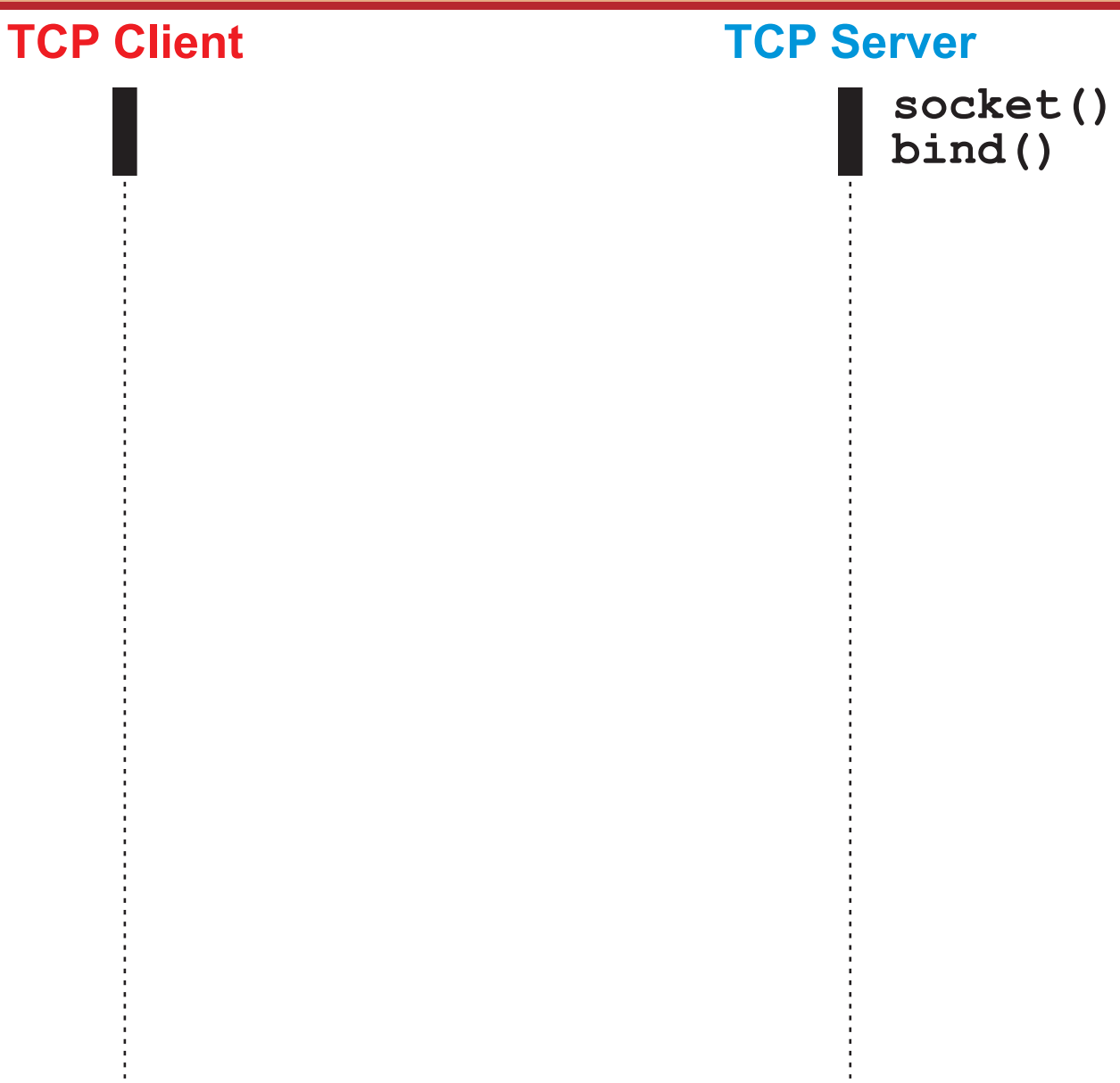
  `gcc` *`fileName`*`_windows.c -o` *`fileName`*`_windows.exe -lws2_32`

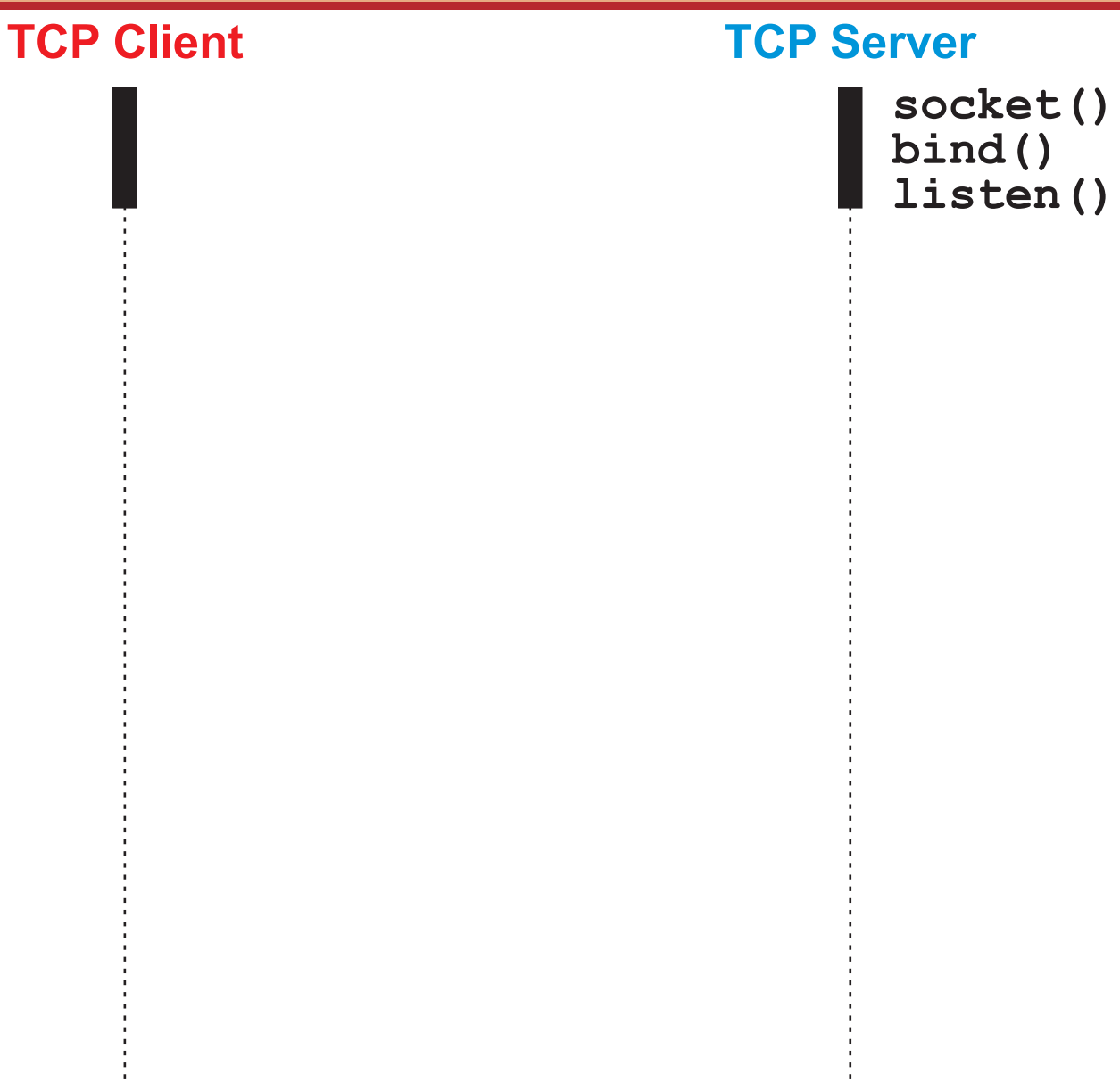
  where `-lws2_32` says "link against Winsock"
- Linux: Compile as

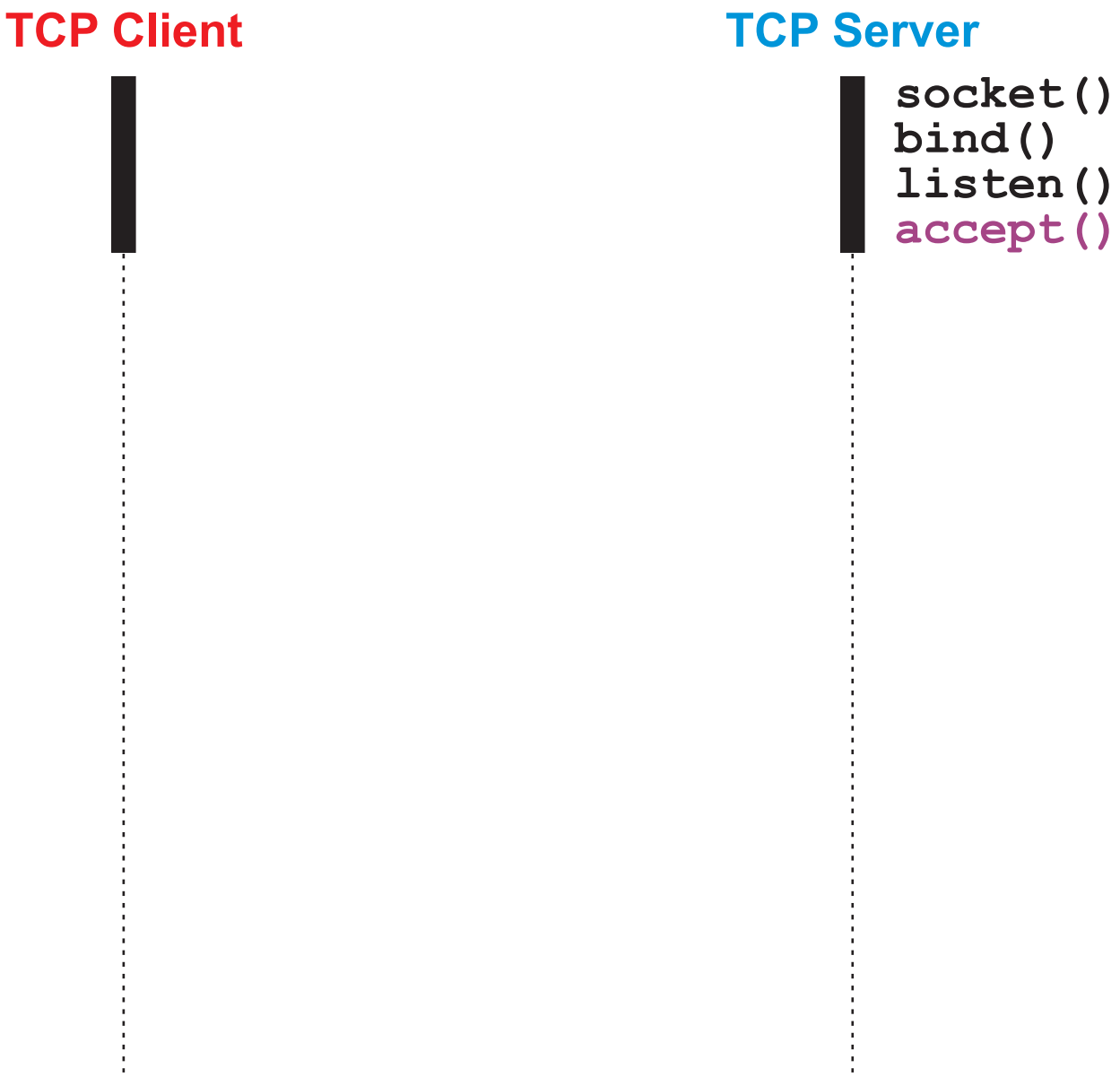
  `gcc` *`fileName`*`_linux.c -o` *`fileName`*`_linux`

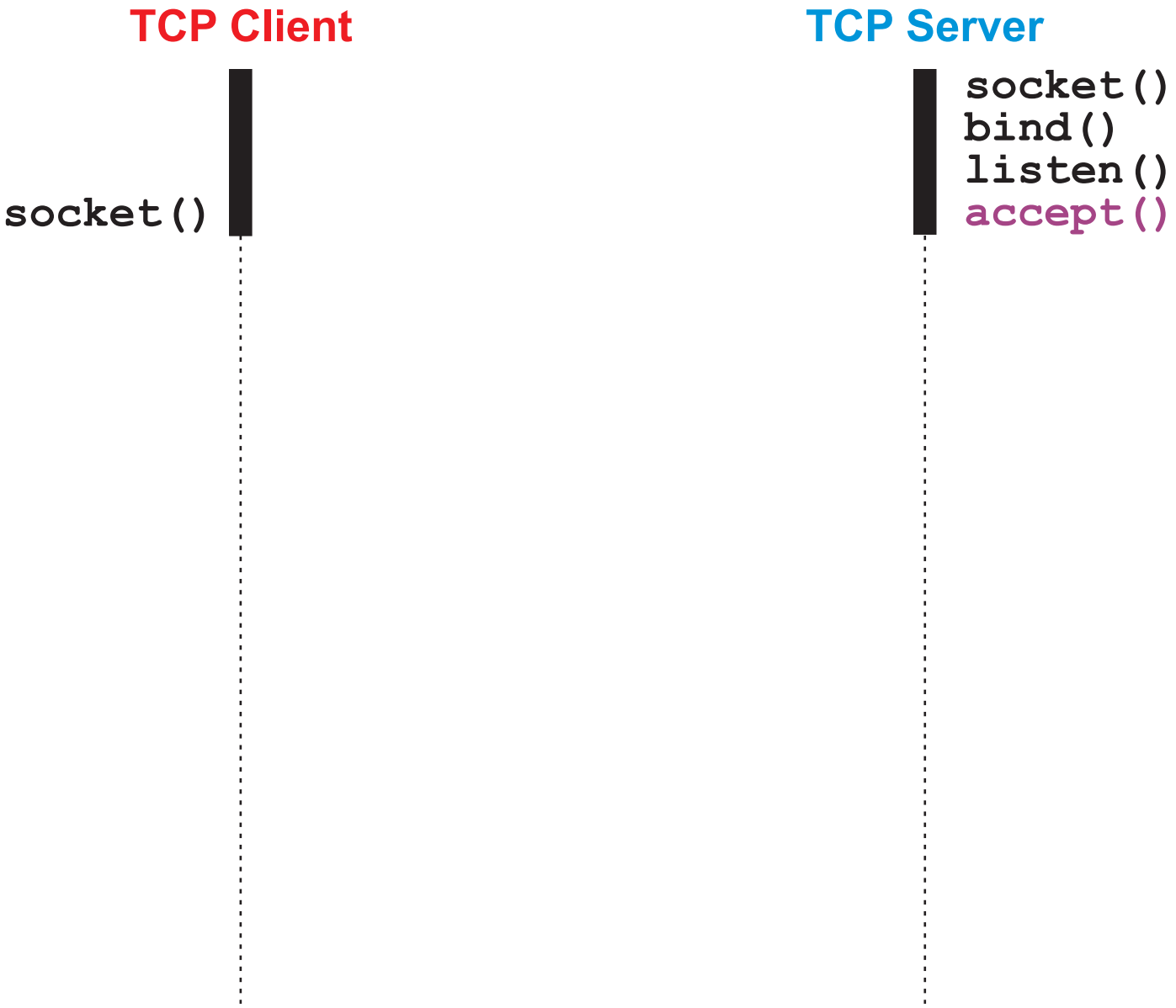

TCP Client
TCP Server

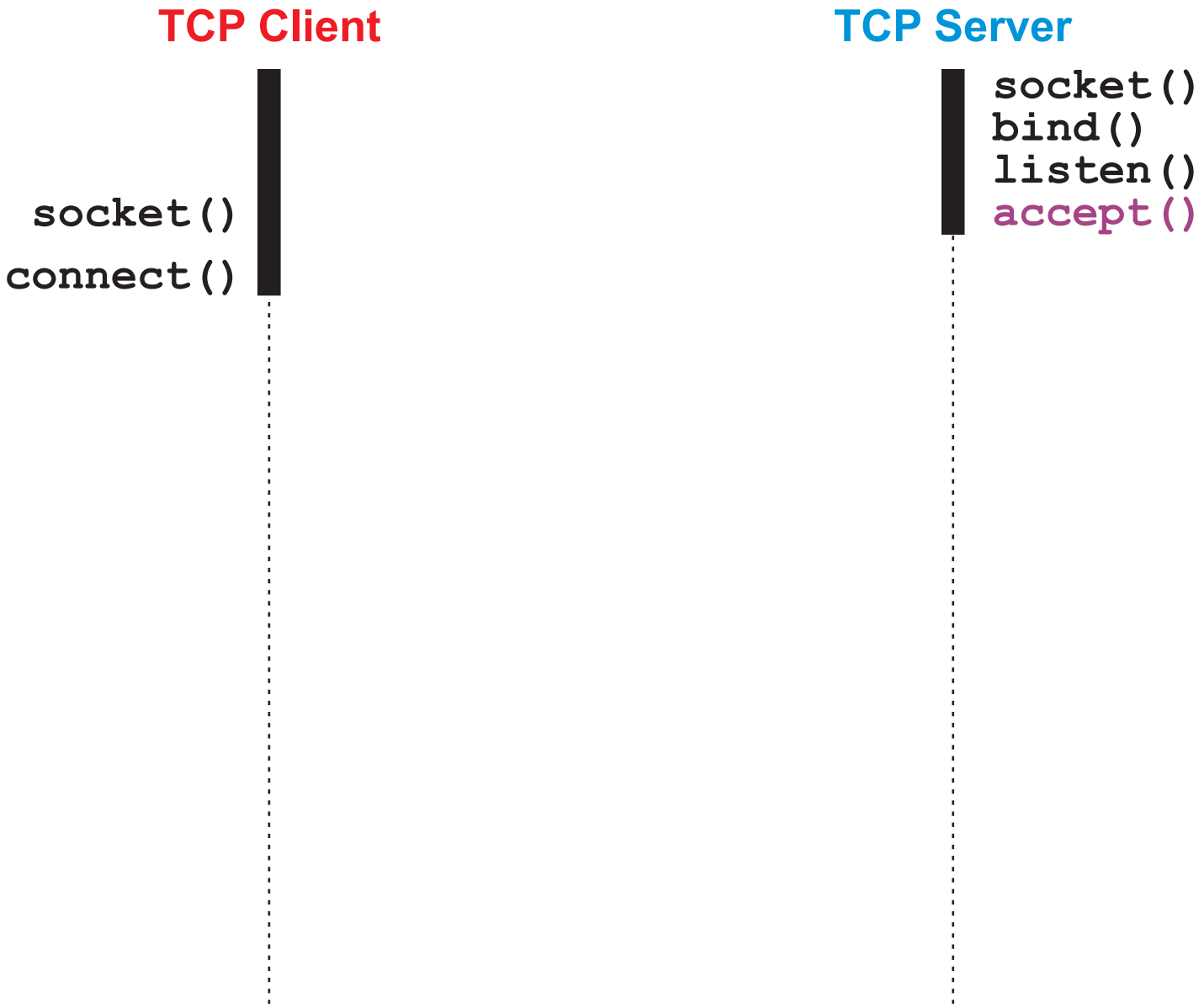

TCP Client
TCP Server
socket()

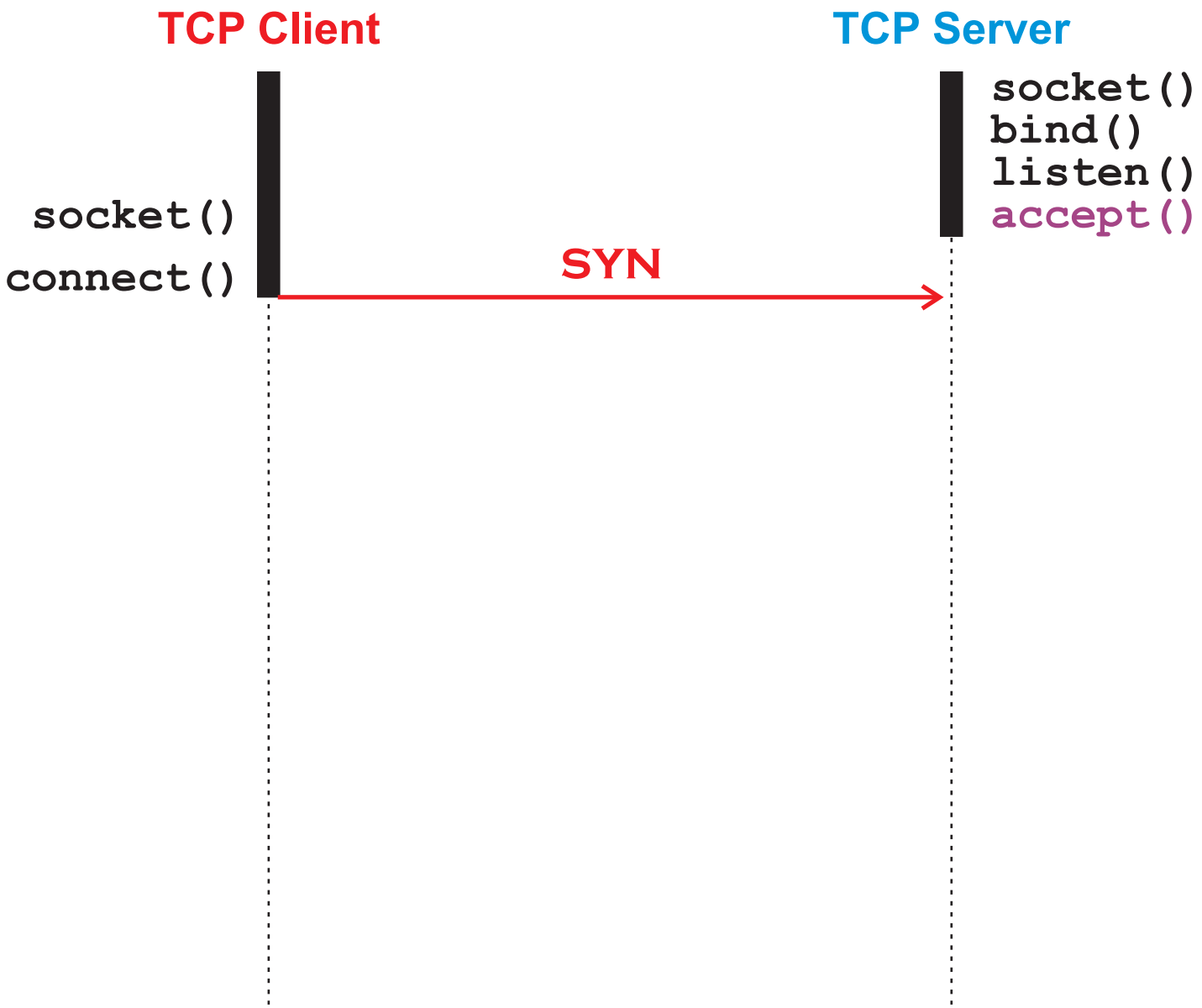

TCP Client
TCP Server
socket()
bind()

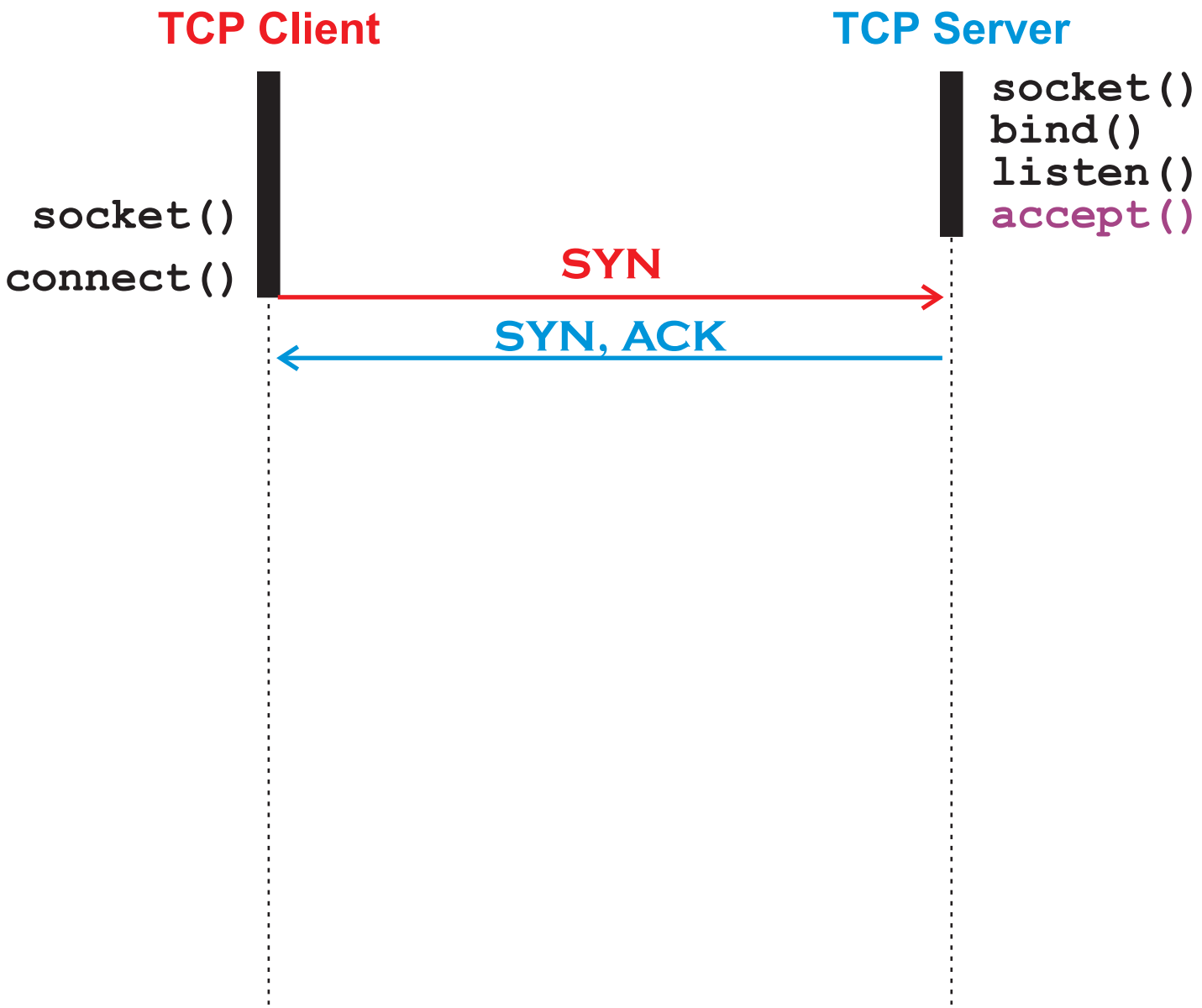

TCP Client
TCP Server
socket()
bind()
listen()

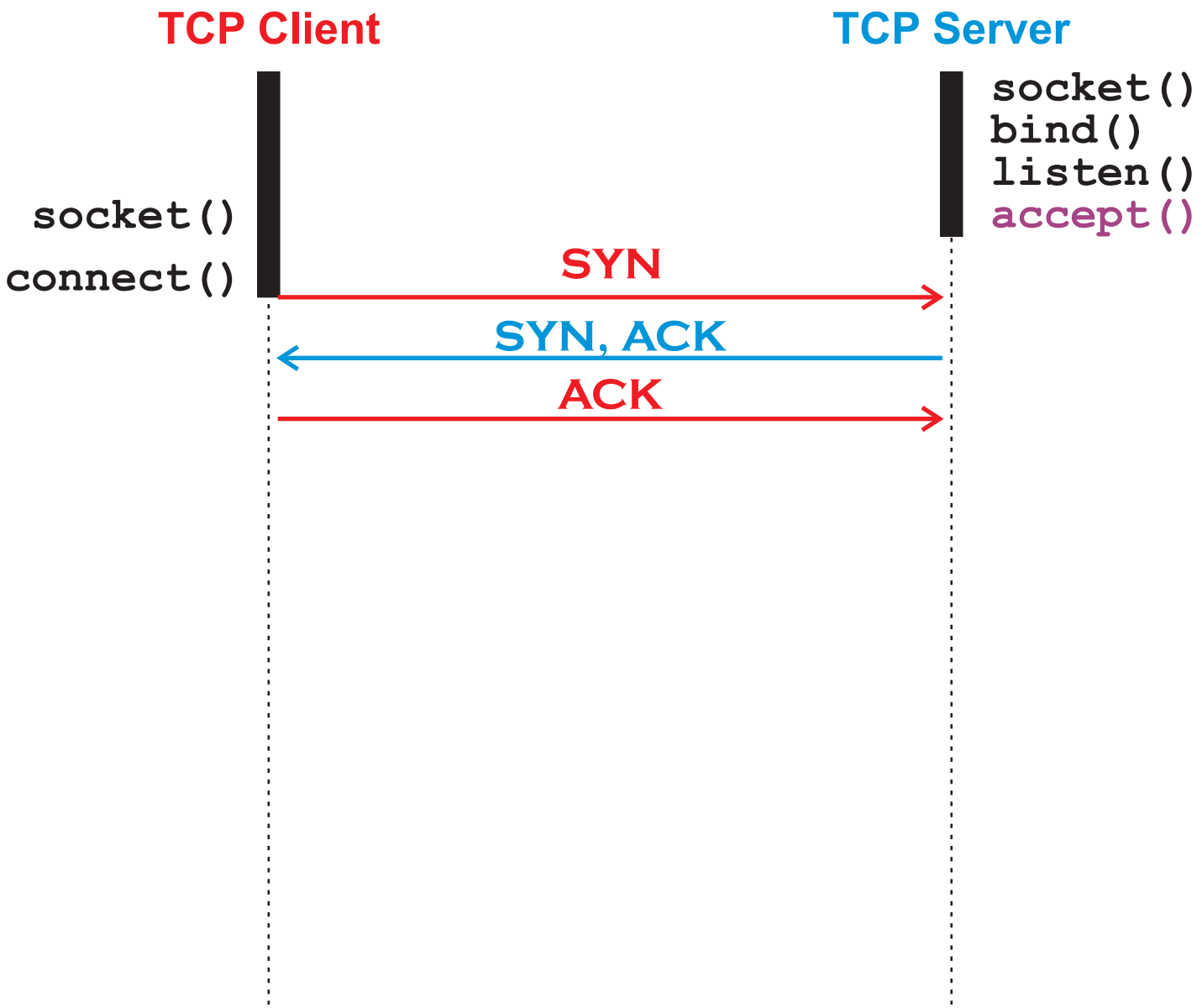

TCP Client
TCP Server
socket()
bind()
listen()
accept()

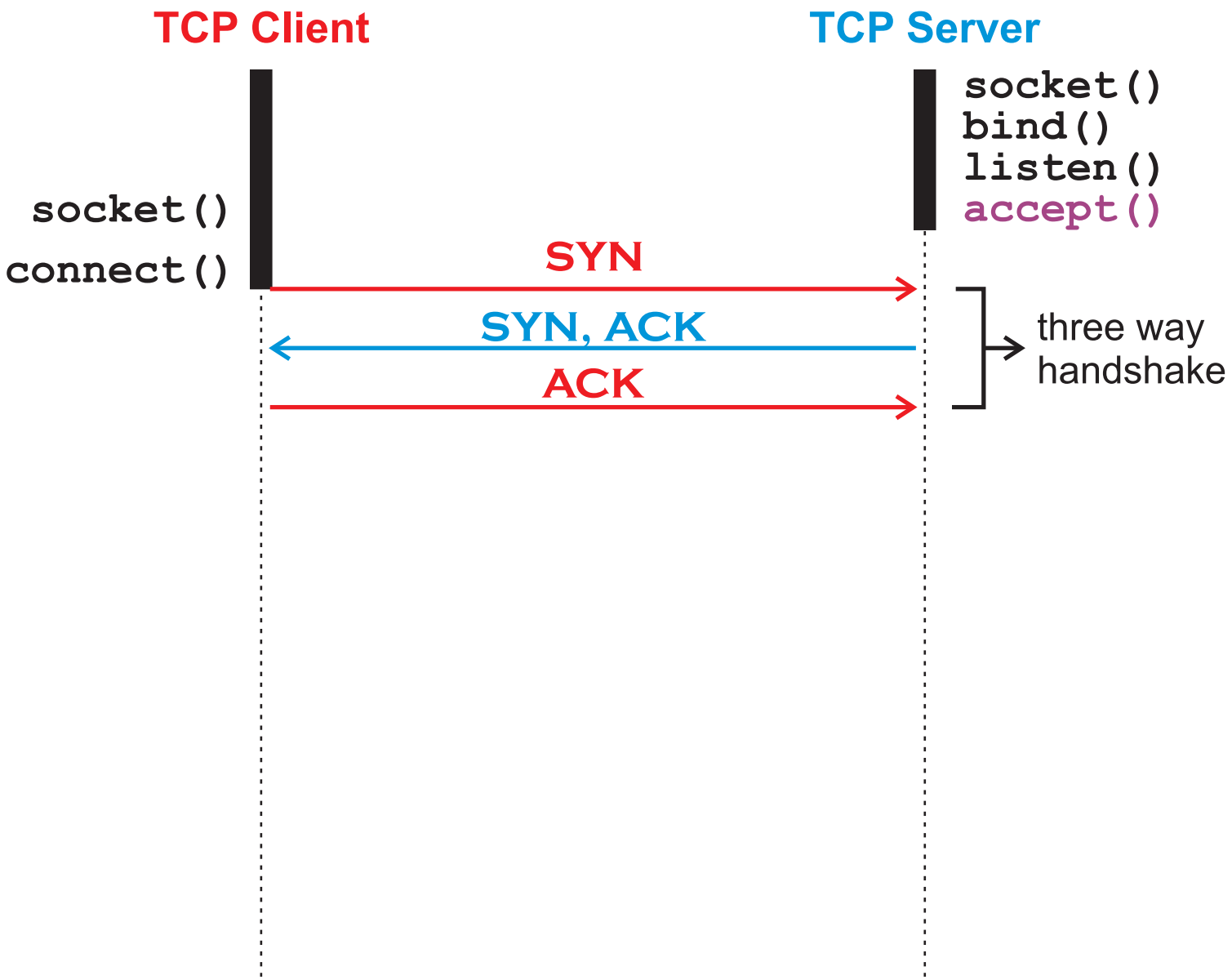

TCP Client
TCP Server
socket()
socket()
bind()
listen()
accept()

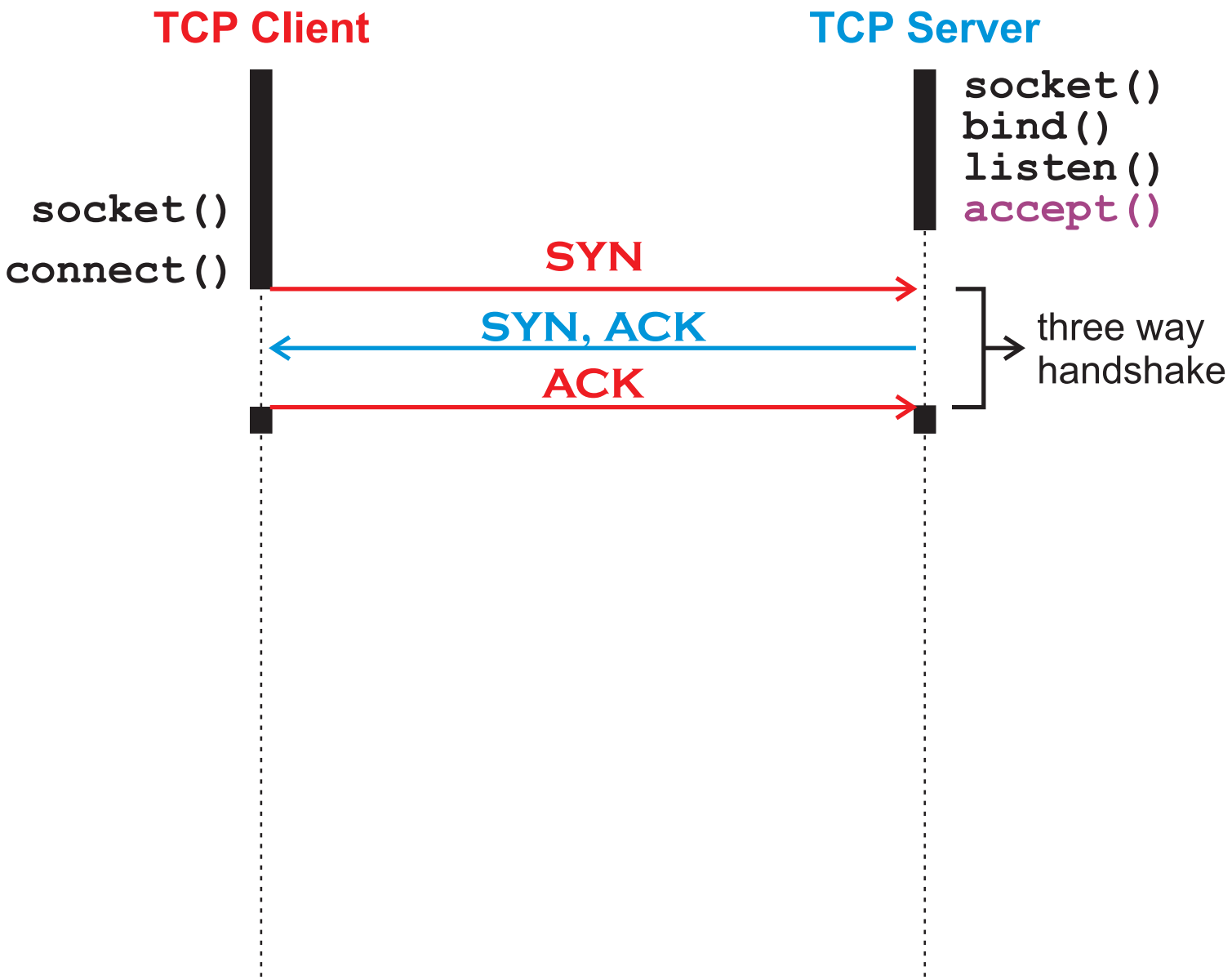

TCP Client
TCP Server
socket()
connect()
socket()
bind()
listen()
accept()

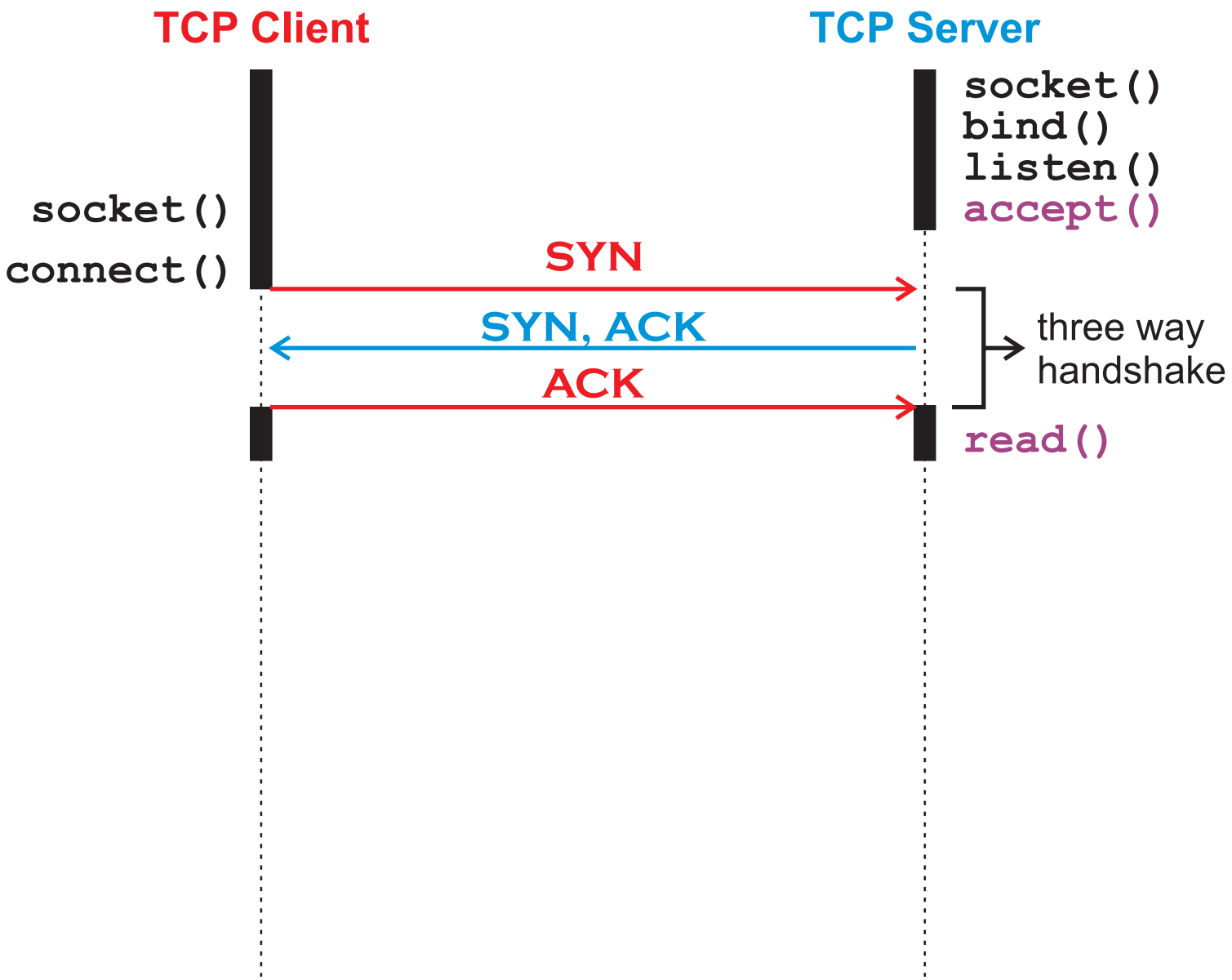

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN

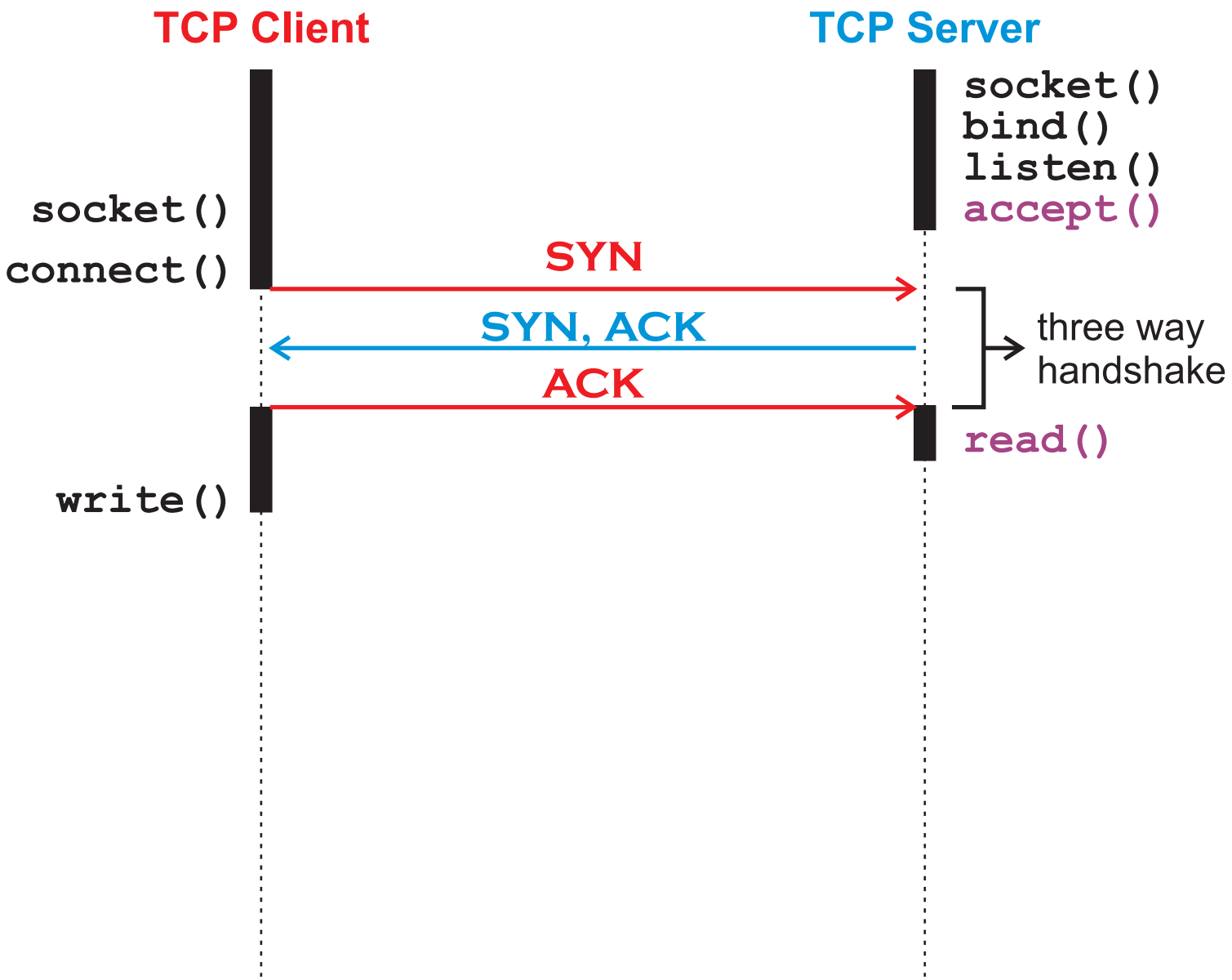

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK

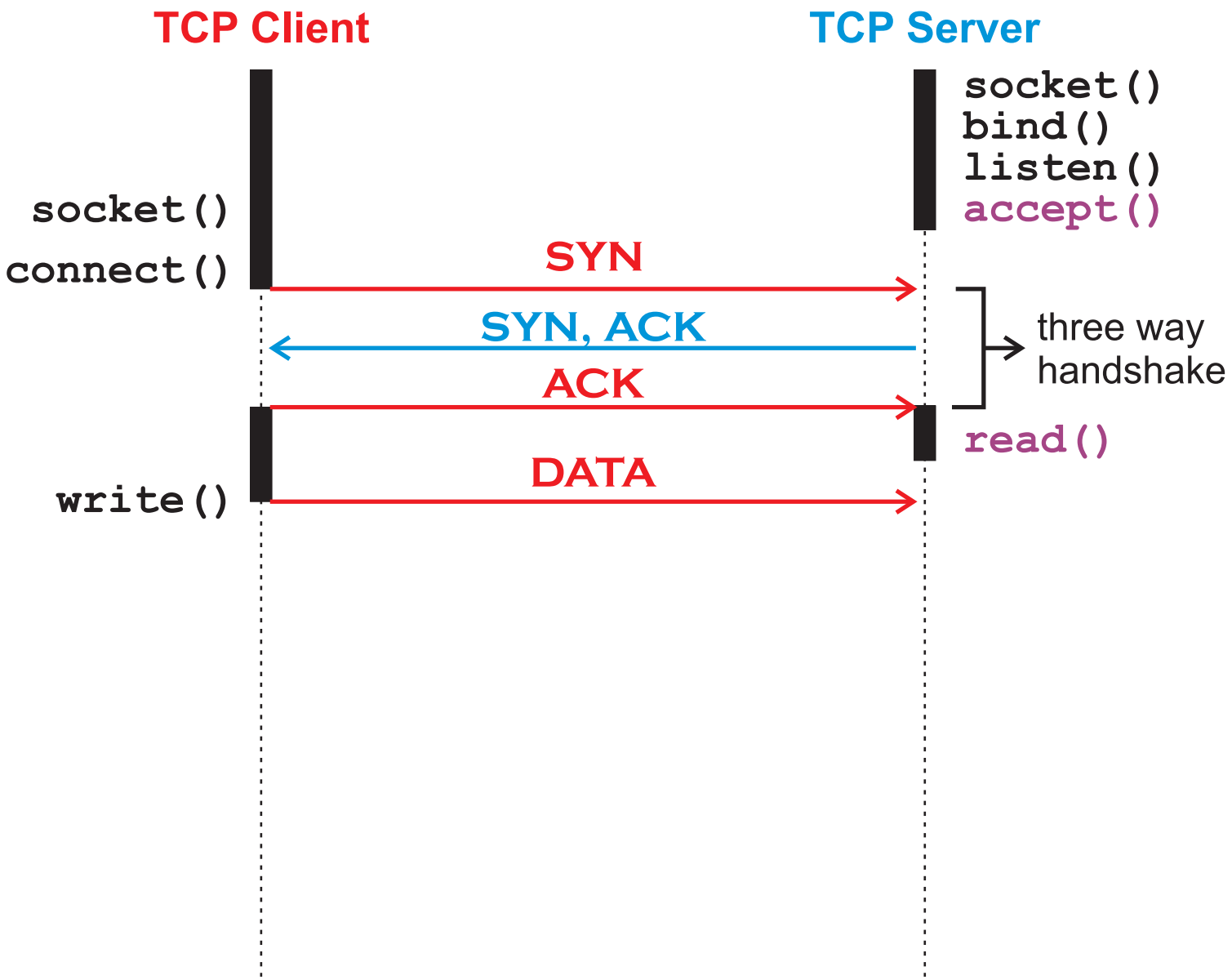

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK

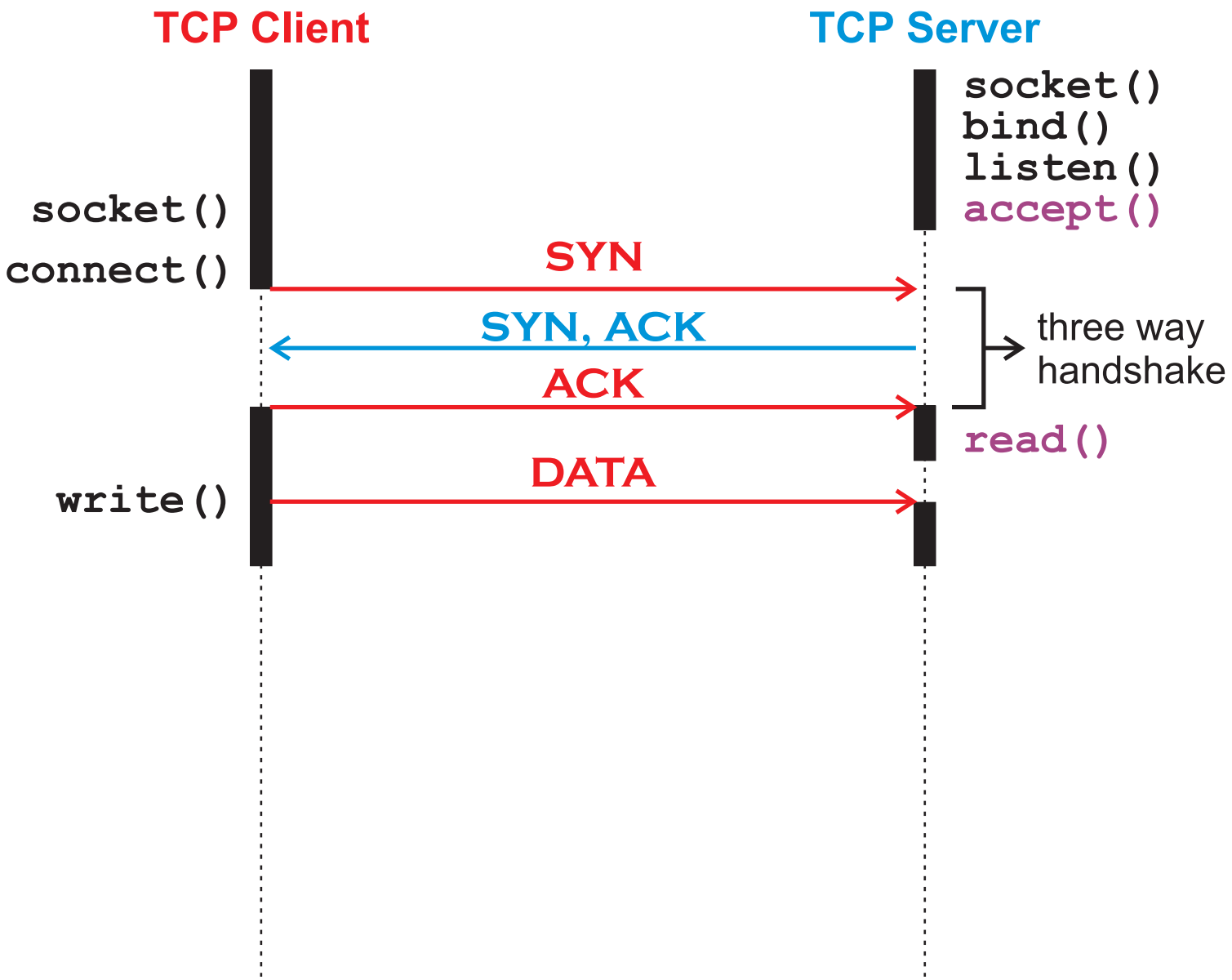

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way
handshake

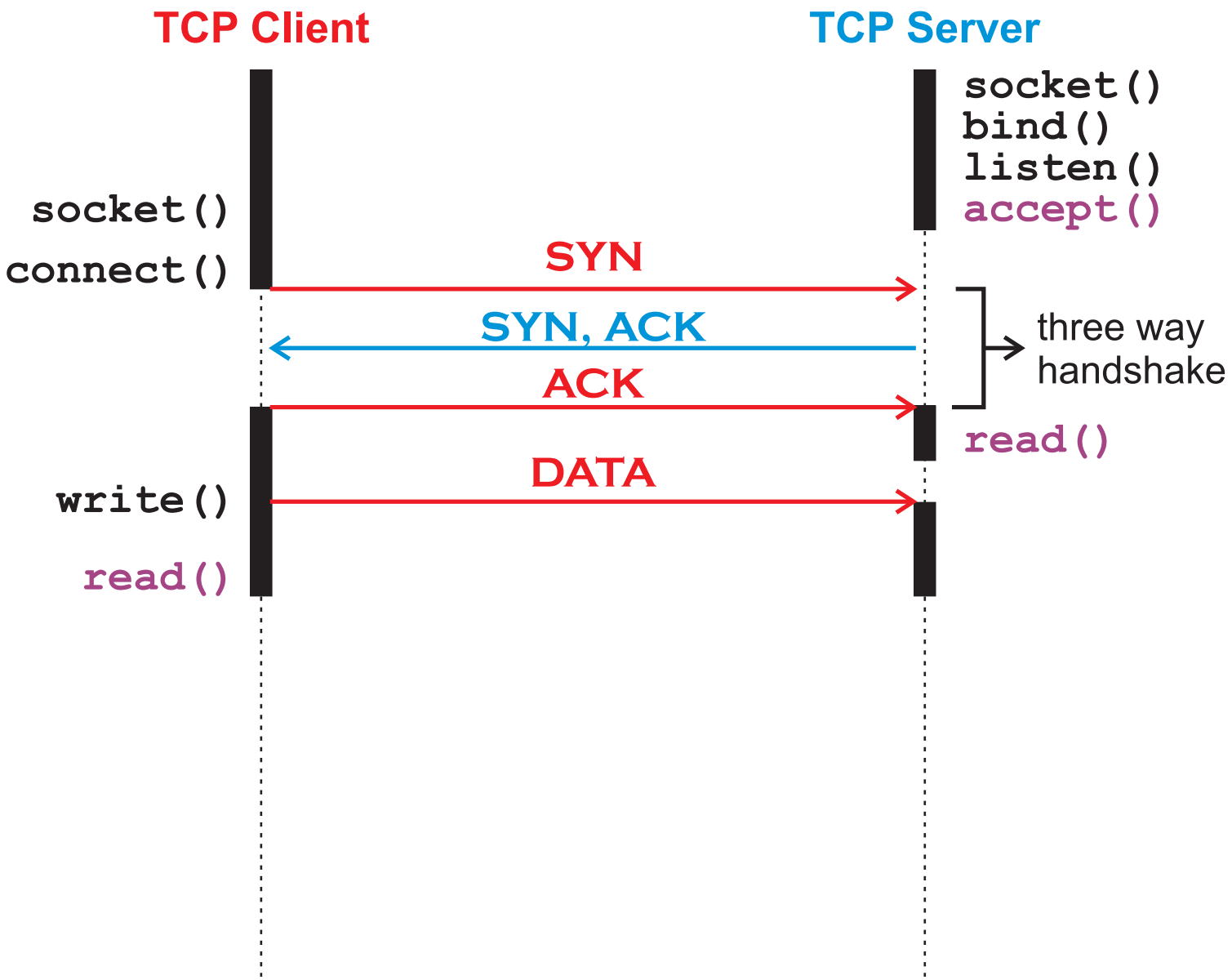

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way
handshake

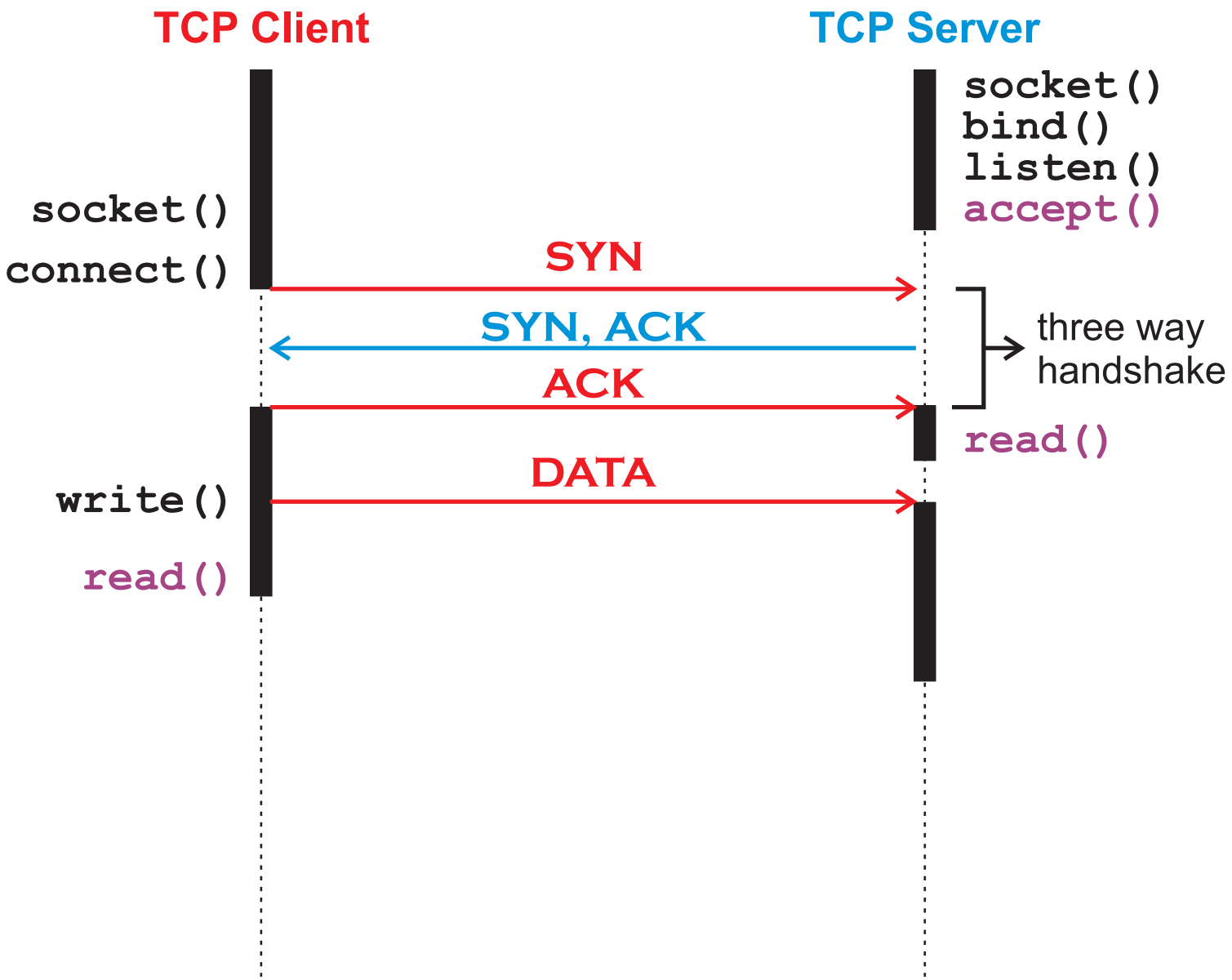

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way
handshake
read()

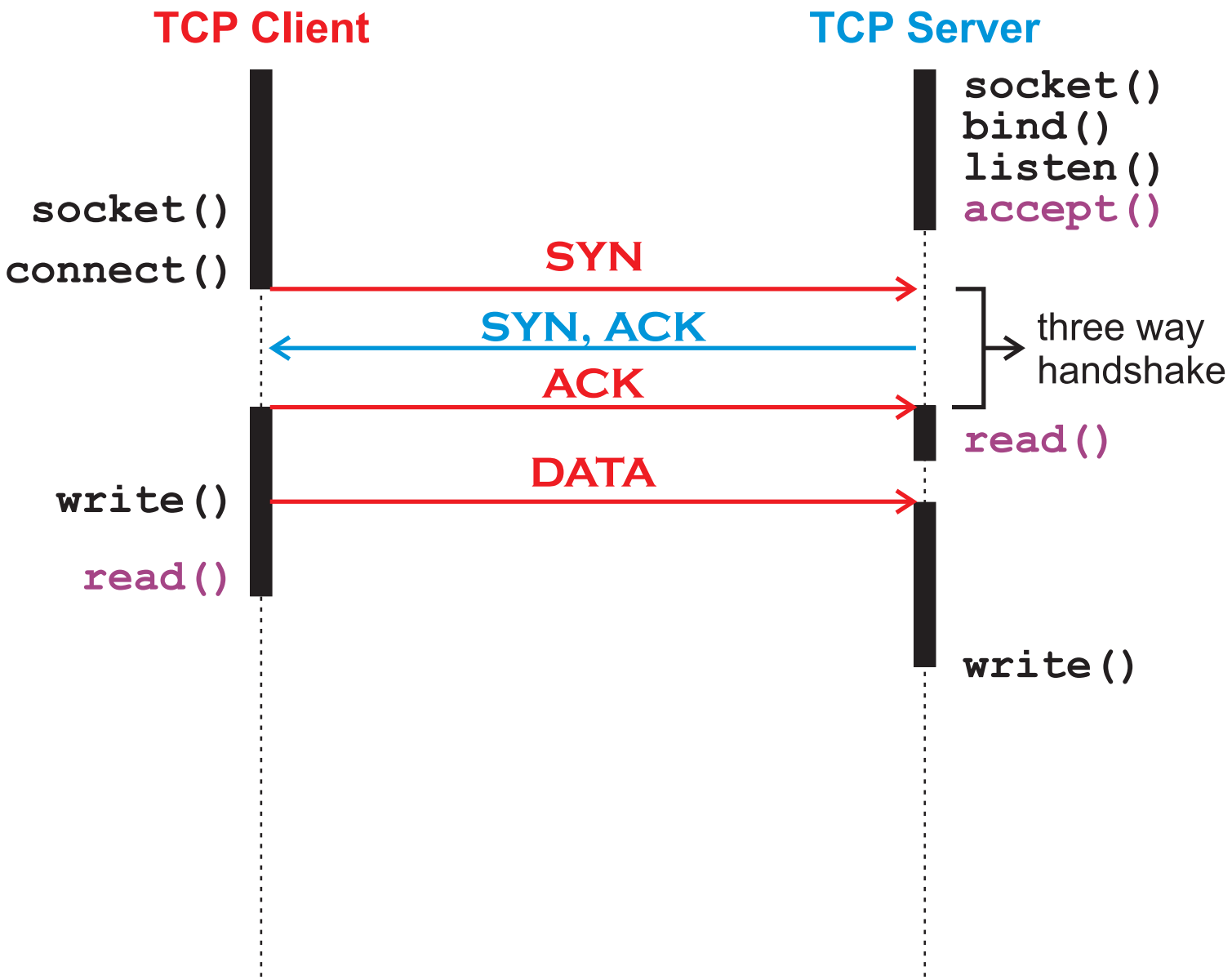

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way
handshake
read()
write()

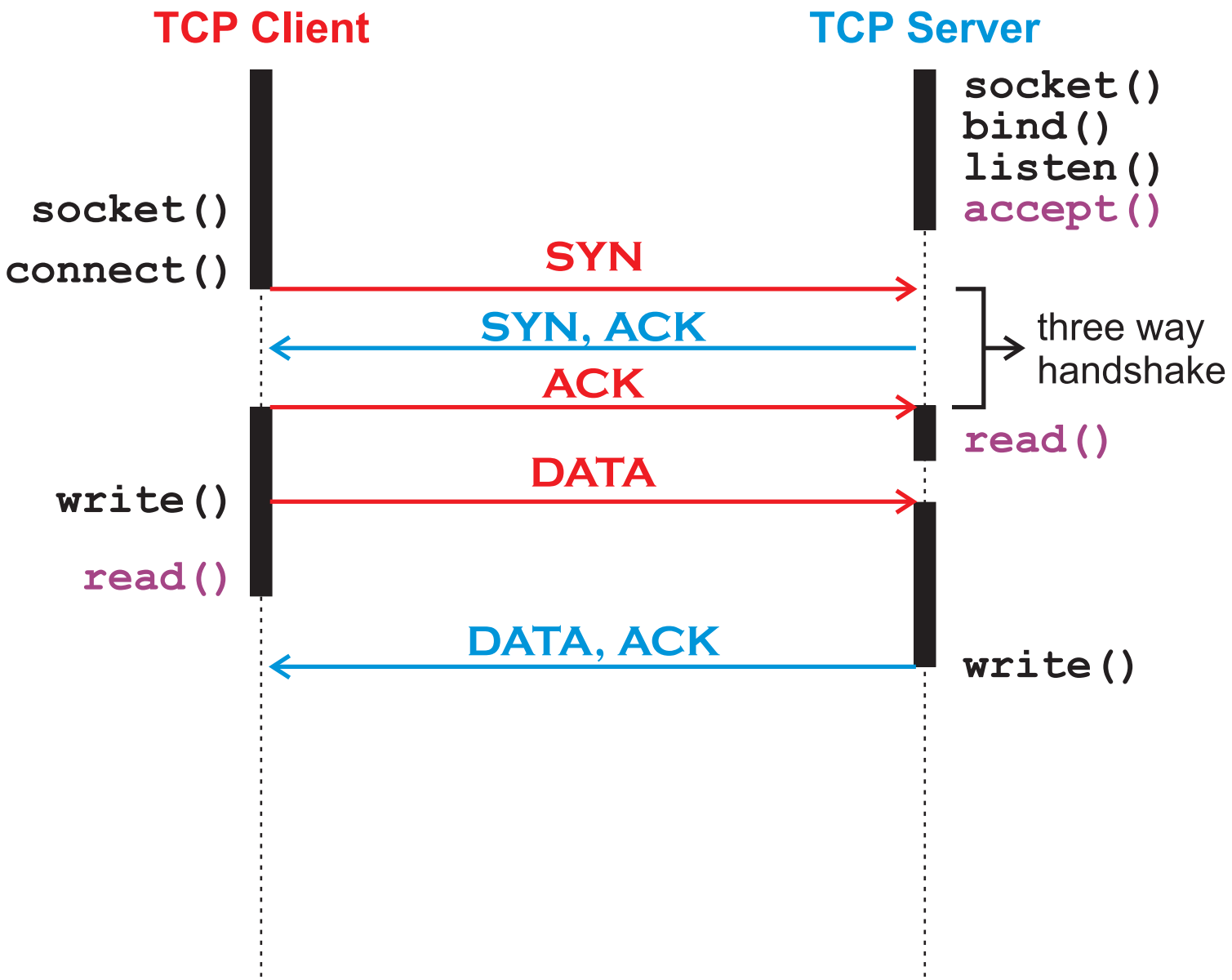

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()

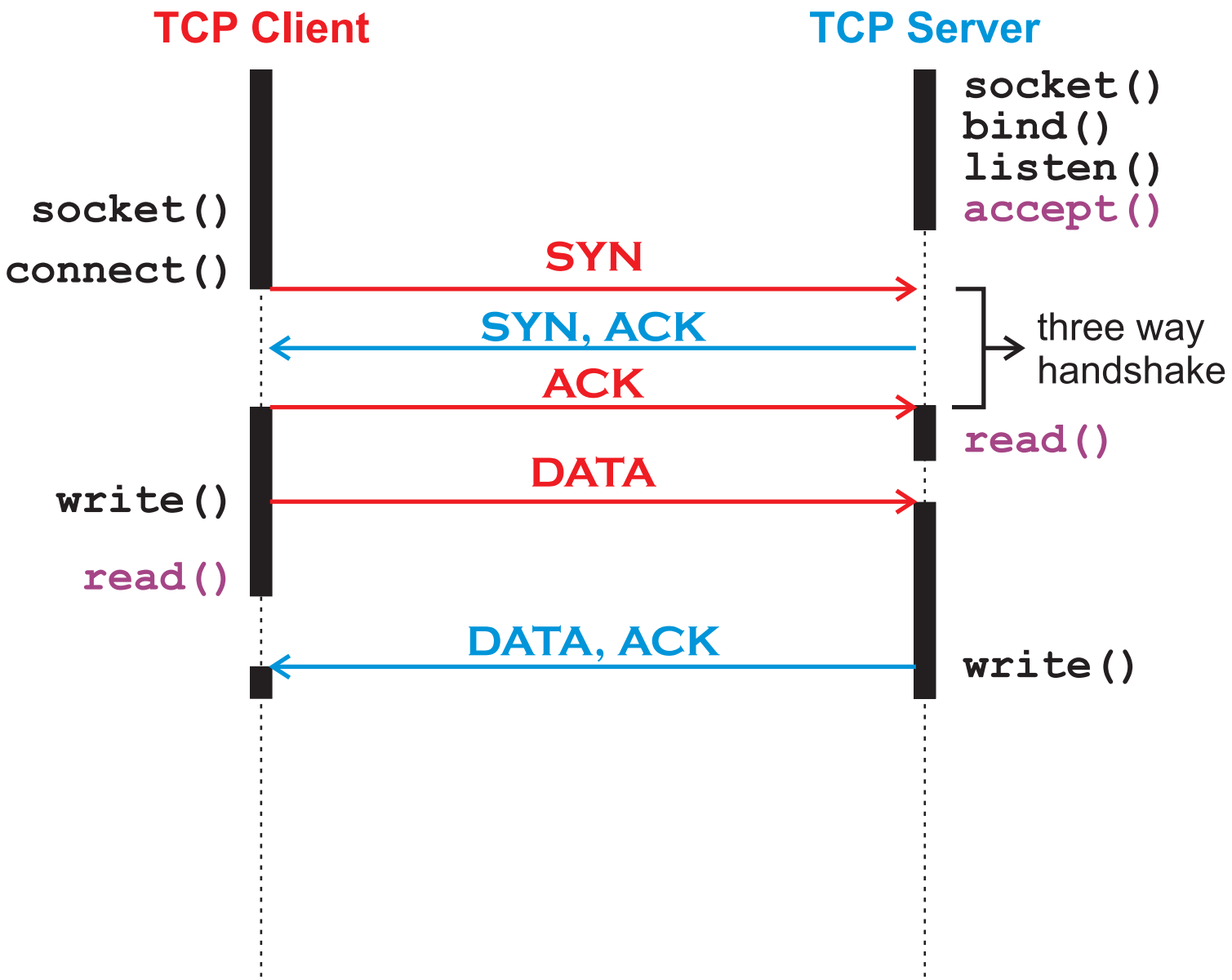

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()

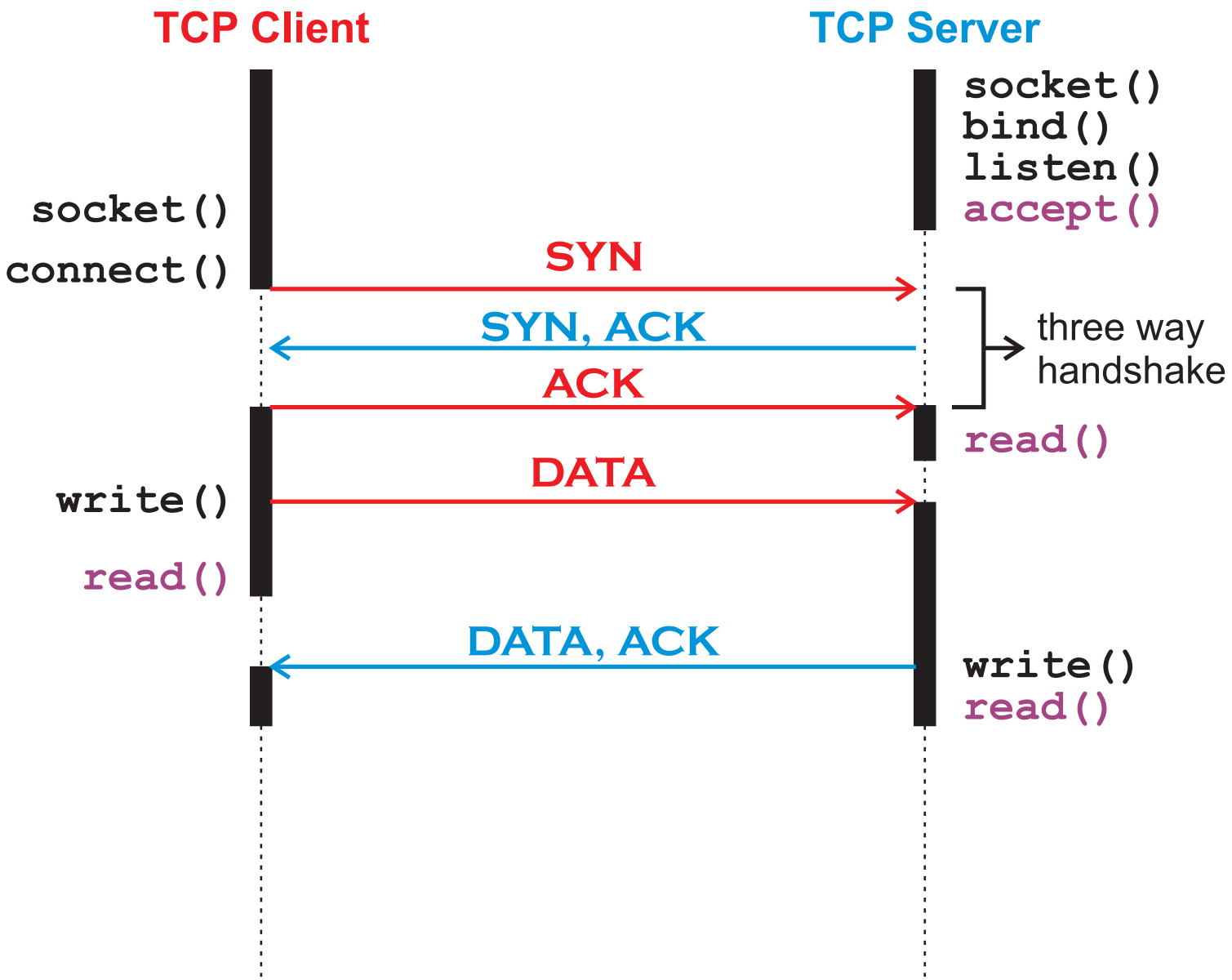

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()

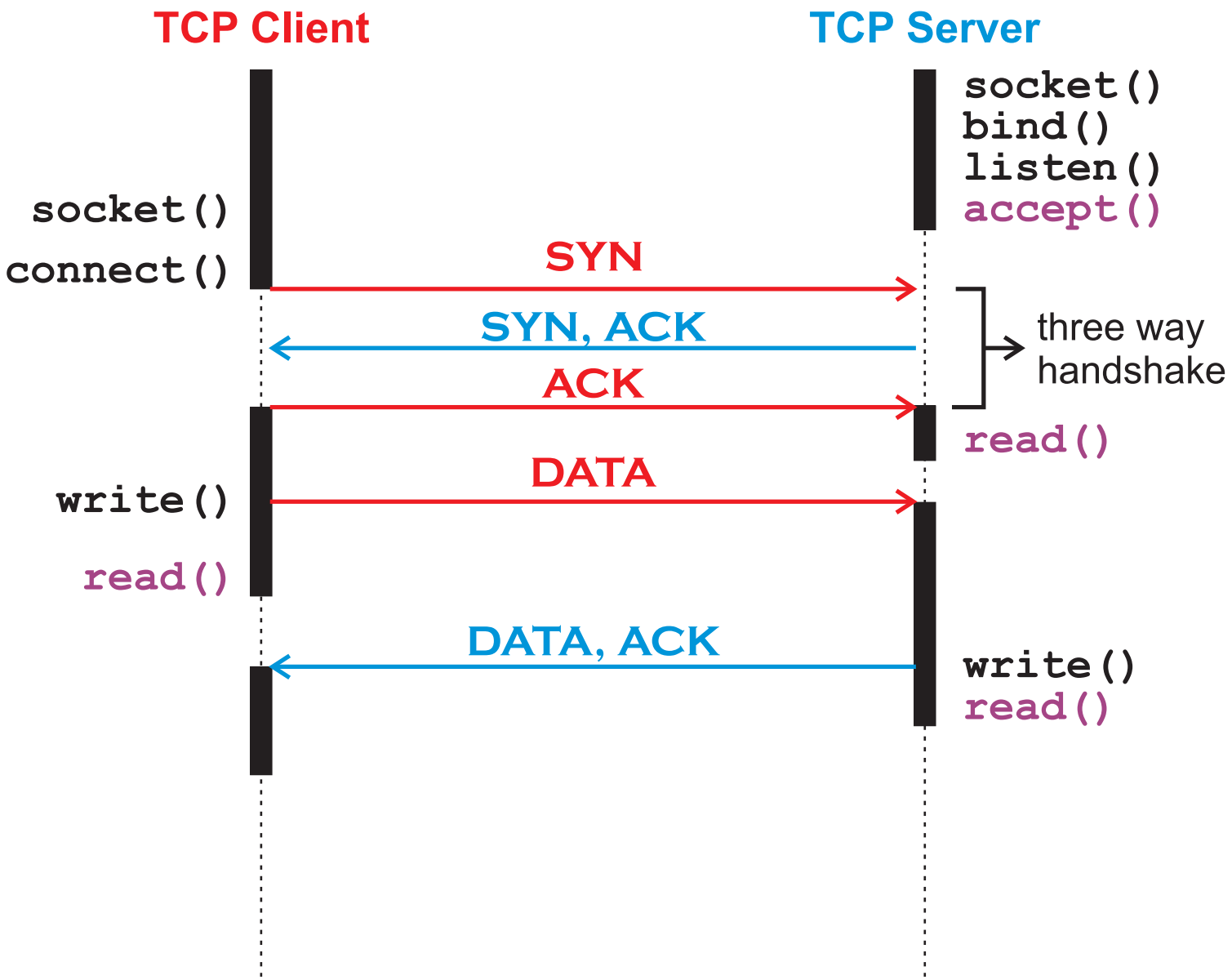

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()

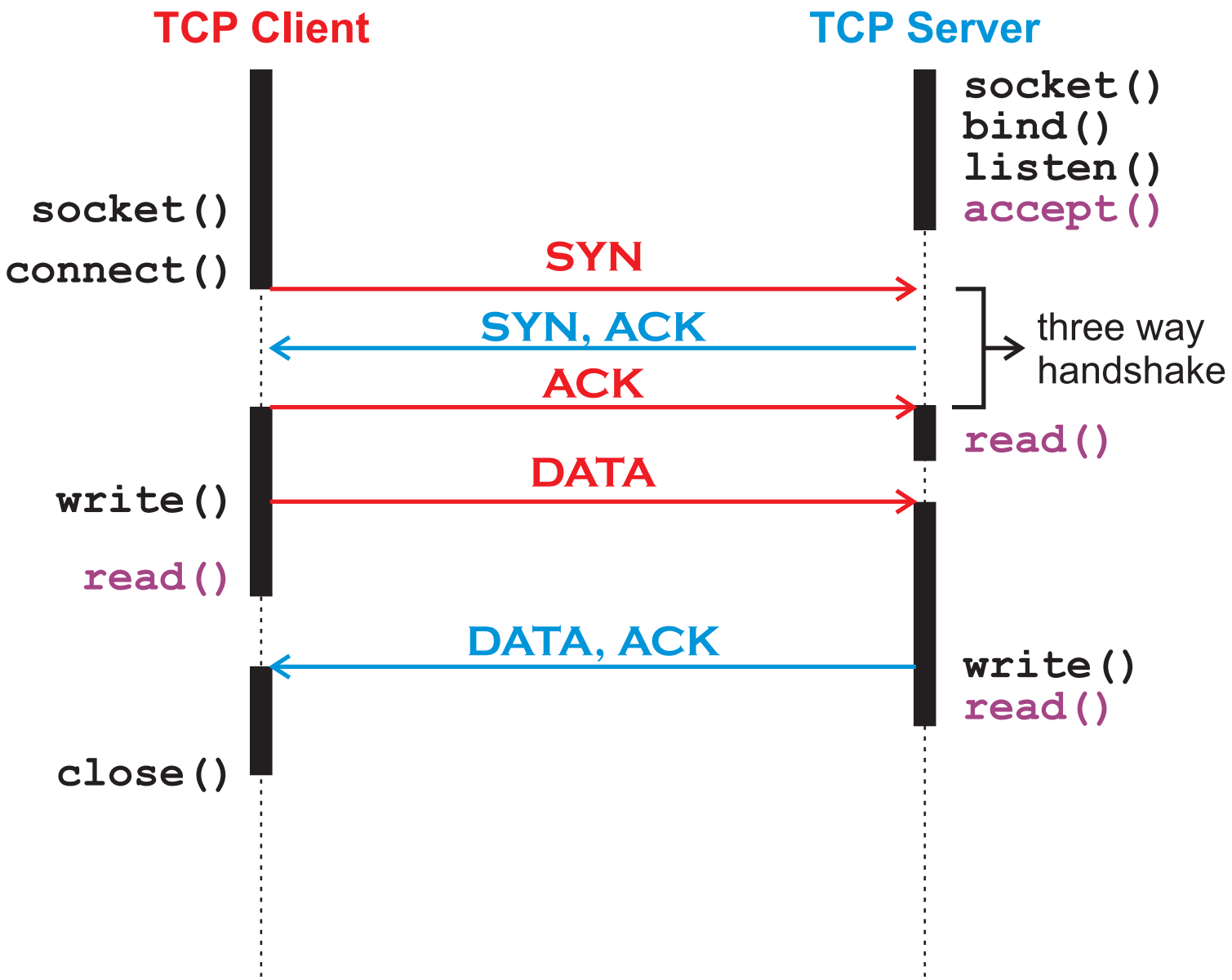

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()
write()

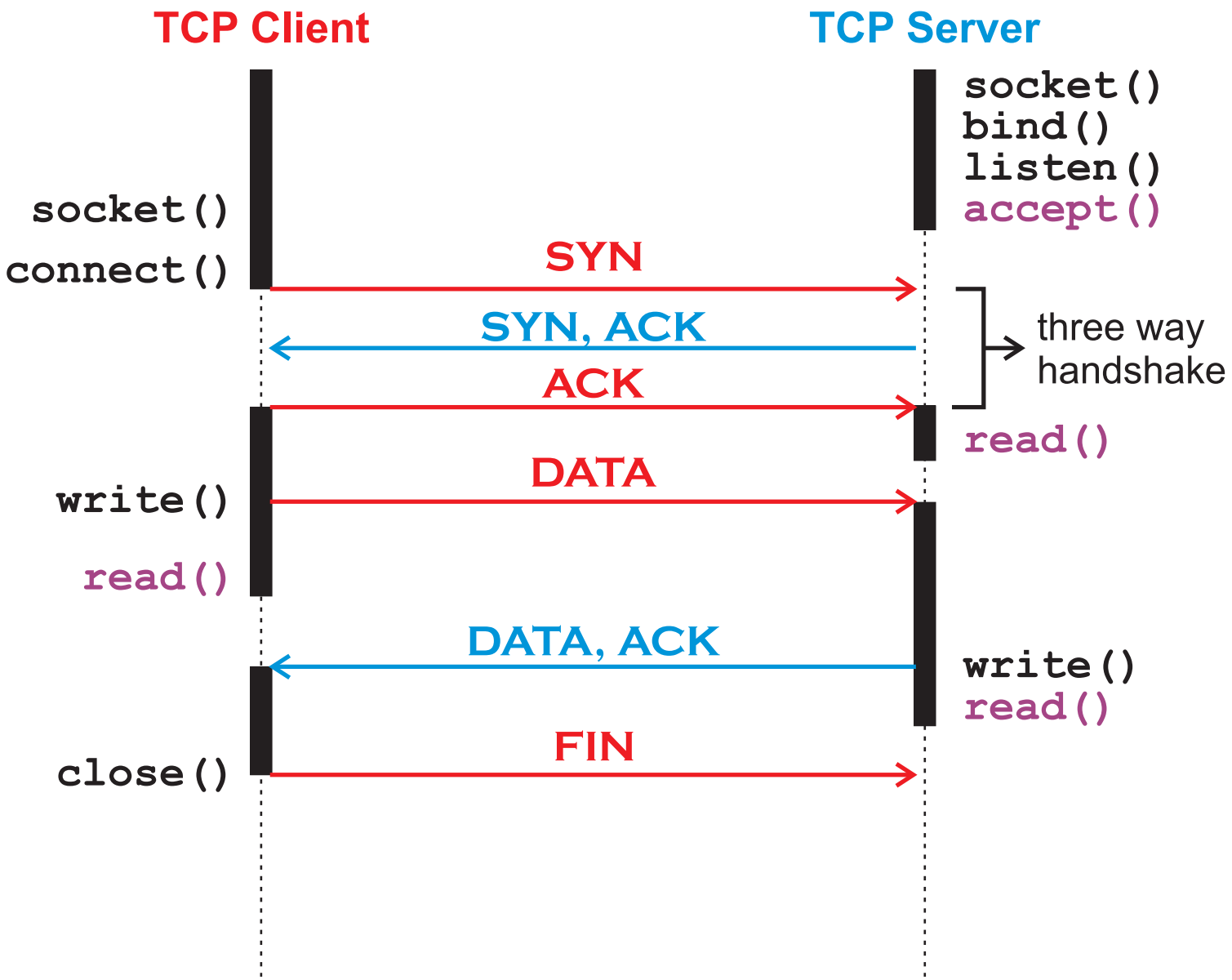

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()
DATA, ACK
write()

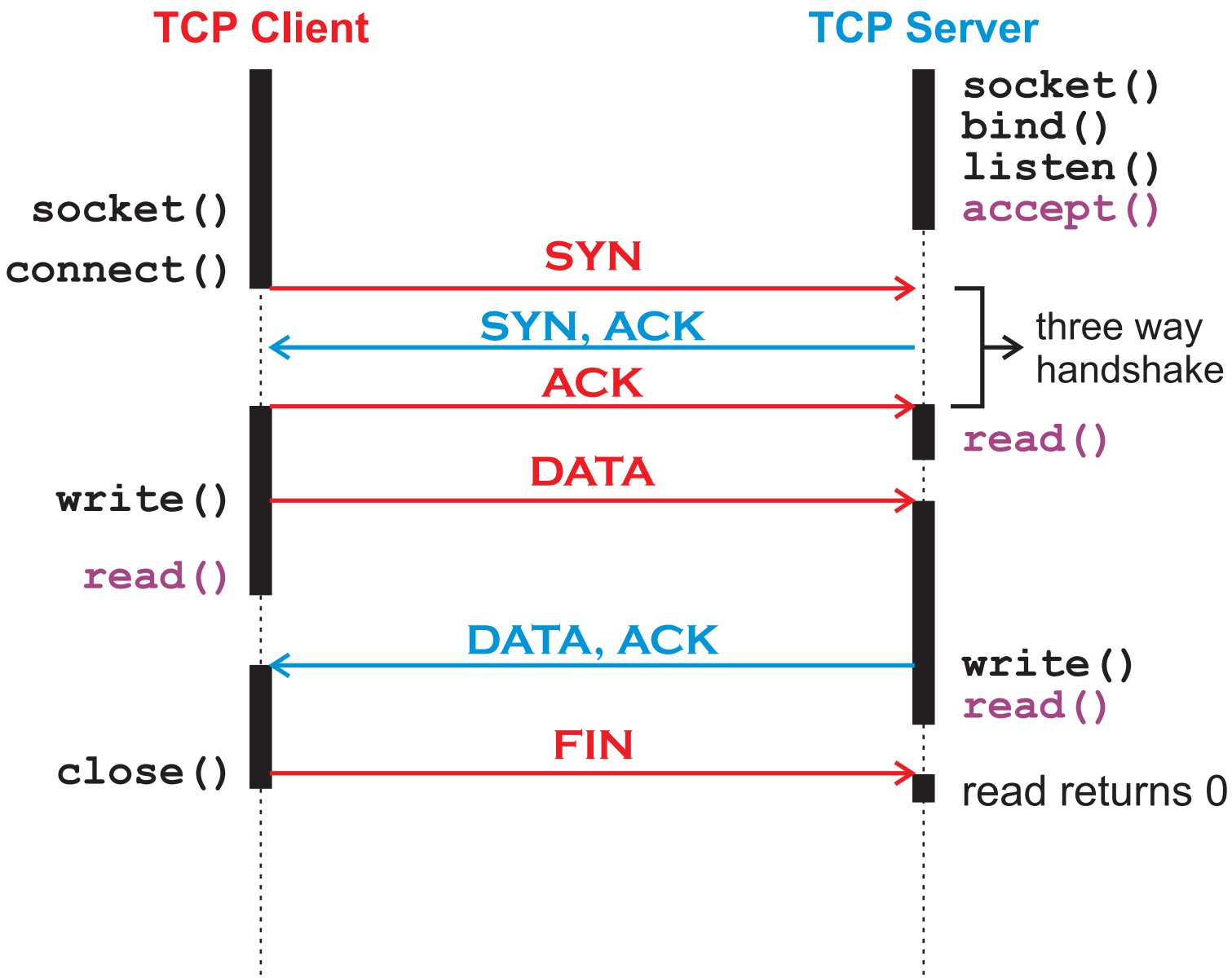

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()
DATA, ACK
write()

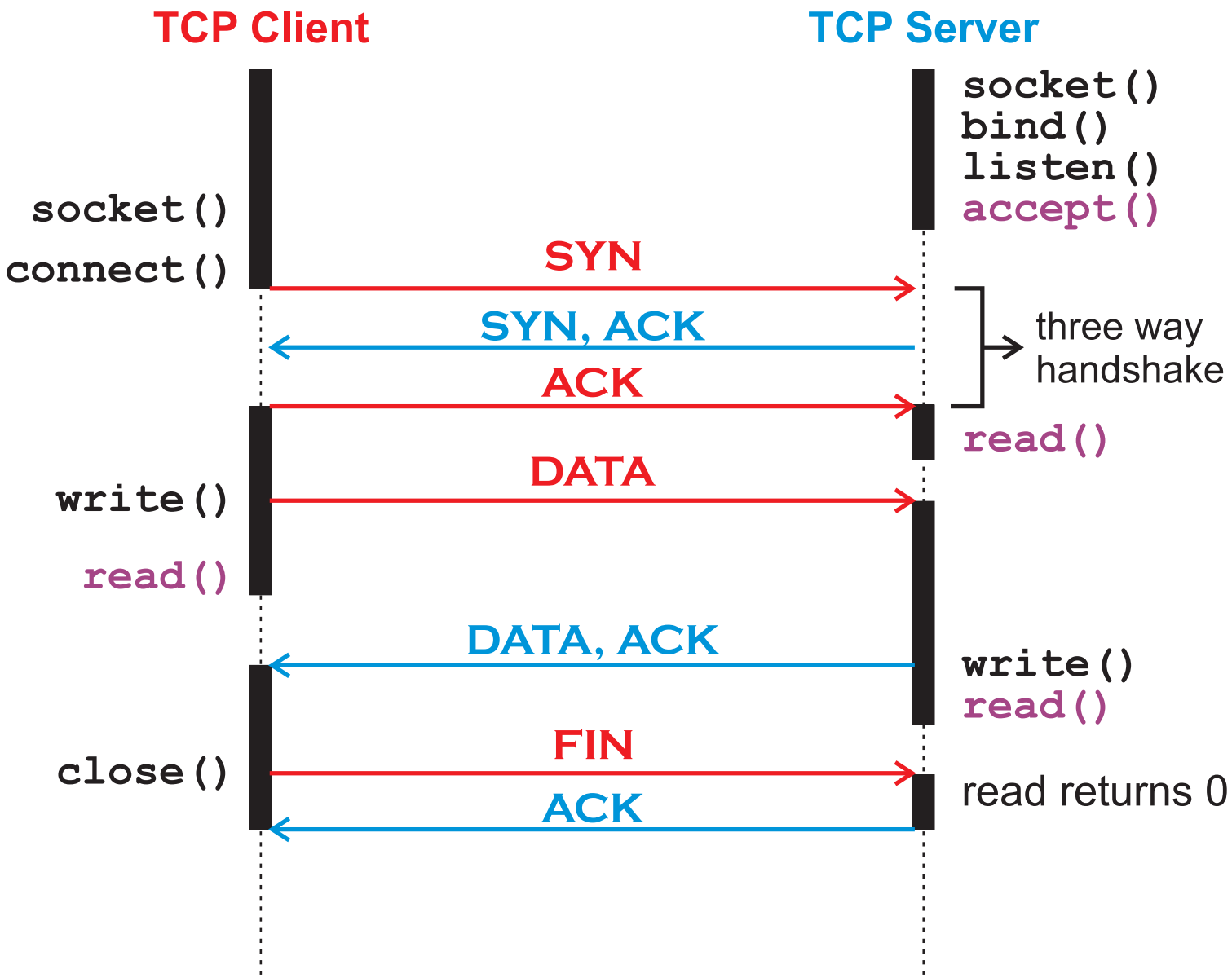

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()
DATA, ACK
write()
read()

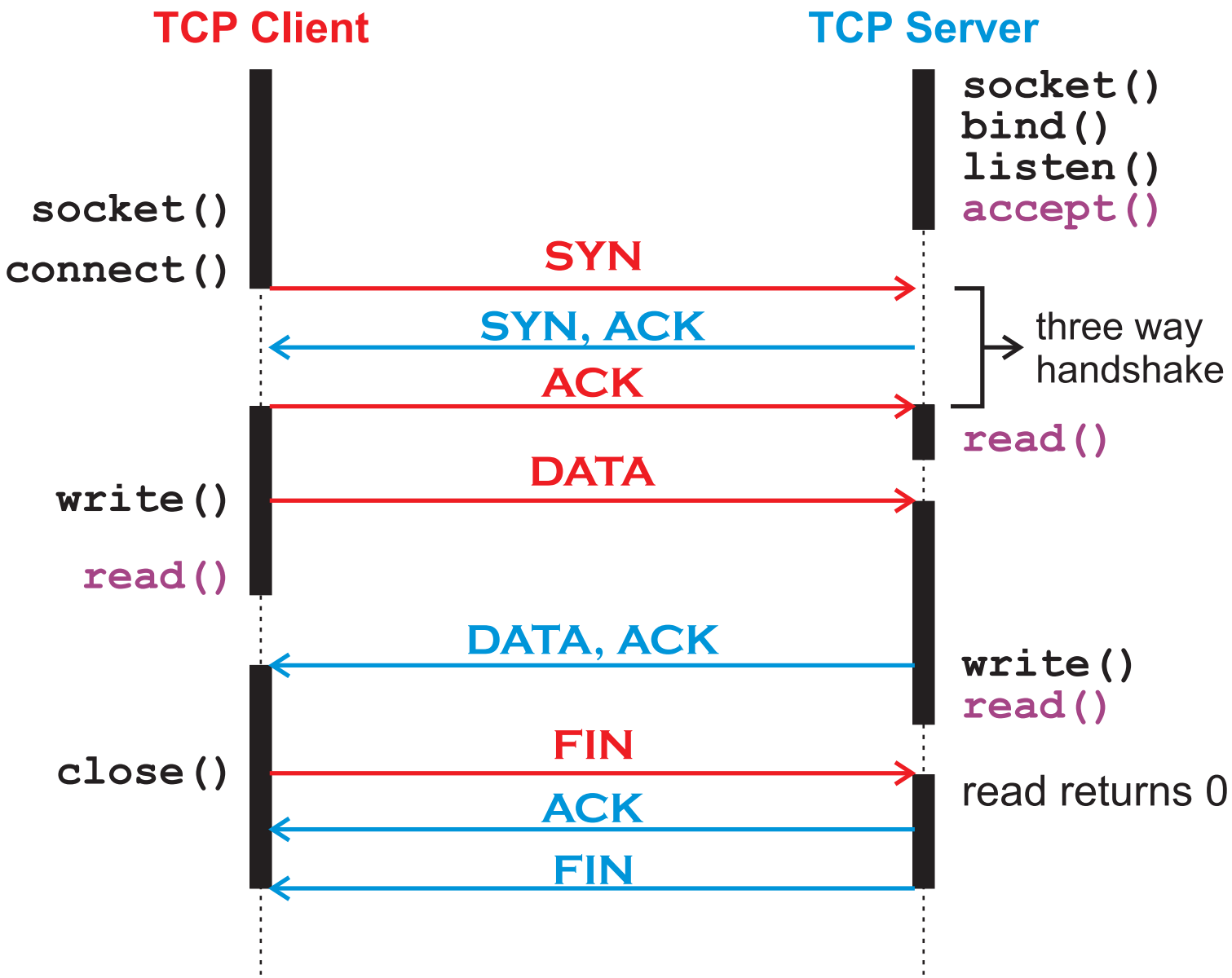

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()
DATA, ACK
write()
read()

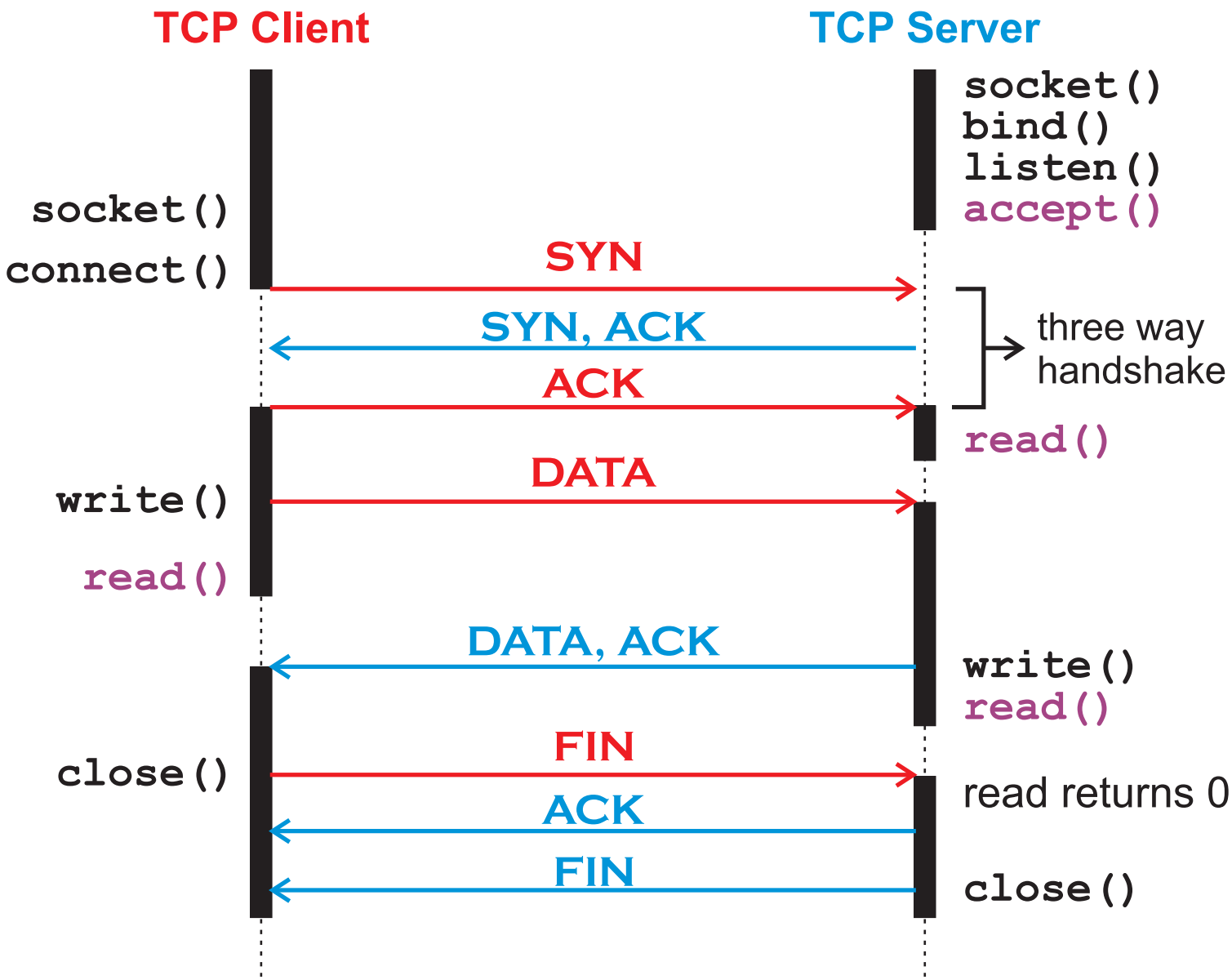

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
three way
handshake
ACK
read()
DATA
write()
read()
DATA, ACK
write()
read()
close()

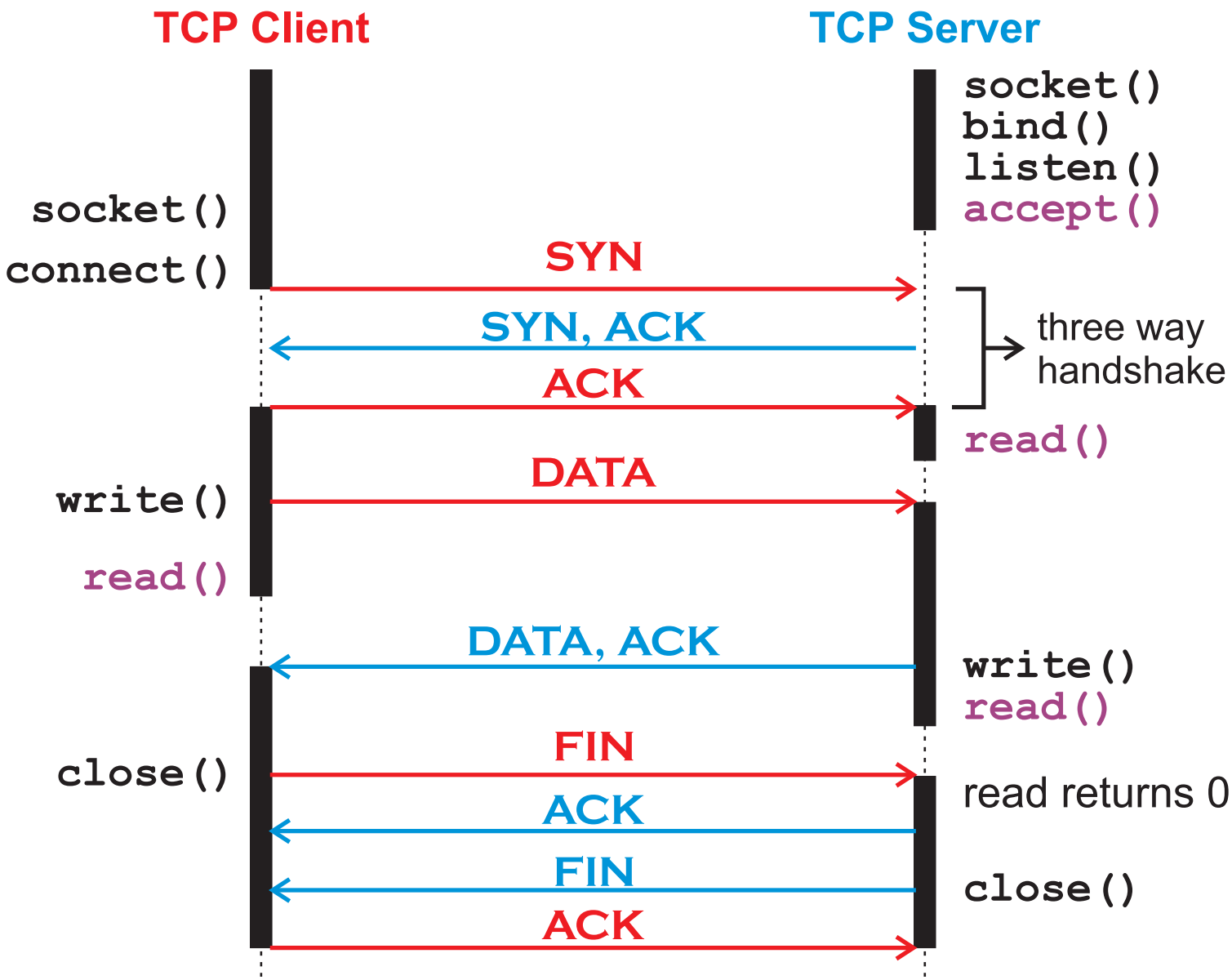

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way
handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()

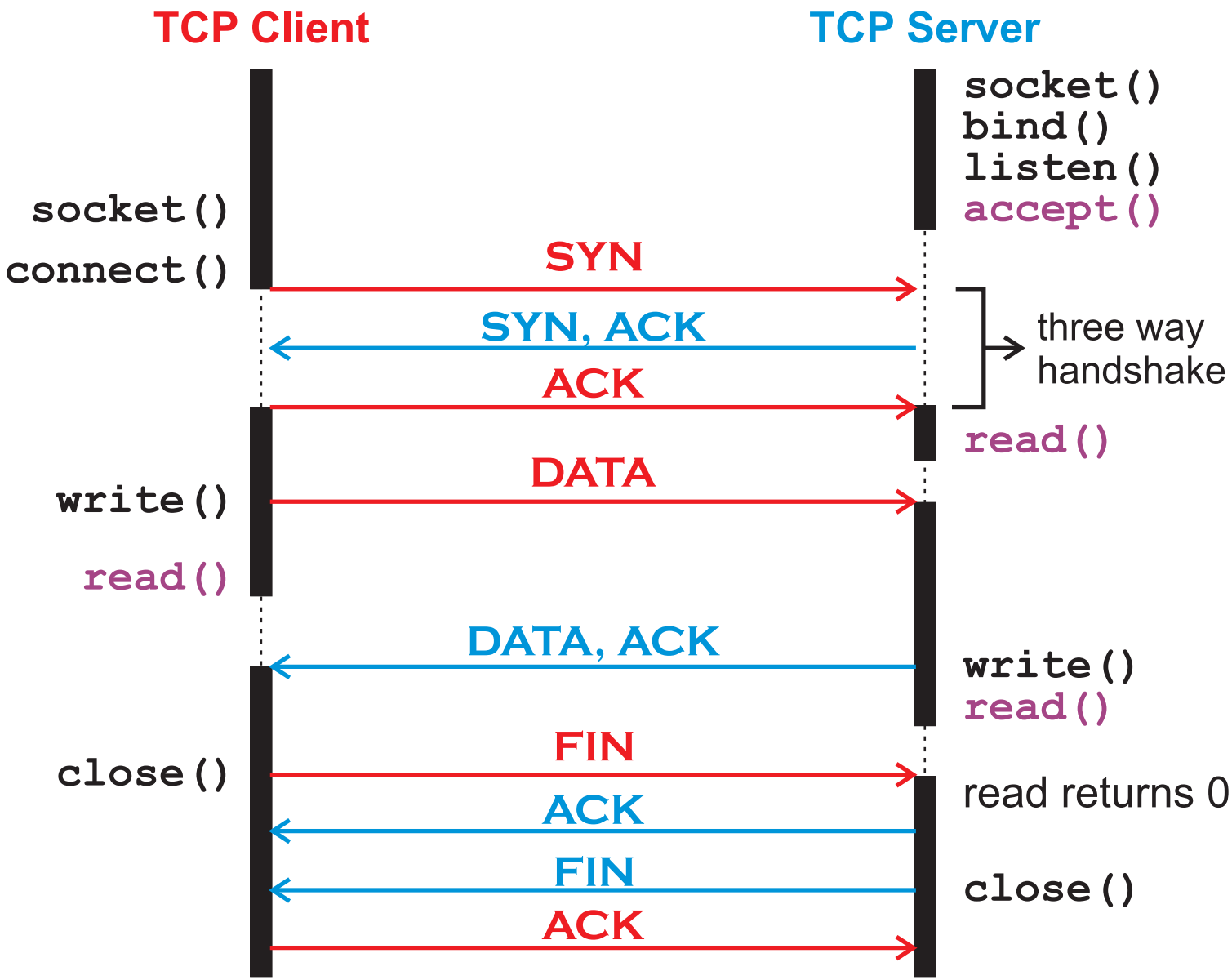

TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way
handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()
read returns 0


TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()
read returns 0
ACK


TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()
read returns 0
ACK
FIN


TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()
read returns 0
ACK
FIN
close()


TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()
read returns 0
ACK
FIN
close()
ACK


TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
SYN
SYN, ACK
ACK
three way handshake
read()
DATA
write()
read()
DATA, ACK
write()
read()
FIN
close()
read returns 0
ACK
FIN
close()
ACK

TCP Sockets in C/Windows: Server

Listing: TCP Server in C/Windows (gcc TCPServer_windows.c -o TCPServer_windows.exe -lws2_32)

```
#include <stdio.h>//compile: gcc TCPServer_windows.c -o TCPServer_windows.exe -lws2_32
#include <winsock.h> //Warning: This program does not perform any error handling.

int main(int argc, char *argv[])  {
  int                server, j, client, addrSize;
  struct sockaddr_in serverAddr, clientAddr;
  WSADATA            wsaData;
  char               data;

  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin_family      = AF_INET; //IPv4 address
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY); //don't care network interface
  serverAddr.sin_port        = htons(9999); //bind to port 9999
  addrSize                   = sizeof(clientAddr);

  WSAStartup(MAKEWORD(2, 0), &wsaData);               //Initialize WinSock
  server = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //Allocate TCP socket
  bind(server, (struct sockaddr *) &serverAddr, sizeof(serverAddr));//1)
  listen(server, 5);  //2)

  for (j = 5; (--j) >= 0;)  {
    client = accept(server, (struct sockaddr *) &clientAddr, &addrSize); //3)
    printf("New connection from %s\n", inet_ntoa(clientAddr.sin_addr));
    // now receive 1 byte of data to client, flags=0
    if(recv(client, &data, 1, 0) == 1) { printf("%d\n", data);  } //4 + 3)
    closesocket(client); //4)
  }
  closesocket(server);  //5)
  WSACleanup(); //Finalize WinSock
}
```

TCP Sockets in C/Windows: Client

Listing: TCP Client in C/Windows (gcc TCPClient_windows.c -o TCPClient_windows.exe -lws2_32)

```c
#include <stdio.h>//compile: gcc TCPClient_windows.c -o TCPClient_windows.exe -lws2_32
#include <winsock.h> //Warning: This program does not perform any error handling.

int main(int argc, char *argv[])  {
  int         client;             struct sockaddr_in address;
  WSADATA     wsaData;            char               data;

  WSAStartup(MAKEWORD(2, 0), &wsaData); //Initialize WinSock
  client = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //Allocate TCP Socket

  memset(&address, 0, sizeof(address)); //clear socket address
  address.sin_family      = AF_INET;  //IPv4 address
  address.sin_addr.s_addr = inet_addr("127.0.0.1");//set to (loopback) IP address
  address.sin_port        = htons(9999); //make port in network byte order

  connect(client, (struct sockaddr *)&address, sizeof(address)); //1+2)

  data = 2;
  send(client, &data, 1, 0);  //3) send 1 byte of data to client, flags=0

  closesocket(client);  //4)
  WSACleanup();  //Finalize WinSock
  return 0;
  }
```

TCP Sockets in C/Linux: Server

Listing: TCP Server in C/Linux (gcc TCPServer_linux.c -o TCPServer_linux -lpthread)

```
#include <stdio.h>      //compile: gcc TCPServer_linux.c -o TCPServer_linux
#include <sys/socket.h> //Warning: This program does not perform any error handling.
#include <netinet/in.h> //In any real program, you need to handle errors.
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])  {
  int                server, j, client;
  socklen_t          addrSize;
  struct sockaddr_in serverAddr, clientAddr;
  char               data;

  memset(&serverAddr, 0, sizeof(serverAddr));//clear socket address
  serverAddr.sin_family      = AF_INET;      //IPv4 address
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);//don't care network interface
  serverAddr.sin_port        = htons(9999); //bind to port 9999
  addrSize                   = sizeof(clientAddr);

  server = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //Allocate TCP socket
  bind(server, (struct sockaddr *) &serverAddr, sizeof(serverAddr)); //1)
  listen(server, 5);  //2)

  for (j = 5; (--j) >= 0;)  {
    client = accept(server, (struct sockaddr *) &clientAddr, &addrSize); //3)
    printf("New connection from %s\n", inet_ntoa(clientAddr.sin_addr));
    // now receive 1 byte of data to client, flags=0
    if(recv(client, &data, 1, 0) == 1) {   printf("%d\n", data);  } //4 + 3)
    close(client); //4)
  }
  close(server); //5)
}
```

TCP Sockets in C/Linux: Client

Listing: TCP Client in C/Linux (gcc TCPClient_linux.c -o TCPClient_linux -lpthread)

```c
#include <stdio.h>      //compile: gcc TCPClient_linux.c -o TCPClient_linux
#include <sys/socket.h> //Warning: This program does not perform any error handling.
#include <arpa/inet.h>  //In any real program, you need to handle errors.
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])  {
  int  client;  struct sockaddr_in address;   char data;

  client = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //Allocate TCP Socket

  memset(&address, 0, sizeof(address)); //clear socket address
  address.sin_family      = AF_INET; //IPv4 address
  address.sin_addr.s_addr = inet_addr("127.0.0.1");//set to (loopback) IP address
  address.sin_port        = htons(9999);  //make port in network byte order

  connect(client, (struct sockaddr *)&address, sizeof(address)); //1+2)

  data = 2;
  send(client, &data, 1, 0); //3) send 1 byte of data to client, flags=0

  close(client);  //4)
  return 0;
  }
```

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)
  5) close server socket when finished with everything

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)
  5) close server socket when finished with everything
- **Client socket**

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)
  5) close server socket when finished with everything
- **Client socket**
  1) client socket bound to random free port

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)
  5) close server socket when finished with everything
- **Client socket**
  1) client socket bound to random free port
  2) send packet to server

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)
  5) close server socket when finished with everything
- **Client socket**
  1) client socket bound to random free port
  2) send packet to server
  3) maybe receive packets from server

UDP Sockets

- UDP [19] is an unreliable and connection-less protocol
- Server sockets therefore wait for incomming packets (which may come from anywhere) instead of connection
- Clients try to send packets to server without establishing connections
- **Server socket**
  1) is bound to a specific (usually well-known) port
  2) accepts packets at that port from clients
  3) processes packet data on arrival
  4) maybe also send answer packet (go to 2)
  5) close server socket when finished with everything
- **Client socket**
  1) client socket bound to random free port
  2) send packet to server
  3) maybe receive packets from server
  4) close the socket

UDP Sockets in Java: Server

Listing: UDPServer.java UDP Server in Java

```
import java.io.OutputStream;       import java.net.DatagramPacket;
import java.net.DatagramSocket;    import java.net.InetAddress;

public class UDPServer {
  public static final void main(final String[] args) {
    DatagramSocket  server;     DatagramPacket  p;

    try {
      server = new DatagramSocket(9998); //create socket 1)

      for(int j = 5; (--j) >= 0; ){
        p = new DatagramPacket(new byte[1], 1); //create package
        server.receive(p); //wait for and receive incoming data 2)

        System.out.println("New message " + p.getSocketAddress());
        if (p.getLength() > 0) { //is there data? 3)
          System.out.println(p.getData()[0]);  //3)
        }
      }
      server.close();  //5)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

UDP Sockets in Java: Client

Listing: UDPClient.java UDP Client in Java

```
import java.io.OutputStream;          import java.net.DatagramPacket;
import java.net.DatagramSocket;       import java.net.InetAddress;

public class UDPClient {

  public static final void main(final String[] args) {
    DatagramSocket client;       InetAddress    ia;
    DatagramPacket p;            byte[]         data;

    try {
      ia     = InetAddress.getByName("localhost");

      client = new DatagramSocket(); //create socket 1)

      data   = new byte[] { 1 }; //allocate data for package
      p      = new DatagramPacket(data, 1, ia, 9998); //create package
      client.send(p); //send package to localhost:9998 2)

      client.close(); //dispose socket 4)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

UDP Server Socket (Java Try-With-Resource)

Listing: UDPServerJava17.java UDP Server in Java

```
import java.io.OutputStream;       import java.net.DatagramPacket;
import java.net.DatagramSocket;    import java.net.InetAddress;

public class UDPServerJava17 {
  public static final void main(final String[] args) {
    DatagramPacket  p;

    try(DatagramSocket server = new DatagramSocket(9998)) { //1)
      for(int j = 5; (--j) >= 0; ){ //only five times...
        p = new DatagramPacket(new byte[1], 1); //create package
        server.receive(p); //wait for and receive package 2)

        System.out.println("New message " + p.getSocketAddress());
        if (p.getLength() > 0) { //is there data? 3)
          System.out.println(p.getData()[0]);  //3)
        }
      }
    //5)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

UDP Client Socket (Java Try-With-Resource)

Listing: UDPClientJava17.java UDP Client in Java

```
import java.io.OutputStream;          import java.net.DatagramPacket;
import java.net.DatagramSocket;       import java.net.InetAddress;

public class UDPClientJava17 {

  public static final void main(final String[] args) {
    InetAddress  ia;         DatagramPacket p;         byte[] data;

    try {
      ia = InetAddress.getByName("localhost"); //get local host address

      try(DatagramSocket client = new DatagramSocket()) {  //1)
        data   = new byte[] { 1 }; //allocate data
        p      = new DatagramPacket(data, 1, ia, 9998); //create package
        client.send(p); //send package to localhost:9998 2)
      } //4)
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

UDP Sockets in C/Windows: Server

Listing: UDP Server in C/Windows (gcc UDPServer_windows.c -o UDPServer_windows.exe -lws2_32)

```
#include <stdio.h>//compile: gcc UDPServer_windows.c -o UDPServer_windows.exe -lws2_32
#include <winsock.h> //Warning: This program does not perform any error handling.

int main(int argc, char *argv[])  {
  int                server, j, addrSize;
  struct sockaddr_in serverAddr, clientAddr;
  WSADATA            wsaData;
  char               data;

  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin_family      = AF_INET;          //IPv4 address
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);//don't care network interface
  serverAddr.sin_port        = htons(9998);      //set port 9998
  addrSize                   = sizeof(clientAddr);

  WSAStartup(MAKEWORD(2, 0), &wsaData);             //Initialize WinSock
  server = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);//Allocate UDP socket
  bind(server, (struct sockaddr *) &serverAddr, sizeof(serverAddr)); //1)

  for (j = 5; (--j) >= 0;)  { // then receive 1 byte package data and get client
      address, with flags=0
    recvfrom(server, &data, 1, 0, (struct sockaddr *) &clientAddr, &addrSize); //2)
    printf("New message %d from %s\n", data, inet_ntoa(clientAddr.sin_addr));  //3)
  }
  closesocket(server); //5)
  WSACleanup();        //Finalize WinSock
}
```

UDP Sockets in C/Windows: Client

Listing: UDP Client in C/Windows (gcc UDPClient_windows.c -o UDPClient_windows.exe -lws2_32)

```
#include <stdio.h>//compile: gcc UDPClient_windows.c -o UDPClient_windows.exe -lws2_32
#include <winsock.h> //Warning: This program does not perform any error handling.

int main(int argc, char *argv[])  {
  int         client;             struct sockaddr_in address;
  WSADATA     wsaData;            char               data;

  WSAStartup(MAKEWORD(2, 0), &wsaData);  //Initialize WinSock
  client = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP); //Allocate client socket

  memset(&address, 0, sizeof(address)); //Clear socket address
  address.sin_family      = AF_INET;    //IPv4 address
  address.sin_addr.s_addr = inet_addr("127.0.0.1");//Set to (loopback) IP address
  address.sin_port        = htons(9998); //Make port in network byte order

  data = 2; // then send 1 byte package data to client, with flags=0
  sendto(client, &data, 1, 0, (struct sockaddr *)&address, sizeof(address)); //1+2)

  closesocket(client); //4)
  WSACleanup(); //Finalize Winsock
  return 0;
  }
```

UDP Sockets in C/Linux: Server

Listing: UDP Server in C/Linux (gcc UDPServer_linux.c -o UDPServer_linux -lpthread)

```c
#include <stdio.h>      //compile: gcc UDPServer_linux.c -o UDPServer_linux
#include <string.h>     //Warning: This program does not perform any error handling.
#include <sys/socket.h> //In any real program, you need to handle errors.
#include <arpa/inet.h>
#include <unistd.h>

int main(int argc, char *argv[])  {
  int                server, j;
  socklen_t          addrSize;
  struct sockaddr_in serverAddr, clientAddr;
  char               data;

  memset(&serverAddr, 0, sizeof(serverAddr));    //Clear address struct
  serverAddr.sin_family      = AF_INET;          //IPv4 address
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);//don't care network interface
  serverAddr.sin_port        = htons(9998);      //serve at port 9998
  addrSize                   = sizeof(clientAddr);

  server = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP); //Allocate UDP socket
  bind(server, (struct sockaddr *) &serverAddr, sizeof(serverAddr)); //1)

  for (j = 5; (--j) >= 0;)  { // then receive 1 byte package data and get client
      address, with flags=0
    recvfrom(server, &data, 1, 0, (struct sockaddr *) &clientAddr, &addrSize); //2)
    printf("New message %d from %s\n", data, inet_ntoa(clientAddr.sin_addr));  //3)
  }
  close(server); //5)
}
```

UDP Sockets in C/Linux: Client

Listing: UDP Client in C/Linux (gcc UDPClient_linux.c -o UDPClient_linux -lpthread)

```c
#include <stdio.h>      //compile: gcc UDPClient_linux.c -o UDPClient_linux
#include <string.h>     //Warning: This program does not perform any error handling.
#include <sys/socket.h> //In any real program, you need to handle errors.
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])  {
  int client;  struct sockaddr_in address;    char data;

  client = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP); //Allocate client socket

  memset(&address, 0, sizeof(address));  //Clear socket address
  address.sin_family      = AF_INET;     //IPv4 address
  address.sin_addr.s_addr = inet_addr("127.0.0.1");//Set to (loopback) IP address
  address.sin_port        = htons(9998); //Make port in network byte order

  data = 2; // then send 1 byte package data to client, with flags=0
  sendto(client, &data, 1, 0, (struct sockaddr *)&address, sizeof(address)); //1+2)

  close(client);  //4)
  return 0;
  }
```

Basic Use Summary

- We can now send bytes from one host to another
- We have seen how this is done both in `Java` and `C`
- The example clients and servers written in different languages can communicate with each other $\Longrightarrow$ Distribution allows us to construct heterogeneous systems
- By either using TCP (connection-oriented) or UDP (connection-free)
- However...
  - What if we want to send more complex stuff? `int`s? `double`s? Objects? Text?
  - What do we do if more than one client connects to a server at a time?
  - We do not have seen some more complex examples
- We will now look at these issues in the next lessons

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://www.it-weise.de

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. *Standard for Information Technology – Portable Operating System Interface (POSIX)*, volume 1003.1,2004. Piscataway, NJ, USA: IEEE (Institute of Electrical and Electronics Engineers), 2004.
2. Herbert Schildt. *Java 2: A Beginner's Guide*. Essential Skills for First-Time Programmers. Maidenhead, England, UK: McGraw-Hill Ltd., 2002. ISBN 0072225130 and 9780072225136. URL `http://books.google.de/books?id=YWDJJGYaLG4C`.
3. Learning java, 2007. URL `http://en.wikiversity.org/wiki/Learning_JAVA`.
4. Robert Sedgewick. *Algorithms in Java, Parts 1–4 (Fundamentals: Data Structures, Sorting, Searching)*. Reading, MA, USA: Addison-Wesley Professional, 3rd edition, September 2002. ISBN 0-201-36120-5 and 978-0-201-36120-9. URL `http://books.google.de/books?id=hyvdUQUmf2UC`. With Java consultation by Michael Schidlowsky.
5. Zbigniew Michael Sikora. *Java: Practical Guide for Programmers*. Morgan Kaufmann Practical Guides. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558609091 and 9781558609099. URL `http://books.google.de/books?id=YQLj_AsVN9QC`.
6. Santa Clara, CA, USA: Sun Microsystems, Inc. *Java™ 2 Platform Standard Edition 5.0 – API Specification*, October 19, 2010.
7. James Gosling, William Nelson Joy, Guy Lewis Steele Jr., and Gilad Bracha. *The Java™ Language Specification*. The Java Series. Upper Saddle River, NJ, USA: Prentice Hall International Inc., Santa Clara, CA, USA: Sun Microsystems Press (SMP), and Reading, MA, USA: Addison-Wesley Professional, 3rd edition, May 2005. ISBN 0-321-24678-0 and 978-0321246783. URL `http://java.sun.com/docs/books/jls/`.
8. James Gosling and Henry McGilton. The java language environment – a white paper. Technical report, Santa Clara, CA, USA: Sun Microsystems, Inc., May 1996. URL `http://java.sun.com/docs/white/langenv/`.
9. Guido Krüger. *Handbuch der Java-Programmierung*. 4. aktualisierte edition. ISBN 3-8273-2361-4 and 3-8273-2447-5. URL `http://www.javabuch.de/`.
10. Christian Ullenboom. *Java ist auch eine Insel – Programmieren mit der Java Standard Edition Version 6*. Bonn, North Rhine-Westphalia, Germany: Galileo-Press, 6. aktualisierte und erweiterte edition, 2007. ISBN 3-89842-838-9 and 978-3-89842-838-5. URL `http://www.galileocomputing.de/openbook/javainsel6/`.
11. William Crawford and Jonathan Kaplan. *J2EE Design Patterns*. Patterns of the Real World. Sebastopol, CA, USA: O'Reilly Media, Inc., 2003. ISBN 0596004273 and 9780596004279. URL `http://books.google.de/books?id=x-7_W0P9KGsC`.
12. Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Upper Saddle River, NJ, USA: Pearson Education, 2005. ISBN 0672334046 and 9780672334047. URL `http://books.google.de/books?id=8fXCn3E864sC`.

Bibliography II

13. Herbert Schildt. *C++: A Beginner's Guide*. Essential Skills for First-Time Programmers. Maidenhead, England, UK: McGraw-Hill Ltd., 2002. ISBN 0072194677 and 9780072194678. URL http://books.google.de/books?id=W0siAQAAIAAJ.
14. Randal Albert and Todd Breedlove. *C++: An Active Learning Approach*. Sudbury, MA, USA: Jones & Bartlett Learning, 2008. ISBN 0763757233 and 9780763757236. URL http://books.google.de/books?id=VwOr2hFIaZoC.
15. Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Handbook*. C++ Programming Languages. Reading, MA, USA: Addison-Wesley Professional, 1999. ISBN 0201379260 and 9780201379266. URL http://books.google.de/books?id=n9VEG2Gp5pkC.
16. David Makofske, Michael J. Donahoo, and Kenneth L. Calvert. *TCP/IP Sockets in C#: Practical Guide for Programmers*. Morgan Kaufmann Practical Guides. Essex, UK: Elsevier Science Publishers B.V., 2004. ISBN 0080492320 and 9780080492322. URL http://books.google.de/books?id=YQQXHEj6O4QC.
17. Chandrta Chandrasekar. Sockets in c#, October 29, 2003. URL http://www.codeproject.com/Articles/5252/Sockets-in-C.
18. Alex Martelli. *Python in a Nutshell*. Nutshell Series. Sebastopol, CA, USA: O'Reilly Media, Inc., 2006. ISBN 0596100469 and 9780596100469. URL http://books.google.de/books?id=JnR9hQA3SncC.
19. Charles M. Kozierok. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. San Francisco, CA, USA: No Starch Press, 2005. ISBN 159327047X and 9781593270476. URL http://books.google.de/books?id=Pm4RgYV2w4YC.
20. Douglas Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Upper Saddle River, NJ, USA: Prentice Hall International Inc., 2006. ISBN 0131876716 and 9780131876712. URL http://books.google.de/books?id=jonyuTASbWAC.
21. Lesson: All about sockets, 2009. URL http://docs.oracle.com/javase/tutorial/networking/sockets/.
22. Kenneth L. Calvert and Michael J. Donahoo. *TCP/IP Sockets in Java: Practical Guide for Programmers*. Morgan Kaufmann Practical Guides. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123742552 and 9780123742551. URL http://books.google.de/books?id=lfHo7uMk7r4C.
23. Merlin Hughes, Michael Shoffner, and Derek Hamner. *Java Network Programming: A Complete Guide to Networking, Streams, and Distributed Computing*. Manning Pubs Co. Greenwich, CT, USA: Manning Publications Co., 1999. ISBN 188477749X and 9781884777493. URL http://books.google.de/books?id=xapQAAAAMAAJ.
24. The java tutorials: The try-with-resources statement, March 1, 2013. URL http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

Bibliography III

25. Michael J. Donahoo and Kenneth L. Calvert. *TCP/IP Sockets in C: Practical Guide for Programmers*. Morgan Kaufmann Practical Guides. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd edition, 2009. ISBN 0123745403 and 9780123745408. URL `http://cs.baylor.edu/~donahoo/practical/CSockets/`.