Interprocess Communication

Grado en Ingeniería Informática

Pablo García Sánchez

Departamento de Ingeniería Informática
Universidad de Cádiz
Pablo García Sánchez
Based on the work by Guadalupe Ortiz, Mei Ling-Liu, Marteen Van Steem and A. Tanenbaum

Curso 2017 – 2018

Indice

Section 1 | Interprocess Communications

Interprocess Communications

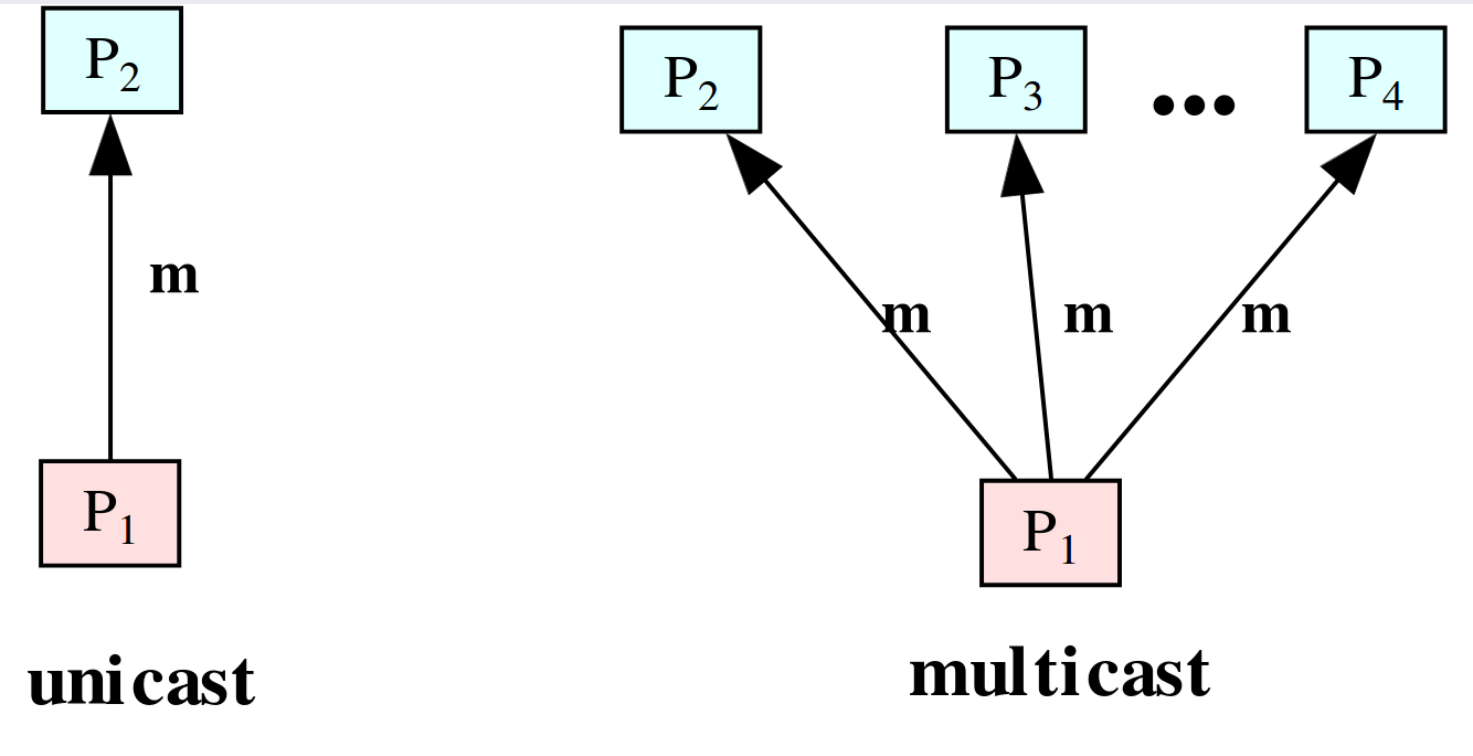
- Operating systems provide facilities for interprocess communications (IPC), such as message queues, semaphores, and shared memory.
- Distributed computing systems make use of these facilities to provide application programming interface which allows IPC to be programmed at a higher level of abstraction.
- Distributed computing requires information to be exchanged among independent processes.

Unicast and Multicast

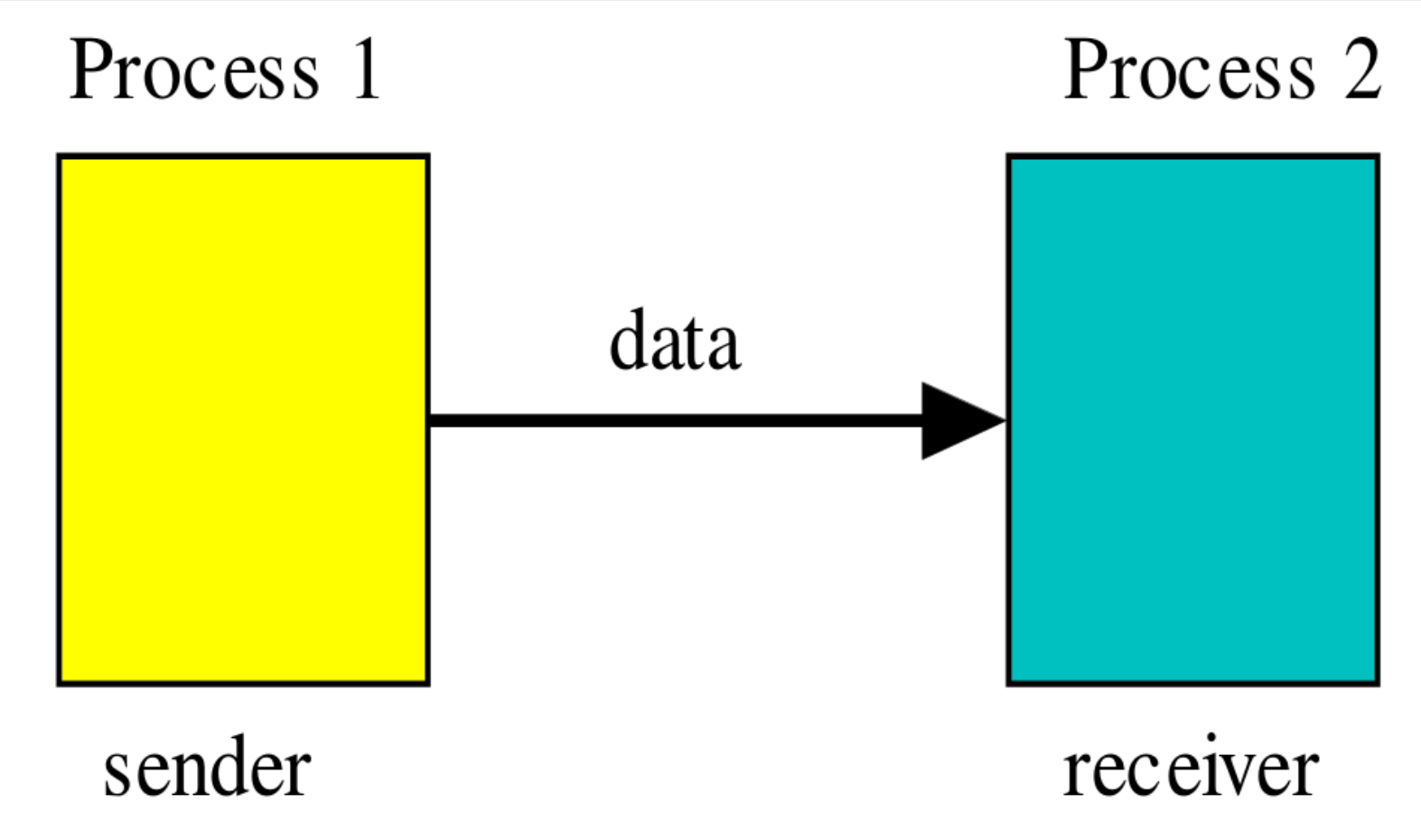
- In distributed computing, two or more processes engage in IPC in a protocol agreed upon by the processes. A process may be a sender at some points during a protocol, a receiver at other points.
- When communication is from one process to a single other process, the IPC is said to be a unicast.
- When communication is from one process to a group of processes, the IPC is said to be a multicast.

Unicast and Multicast

Unicast vs. Multicast



Interprocess Communications in Distributed Computing

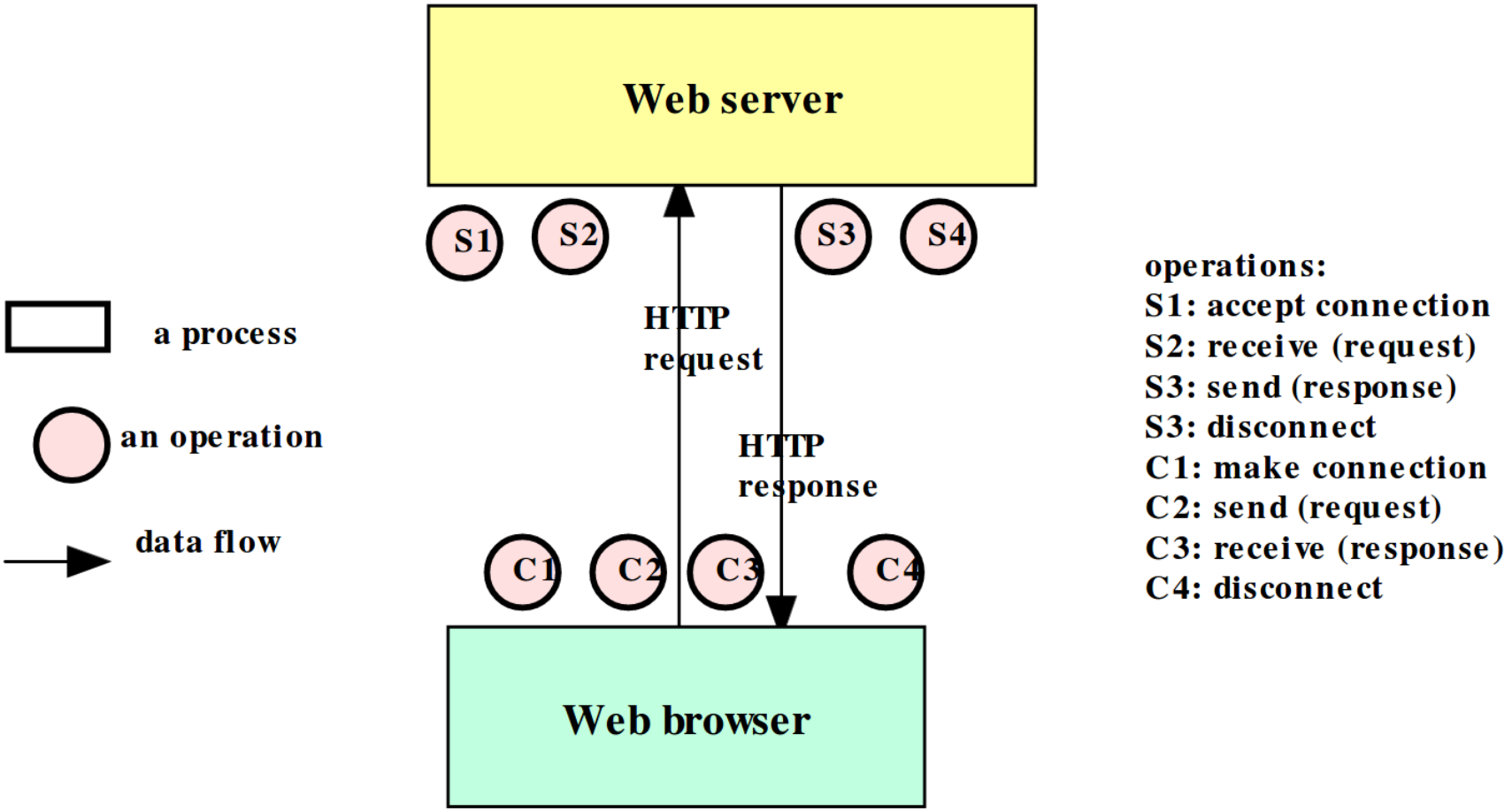


Operations in archetyipal IPCs APIs

Operations

- `Receive ( [sender], message storage object)`
- `Connect (sender address, receiver address)`, for connection-oriented communication.
- `Send ( [receiver], message)`
- `Disconnect (connection identifier)`, for connection oriented communication

Interprocess Communications in Basic HTTP



Section 2 | Event Synchronization

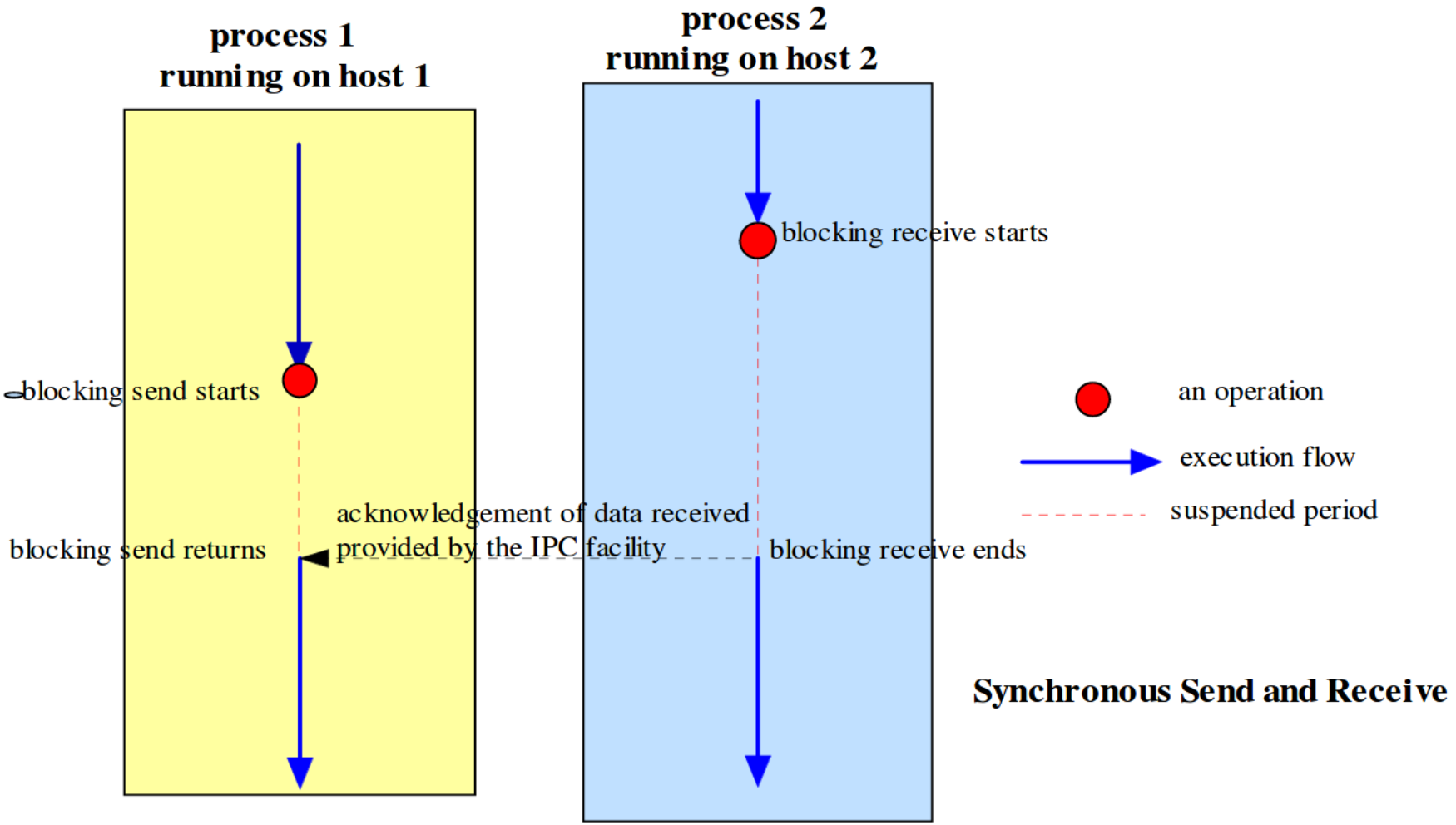
Event Synchronization

- Interprocess communication requires that the two processes synchronize their operations: one side sends, then the other receives until all data has been sent and received.

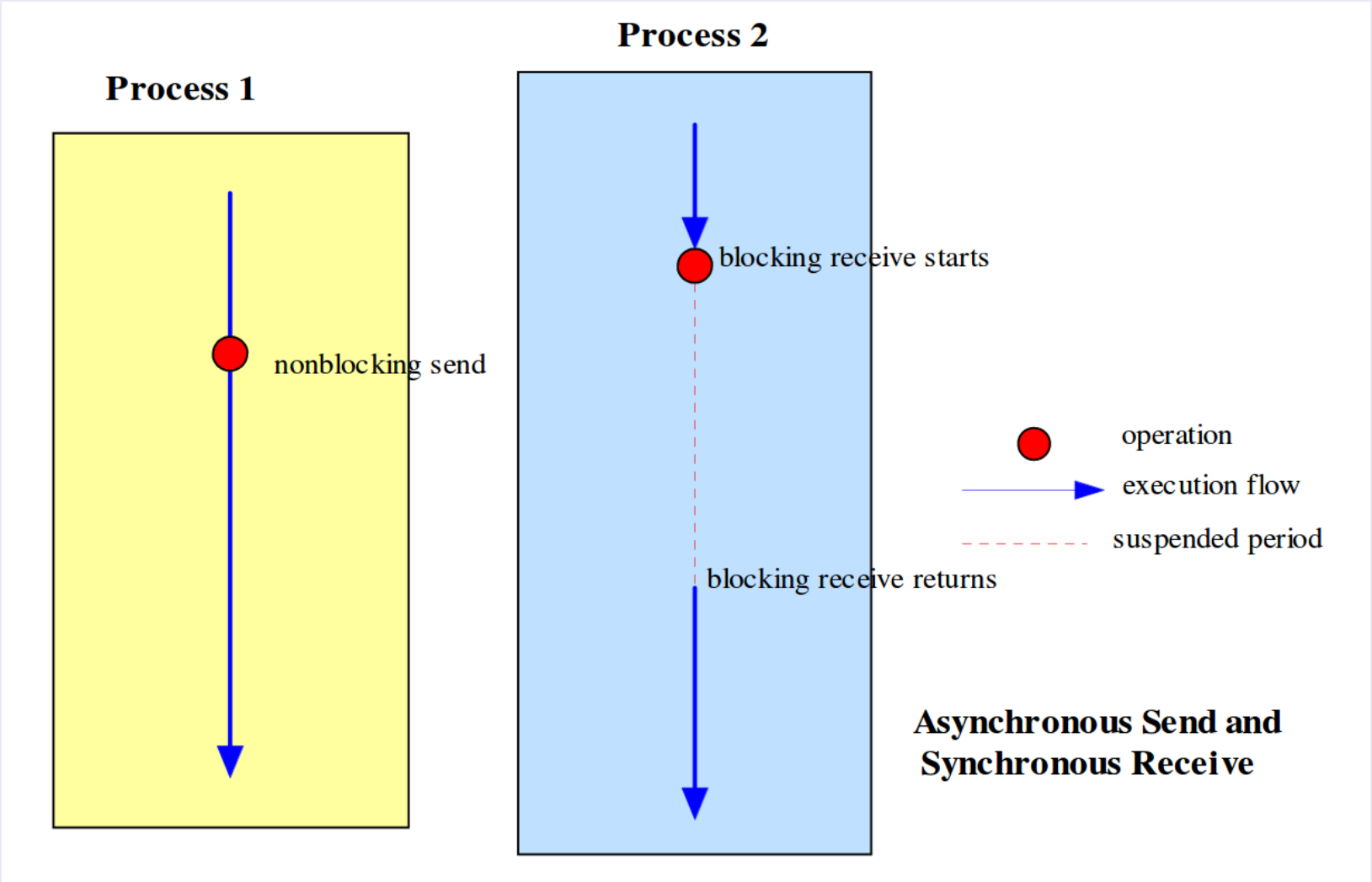
Event Synchronization

- The IPC operations may provide the synchronization necessary using blocking. A blocking operation issued by a process will block further processing of the process until the operation is fulfilled.
- Alternatively, IPC operations may be asynchronous or nonblocking. An asynchronous operation issued by a process will not block further processing of the process. Instead, the process is free to proceed with its processing, and may optionally be notified by the system when the operation is fulfilled.

Synchronous send and receive

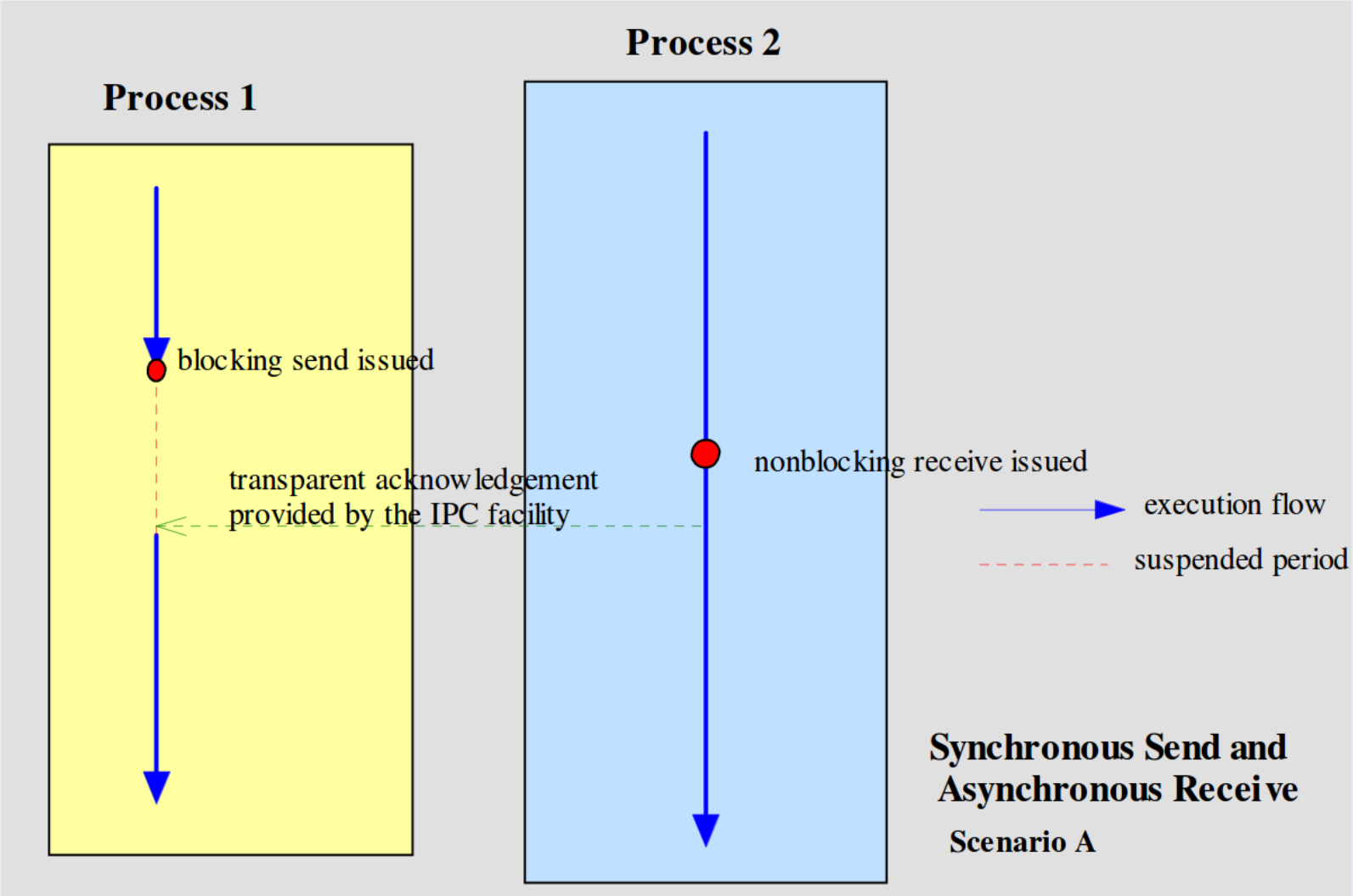


Asynchronous send and synchronous receive



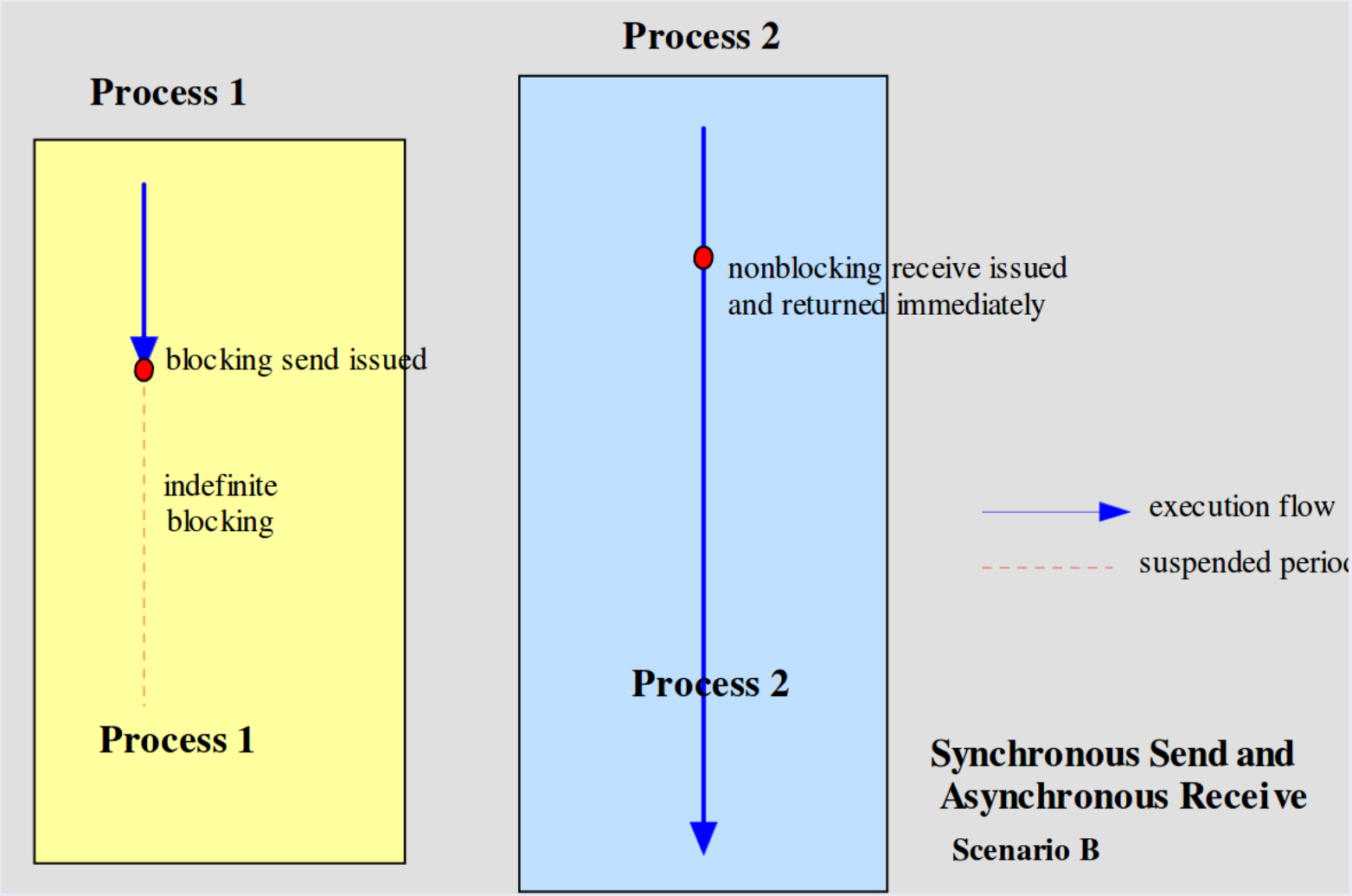
Synchronous send and Asynchronous Receive

Scenario A



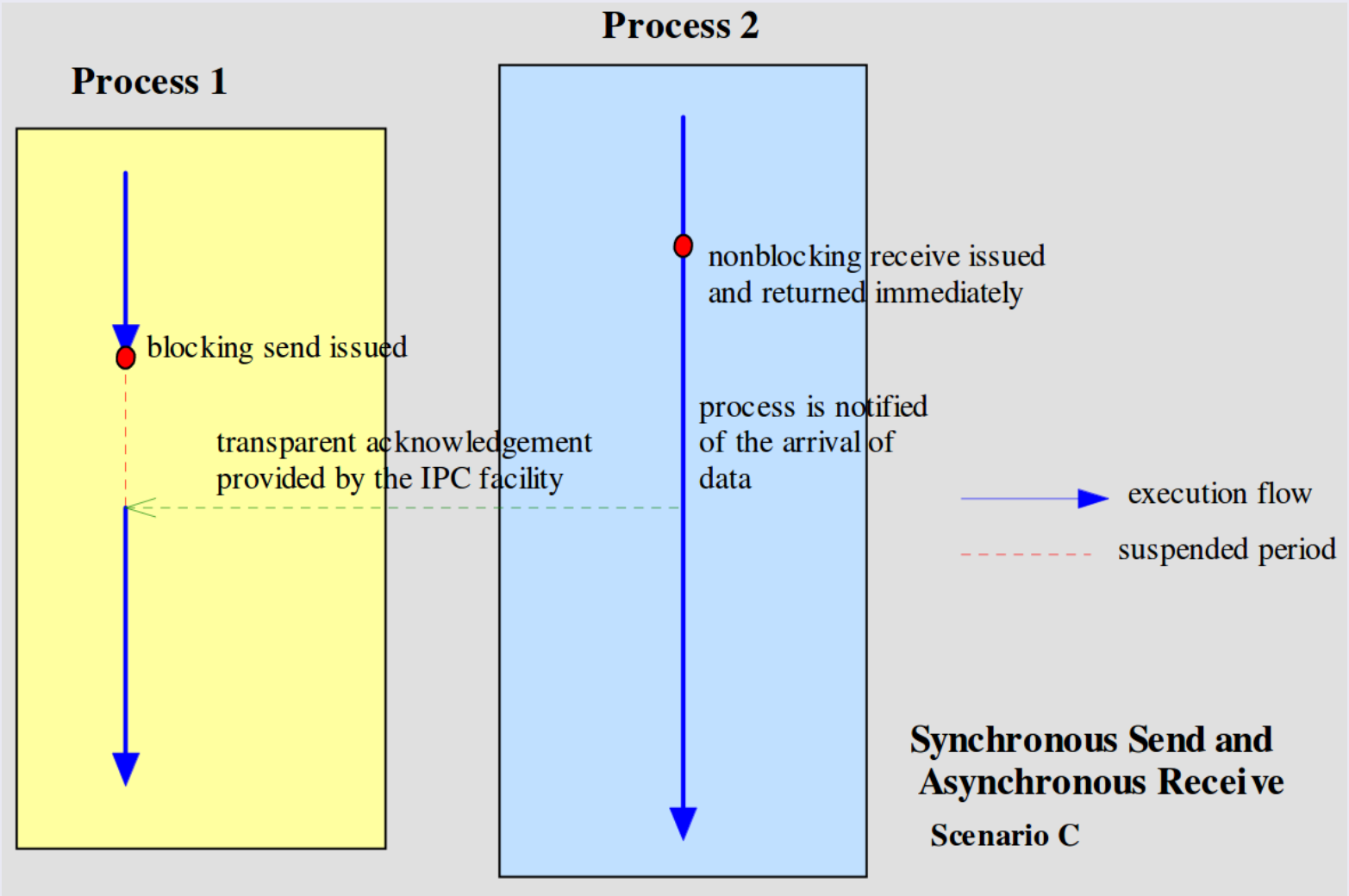
Synchronous send and Asynchronous Receive

Scenario B

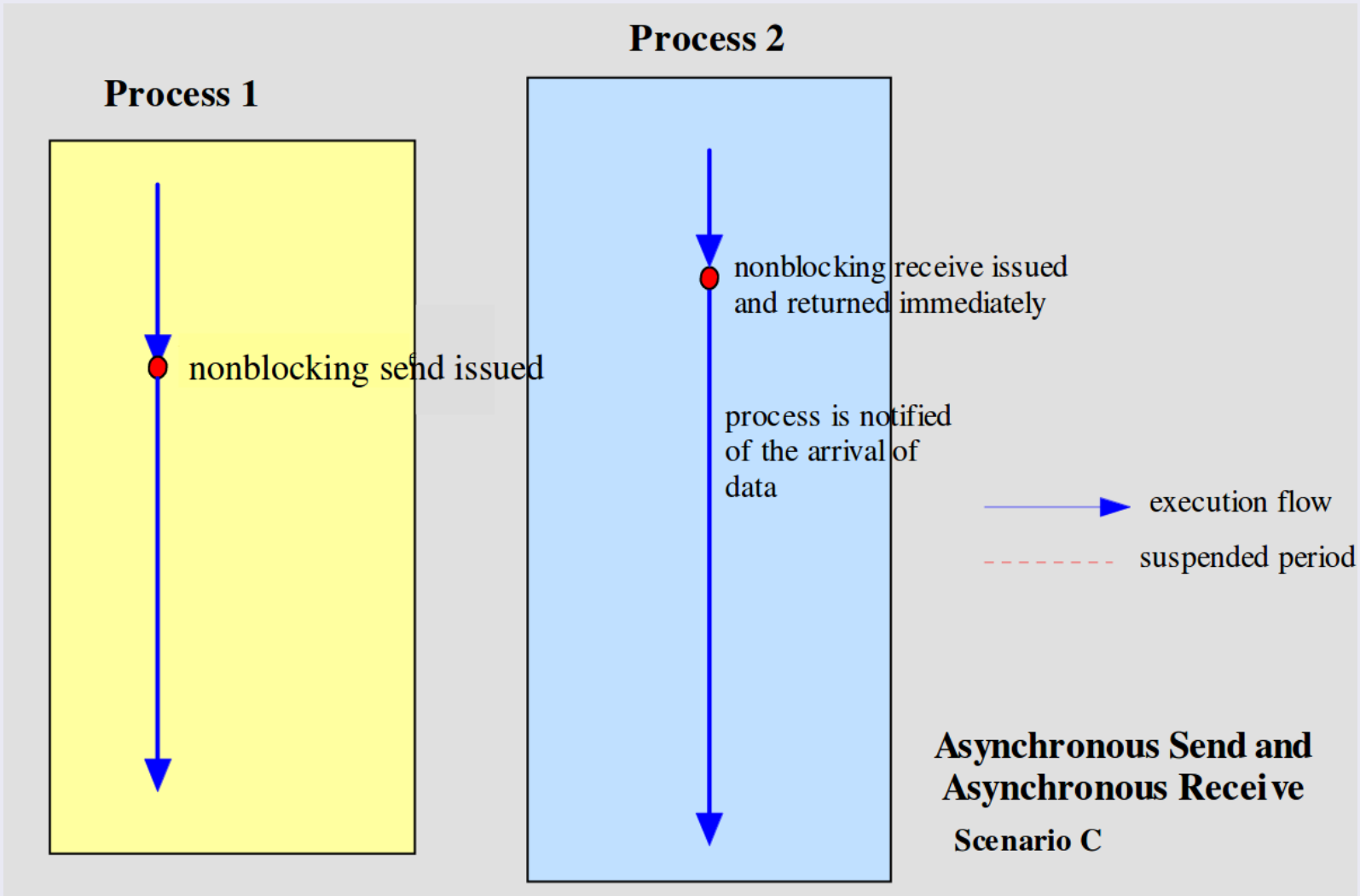


Synchronous send and Asynchronous Receive

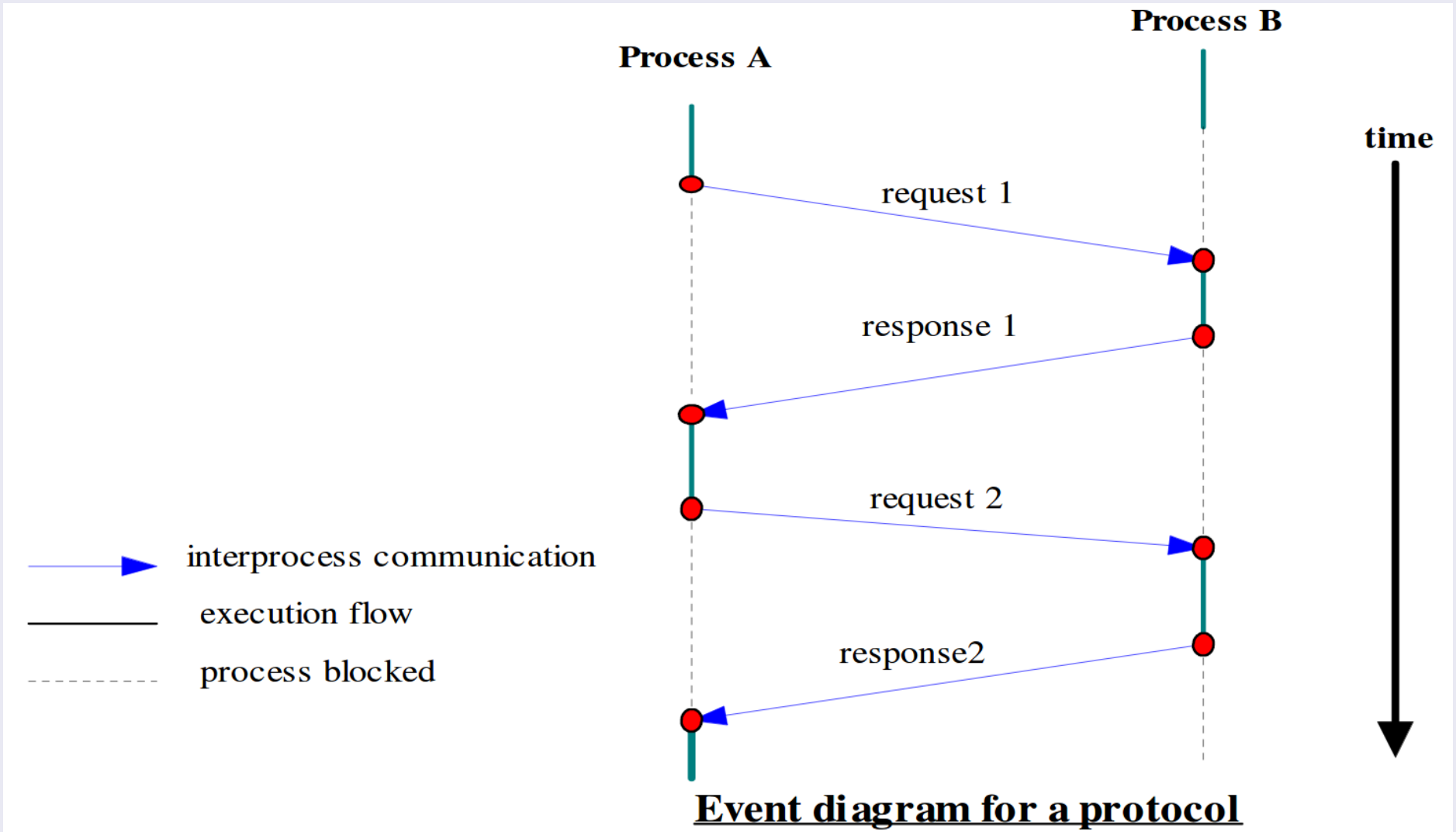
Scenario C



Asynchronous send and Asynchronous receive



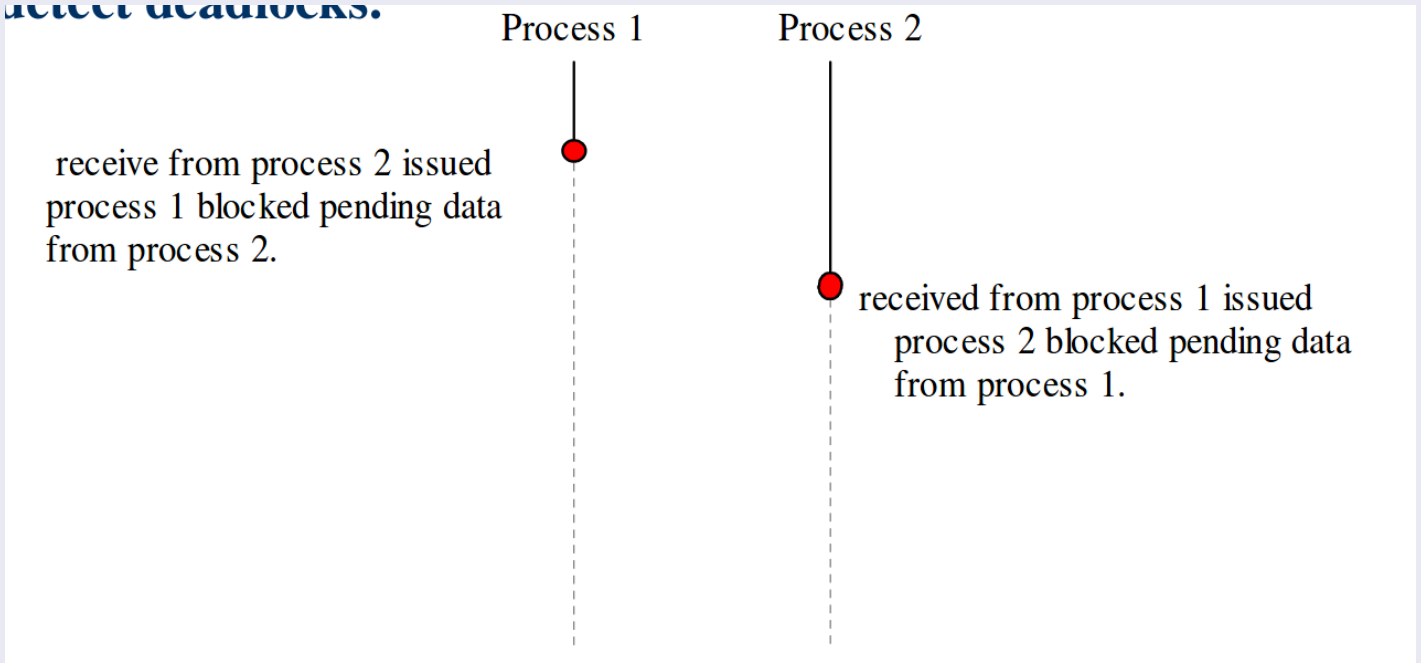
Event Diagram



Event diagram for a protocol

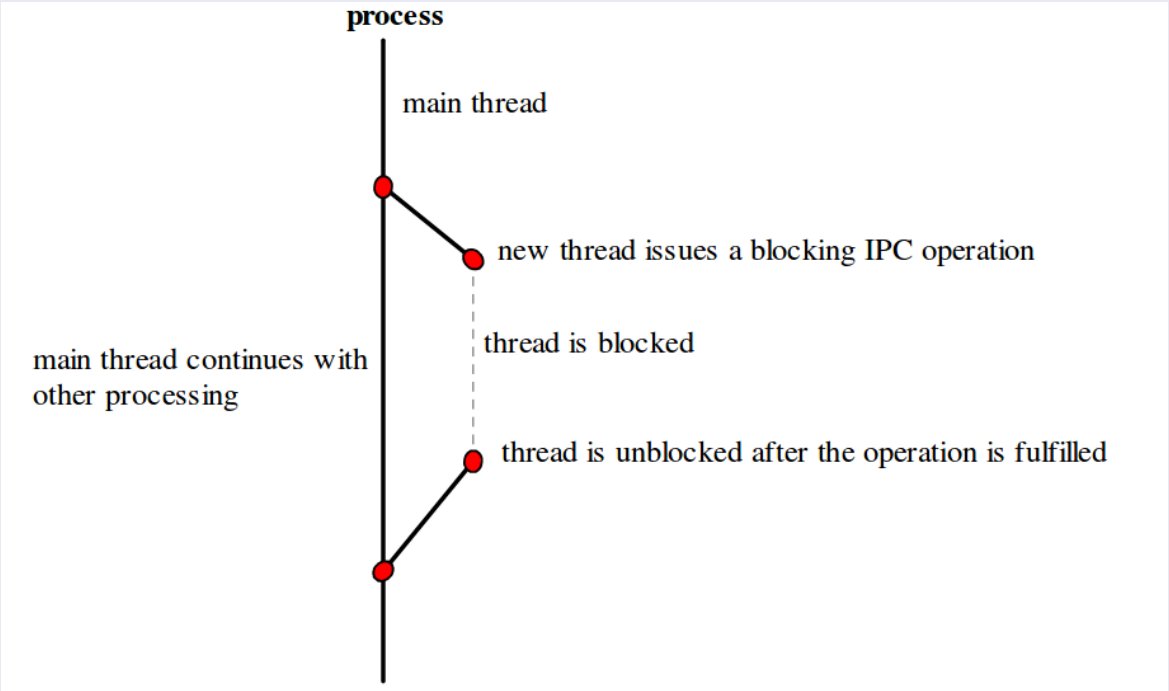
Blocking, deadlock and timeouts

- Blocking operations issued in the wrong sequence can cause deadlocks.
- Deadlocks should be avoided. Alternatively, timeout can be used to detect deadlocks.



Using threads for asynchronous IPC

- When using an IPC programming interface, it is important to note whether the operations are synchronous or asynchronous.
- If only blocking operation is provided for send and /or receive, then it is the programmer's responsibility using child processes or threads if asynchronous operations are desired.

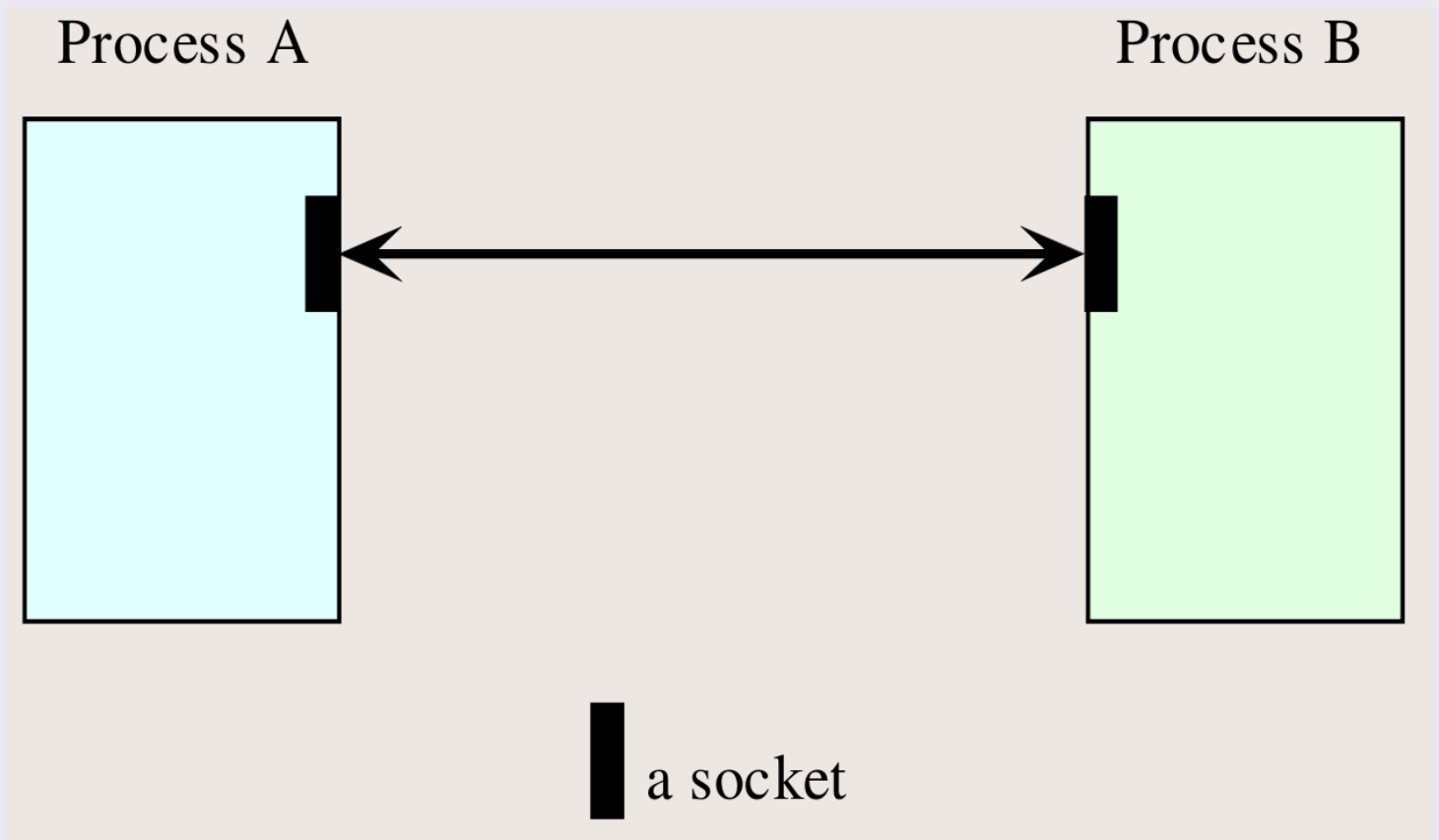


Section 3 | Sockets

Introduction

- The socket API is an Interprocessing Communication (IPC) programming interface originally provided as part of the Berkeley UNIX operating system.
- It has been ported to all modern operating systems, including Sun Solaris and Windows systems.
- It is a *de facto* standard for programming IPC, and is the basis of more sophisticated IPC interface such as *remote procedure call* and *remote method invocation*

The conceptual model of the socket API



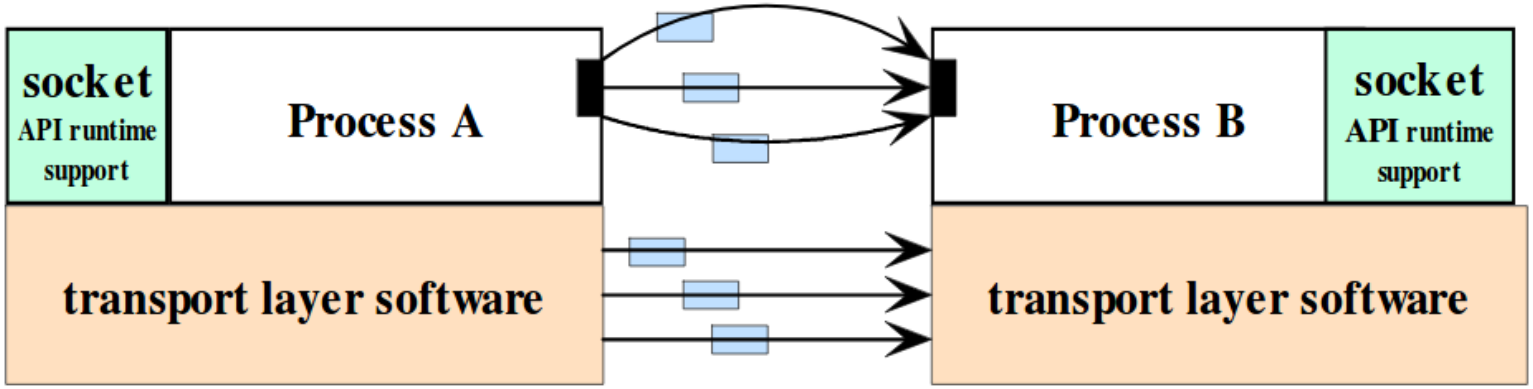
The Socket API

- A socket API provides a programming construct termed a socket. A process wishing to communicate with another process must create an instance, or instantiate, such a construct
- The two processes then issues operations provided by the API to send and receive data.
- A socket programming construct can make use of either the UDP or TCP protocol.
- Sockets that use UDP for transport are known as datagram sockets, sockets while sockets that use TCP are termed stream sockets.

Section 4 | Datagram Sockets

Connection Oriented & connectionless datagram socket

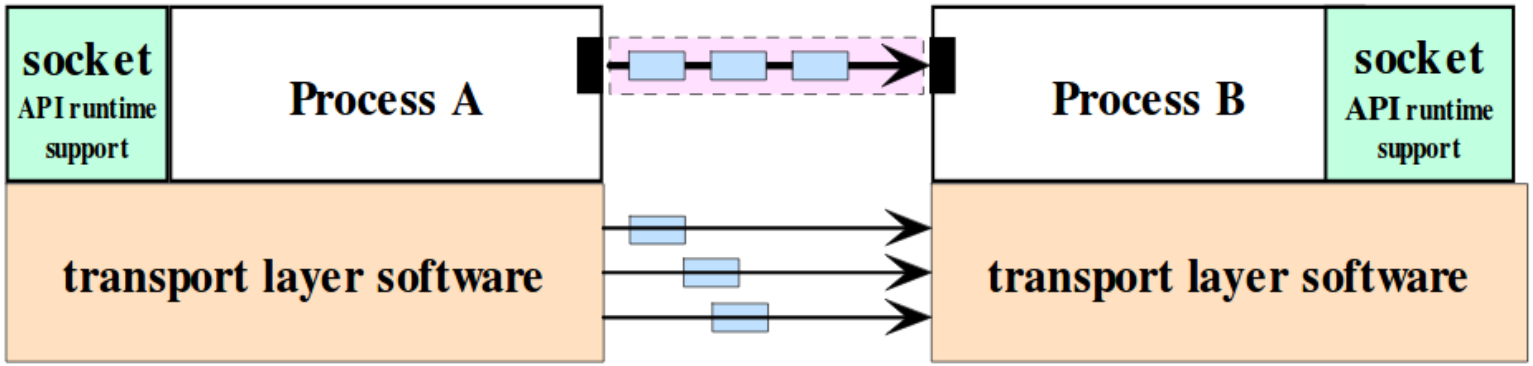
- Datagram sockets can support both connectionless and connection oriented communication at the application layer.
- This is so because even though datagrams are sent or received without the notion of connections at the transport layer, the runtime support of the socket API can create and maintain logical connections for datagrams exchanged between two processes, as you will see in the next section.
- The runtime support of an API is a set of software that is bound to the program during execution in support of the API.



connectionless datagram socket

a datagram

a logical connection created and maintained by the runtime support of the datagram socket API



connection-oriented datagram socket

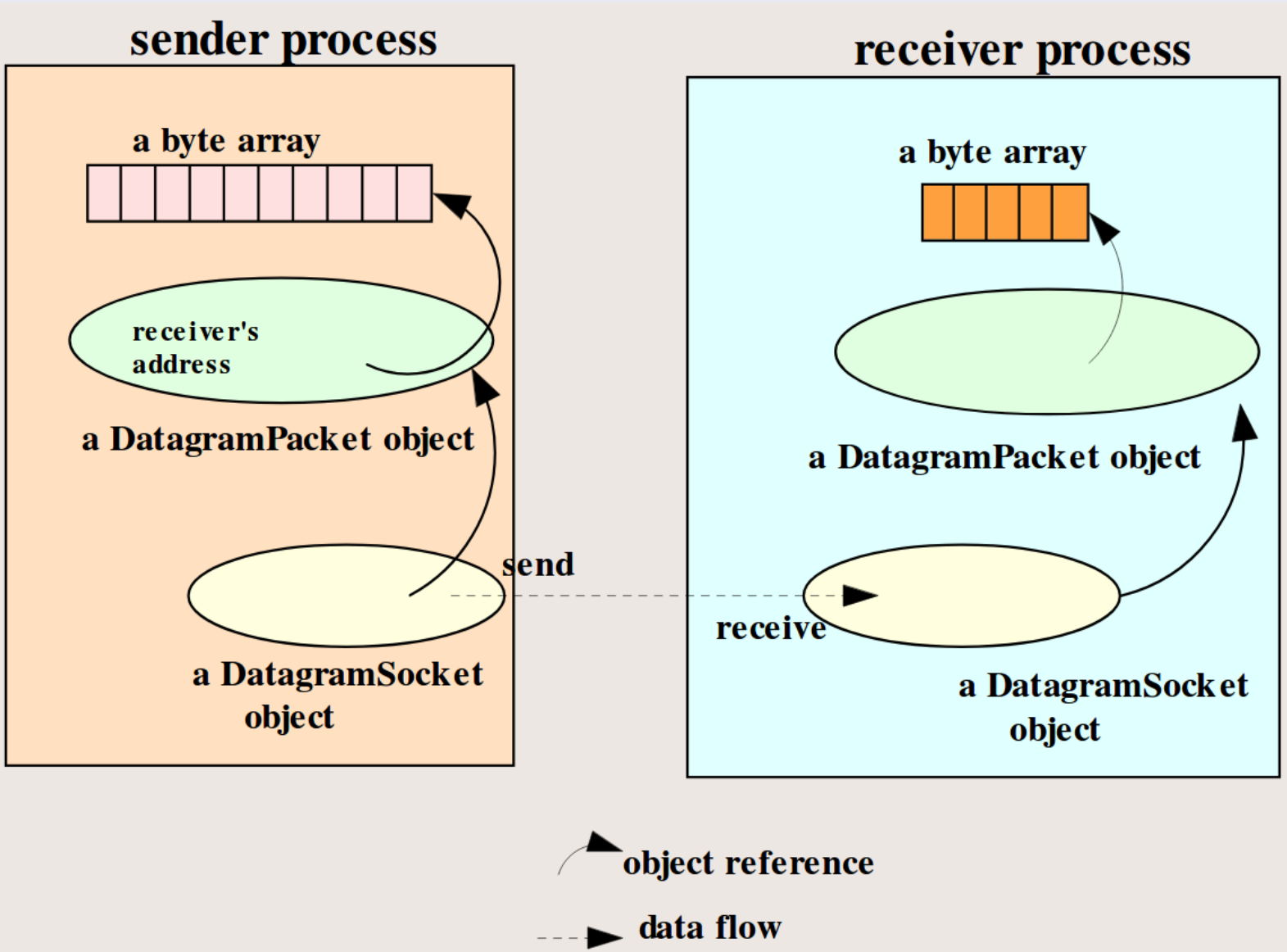
The Java Datagram Socket API

- In Java, two classes are provided for the datagram socket API:
  - the `DatagramSocket` class for the sockets.
  - the `DatagramPacket` class for the datagram exchanged.
- A process wishing to send or receive data using this API must instantiate a DatagramSocket object, or a socket in short. Each socket is said to be bound to a UDP port of the machine local to the process

The Java Datagram Socket API

- In the receiving process, a `DatagramSocket` object must also be instantiated and bound to a local port, the port number must agree with that specified in the datagram packet of the sender.
- To receive datagrams sent to the socket, the process creates a `DatagramPacket` object which references a byte array and calls a receive method in its DatagramSocket object, specifying as argument a reference to the `DatagramPacket` object.

The Data Structures in the sender and receiver programs



The program flow in the sender and receiver programs

Sender Program

- create a datagram socket and bind it to any local port;
- place data in a byte array;
- create a datagram packet, specifying the data array and the receiver's address;
- invoke the send method of the socket with a reference to the datagram packet;

Receiver Program

- create a datagram socket and bind it to a specific local port;
- create a byte array for receiving the data; create a datagram packet, specifying the data array;
- invoke the receive method of the socket with a reference to the datagram packet

Connectionless datagram sockets

- With connectionless sockets, it is possible for multiple processes to simultaneously send datagrams to the same socket established by a receiving process, in which case the order of the arrival of these messages will be unpredictable, in accordance with the UDP protocol

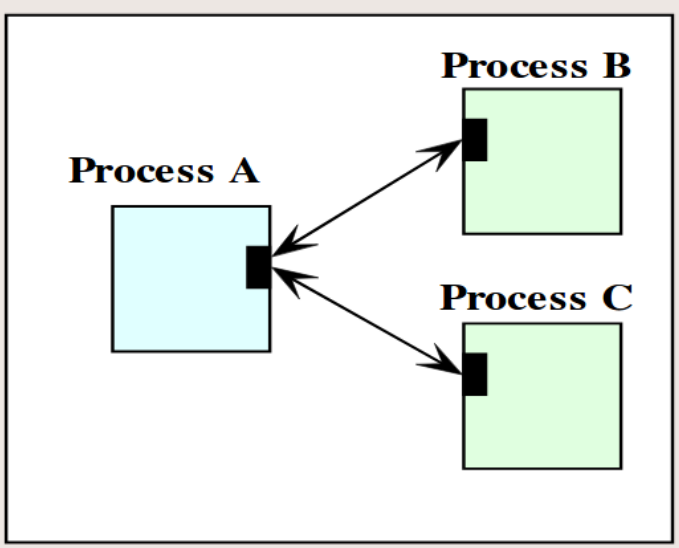


Figure 3a

a connectionless datagram socket

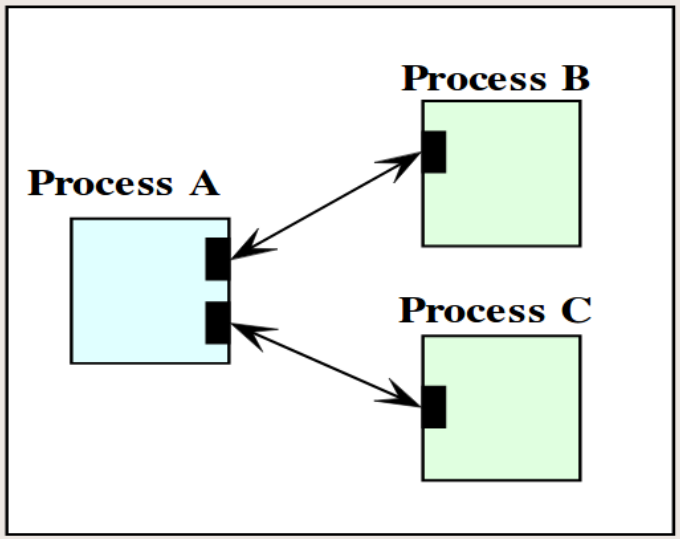
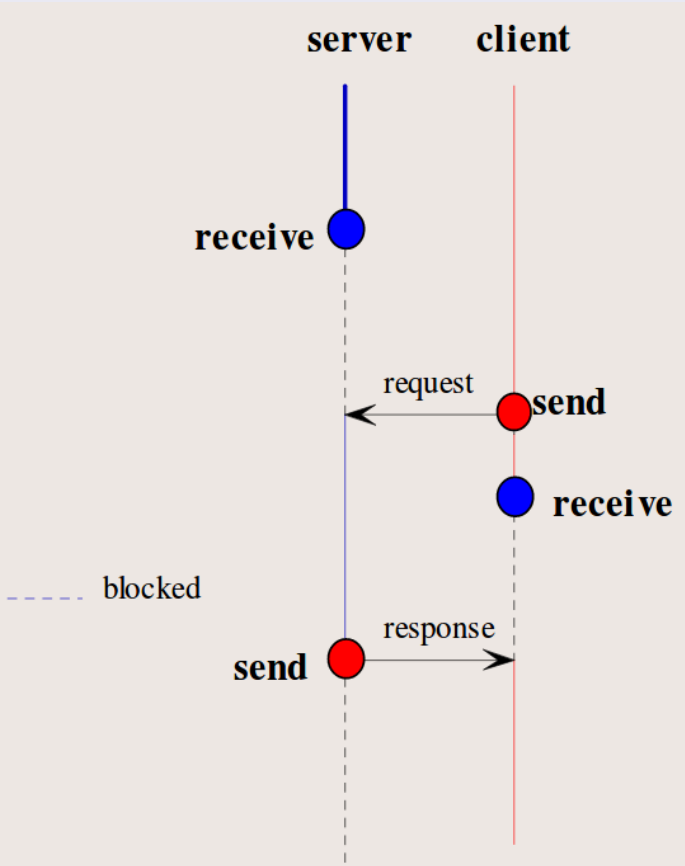


Figure 3b

- It is not common to employ datagram sockets for connection - oriented communication

Event synchronization with the connectionless datagram sockets API



blocking receive, nonblocking send in a request-response protocol

if data is sent before a corresponding receive operation is issued, the data will be discarded by the runtime support and will not be received.

Setting timeout

- To avoid indefinite blocking, a timeout can be set with a socket object:
- `void setSoTimeout(int timeout)`
- Set a timeout for the blocking receive from this socket, in milliseconds.
- Once set, the timeout will be in effect for all blocking operations.

Key Methods and Constructors

- `DatagramPacket (byte[ ] buf, int length)` Construct a datagram packet for receiving packets of length length; data received will be stored in the byte array reference by buf.
- `DatagramPacket (byte[ ] buf, int length, InetAddress address, int port` Construct a datagram packet for sending packets of length length to the socket bound to the specified port number on the specified host ; data received will be stored in the byte array reference by buf
- `DatagramSocket ( )` Construct a datagram socket and binds it to any available port on the local host machine; this constructor can be used for a process that sends data and does not need to receive data.
- `DatagramSocket (int port)` Construct a datagram socket and binds it to the specified port on the local host machine; the port number can then be specified in a datagram packet sent by a sender.
- `void close( )` Close this datagramSocket object
- `void receive ( DatagramPacket p)` Receive a datagram packet using this socket.
- `void send ( DatagramPacket p)` Send a datagram packet using this socket.
- `void setSoTimeout (int timeout)` Set a timeout for the blocking receive from this socket, in milliseconds.

Source

Sending process

```
InetAddress receiverHost=InetAddress.getByName("localHost");
DatagramSocket theSocket = new DatagramSocket( );
String message = "Hello world!";
byte[ ] data = message.getBytes( );
DatagramPacket thePacket //remote port is specified in datagram
= new DatagramPacket(data, data.length, receiverHost, 2345);
theSocket.send(thePacket);
```

Receiver process

```
DatagramSocket ds = new DatagramSocket(2345);
DatagramPacket dp = new DatagramPacket(buffer, MAXLEN);
ds.receive(dp);
len = dp.getLength( );
System.out.Println(len + " bytes received.\n");
String s = new String(dp.getData( ), 0, len);
System.out.println(dp.getAddress( ) + " at port "
+ dp.getPort( ) + " says " + s);
```

Connection-oriented datagram socket

- `public void connect(InetAddress address, int port)` Create a logical connection between this socket and a socket at the remote address and port.
- `public void disconnect()` Cancel the current connection, if any, from this socket.
- A connection is made for a socket with a remote socket.
- Once a socket is connected, it can only exchange data with the remote socket.
- If a datagram specifying another address is sent using the socket, an IllegalArgumentException will occur.
- If a datagram from another socket is sent to this socket, the data will be ignored.

Connection-oriented datagram socket

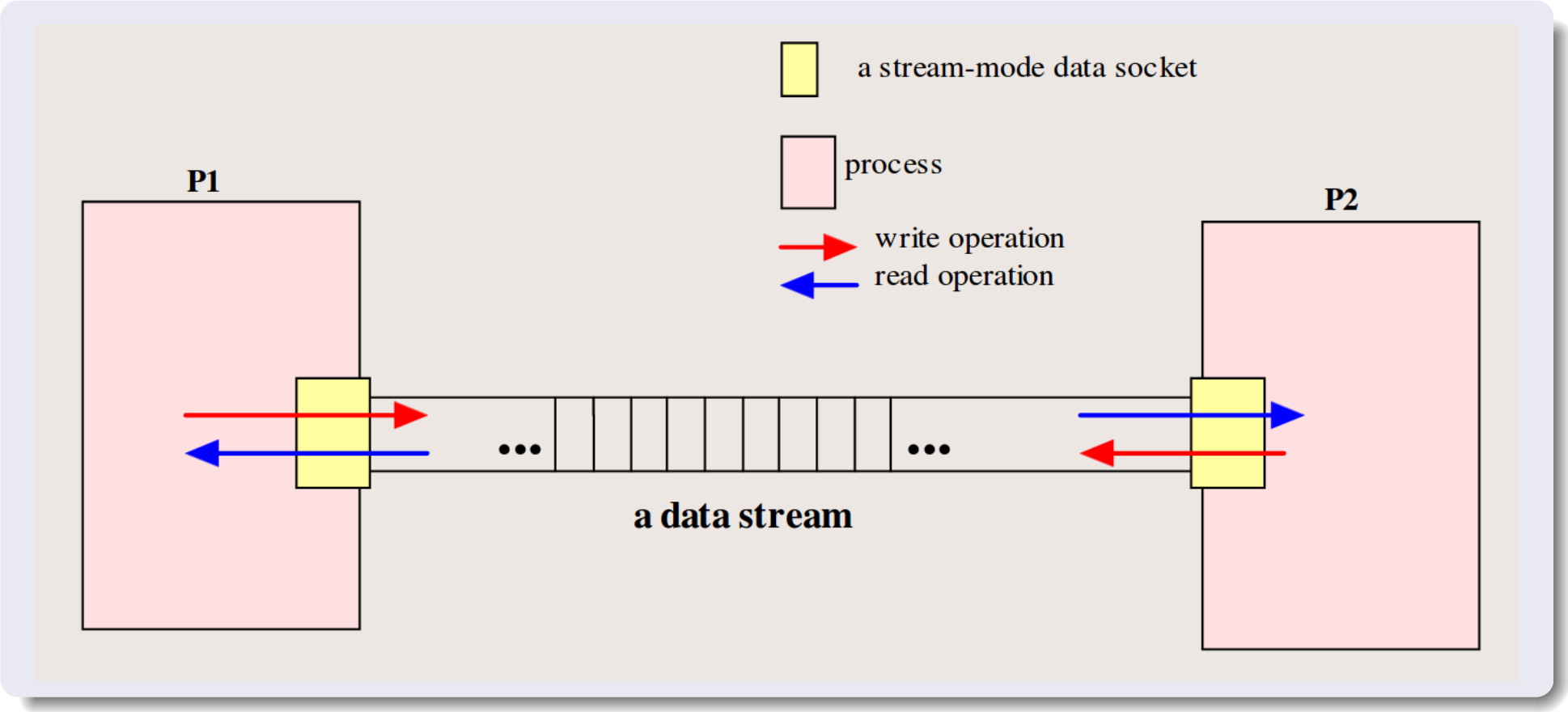
- The connection is unilateral: it is enforced only on one side.
- The socket on the other side is free to send and receive data to and from other sockets, unless it too commits to a connection to the other socket.

Section 5 | Stream sockets

Stream-mode socket

- The datagram socket API supports the exchange of discrete units of data (that is, datagrams).
- But the stream socket API provides a model of data transfer based on the stream-mode I/O of the Unix operating systems.
- By definition, a stream-mode socket supports connection-oriented communication only.

Stream-mode Socket API (connection-oriented socket API)



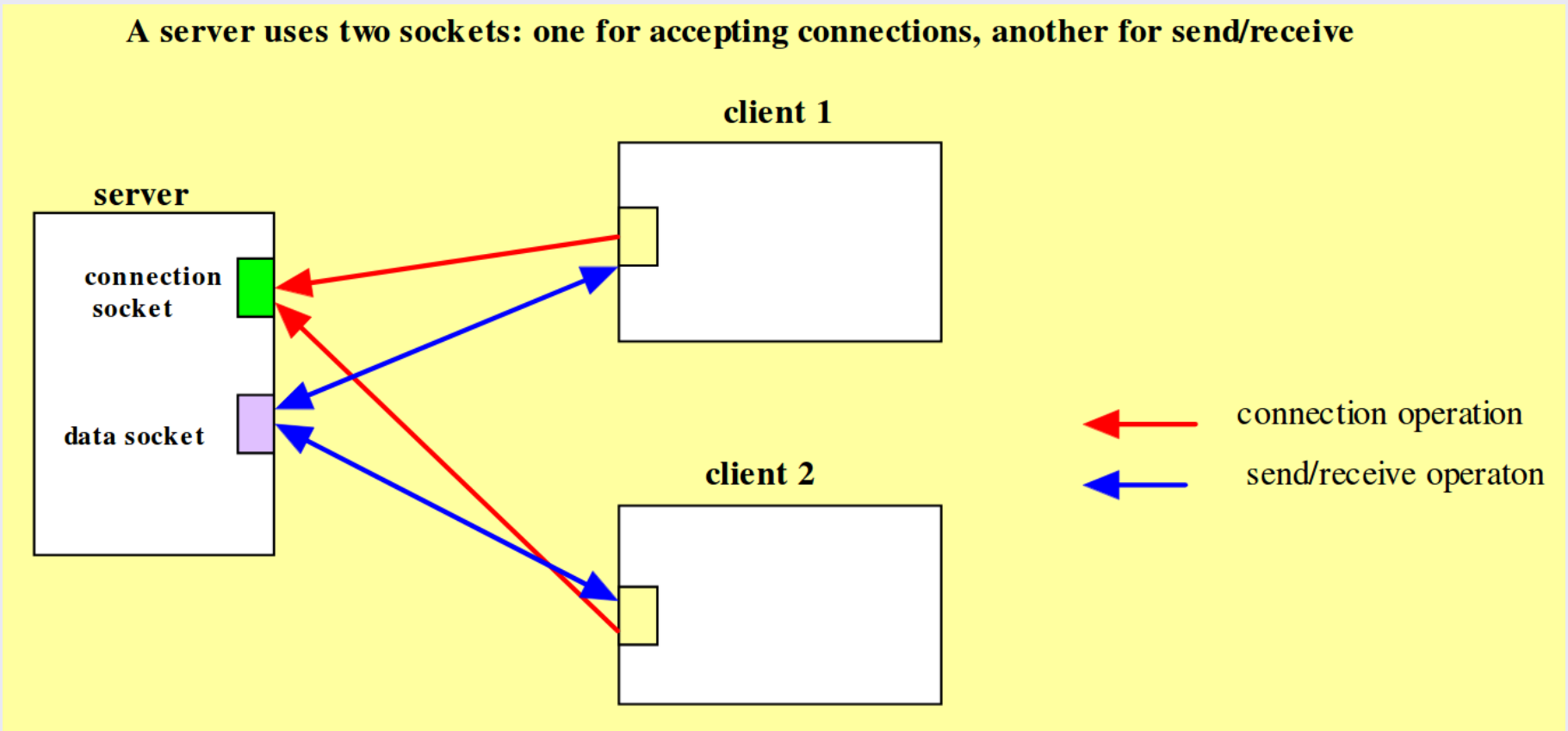
Stream-mode Socket API

- A stream-mode socket is established for data exchange between two specific processes.
- Data stream is written to the socket at one end, and read from the other end.
- A stream socket cannot be used to communicate with more than one process.

Stream-mode Socket API

- In Java, the stream-mode socket API is provided with two classes:
- Server socket: for accepting connections; we will call an object of this class a *connection socket*.
- Socket: for data exchange; we will call an object of this class a *data socket*.

The Server (the connection listener)



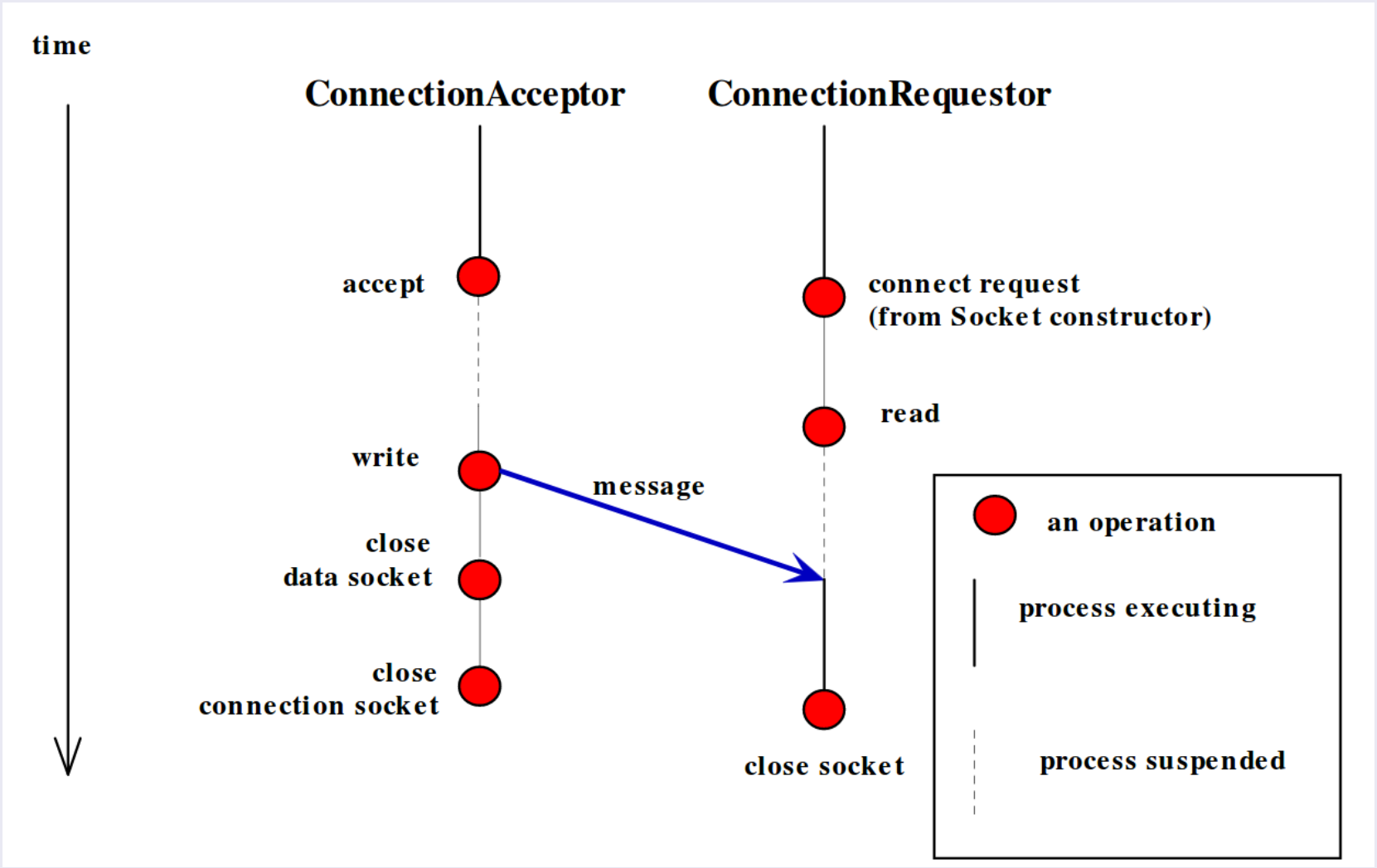
Key methods in the ServerSocket class

- `ServerSocket(int port)` Creates a server socket on a specified port.
- `Socket accept() throws IOException` Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.
- `public void close() throws IOException` Set a timeout period (in milliseconds) so that a call to accept( ) for this socket will block for only this amount of time. If the timeout expires, a java.io.InterruptedIOException is raised
- `void setSoTimeout(int timeout) throws SocketException` Closes this socket
- Note: Accept is a blocking operation.

Key methods in the Socket Class

- `Socket(InetAddress address, int port)` Creates a stream socket and connects it to the specified port number at the specified IP address
- `void close() throws IOException` Closes this socket.
- `InputStream getInputStream( ) throws IOException` Returns an input stream so that data may be read from this socket.
- `OutputStream getOutputStream() throws IOException` Returns an output stream so that data may be written to this socket.
- `void setSoTimeout(int timeout) throws SocketException` Set a timeout period for blocking so that a read( ) call on the InputStream associated with this Socket will block for only this amount of time. If the timeout expires, a java.io.InterruptedIOException is raised
- A read operation on the InputStream is blocking.
- A write operation is nonblocking.

Example: Event Diagram



Source

Connection acceptor

```
ServerSocket connectionSocket = new ServerSocket(19999);
Socket dataSocket = connectionSocket.accept();
OutputStream outStream = dataSocket.getOutputStream();
PrintWriter socketOutput =
new PrintWriter(new OutputStreamWriter(outStream));
socketOutput.println("message");
socketOutput.flush();  dataSocket.close( ); connectionSocket.close( )
```

Connections process

```
InetAddress acceptorHost = InetAddress.getByName("server.com");
Socket mySocket = new Socket(acceptorHost, 19999);
InputStream inStream = mySocket.getInputStream();
BufferedReader socketInput =
new BufferedReader(new InputStreamReader(inStream));
String message = socketInput.readLine( );
System.out.println("\t" + message);
mySocket.close( );
```

References

- Distributed Systems. Marteen Van Steen and Andrew Tanenbaum (2017).
- Mei Ling-Liu. Distributed Computing Algorithms.
- Classroom notes by Guadalupe Ortiz