Networking & NIO

media.net

SYN

whoami

Socket Programming

-What is it ?

-Why bother ?

Basic

- Interface for programming networks at transport level
- It is communication end point
- Used for inter process communication over network
- Need IP address and port number
- Popularly used in client-server computing
- Connection oriented
  – TCP – Phone system – Delivery is guaranteed
- Connectionless
  – UDP – Postal system – Delivery is not guaranteed

Ports

- Represented by a positive (16 bit) integer
- Some ports are reserved for common services
  - FTP 21
  - TELNET 23
  - SMTP 25
  - HTTP 80
- User process generally use port value >= 1024
- Heard of ephemeral ports ?

Code

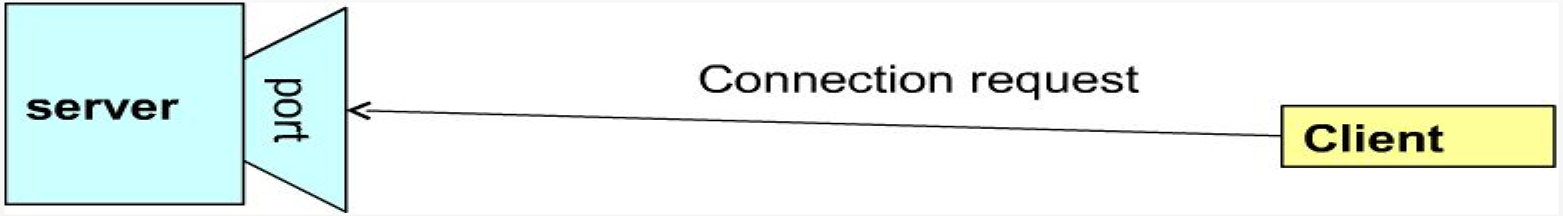
YES ALREADY !!!

ZIP file

- whois/Whois

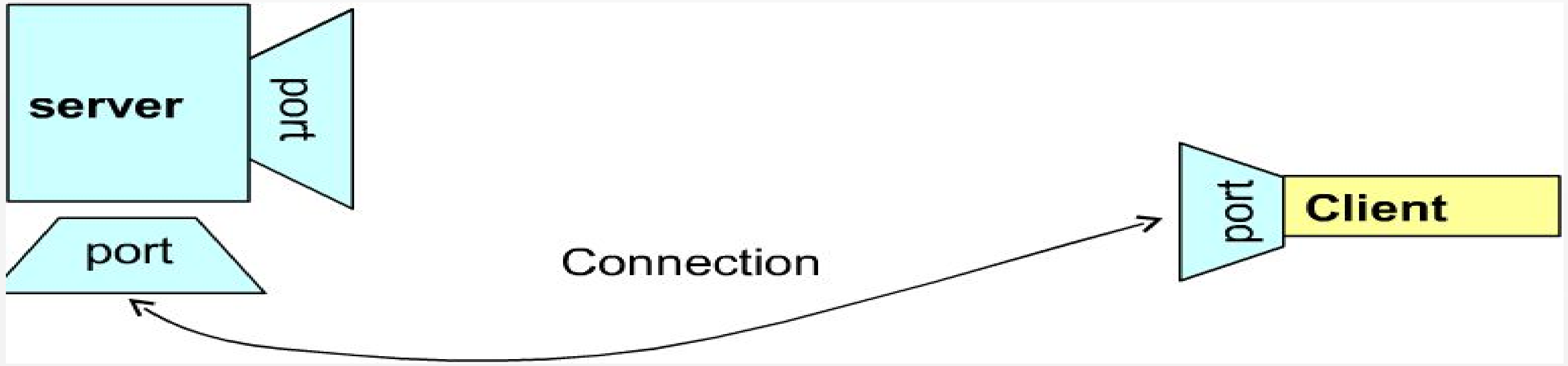
Socket communication

- A server (program) runs on a specific computer and has a socket bound to that port. The server waits and listens to socket for a client to make a connection request



Socket Communication

- Upon acceptance, the server gets a new socket bounds to a different port. It needs a new socket (different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client.



Java socket library

- Through the classes in java.net, program can use TCP / UDP to communicate over the internet
- ‘URL’, ‘URLConection’, ‘Socket’, ‘ServerSockets’ - TCP
- ‘DatagramSocket’ / ‘DatagramPacket’ - UDP
- Raw Sockets and Unix Domain Sockets (No java native support ... have to use 3rd party JNI libs)

TCP / IP in java

- Java.net.InetAddress: Represents an IP address (either IPv4 or IPv6) and has methods for performing DNS lookups
- Java.net.Socket: Represents a TCP socket
- Java.net.ServerSocket: Represents a server socket which is capable of waiting for requests from clients

InetAddress

- Used to encapsulate both the numerical IP address and domain name for that address
- Factory methods to be used to create instance
  - static InetAddress getLocalHost()
  - static InetAddress getByName(String hostName)
  - static InetAddress getAllByName(String hostName)

InetSocketAddress class (check it out)

code

ip/InetAddressTest

Client Socket

- Java wraps OS sockets (over TCP) by the objects of class java.net.Socket
  - Socket(String remoteHost, int remotePort)

- Create TCP socket and connects it to the remote host on the remote port (hand shake)

- Write and read :
  - Using Streams:
    - InputStream getInputStream()
    - OutputStream getOutputStream()
  - Using channel (nio ...)

Server Socket

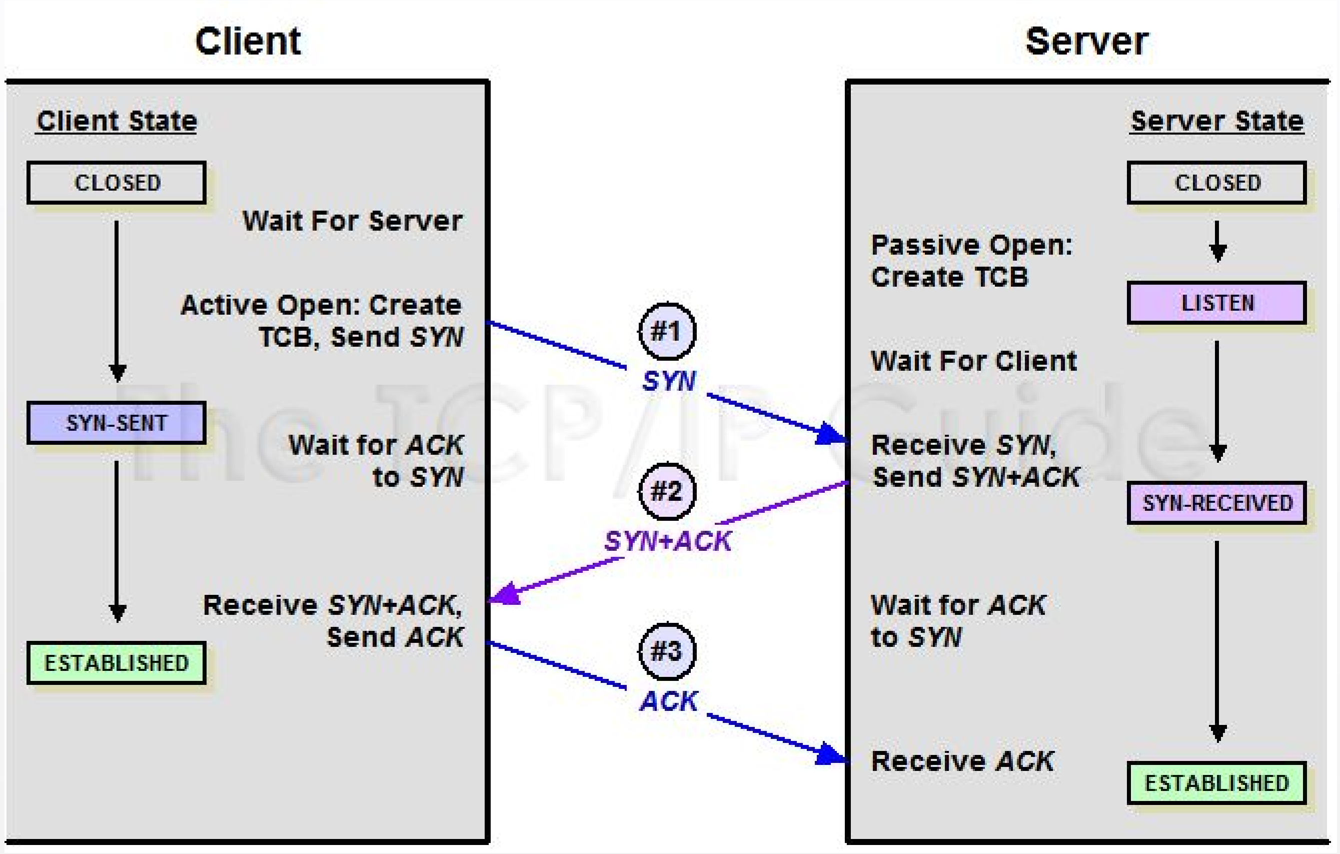
- This class implements server socket. A server socket waits for requests to come in over the network. It performs some operation based on that request, and possibly returns a result to the requester.

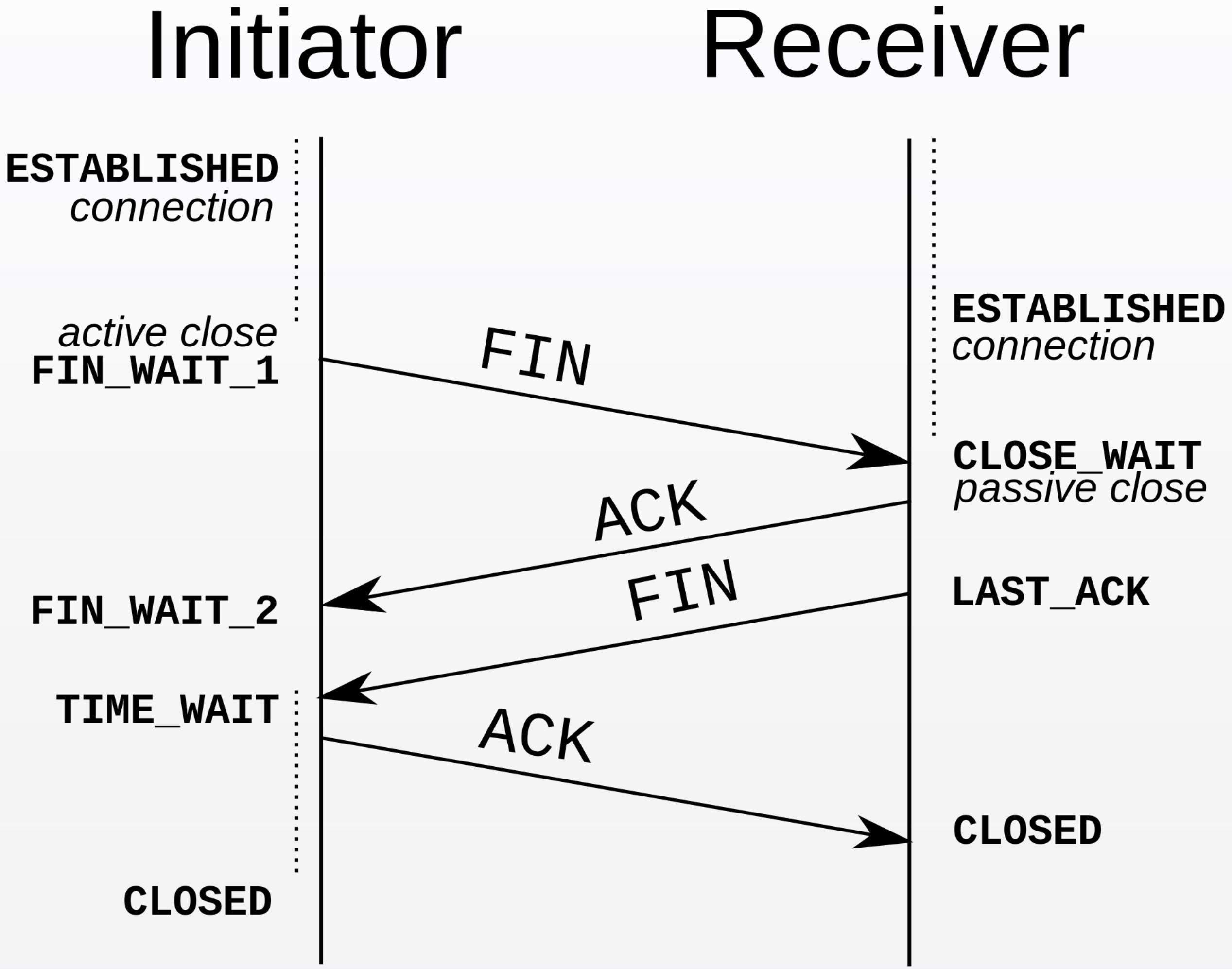
- A server socket is technically not a socket: when a client connects to a server socket, a TCP connection is made, and a (normal) socket is created for each end point.

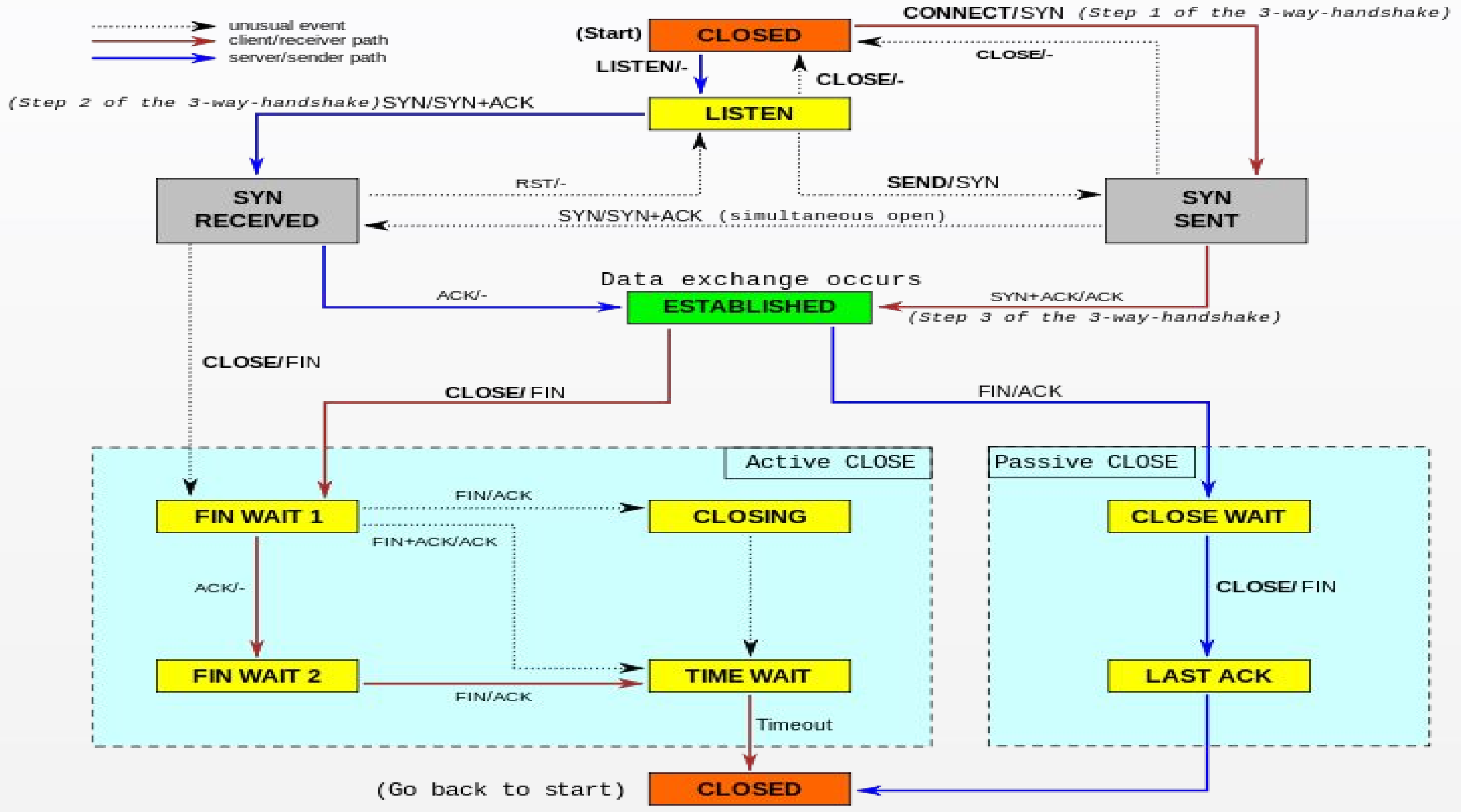
code

echo/EchoServer

TCP States

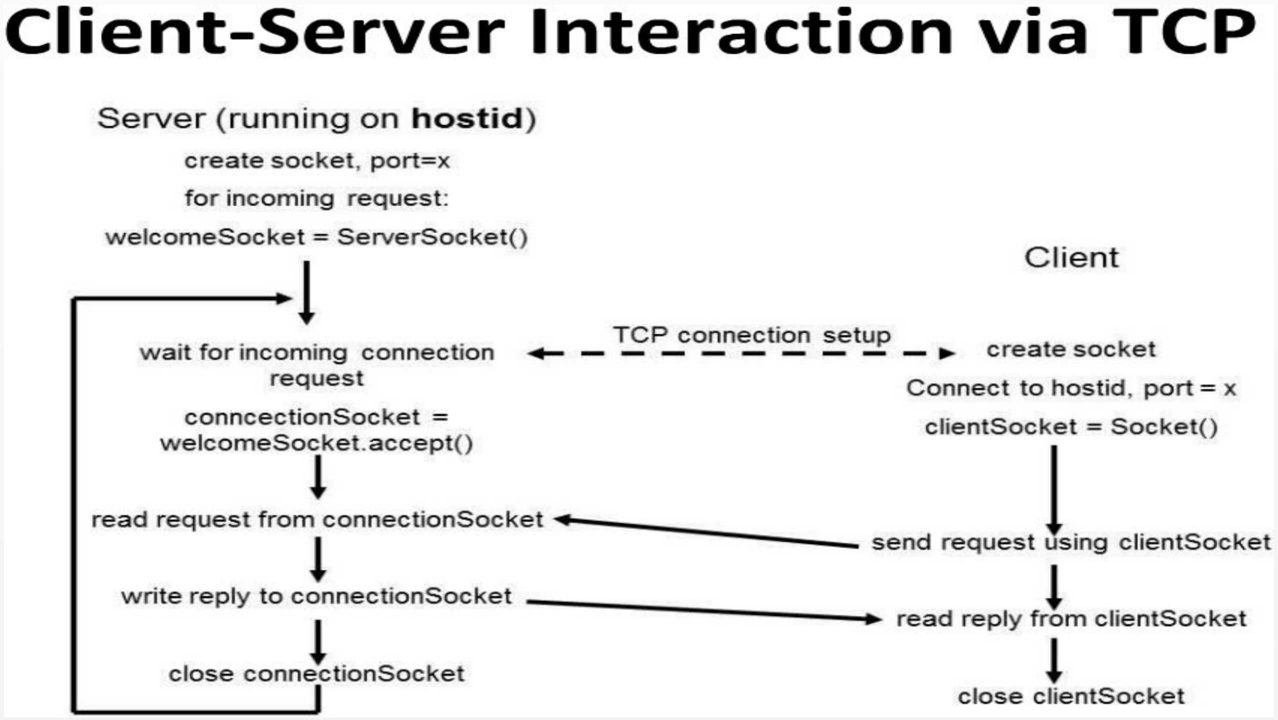

Client
Server
Client State
CLOSED
Wait For Server
Active Open: Create TCB, Send SYN
SYN-SENT
Wait for ACK to SYN
Receive SYN+ACK, Send ACK
ESTABLISHED
#1
SYN
#2
SYN+ACK
#3
ACK
Server State
CLOSED
Passive Open: Create TCB
LISTEN
Wait For Client
Receive SYN, Send SYN+ACK
SYN-RECEIVED
Wait for ACK to SYN
Receive ACK
ESTABLISHED


Initiator
Receiver
ESTABLISHED
connection
active close
FIN_WAIT_1
FIN
ESTABLISHED
connection
CLOSE_WAIT
passive close
ACK
FIN
LAST_ACK
FIN_WAIT_2
TIME_WAIT
ACK
CLOSED
CLOSED

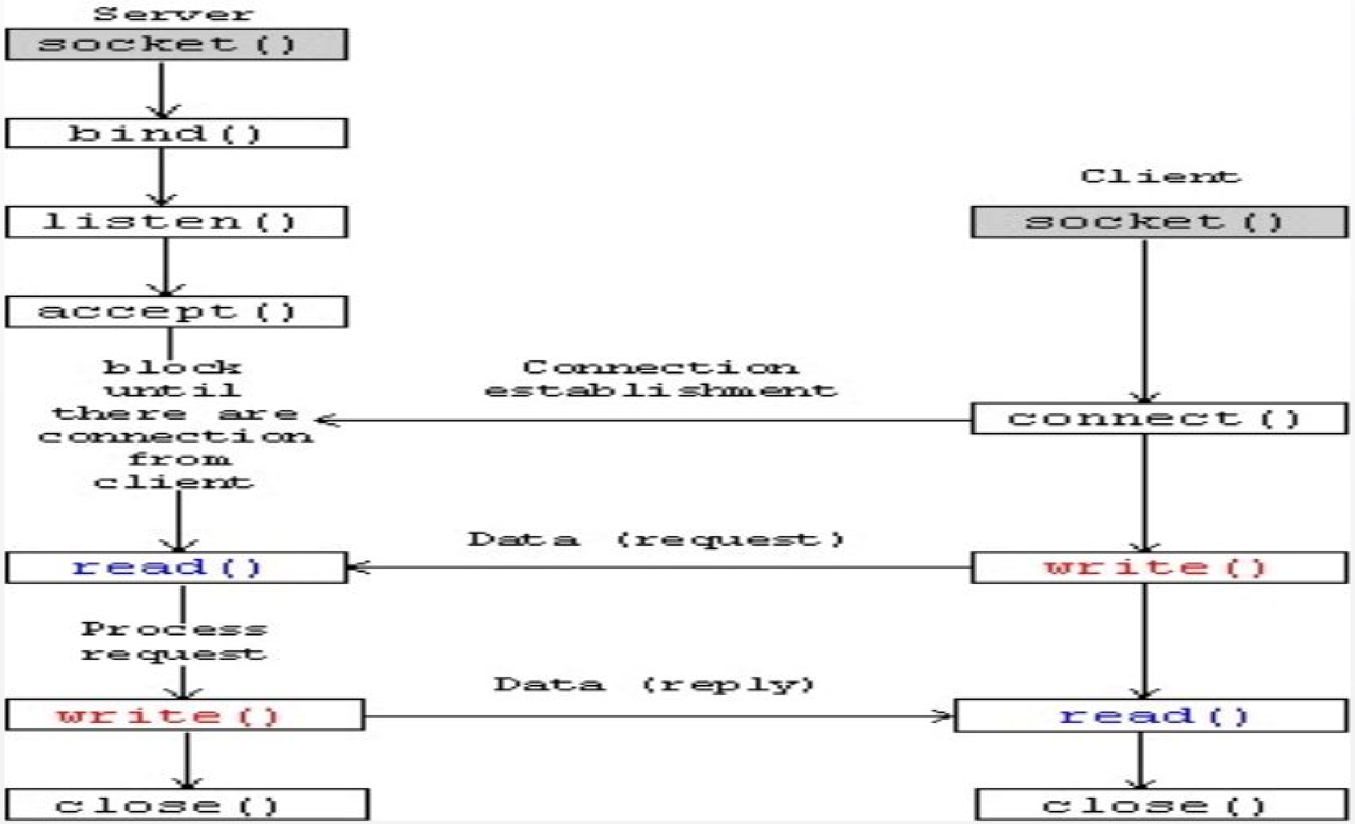


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake) SYN/SYN+ACK
LISTEN
RST/-
SEND/SYN
SYN RECEIVED
SYN/SYN+ACK (simultaneous open)
SYN SENT
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/ FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
(Go back to start)
CLOSED

Implementing a server


Client-Server Interaction via TCP
Server (running on **hostid**)
create socket, port=x
for incoming request:
welcomeSocket = ServerSocket()
Client
TCP connection setup
wait for incoming connection request
create socket
Connect to hostid, port = x
clientSocket = Socket()
conncectionSocket = welcomeSocket.accept()
read request from connectionSocket
send request using clientSocket
write reply to connectionSocket
read reply from clientSocket
close connectionSocket
close clientSocket

Sockets

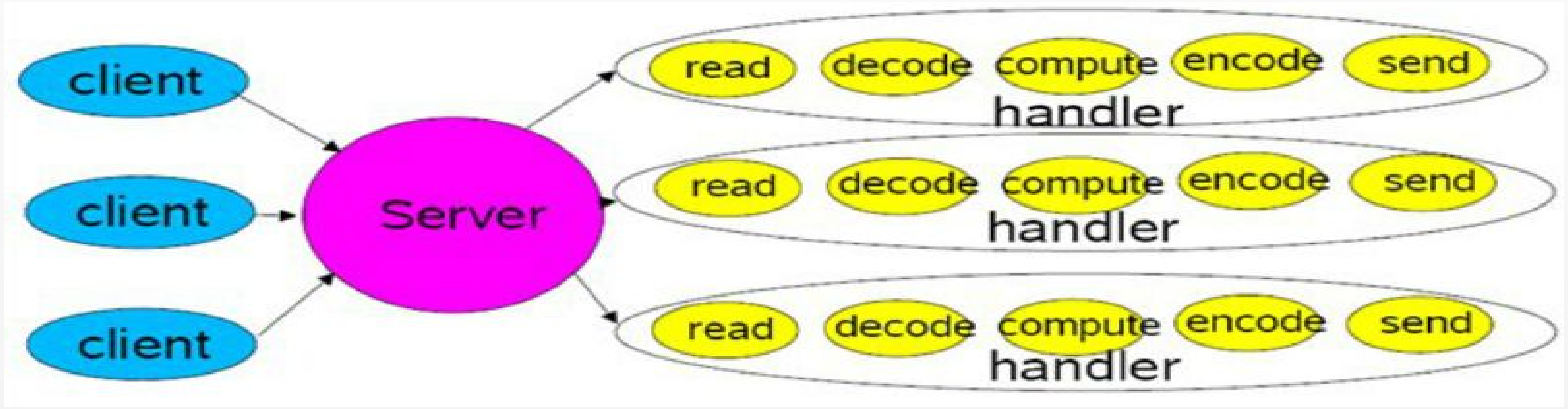

Server
socket()
bind()
listen()
accept()
block until there are connection from client
read()
Process request
write()
close()
Client
socket()
connect()
write()
read()
close()
Connection establishment
Data (request)
Data (reply)

Server with Multithreading support

Code

- knock/singleThread
- knock/multiThreded

Each Handler starts in its own thread



Multithreaded model

synchronous: you handle one request at a time, each in turn.

pros: simple

- ***cons***: any one request can hold up all the other requests

fork: you start a new process to handle each request.

- ***pros***: easy
- ***cons***: does not scale well, hundreds of connections means hundreds of processes.
- fork() is the Unix programmer's hammer. Because it's available, every problem looks like a nail. *It's usually overkill*

threads: start a new thread to handle each request.

- ***pros***: easy, and kinder to the kernel than using fork, since threads usually have much less overhead
- ***cons***: threaded programming can get very complicated very fast, with worries about controlling access to shared resources

I/O ???

- I/O -- or input/output -- refers to the interface between a computer and the rest of the world, or between a single program and the rest of the computer.
- It is such a crucial element of any computer system that the bulk of any I/O is actually built into the operating system. Individual programs generally have most of their work done for them.
- In Java programming, I/O has until recently been carried out using a stream metaphor. All I/O is viewed as the movement of single bytes, one at a time, through an object called a stream. Stream I/O is used for contacting the outside world. It is also used internally, for turning objects into bytes and then back into objects.

NIO

NIO was created to allow Java programmers to implement high-speed I/O without having to write custom native code. NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a great increase in speed.

HEADACHE !!!

```
public class HelloWorld
{
   public static void main (String [] argv)
   {
         System.out.println ("Hello World");
   }
}
```

```
import java.nio.ByteBuffer;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;

public class HelloWorldNio
{
   public static void main (String [] argv)
         throws Exception
   {
         String hello = "Hello World" + System.getProperty ("line.separator");
         ByteBuffer bb = ByteBuffer.wrap (hello.getBytes ("UTF-8"));
         WritableByteChannel wbc = Channels.newChannel (System.out);

         wbc.write (bb);
         wbc.close();
   }
}
```

Aur kya deti hain ?

New Abstractions

- Buffers
- Channels
- Selectors

New I/O Capabilities

- Non-Blocking Sockets
- Readiness Selection
- File Locking
- Memory Mapped Files

New Non-I/O Features

- Regular Expressions
- Pluggable Charset Transcoders

Kaise deti hain ?

- Buffers
  - Data container objects
- Channels
  - Transfer data between buffers and I/O services
  - Channels and Buffers are the central objects in NIO, and are used for just about every I/O operation. Channels are analogous to streams in the original I/O package. All data that goes anywhere (or comes from anywhere) must pass through a Channel object. A Buffer is essentially a container object. All data that is sent to a channel must first be placed in a buffer; likewise, any data that is read from a channel is read into a buffer.
- Selectors
  - Provide status information about channels
- Regular Expressions (DIY)
  - Perform pattern matching against character sequences
- Character Set Coding (DIY)
  - Perform encoding/decoding of character sequences to/from byte streams

Buffer

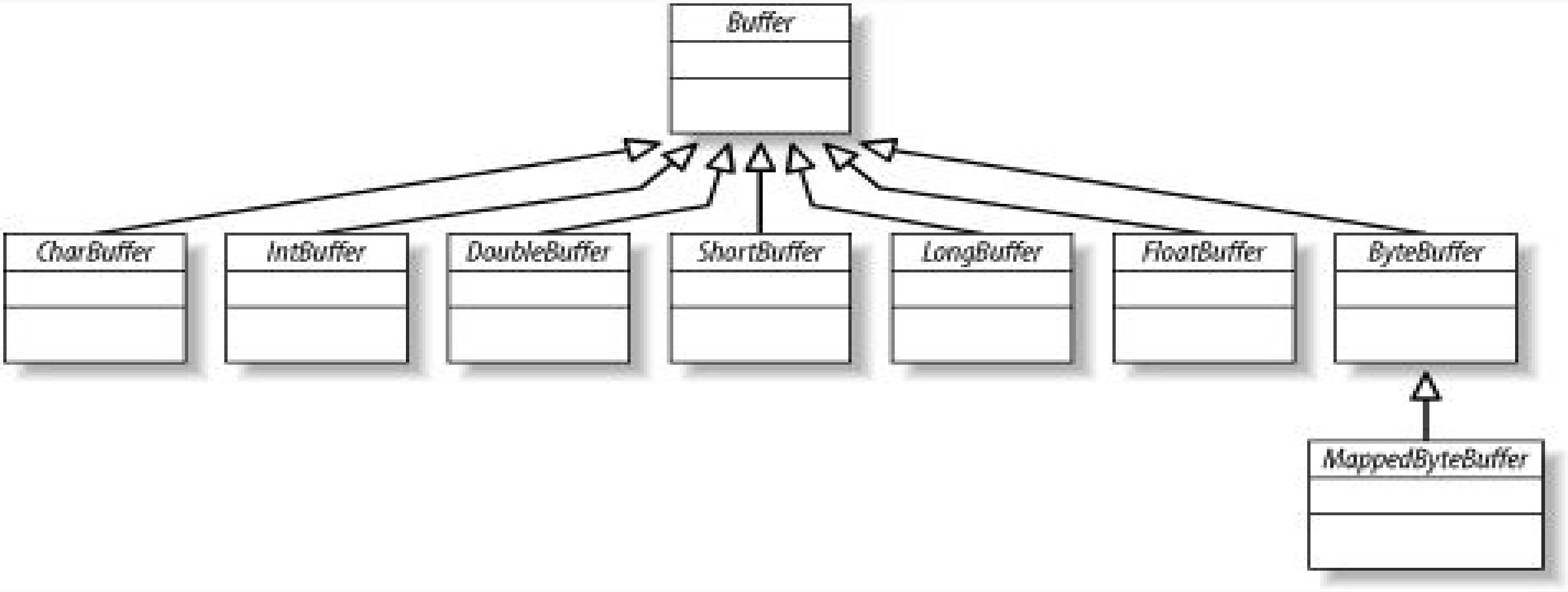
- A Buffer is an object, which holds some data, that is to be written to or that has just been read from.

- The addition of the Buffer object in NIO marks one of the most significant differences between the new library and original I/O. In stream-oriented I/O, you wrote data directly to, and read data directly from, Stream objects.

- In the NIO library, all data is handled with buffers. When data is read, it is read directly into a buffer. When data is written, it is written into a buffer. Anytime you access data in NIO, you are pulling it out of the buffer.

- A buffer is essentially an array. Generally, it is an array of bytes, but other kinds of arrays can be used. But a buffer is more than just an array. A buffer provides structured access to data and also keeps track of the system's read/write processes.

Buffer Types



code

buffer/CreateBuffer

buffer/TypesInByteBuffer

buffer/FastCopyFile

Channel

- A Channel is an object from which you can read data and to which you can write data. Comparing NIO with original I/O, a channel is like a stream. As previously mentioned, all data is handled through Buffer objects.

- You never write a byte directly to a channel; instead you write to a buffer containing one or more bytes. Likewise, you don't read a byte directly from a channel; you read from a channel into a buffer, and then get the bytes from the buffer.

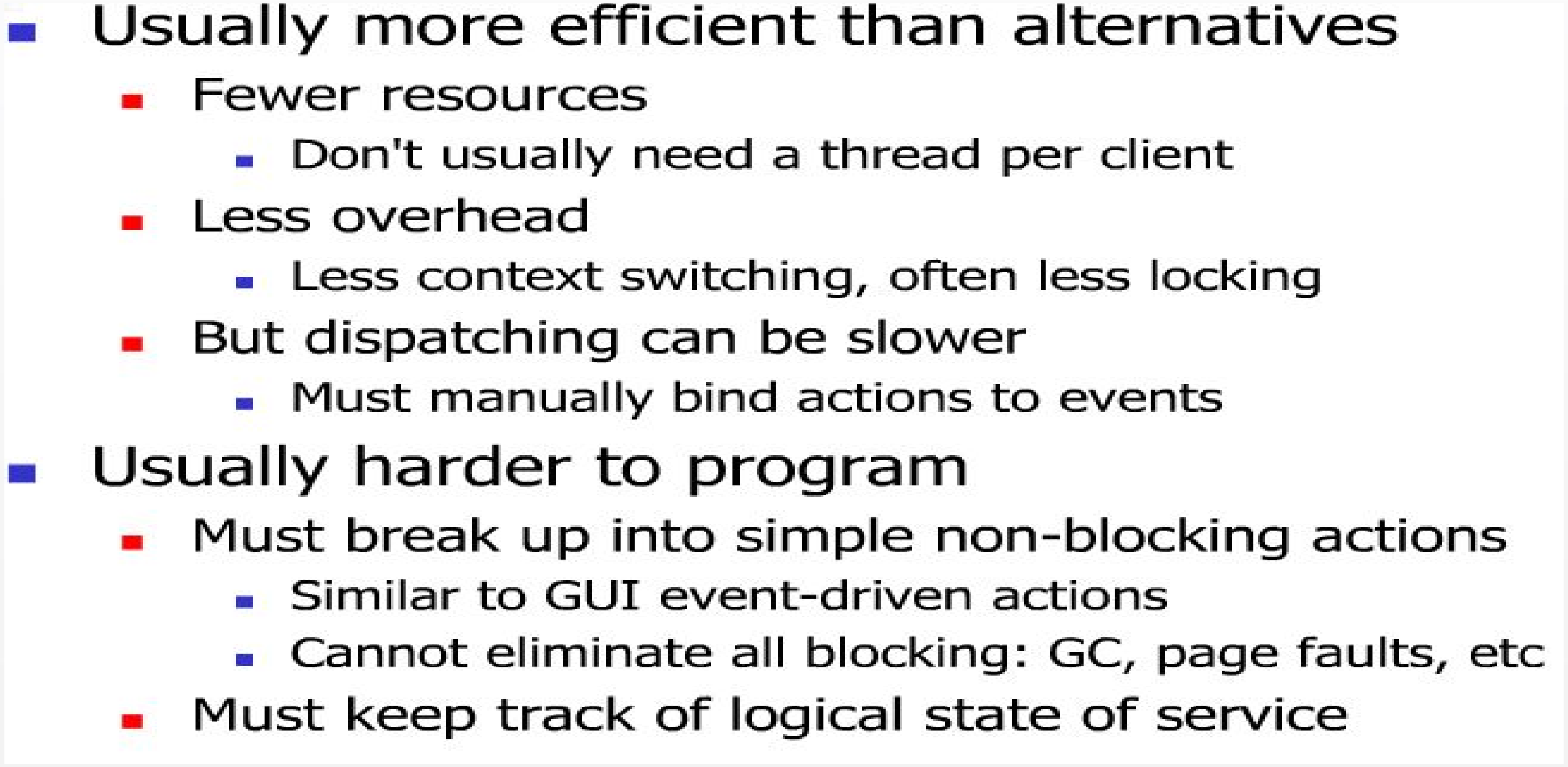
- Channels differ from streams in that they are bi-directional. Whereas streams only go in one direction (a stream must be a subclass of either InputStream or OutputStream), a Channel can be opened for reading, for writing, or for both.

- Because they are bi-directional, channels better reflect the reality of the underlying operating system than streams do. In the UNIX model in particular, the underlying operating system channels are bi-directional.

code

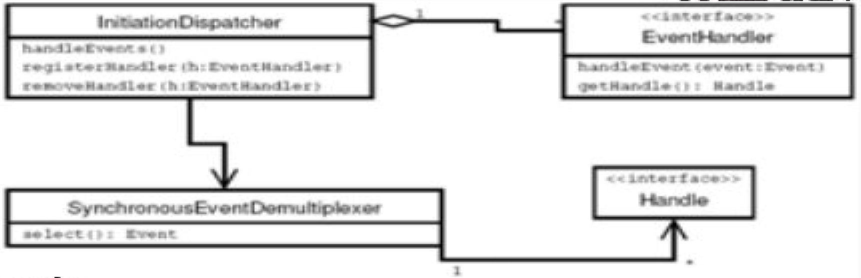
- nio/channel/ReadAndShow
- nio/channel/WriteSomeBytes

Event Driven Model



- Usually more efficient than alternatives
  - Fewer resources
    - Don't usually need a thread per client
  - Less overhead
    - Less context switching, often less locking
  - But dispatching can be slower
    - Must manually bind actions to events
- Usually harder to program
  - Must break up into simple non-blocking actions
    - Similar to GUI event-driven actions
    - Cannot eliminate all blocking: GC, page faults, etc
  - Must keep track of logical state of service

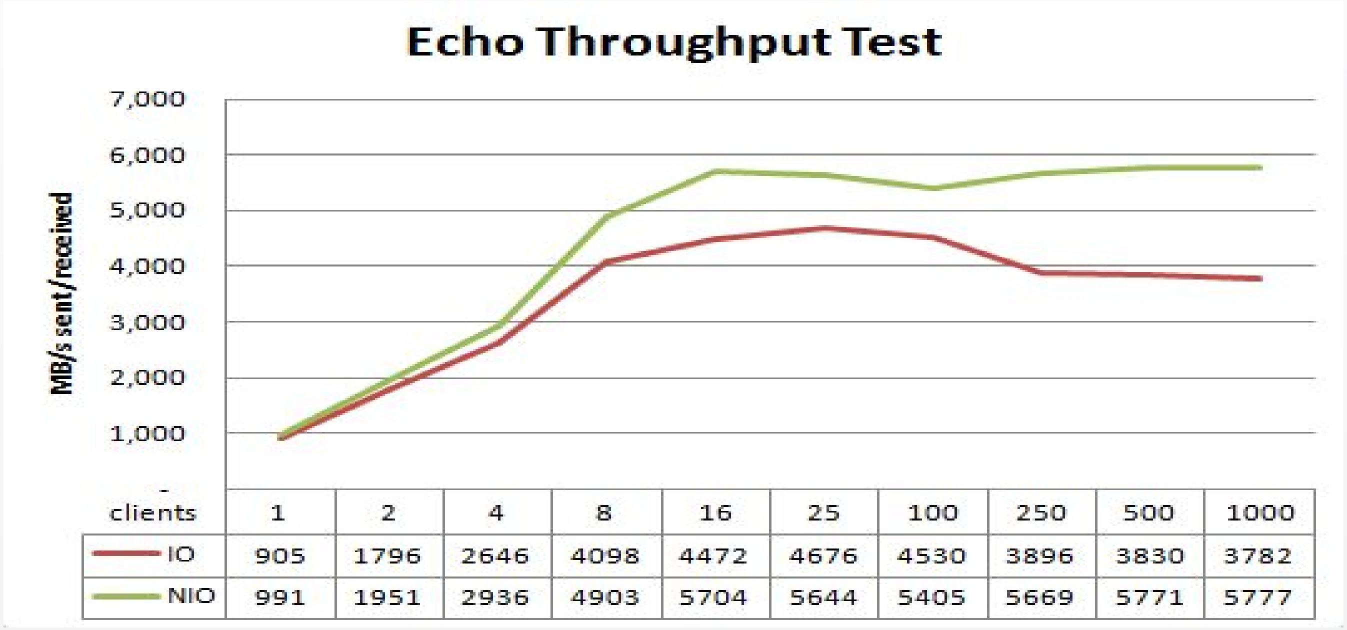
Reactor Pattern Structure



- Handle
  - Receives events; E.g. a network connection, timer, user interface device
- Synchronous Event Demultiplexer
  - select() waits until an event is received on a Handle and returns the event.
  - Often implemented as part of an operating system.
- Initiation Dispatcher
  - Uses the Synchronous Event Demultiplexer to wait for events.
  - Dispatches events to the Event Handlers.
- Event Handler
  - Application-specific event processing code.

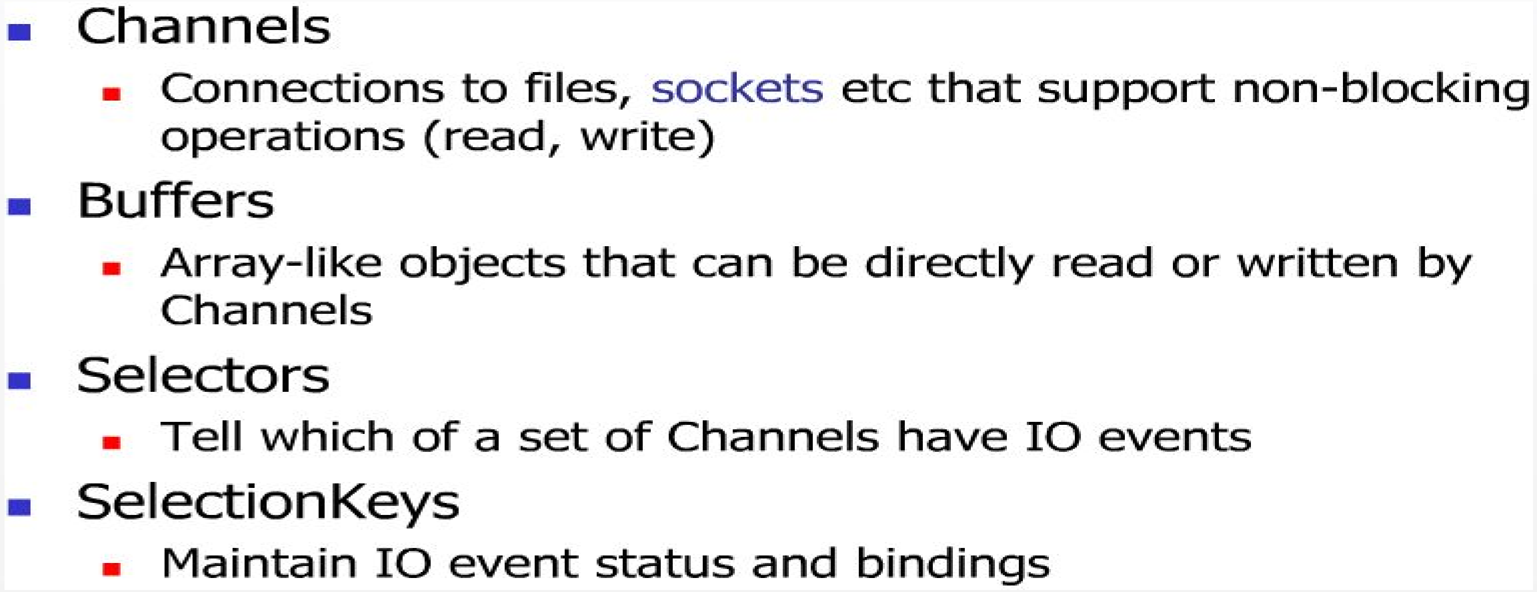
- Setup
  - Create Initiation Dispatcher.
  - Register Event Handlers with Initiation Dispatcher.
- Main loop
  - Call handleEvents in Initiation Dispatcher repeatedly.
  - Initiation Dispatcher calls select in Synchronous Event Demultiplexer, blocking until an event is received.
  - The Initiation Dispatcher calls handleEvent in the corresponding Event Handler, passing it the event.
- End
  - Unregister Event Handlers from Initiation Dispatcher.

Small benchmark



clients	1	2	4	8	16	25	100	250	500	1000
🟥 IO	905	1796	2646	4098	4472	4676	4530	3896	3830	3782
🟩 NIO	991	1951	2936	4903	5704	5644	5405	5669	5771	5777

Non-blocking Socket Implementation



- Channels
  - Connections to files, sockets etc that support non-blocking operations (read, write)
- Buffers
  - Array-like objects that can be directly read or written by Channels
- Selectors
  - Tell which of a set of Channels have IO events
- SelectionKeys
  - Maintain IO event status and bindings

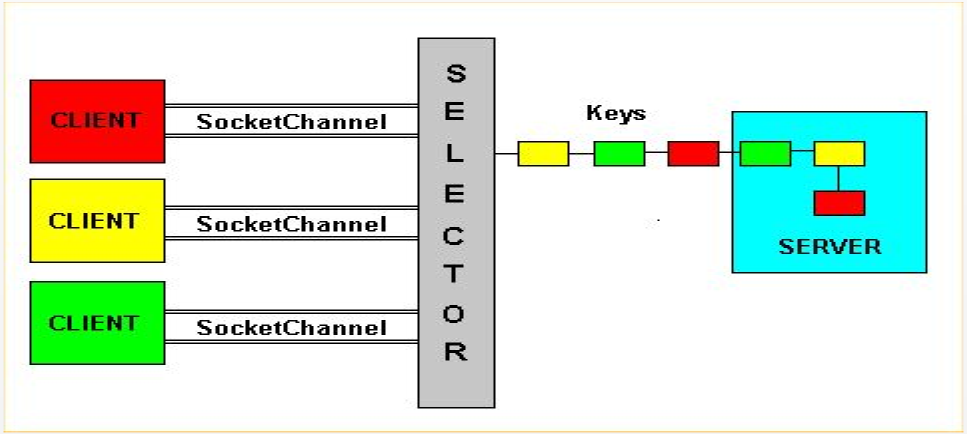
code

- nio/ConnectAsync

Non-blocking System Model

- Server: the application receiving requests.
- Client: the set of applications sending requests to the server.
- Socket channel: the communication channel between client and server. It is identified by the server IP address and the port number. Data passes through the socket channel by buffer items.
- Selector: the main object of all non-blocking technology. It monitors the recorded socket channels and serializes the requests, which the server has to satisfy.
- Keys: the objects used by the selector to sort the requests. Each key represents a single client sub-request and contains information to identify the client and the type of the request.

Non-blocking socket architecture



code

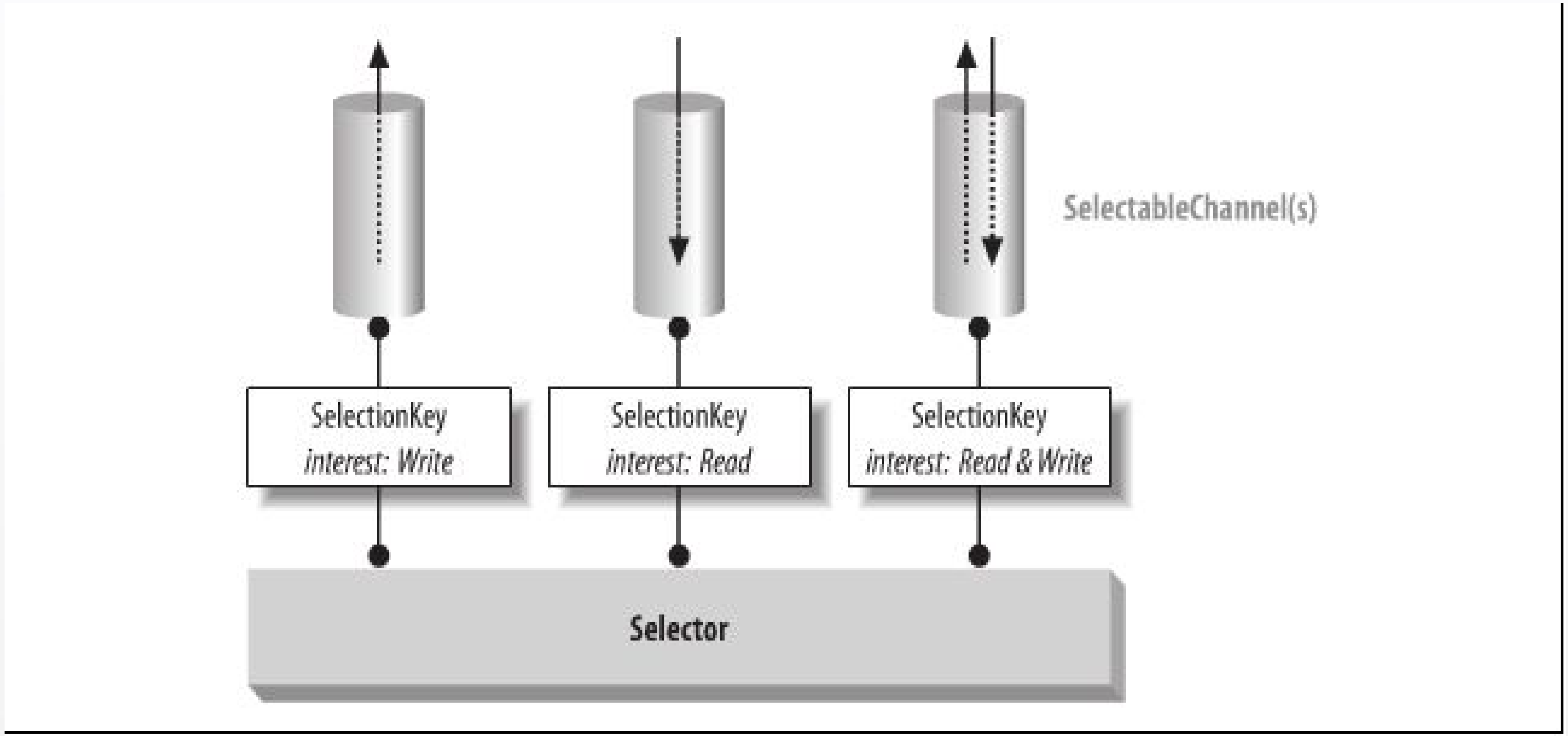
- nio/EchoServer
  - nc localhost 12345

Channels

- They can operate in non-blocking mode and are selectable
- Channel represents specific connection to a specific I/O service and encapsulates the state of that connection
- Buffers are the internal endpoints used by channel to send and receive data

Selectors

- This class manages information about a set of registered channels and their readiness status
- Each instance of Selector can monitor more socket channels, and thus more connections
- When something interesting happens on the channel, the selector informs the application to process the request



Relationships of the selection classes

Code test

- Are selectors working ???
- nc `whatsmyip` 12345
  - Bomb away

Selection keys

- Key represents the registration of particular channel object with a particular selector object.
- Interest
  - OP_READ
  - OP_WRITE
  - OP_CONNECT
  - OP_ACCEPT

The Selection Process

- Registered key set
  - Set of currently registered keys associated with selector

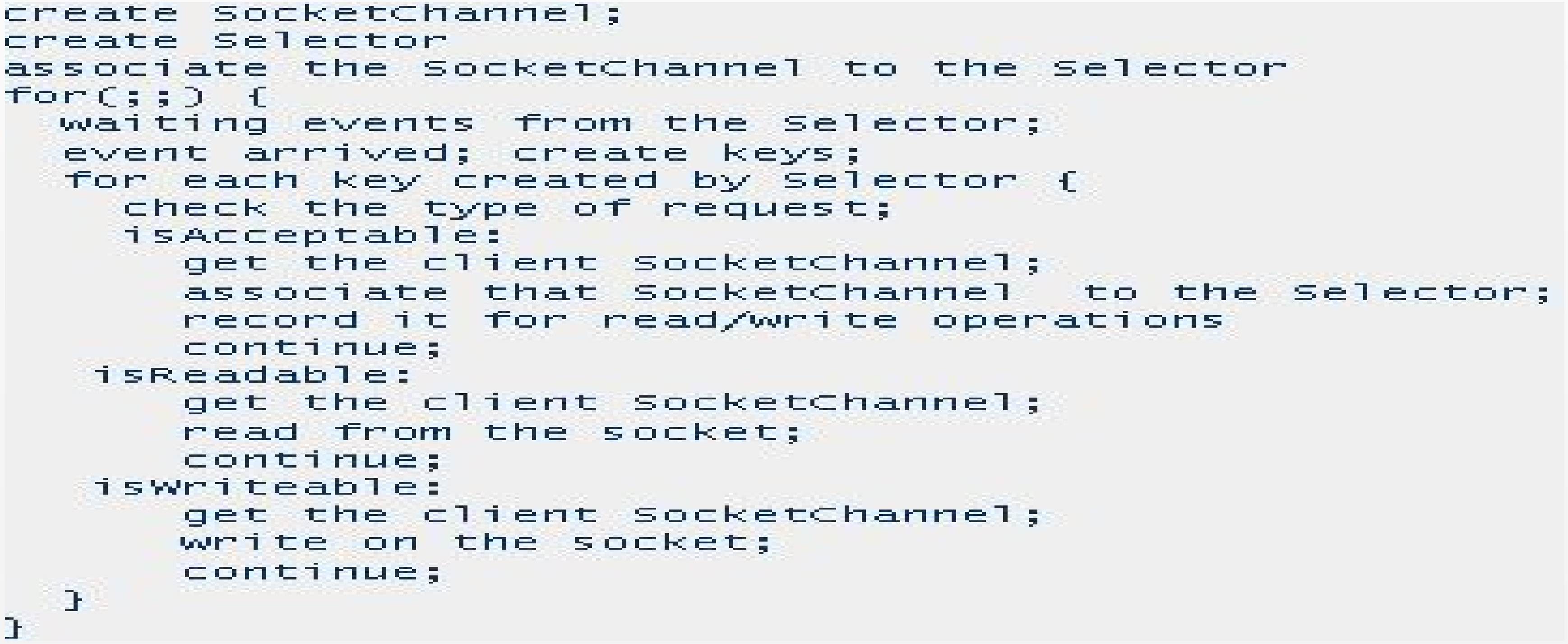
- Selected key set
  - key whose associated channel was determined to be ready for at least one of the operations in the key’s interest

- Cancelled key set
  - Keys whose cancel() method have been called

The Selection Process

- Selector class’s select() method
  - Public abstract int select() throws IOException;
    - This call block indefinitely if no channels are ready but it can return 0 if the wakeup() method of the selector is invoked by another thread
  - Public abstract int select (long timeout) throws IOException;
    - Limit the amount of time a thread will wait for a channel to become ready
  - Public abstract int selectNow() throws IOException;
    - This is totally non-blocking, If no channel is currently ready, it immediately returns 0

General algo of non-blocking server



```
create SocketChannel;
create Selector
associate the SocketChannel to the Selector
for(;;) {
  waiting events from the Selector;
  event arrived; create keys;
  for each key created by Selector {
    check the type of request;
    isAcceptable:
      get the client SocketChannel;
      associate that SocketChannel  to the Selector;
      record it for read/write operations
      continue;
   isReadable:
      get the client SocketChannel;
      read from the socket;
      continue;
   isWriteable:
      get the client SocketChannel;
      write on the socket;
      continue;
  }
}
```

‘Event-driven' model

- *pros*:
  - efficient and elegant
  - scales well - hundreds of connections means only hundreds of socket/state objects, not hundreds of threads or processes.

- *cons*:
  - more complex - you may need to build state machines.
  - requires a fundamentally different approach to programming that can be confusing at first

Use kab karoon ?

- Move large amounts of data efficiently
  - NIO is primarily block oriented – **java.io** uses streams
  - *Direct* buffers to do raw, overlapped I/O – bypassing the JVM

- Multiplex large numbers of open sockets
  - NIO sockets can operate in non-blocking mode
  - One thread can manage huge numbers of socket channels
  - Better resource utilization

- Use OS-level file locking or memory mapping
  - Locking: Integration with legacy/non-Java applications
  - Mapped Files: Non-traditional I/O model - leverages virtual memory

- Do custom character set Transcoding
  - Control translation of chars to/from byte streams

- **Efficiency – The Need For Speed**
  - Byte/char-oriented pipelines are flexible but relatively inefficient
  - The OS provides high-**performance** I/O services - the JVM gets in the way

- **Scalability – Livin' Large**
  - Big applications have big appetites, they consume large amounts of data
  - Traditional method of handling large numbers of I/O streams does not scale
  - Multiplexing can only be done effectively with OS support

- **Reliability – Less Code Written = Fewer Bugs**
  - These I/O needs are generic and should be provided by the Java platform
  - Application programmers should write application code, not infrastructure

- **No Longer CPU Bound**
  - Moving data has become the bottleneck, not bytecode execution speed

- **JSR 51** (http://www.jcp.org/en/jsr/detail?id=51)
  - Requested I/O features widely available on most OSs but missing from Java

Java NIO Projects

- Netty -
  - **Brilliant abstractions for NIO**

- Mina -
- Grizzly -

Listening ..... aNy-I/O

code

- Write a chat bot using NIO !!!

FIN/ACK ?