TCP - Socket communication

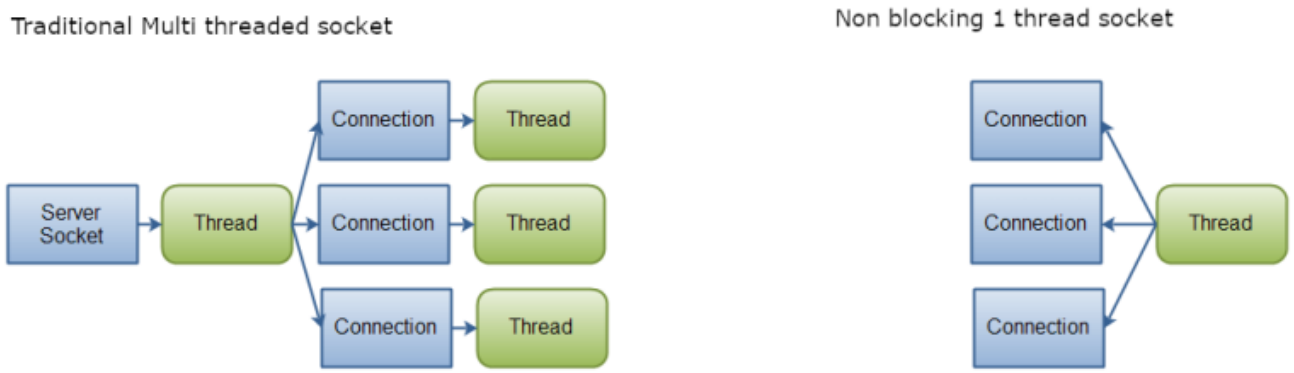
The client sends requests to the server over a TCP socket connection, and the server responds to these requests. Here are the basic steps involved in integrating software systems or components using TCP socket communication:

1. **Select a protocol**: TCP/IP is a common protocol for socket communication, but other protocols like UDP can also be used depending on the requirements.
2. **Determine the message format**: Decide on the format of the messages that will be exchanged between the client and server. This could be a simple text-based format or a more complex binary format.
3. **Define the communication interface**: Define the functions or APIs that will be used for communication between the client and server.
4. **Create the server**: Write the code for the server that listens for incoming client connections and handles incoming requests.
5. **Create the client**: Write the code for the client that connects to the server and sends requests.
6. **Handle errors**: Implement error handling mechanisms to ensure that communication errors are handled gracefully and do not cause the system to crash or become unstable.
7. **Test and iterate**: Test the system thoroughly and make any necessary changes or improvements to ensure that it is functioning correctly.

Features:

- Socket ::= IP address + (TCP/UPD) port number. A Socket is a combination of ip address and port number.
- TCP Sockets provides 'real-time' data transfer
    - binary data transfer but can be normal text or JSON, XML as well
    - no direct method sharing (can be implemented by hand)
    - TCP and UDP connections are possible. UDP is min 3 times quicker but one-way communication
- Persistent or On-Demand communication channel
    - because of connection time-loss usually persistent channels are better, but periodically 'ping' messages should be sent. (in order to avoid connection closing). In case of any problems reconnection is possible
    - in case of UDP channels an extra TCP channel is available for synchronizing - in online games
- Results in the fastest possible transmission:
    - Where the number of transactions per second up to ~ 50 transactions, there should have been applied. (20ms / sec transfer)

Blocking and non-blocking TCP sockets in Java



Blocking and non-blocking TCP sockets differ mainly in handling input/output (I/O) operations, particularly regarding how a program's execution flow is managed when waiting for operations to complete. Here's a breakdown of the main differences:

Blocking sockets

Execution Flow: In the case of blocking sockets, when a socket operation (like recv or send) is called, the program's execution is “blocked” until the operation completes. For example, if you call recv to read data from a socket, the call will wait forever until data is received.

Ease of Use: Blocking sockets are straightforward to use and understand because the operations appear sequential and synchronous. The program will not proceed until the current operation finishes, simplifying the logic, especially for simple networked applications.

Performance Considerations: While *blocking sockets* are easier to work with, they can lead to inefficient use of resources. For instance, if a server implemented with blocking sockets handles multiple connections, it would need *to spawn a new thread or process for each connection to avoid one operation blocking the entire server*.

Non-Blocking sockets

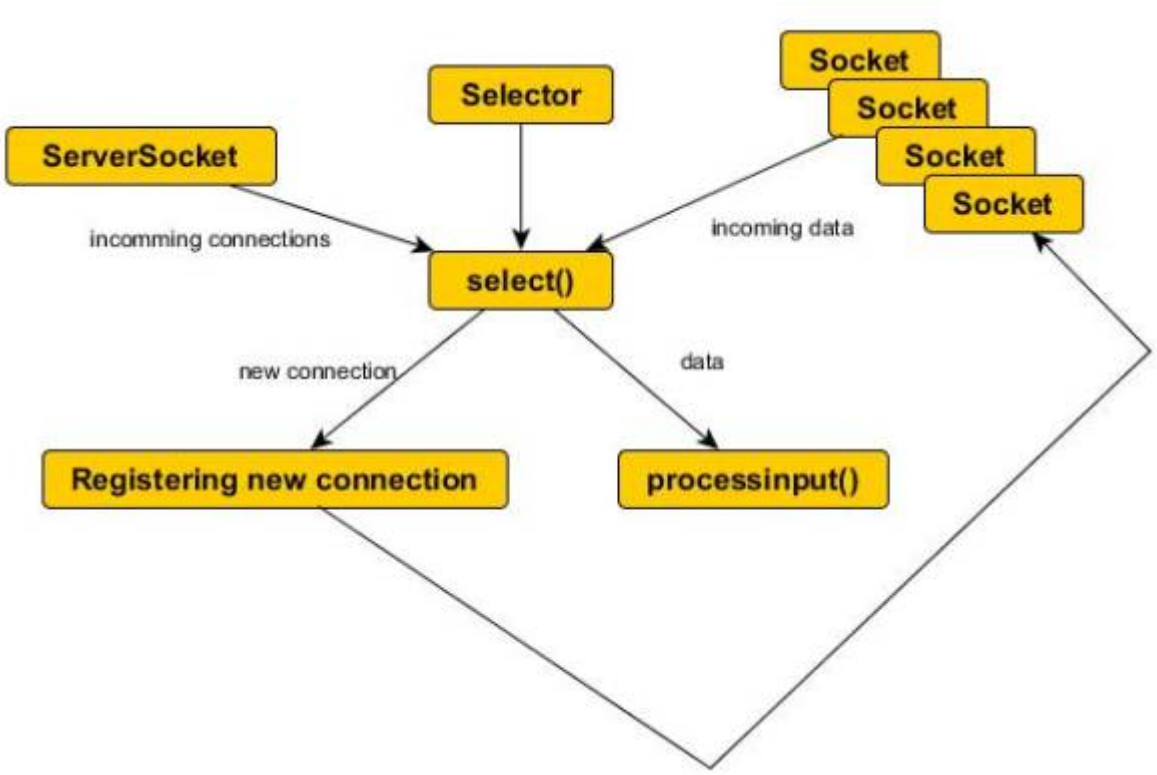
Execution Flow: Non-blocking sockets, on the other hand, allow the program's execution to continue immediately, even if the socket operation cannot be completed at that moment.

Complexity: Because the program continues to run even when data is not immediately available, using non-blocking sockets can lead to more complex program structures. Developers often use event loops or select/poll mechanisms to efficiently manage these sockets, especially when handling multiple connections simultaneously.

Performance and Scalability: Non-blocking sockets can lead to more efficient and scalable applications. A single process or thread can manage multiple socket connections without spawning new threads or processes for each connection, using system resources better and enabling the server to handle many connections concurrently.

Reading:

- http://tutorials.jenkov.com/java-nio/nio-vs-io.html
- http://www.javaworld.com/article/2073344/core-java/use-select-for-high-speed-networking.html



Non-blocking loop

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);  // non blocking enabled

while(true){
 SocketChannel socketChannel = serverSocketChannel.accept();

 if(socketChannel != null){
     // the connection is accepted
 }
}
```