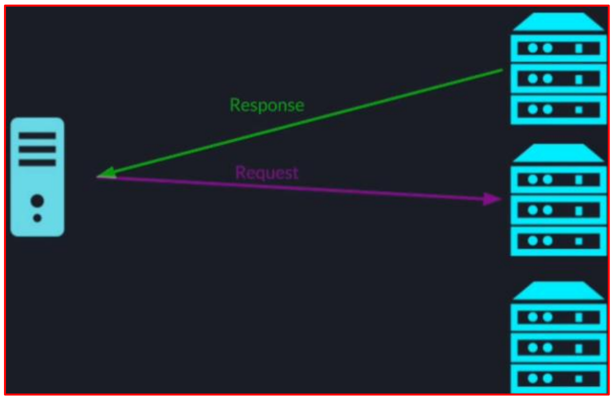
HTTP Client - Implementation & Network Packet Analysis

In this section, we will complete the other side of network communication – the HTTP client. After we're done implementing the HTTP client, we'll test our implementation by sending tasks to multiple worker instances. Finally, we will inspect our network communication using a potent tool called Wireshark.

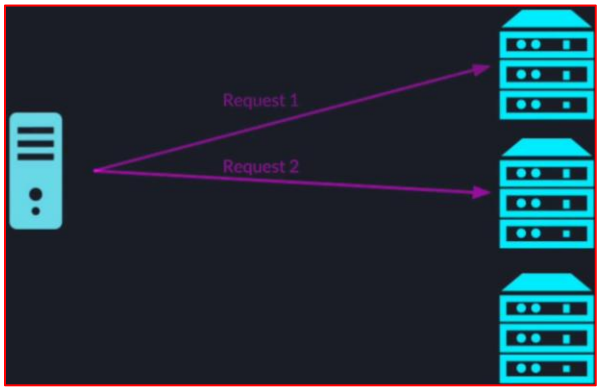
Build HTTP client

Let's dive into building our HTTP client code. In our discussions, we will use the HTTP client that comes built into the JDK, starting with JDK 11. I'd like to point out that there are numerous third-party implementations of both the HTTP server and client, so feel free to choose the right library for your needs when you build your application. The key performance features we care about in an HTTP client are the abilities to send HTTP requests asynchronously and to maintain a connection pool to our downstream servers. Let's talk about these two crucial features in detail.

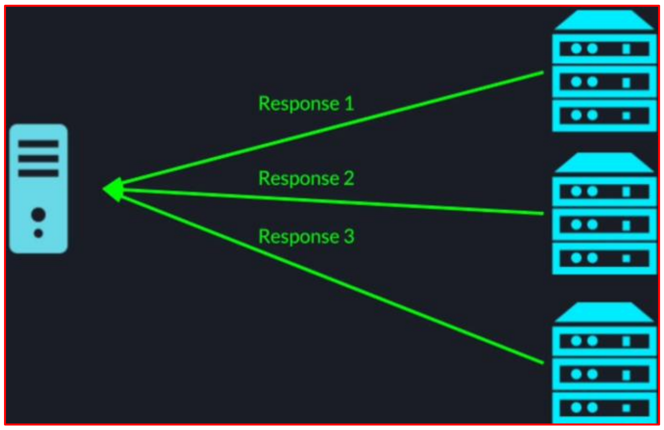
In a *synchronous* communication mode, once we send a request to a server, our thread remains blocked until we receive a response or an error from the server. We can only send the next request to the server once the previous transaction is complete. This approach simplifies writing the code but inhibits our ability to perform multiple tasks in parallel by different nodes.



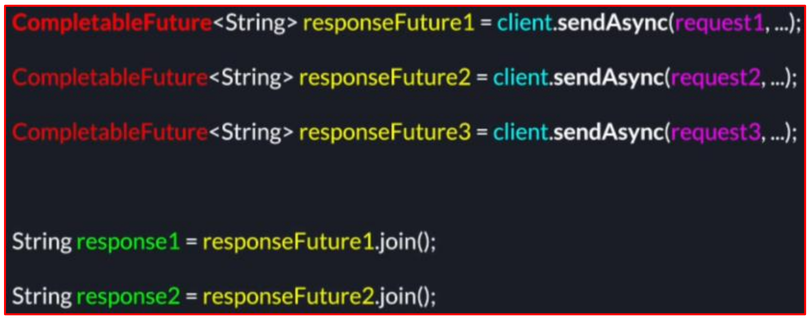
Conversely, in *asynchronous* mode, we can send a request to the server without suspending a thread waiting for the response.



Once we've dispatched all the requests, we wait until all the responses arrive so we can aggregate them.



The JDK's HTTP client enables us to do this easily by calling the `sendAsync` method, which returns immediately and provides us with a `CompletableFuture` representing a future response.



```
CompletableFuture<String> responseFuture1 = client.sendAsync(request1, ...);

CompletableFuture<String> responseFuture2 = client.sendAsync(request2, ...);

CompletableFuture<String> responseFuture3 = client.sendAsync(request3, ...);


String response1 = responseFuture1.join();

String response2 = responseFuture2.join();
```

The second feature vital for performance is Connection Pooling. If we're continually sending requests to the same server, we need to ensure that our HTTP client does not close the connection immediately after the response returns from the server. Otherwise, we incur an unnecessary cost of establishing and closing the connection for every HTTP transaction. Typically, if both the HTTP server and client support HTTP2, we have the connection pooling enabled by default, as the protocol version itself is designed to keep all communication on a single connection at all times.

If at least one of the peers does not support HTTP2, we need to ensure the connection pooling is either enabled by default or that we have to enable it explicitly. Luckily, the built-in HTTP client supports connection pooling with HTTP 1.1 by default. Still, in most third-party implementations, we need to configure it explicitly by setting the *keepalive* parameter to true.

As of JDK 11, the built-in HTTP server does not support HTTP2, so we will see how both entities agree on a protocol version in the case of a mismatch.

Let's switch to the IntelliJ IDE and create the `WebClient` class for our HTTP client. Inside the class, the only private variable we will store is an instance of our HTTP client. When we create our `WebClient` instance, we will instantiate our HTTP client by calling the `newBuilder` method and set the preferred HTTP version to HTTP2. We will keep all the remaining settings as default and build the HTTP client by calling the build method.

```
public WebClient() {
    this.client = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .build();
}
```

Now, the only method we will have in this class is the `sendTask` method. It takes the address where we want to send a request and the HTTP message body in the form of a byte array. We need to create an HTTP request first, so we call the post method and pass it the request message body. We then set the address of the destination and build the request. Finally, we call the `sendAsync` method, which sends our requests asynchronously without waiting for the response.

Once the response arrives, we call the body method to get just the message body without any headers. At this point, our `WebClient` is ready.

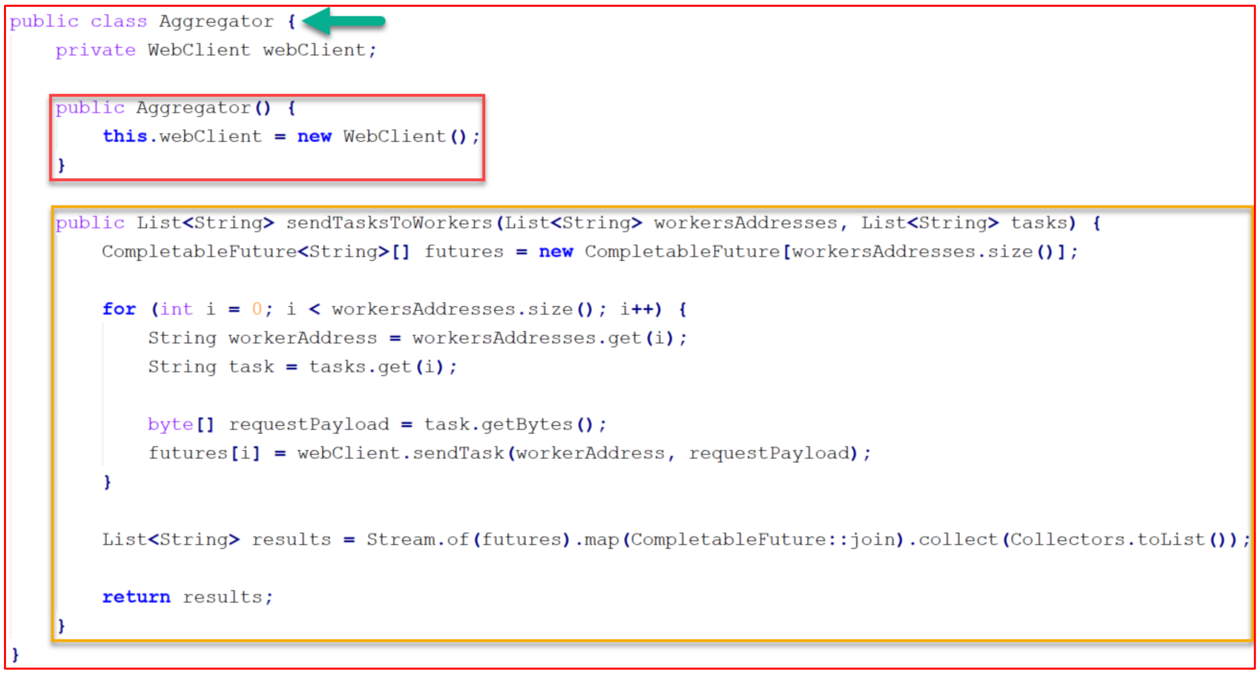
```
public CompletableFuture<String> sendTask(String url, byte[] requestPayload) {
    HttpRequest request = HttpRequest.newBuilder()
            .POST(HttpRequest.BodyPublishers.ofByteArray(requestPayload))
            .uri(URI.create(url))
            .build();

    return client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
            .thenApply(HttpResponse::body);
}
```

Next, let's move to the class that will call this method to send multiple requests and later aggregate them. Let's create a new class called Aggregator. The class will store a reference to the `WebClient`, instantiated inside the constructor. This Aggregator class will also have a single method called `sendTasksToWorkers`, which will take a list of HTTP server worker addresses and a list of tasks.

Inside the method, we create an array of `CompletableFutures`, which we use to store the future representations of the HTTP responses we get from the worker servers. We then iterate through all the worker addresses, in each iteration, we get the worker address and tasks we need to send to the current worker. We then convert the string representation of the task into a byte array and finally, we send that byte array payload to the corresponding worker by calling the `sendTask` method of the `WebClient` we just implemented.

After we've sent all the requests to the servers, we allocate an `ArrayList` to store the results. We then run a loop and call the join method of each future to wait and get the response for the corresponding request. Once we've obtained all the responses from all the workers, we simply return the aggregated responses to the caller.

If you are familiar with the streaming API in Java, we can rewrite this loop into a single line. It accomplishes the same task but looks a bit more elegant the responses.



```java
public class Aggregator {
    private WebClient webClient;

    public Aggregator() {
        this.webClient = new WebClient();
    }

    public List<String> sendTasksToWorkers(List<String> workersAddresses, List<String> tasks) {
        CompletableFuture<String>[] futures = new CompletableFuture[workersAddresses.size()];

        for (int i = 0; i < workersAddresses.size(); i++) {
            String workerAddress = workersAddresses.get(i);
            String task = tasks.get(i);

            byte[] requestPayload = task.getBytes();
            futures[i] = webClient.sendTask(workerAddress, requestPayload);
        }

        List<String> results = Stream.of(futures).map(CompletableFuture::join).collect(Collectors.toList());

        return results;
    }
}
```

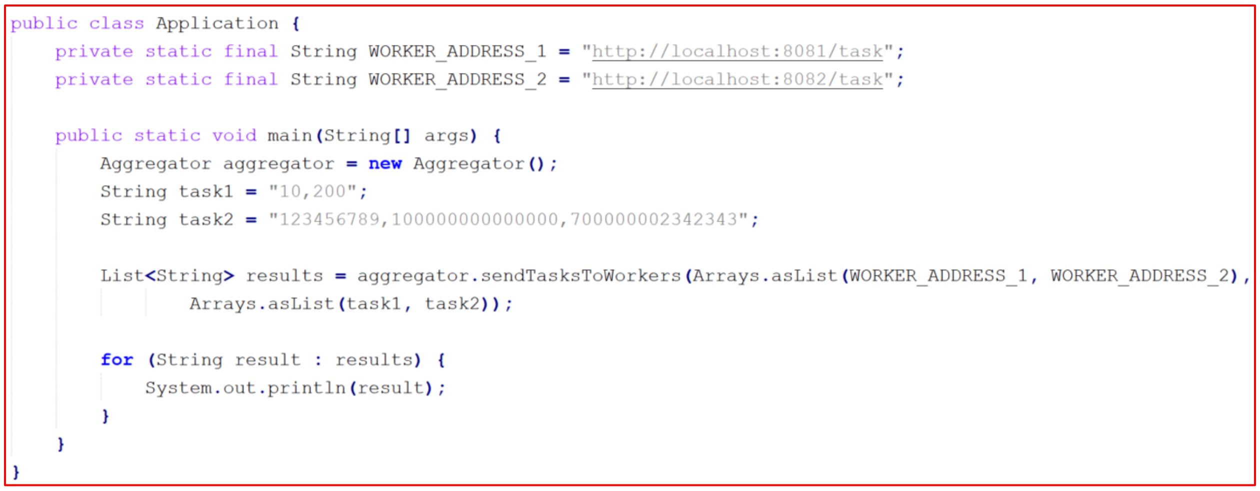
Send tasks from an HTTP client to multiple HTTP server workers

Now, let's proceed to test our HTTP client implementation by sending tasks to two instances of an HTTP server, which we implemented in a previous section. To do this, we'll create the Application class and define the main method as the entry point to our application.

At the top of the class, we'll declare the address of the first worker, which will listen on port 8081 and expect the task to arrive at the 'task' endpoint. Similarly, the second worker will run on the same machine, listening on port 8082.

Inside the main method, we'll create our Aggregator object, followed by defining two tasks. The first task will involve calculating the product of two small integers, 10 and 200. The second task will involve calculating the product of three large integers.

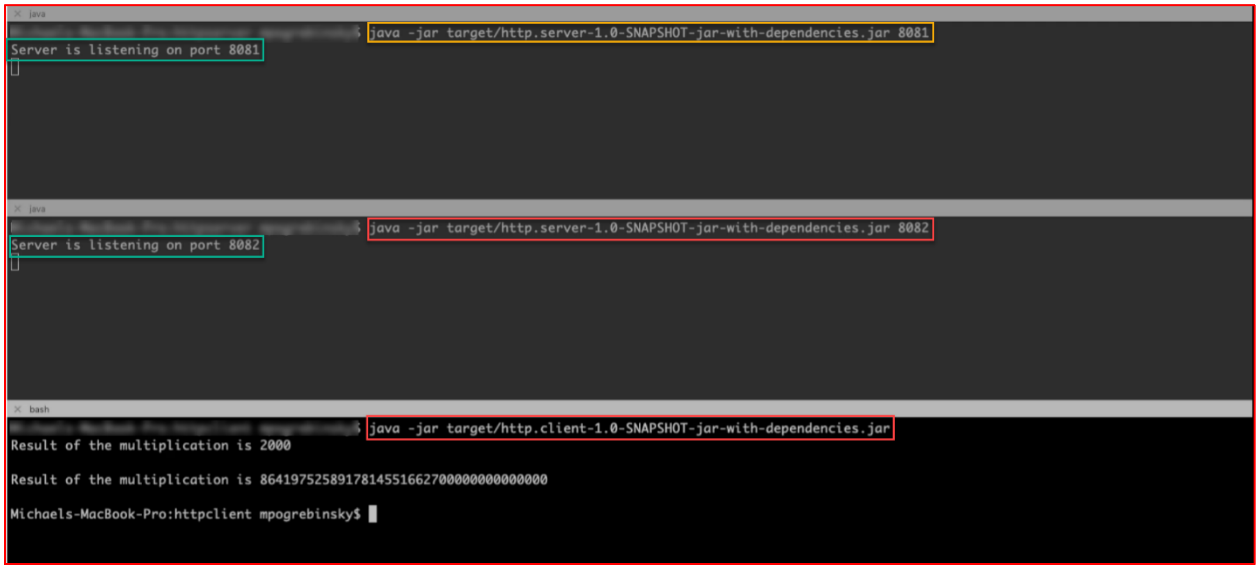
Now, let's dispatch those tasks to our workers for parallel computation by calling the `sendTask` method. We'll pass it a list of workers and tasks. Once we have all the results, we'll iterate through and print them all to the screen.



```java
public class Application {
    private static final String WORKER_ADDRESS_1 = "http://localhost:8081/task";
    private static final String WORKER_ADDRESS_2 = "http://localhost:8082/task";

    public static void main(String[] args) {
        Aggregator aggregator = new Aggregator();
        String task1 = "10,200";
        String task2 = "123456789,100000000000000,700000002342343";

        List<String> results = aggregator.sendTasksToWorkers(Arrays.asList(WORKER_ADDRESS_1, WORKER_ADDRESS_2),
                Arrays.asList(task1, task2));

        for (String result : results) {
            System.out.println(result);
        }
    }
}
```

That's it! Let's build and package our application using the `'mvn clean package'` command. Once the packaging is complete, we'll switch to our terminal.

In the top window, we'll launch one worker listening on port 8081. In the middle window, we'll launch another worker listening on port 8082. In the bottom window, we'll launch our HTTP client, which will send the tasks to our workers.



```
[redacted]$ java -jar target/http.server-1.0-SNAPSHOT-jar-with-dependencies.jar 8081
Server is listening on port 8081
```

```
[redacted]$ java -jar target/http.server-1.0-SNAPSHOT-jar-with-dependencies.jar 8082
Server is listening on port 8082
```

```
[redacted]$ java -jar target/http.client-1.0-SNAPSHOT-jar-with-dependencies.jar
Result of the multiplication is 2000

Result of the multiplication is 8641975258917814551662700000000000000

Michaels-MacBook-Pro:httpclient mpogrebinsky$ 
```

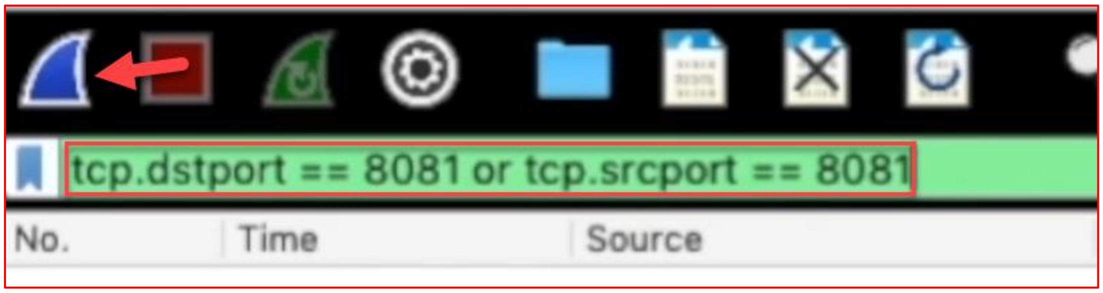
As we can see, we get the results for all our tasks. Our workers performed these tasks completely in parallel, and even the complex computation sent to the second worker was executed successfully.

As an exercise, I encourage you to modify the HTTP client to pass the 'X-debug' header to observe how long it takes the workers to perform each of these computations.

Inspect the network traffic using Wireshark

Now that we have a fully functional implementation of both an HTTP client and an HTTP server, let's learn how to use a compact but incredibly powerful tool called *Wireshark*. Wireshark is a free, open-source packet analyzer that allows us to inspect network traffic entering and leaving a specific computer. It's one of the best tools for troubleshooting and debugging network issues that might otherwise be difficult, or sometimes impossible, to diagnose within our code. It's also a great tool for understanding how protocols actually work, as any packet can be easily inspected down to individual bytes.

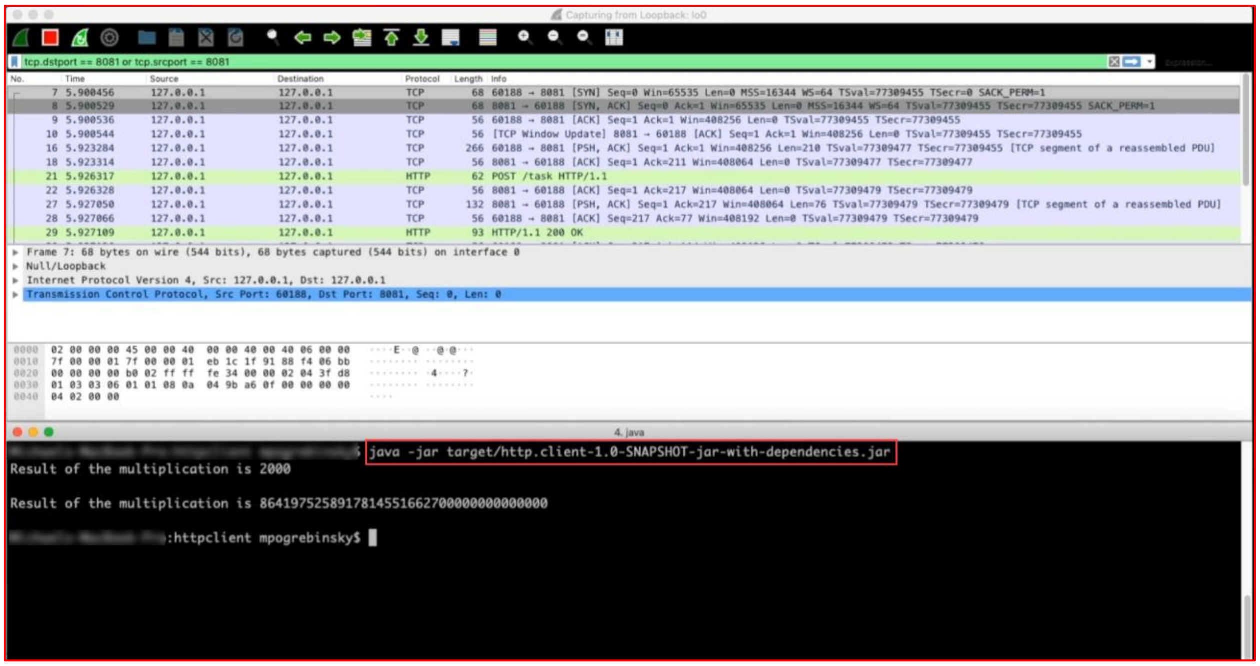
I've gone ahead and opened Wireshark, with a small terminal window at the bottom of the screen. At the top of the Wireshark window, we can filter the messages we're interested in, so we don't get overwhelmed once we start capturing network packets. In this example, we're going to inspect the network traffic from our HTTP client to one of our HTTP servers. We're going to filter on the TCP packets whose destination port is 8081 or source port is 8081. This way, we're going to capture all the requests and responses between the HTTP client and the server listening on that specific port.



```
tcp.dstport == 8081 or tcp.srcport == 8081
```

Let's begin capturing packets by clicking the 'Start' button. Then, we'll navigate to our terminal and send a request from the HTTP client to our worker by running our program. As we can see, once we do that, Wireshark captures all the TCP packets corresponding to our transaction. For each packet, we can see the direction of the packet. For instance, the first packet is going from the auto-generated port on the client to port 8081 on the HTTP server. The second packet's direction is the opposite, which means it went back from the HTTP server to the HTTP client as a response to the first packet.

Using Wireshark, we can also see how many messages it takes to accomplish a simple HTTP request/response transaction between two entities. Specifically, Wireshark highlights the connection establishment and the connection shutdown overhead associated with this transaction, using a dark gray background.


Capturing from Loopback: lo0
tcp.dstport == 8081 or tcp.srcport == 8081
4. java

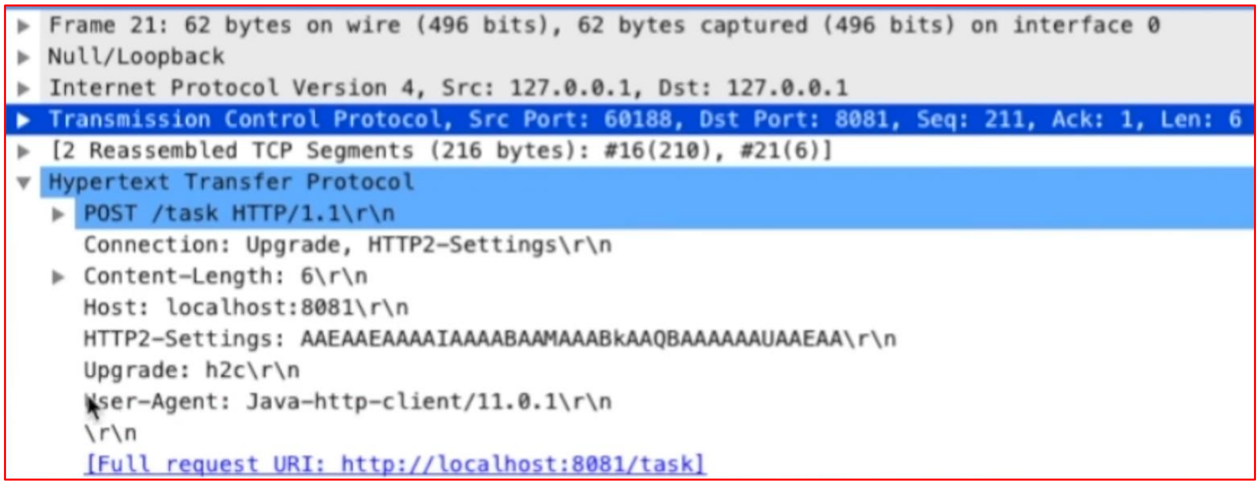
```
No.   Time      Source     Destination  Protocol Length Info
    7 5.900456  127.0.0.1  127.0.0.1    TCP       68 60188 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=77309455 TSecr=0 SACK_PERM=1
    8 5.900529  127.0.0.1  127.0.0.1    TCP       68 8081 → 60188 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=77309455 TSecr=77309455 SACK_PERM=1
    9 5.900536  127.0.0.1  127.0.0.1    TCP       56 60188 → 8081 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=77309455 TSecr=77309455
   10 5.900544  127.0.0.1  127.0.0.1    TCP       56 [TCP Window Update] 8081 → 60188 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=77309455 TSecr=77309455
   16 5.923284  127.0.0.1  127.0.0.1    TCP      266 60188 → 8081 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=210 TSval=77309477 TSecr=77309455 [TCP segment of a reassembled PDU]
   18 5.923314  127.0.0.1  127.0.0.1    TCP       56 8081 → 60188 [ACK] Seq=1 Ack=211 Win=408064 Len=0 TSval=77309477 TSecr=77309477
   21 5.926317  127.0.0.1  127.0.0.1    HTTP      62 POST /task HTTP/1.1
   22 5.926328  127.0.0.1  127.0.0.1    TCP       56 8081 → 60188 [ACK] Seq=1 Ack=217 Win=408064 Len=0 TSval=77309479 TSecr=77309479
   27 5.927050  127.0.0.1  127.0.0.1    TCP      132 8081 → 60188 [PSH, ACK] Seq=1 Ack=217 Win=408064 Len=76 TSval=77309479 TSecr=77309479 [TCP segment of a reassembled PDU]
   28 5.927066  127.0.0.1  127.0.0.1    TCP       56 60188 → 8081 [ACK] Seq=217 Ack=77 Win=408192 Len=0 TSval=77309479 TSecr=77309479
   29 5.927109  127.0.0.1  127.0.0.1    HTTP      93 HTTP/1.1 200 OK
```

```
▶ Frame 7: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 0
▶ Null/Loopback
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 60188, Dst Port: 8081, Seq: 0, Len: 0
```

```
0000  02 00 00 00 45 00 00 40  00 00 40 00 40 06 00 00   ····E··@ ··@·@···
0010  7f 00 00 01 7f 00 00 01  eb 1c 1f 91 88 f4 06 bb   ········ ········
0020  00 00 00 00 b0 02 ff ff  fe 34 00 00 02 04 3f d8   ········ ·4····?·
0030  01 03 03 06 01 01 08 0a  04 9b a6 0f 00 00 00 00   ········ ········
0040  04 02 00 00                                        ····
```

```
[redacted]$ java -jar target/http.client-1.0-SNAPSHOT-jar-with-dependencies.jar
Result of the multiplication is 2000

Result of the multiplication is 8641975258917814551662700000000000000

[redacted]:httpclient mpogrebinsky$
```

If we click on the HTTP POST request, we can observe all the networking layers we discussed in the section about networking. For example, we can see the frame that encapsulates the entire packet, which comes from the data link layer. Inside the frame, we can see the IP headers associated with the network layer. Underneath that, we can see the TCP headers associated with the transport layer, where, among other headers, we can see the source and destination port for that specific message. Finally, we can see the HTTP envelope which corresponds to the application layer.



```
▶ Frame 21: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
▶ Null/Loopback
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 60188, Dst Port: 8081, Seq: 211, Ack: 1, Len: 6
▶ [2 Reassembled TCP Segments (216 bytes): #16(210), #21(6)]
▼ Hypertext Transfer Protocol
  ▶ POST /task HTTP/1.1\r\n
    Connection: Upgrade, HTTP2-Settings\r\n
  ▶ Content-Length: 6\r\n
    Host: localhost:8081\r\n
    HTTP2-Settings: AAEAAEAAAAIAAAABAAMAAABkAAQBAAAAAAUAAEAA\r\n
    Upgrade: h2c\r\n
    User-Agent: Java-http-client/11.0.1\r\n
    \r\n
    [Full request URI: http://localhost:8081/task]
```

Also, note that the HTTP client is using the HTTP 1.1 version. However, it also sends two headers suggesting an upgrade to HTTP 2. This means that if the server does not support HTTP 2, it can still respond to the request by simply ignoring those two headers. But if the server does support HTTP 2, it would upgrade the transaction and continue the communication on a higher version of the protocol.

Additionally, if we scroll to the bottom of the request, we can see the actual message body in binary format. Take note of the amount of overhead and data that has to travel through the network just to deliver those six bytes of data. It's crucial to remember this picture every time we want to send a message and consider if we could batch some requests to improve efficiency.

Now let's also explore how we can verify if connection pooling is enabled and how it can save us some network overhead. Let's return to our main method and instead of sending one message to each worker, we'll send both messages to the same worker. Also, as we mentioned earlier, we can reuse the same connection only if the entire transaction that occupied that connection has already been completed. To ensure that, for testing purposes, let's add a short delay between the first and the second request to allow enough time for the response to the first request to arrive at the client.

Next, let's rebuild our application, return to our terminal, and send both requests to the same server. As we can see in Wireshark, thanks to the connection pooling enabled by default, we establish communication only once, then reuse the same connection for both transactions and shut down the connection in the end.

Summary

In this section, we learned how to use an HTTP client to send tasks to multiple worker nodes. Utilizing the asynchronous HTTP client API, we were able to dispatch these tasks to be performed in parallel by all the workers. We also learned how to use Wireshark to inspect network communication between two different entities.

In this specific example, we discovered how protocol negotiation works and realized that one of our entities only supports HTTP 1.1. So, if the number of connections becomes an issue, we may consider changing our HTTP server implementation to another library. We also observed the significant overhead involved in sending just a few bytes of data, which is inevitable, but perhaps we can reconsider our strategy and batch multiple tasks in one request to improve our communication efficiency. Finally, we saw how connection pooling works and how it allows us to reuse an existing connection instead of creating a new one for each individual transaction.