**Non-Blocking IO (Java NIO)**

Let's understand the different methods a Server can handle load/connections:

1. Simple blocking server (everything handled by Main Thread)
2. Threaded blocking server (Every req handled by a different Thread)
3. Threaded pool blocking servers(Using executorService to handle Requests)
4. Java NIO blocking(Single-threaded model, using NIO API)
5. Java NIO non-blocking pooling (Polling at every socket so see if anything coming)
6. Java NIO non-blocking selector

# Simple Blocking Server

Let's say we have a simple Server, which inverts the case and returns back the changed output.

To test this server, telnet into this port and write any input. You will receive the Output with the inverted case:

**telnet localhost 8080> hElLoHeLlO**

*Issues with this server:  
-****accept() is blocking****, so it can not do anything until the socket connection is opened.  
- Say we have multiple telnet connections opened, but only****ONE****will be able to communicate and open socket. Others can only do so one the first one is closed.****THIS IS A VERY BAD SERVER.***

# Threaded blocking server

As you might have guessed, to handle the multiple connections, we can use different threads to do so:

Here the handle method runs on a different Thread.  
Now multiple telnet connections can send and receive output.

*Issues with this:  
- Say if we have many high numbers of connections, the server may crash/go outOfMemory/etc. Also, the number of threads we can create for a process is limited(even if they are idle).  
So there is still a limited number of connections we can make.*

Example java program to test this by creating multiple connections:

|  |
| --- |
| import java.io.IOException; |
|  | import java.net.Socket; |
|  |  |
|  | public class SimulateHeavyLoad { |
|  | public static void main(String[] args) throws InterruptedException { |
|  | final Socket[] sockets = new Socket[3000]; // Create 3000 connections |
|  | for (int x = 0; x < sockets.length; x++) { |
|  | try { |
|  | sockets[x] = new Socket("localhost", 8080); |
|  | } catch (final IOException e) { |
|  | System.err.println(e); |
|  | } |
|  | } |
|  | // To enable the sockets to live. for awhile |
|  | Thread.sleep(Long.MAX\_VALUE); |
|  | } |
|  | } |

Run this while the ThreadedSingleserver is running, and it may cause the server to crash, with such a high number of simultaneous connections.

# Thread Pool Blocking Server

As you might have guessed again: instead of creating the threads ourselves, let's leave it to the Java API which reuses a given number of threads. Any new operations beyond that number will wait in the queue until a thread is active again.  
**Thread pool executor Service**is such an API.

*Issues with this:  
- Although the server will never crash for OutOfMemory, only a fixed number of connections can sustain at a given time(as we have given a fixed number of threads).  
i.e. the new connection will be blocked until the thread frees up.  
This blocking issue is solved by java NIO.*

# NIO(New IO) Blocking Server

This uses different methods to open Socket.  
Also, the Input/Output streams are not familiar to this and instead rely on **Byte Buffers**(a very good thing to learn- would be writing on ButeBuffers very shortly).

***Byte Buffer:***

*It is Bytes with a bunch of other data inside. It knows how much data is inside it.  
So we start reading from position=0 and limit=80(as defined while allocating it).*

*Say it has content = “hello”, now its position is at 5 and limit=80 still.*

*To read, we set position=0 and limit=5. This is done using****flip()****method.*

*After the entire content is read, and written to the socket, we reset the position and limit to 0 and 80 resp. This is done using****compact()****method.*

*To perform the operation on the ByteButter, read everybyte from pos->limit and perform the operation on every byte.*

This uses NIO API, but it's the same as the first one- Single blocking server. It can only accept a single connection at a time. As everything is happening in Main Thread.

# Java NIO non-blocking pooling (BAD Approach)

Here the server polls at all the sockets to listen if they want to say anything. But it's bad as the number of sockets inc, it will screw the performance.

NIO has **configureBlocking**API and setting it false makes it nonBLocking, but an issue with this is that now accept() may return null, as there may not be anything on the socket always.

This will work for multiple connections but is a really very bad approach.

# NIO non-blocking selector

This is the best of all the Approaches we have discussed. It is non-blocking and it only operates once a particular event is received to do on the socket.

This wakes up(the thread handling the request), whenever something comes up on any socket(instead of us polling every socket).  
So we register the serverSocketChannel with the selector(which can accept the connection and sends callback).

So we register the events we are interested in.  
Say Socket1 is ready for connection, the Accept event(selectionKey==ACCEPT) will be received.  
On **accept**key, we accept the socket and create the connection, then we listen to the READ event for this socket.  
Now say Socket1 has to say anything, the **Read**event(selectionKey==Read) will be received. On Read key, we read the input in that socket, and perform the operation on it. Now to write it to the socket, we listen to write Operation.  
Now say Socket1 is ready to take the output, the **Write**event(selectionKey==Write) will be received. We write back the output which Read prepared before.

NOI stands for Network Operations and Intelligence. It is a term used in the field of networking to describe the process of monitoring, analyzing, and optimizing network performance and operations.

Network Operations refers to the activities and tasks involved in managing and maintaining a network infrastructure. This includes tasks such as configuring network devices, monitoring network health and performance, troubleshooting network issues, and ensuring network security.

Network Intelligence refers to the ability to gather and analyze data from the network to gain insights and make informed decisions. This involves collecting network data, such as traffic flows, packet statistics, and device performance metrics, and using analytical tools and techniques to interpret and extract meaningful information from the data.

NOI combines these two aspects, network operations and network intelligence, to provide a comprehensive approach to managing and optimizing network performance. By monitoring and analyzing network data, network administrators can identify performance bottlenecks, detect security threats, and make informed decisions to improve network efficiency and reliability.

NOI solutions often include network monitoring tools, analytics platforms, and automation capabilities to streamline network management processes and enhance network performance. These solutions can be particularly valuable in large-scale enterprise networks or service provider environments where network uptime and performance are critical.

NOI implemented in JAVA

ChatGPT

Implementing a complete Network Operations and Intelligence (NOI) solution in Java would be a complex task that requires a deep understanding of networking protocols, Java programming, and various libraries and frameworks. However, I can provide you with a high-level overview of how you could approach implementing certain components of a NOI system in Java. Here are a few key aspects to consider:

1. Network Monitoring:
   * Use Java libraries such as pcap4j or jNetPcap to capture network traffic.
   * Analyze captured packets using libraries like jNetFlow or Apache Kafka for flow analysis.
   * Extract relevant information from packets, such as source/destination IP addresses, ports, and protocols.
   * Store the collected data in a database or a data store for further analysis.
2. Data Analysis:
   * Utilize data processing and analysis frameworks such as Apache Spark or Apache Flink to process the captured network data.
   * Develop algorithms or use machine learning techniques to extract insights from the network data.
   * Perform anomaly detection, traffic pattern analysis, or security threat detection.
3. Visualization and Reporting:
   * Use Java-based visualization libraries like JavaFX or Swing to create user interfaces for displaying network performance metrics, alerts, and reports.
   * Generate visualizations, charts, and graphs to present network statistics and trends.
   * Develop reporting functionalities to provide detailed reports on network performance, security incidents, or any other relevant metrics.
4. Automation and Management:
   * Implement automated tasks and workflows using Java frameworks like Spring or Apache Camel to streamline network management processes.
   * Develop automation scripts to configure network devices, deploy network changes, or perform routine network operations.
   * Integrate with network management systems (NMS) or orchestration platforms to enable centralized control and management of the network.

It's important to note that implementing a complete NOI solution in Java requires significant expertise in networking, Java programming, and data analysis. Additionally, you may need to leverage additional libraries, frameworks, or technologies depending on the specific requirements of your NOI system.

Non-blocking I/O (input/output) is a programming approach that allows a program to continue executing other tasks without waiting for the completion of I/O operations. It enables asynchronous processing and is commonly used in scenarios where the program needs to perform multiple I/O operations concurrently or where responsiveness is crucial.

In traditional blocking I/O, when a program initiates an I/O operation, it typically waits until the operation completes before moving on to the next task. This can lead to inefficient resource utilization and delays, especially when dealing with slow or unpredictable I/O devices.

Non-blocking I/O, on the other hand, allows the program to initiate an I/O operation and continue execution immediately without waiting. The program can periodically check the status of the operation or be notified when it completes. This way, it can perform other tasks or initiate additional I/O operations while waiting for the completion of previous ones.

Non-blocking I/O is often implemented using mechanisms such as callbacks, polling, or event-driven architectures. Here are a few commonly used techniques:

1. Callbacks: The program specifies a callback function to be executed when the I/O operation completes. The I/O system invokes the callback, passing the result or error information to the program.
2. Polling: The program periodically checks the status of the I/O operation by querying the I/O system. It can continue execution if the operation is not yet complete and come back to check later.
3. Event-driven architecture: The program registers its interest in specific I/O events with the operating system or a dedicated I/O framework. When an I/O operation completes or a relevant event occurs, the program is notified, and it can handle the event asynchronously.

Non-blocking I/O is particularly useful in scenarios where the program needs to handle multiple concurrent I/O operations efficiently, such as network programming, web servers, or user interfaces that need to remain responsive while performing I/O tasks.

It's worth noting that non-blocking I/O is different from multithreading or multiprocessing. Non-blocking I/O allows a single thread or process to handle multiple I/O operations concurrently without blocking, while multithreading/multiprocessing involves using multiple threads or processes to handle tasks simultaneously, which can also achieve concurrency but with different programming models and trade-offs.

Non blocking IO implemented in JAVA.

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.channels.SocketChannel;

public class NonBlockingExample {

public static void main(String[] args) {

try {

// Create a SocketChannel

SocketChannel socketChannel = SocketChannel.open();

socketChannel.configureBlocking(false); // Set the channel to non-blocking mode

// Connect to a server

socketChannel.connect(serverAddress);

// Check if the connection is complete

if (socketChannel.finishConnect()) {

// Connection successful

ByteBuffer buffer = ByteBuffer.allocate(1024);

buffer.put("Hello, Server!".getBytes());

buffer.flip();

// Write to the channel

while (buffer.hasRemaining()) {

socketChannel.write(buffer);

}

buffer.clear();

// Read from the channel

int bytesRead = socketChannel.read(buffer);

while (bytesRead != -1) {

buffer.flip();

while (buffer.hasRemaining()) {

System.out.print((char) buffer.get());

}

buffer.clear();

bytesRead = socketChannel.read(buffer);

}

// Close the channel

socketChannel.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}} } }

In this example, we create a **SocketChannel** and configure it to use non-blocking mode by calling **configureBlocking(false)**. We then initiate a connection to a server using **connect()**.

We can check if the connection is complete by calling **finishConnect()**. If the connection is successful, we can perform non-blocking I/O operations.

In the provided example, we write a message to the channel using **write()** in a loop until all the data is written. We then read from the channel using **read()** in a loop until we receive the end of the stream (-1). The received data is printed to the console.

Finally, we close the **SocketChannel** using **close()**.

It's important to note that non-blocking I/O in Java NIO typically requires a loop that checks the readiness of the channel for reading or writing using selectors. However, in this simplified example, we use a blocking loop to demonstrate the basic concepts of non-blocking I/O. In practice, you would typically use a **Selector** object to manage multiple channels efficiently.

Top of Form