CoTCP – A New Approach to The Concurrent TCP

Doan Phi Hung
Switching Technology Research Center
Viettel High Technology Industries Corporation
Hanoi, Vietnam
hungdp6@viettel.com.vn

Nguyen Tai Hung
Faculty of Communication Engineering
Hanoi University of Science and Technology
Hanoi, Vietnam
hung.nguyentai@hust.edu.vn

Dinh Viet Quan
Switching Technology Research Center
Viettel High Technology Industries Corporation
Hanoi, Vietnam
quandv@viettel.com.vn

Do Ngoc Thanh
Switching Technology Research Center
Viettel High Technology Industries Corporation
Hanoi, Vietnam
thanhdn1@viettel.com.vn

Ba Dinh Hoai
Switching Technology Research Center
Viettel High Technology Industries Corporation
Hanoi, Vietnam
hoaibd1@viettel.com.vn

***Abstract*—The Transmission Control Protocol (TCP) were intentionally designed for the sake of service reliability but with the cost of application performance on which TCP clients need to use multiple connections to achieve concurrency and to reduce latency. To address this weakness of TCP, and on this paper, we proposed a new application-level protocol that makes use of TCP as transportation, named as CoTCP (Concurrent request - response over TCP). The new proposed protocol allows sending and receiving multiple messages concurrently on one connection. We also evaluated and tested the performance of CoTCP in various application scenarios on the specific hardware platform. Numerical results show that CoTCP can lead to higher concurrency and lower latency.**

Keywords—TCP, CoTCP, Connection, Concurrency, Latency

I. Problem Statement

TCP is one of the most commonly used protocol that is designed to send packets across the Internet and ensures the integrity of data sent over the network [1], [2]. In order to transmit data, TCP establishes a connection between a source and its destination. TCP can only transfer one message at a time per connection. A normal TCP transaction operated like this, the client establishes a connection to the server; the client sends a request to the server and wait for the response; the server responses to the client; the connection is closed or is reserved to be used for next transactions. A transaction needs to wait for other transaction to be completed before it can be started. A common strategy is to open multiple connections to serve multiple transactions at a time, which can help to improve concurrency and to reduce latency, but to open too many connections can be costly.

TCP uses a three-way hand shaking to establish a connection between the client and the server [3]. A three-way hand shaking process is expensive because it requires three packets (SYNC, SYNC-ACK and ACK) to be transferred. To avoid having to open the connection many times, a TCP connection can be made persistent to be reused. However, additional resources are required to maintain each persistent TCP connection. Multiple TCP clients where each one opens several connections to the server can cause the server to be overloaded.

To address that weakness of TCP for our own application in 4/5G mobile core networks, our R&D team has come up with a proposition of a new application protocol names as CoTCP. That said, CoTCP is designed to solve the concurrency problem of TCP. On our design, a CoTCP transaction is operated in asynchronous mode so that multiple ones can be executed concurrently over the same TCP connection thus make CoTCP able to meet the requirement of high number of transactions per second (TPS) and low latency system while ensuring low number of (TCP) connections.

This work is supported by Switching Technology Research Center.

The rest of the paper is organized as follows. The investigation result of similar works will be given on section II. Section III will present the details of our new proposed protocol, named as CoTCP. Experimental setup and performance evaluation will be presented in section IV. Finally, section V concludes our paper.

II. Related Works

In this section, we will discuss about current researches on the problem of high-performance TCP client-server system, and how to scale up TCP for handling of a large number of concurrent clients. This problem is always the remaining hot (research) topic for decades.

The C10K problem [4] was coined in 1999 by software engineer Dan Kegel. It is the problem of optimizing network sockets to handle 10,000 connections at the same time. C10K problem is currently solved by certain web servers such as Nginx [5] which applies event-driven architecture to disorder the execution flow of network programs, and maximizes the utilization of CPU. By the early of 2020s, the problem is scaled up to C10M which means to concurrently handle 10,000,000 connections. Several solutions have been proposed to solve the C10M problem also, such as in [6], [7], [8]. Those solutions mainly focus on optimizing or bypassing the kernel and therefore utilize multi-core processors and reduce system calls and context switching overheads. Recently, there are high-speed packet I/O frameworks such as DPDK [9], netmap [10], and PF RING [11] that allow user-space applications to exchange packets with the kernel networking stack, providing unprecedented network performance for applications.

All of the above-mentioned solutions solve the concurrency problem of TCP by trying to increase the number of concurrent connections but none has been focused on utilizing a single connection to handle multiple (application) transactions concurrently. With the introduction of coroutine in modern programming languages such as golang [12], python [13], and kotlin [14]; handling millions of transactions at the same time

becomes significantly less expensive. Coroutine is a light-weight thread managed by user-space which allows execution to be suspended or resumed without context switching overheads [15]. Using coroutines, applications can easily handle millions of concurrent transactions but to open and to manage millions of concurrent connections is still a challenge to this day.

III. Proposed Solution

A. Protocol Design

As said above, on this work, we've proposed a new application-level protocol to solve the concurrency problem of TCP. The new protocol was named as CoTCP (Concurrent request - response over TCP) and this section will give a detailed presentation of its design and operation.

The core part of new proposed protocol are its transactions. CoTCP transactions are designed asynchronously in which a request could be sent before the response of another request is received as depicted on figure 2.

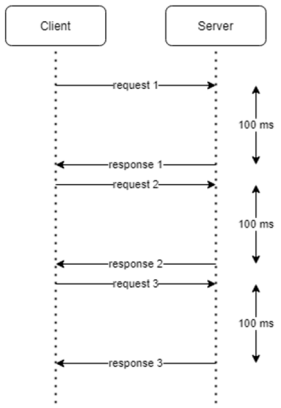

Fig. 1. Requests and responses are sent sequentially.

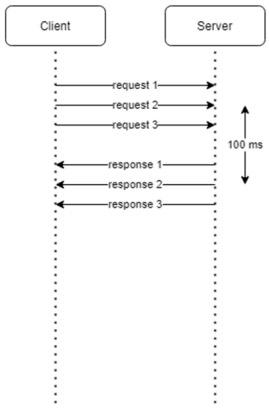

Fig. 2. Requests and responses are sent concurrently.

In this asynchronous transaction mode, responses could be received out of the order in which requests were sent. And to make this possible, we assigned each request with a unique identifier (ID) and then the corresponding response must have the same ID so that it can be matched to its own request. As such, the proposed structure of a CoTCP message is depicted as on the figure 3.

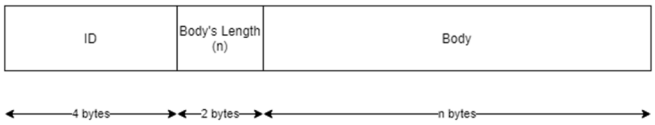

Fig. 3. Message's structure.

The message is composed of three parts:

- ID: A 4 bytes, unique integer that identifies a pair of request and response.
- Body's Length: A 2 bytes integer that indicates the size of message's body.
- Body: The actual content of the message that is stored in binary format.

The working procedure of CoTCP on client side is illustrated as on figure 4 below.

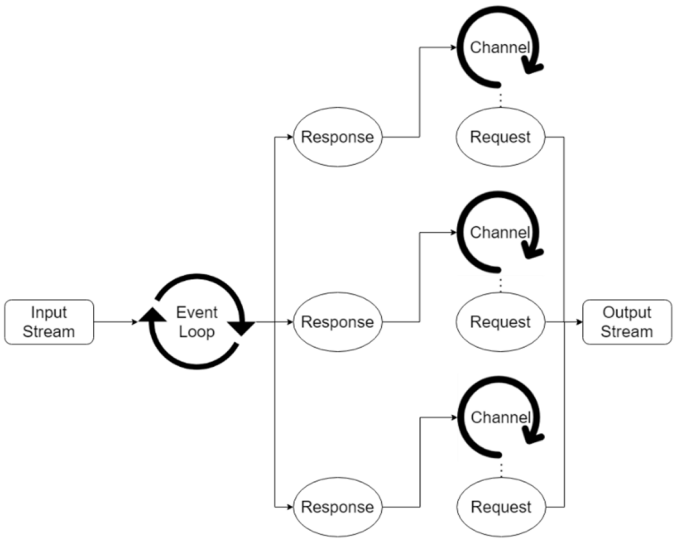

Fig. 4. Application architecture of CoTCP Client

The procedure is a sequence of steps as follows:

- Step 1: Establishes a new connection to the server.
- Step 2: Create an event loop to listen on the established connection.
- Step 3: Generate a unique ID for each request message.
- Step 4: Send request message and open a channel to wait for the response.
- Step 5: When the event loop receives data, split the data stream into messages and send them to the corresponding channels.
- Step 6: Read the response message from the channel and close the channel.

From the server side, the CoTCP working procedure is as depicted on the figure 5.

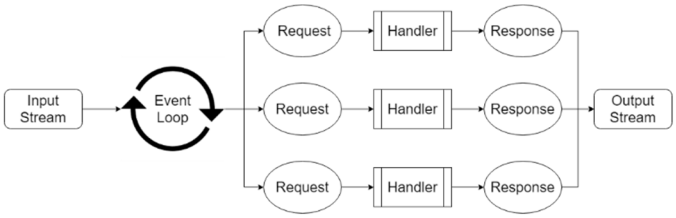

Fig. 5. Application architecture of CoTCP server.

It also goes through steps as follows:

- Step 1: Accept a new connection from the client.
- Step 2: Create an event loop to listen on the established connection.

- Step 3: When the event loop receives data, split the data stream into messages and handle them concurrently.
- Step 4: Assign the request's ID to the corresponding response and send response to the client.

B. Implementation

In this sub-section, we will explain how to implement CoTCP client and server applications that can provide high concurrency and low latency with small overheads.

We choose golang as the programming language to implement CoTCP because its built-in co-routines are suitable for building high concurrency applications.

The core of the CoTCP application is its event loop. Each TCP connection is managed by one event loop running on an independent coroutine. The responsibility of the event loop is to listen on the connection for incoming messages and to handle them concurrently. Because TCP transmits data in stream and there is no boundary between TCP packets, it has the problem of packets sticking together. In order to solve this problem, each CoTCP message has a length field that can be used to split the data stream into messages. For CoTCP server, the event loop will scan on the input data stream for request messages and will spawn a coroutine to handle each one; the response message is then sent back to the client through the same connection of its request. For CoTCP client, the event loop will scan on the input data stream for response messages and will send them to the corresponding waiting channels.

In order to match the response message to its request, each request is assigned to a channel that waits for response from the event loop; this mechanism makes a transaction look like a synchronous process. The list of waiting channels is stored in a hash table that can be used to lookup the channel by the ID of the response message.

IV. Performance Evaluation

To evaluate the performance of the new protocol, we have setup the test-bed (figure 6) and conducted three performance benchmarks with different application configurations where each one was taken for both TCP and CoTCP.

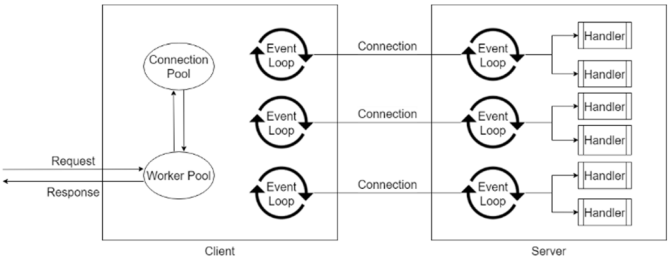

Fig. 6. Performance benchmark's setup.

The benchmarks were performed as following procedure.

- Step 1: Initiate the server with predefined configurations.
- Step 2: The server waits for incoming requests and responses after a delay.
- Step 3: Initiate the client with predefined configurations.
- Step 4: The client establishes a fixed number of connections to the server.
- Step 5: The client initiates a pool of worker coroutines to send request to the server and wait for the response.
- Step 6: The average number of transactions per second (TPS) and average latency is calculated where a transaction is started from the time of sending request until receiving response.
- Step 7: For TCP benchmark, transactions on the same connection are executed sequentially.
- Step 8: For CoTCP benchmark, transactions on the same connection are executed concurrently.

Each benchmark includes one server to handle requests and one benchmark tool acting as the client:

- The server is configurable with the following parameters: number of CPUs used, delay duration before sending responses back to the client, and size of the response's body.
- The client is configurable with the following parameters: number of CPUs used, number of opened connections to the server, number of worker coroutines used to send requests to the server, and size of the request's body.

A. First Benchmark

1) Configurations

TABLE I. Server's Configurations

Parameter	Value
Type of CPU	Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
Number of CPUs	8
Reponse delay	0ms
Size of response's body	10 bytes

TABLE II. Client's Configurations

Parameter	Value
Type of CPU	Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
Number of CPUs	8
Number of worker coroutines	500
Size of request's body	10 bytes

2) Results

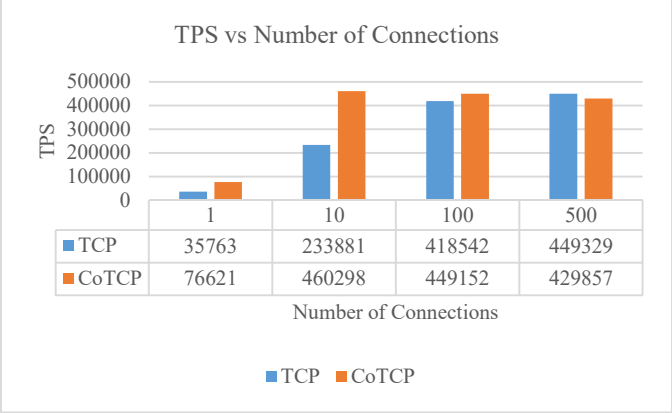

Fig. 7. TPS vs Number of Connections

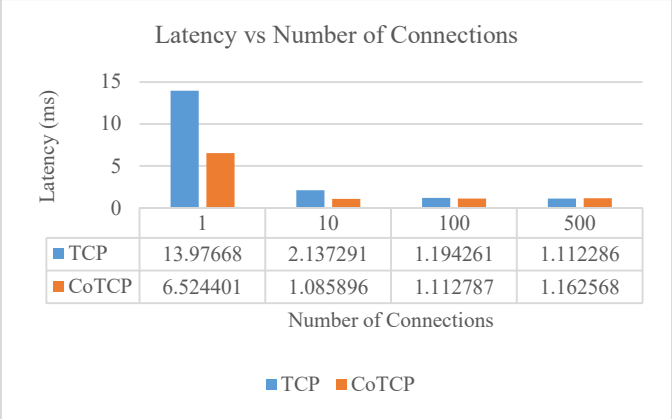

Fig. 8. Latency vs Number of Connections

According to figure 7 and figure 8, we can conclude that:

- For small number of connections (10 connections and below), the performance of CoTCP is about two times better than the performance of TCP.
- For big number of connections (100 connections and above), the performance of CoTCP is similar to the performance of TCP.
- The optimal number of connections for TCP is 500 which is equal to the number of worker coroutines. Increasing the number of connections beyond 500 will not improve the concurrency but produce idle connections.
- The optimal number of connections for CoTCP is about 10 connections. As the number of connections grows, the performance of CoTCP slightly decreases due to the overheads for maintaining extra connections.

B. *Second Benchmark*

1) *Configurations*

TABLE III. SERVER'S CONFIGURATIONS

Parameter	Value
Type of CPU	Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
Number of CPUs	8
Reponse delay	10ms
Size of response's body	10 bytes

TABLE IV. CLIENT'S CONFIGURATIONS

Parameter	Value
Type of CPU	Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
Number of CPUs	8
Number of worker coroutines	500
Size of request's body	10 bytes

2) *Results*

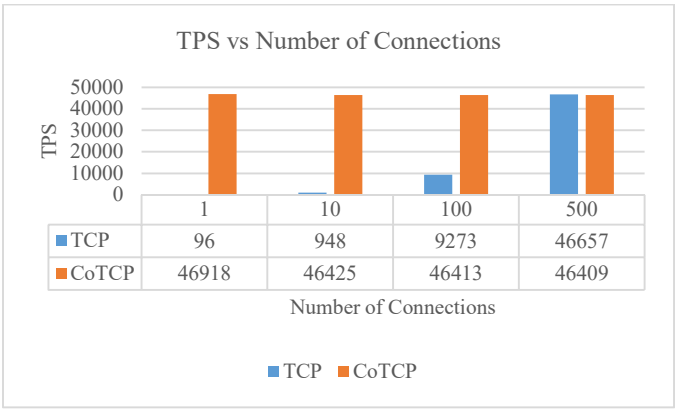

Fig. 9. TPS vs Number of Connections

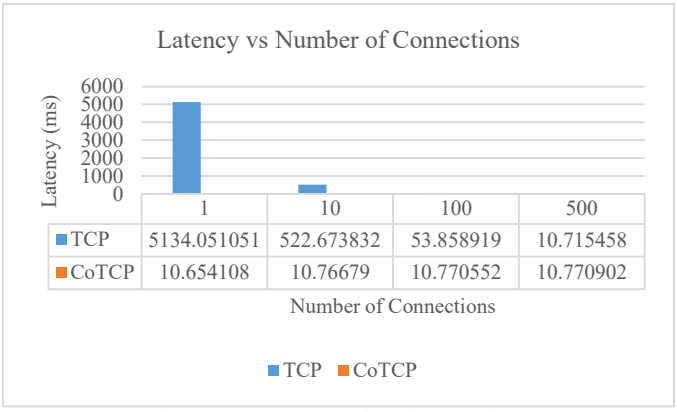

Fig. 10. Latency vs Number of Connections

According to figure 9 and figure 10, we can conclude that:

- The performance of TCP linearly increases as the number of connections increases from 1 to 500.
- The performance of CoTCP is the same for any number of connections.

- At 500 connections, the performance of TCP is similar to the performance of CoTCP and is close to ideal which are 50,000 TPS and 10ms latency.

C. Third Benchmark

1) Configurations

TABLE V. SERVER'S CONFIGURATIONS

Parameter	Value
Type of CPU	Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
Number of CPUs	8
Reponse delay	100ms
Size of response's body	10 bytes

TABLE VI. CLIENT'S CONFIGURATIONS

Parameter	Value
Type of CPU	Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
Number of CPUs	8
Number of worker coroutines	500
Size of request's body	10 bytes

2) Results

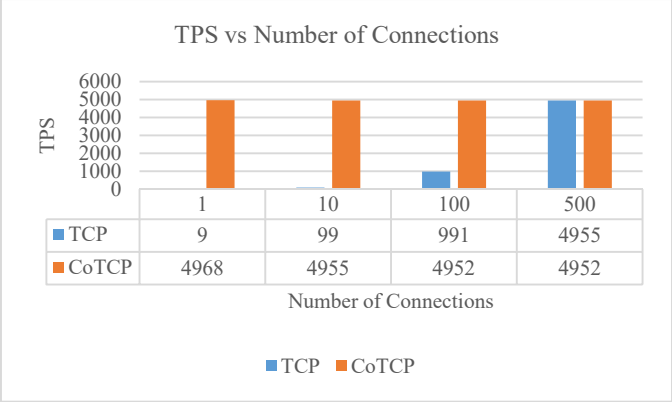

Fig. 11. TPS vs Number of Connections

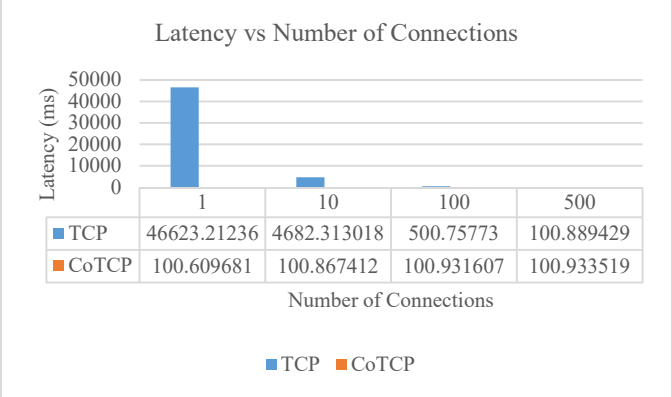

Fig. 12. Latency vs Number of Connections

According to figure 11 and figure 12, we can conclude that:

- The performance of TCP linearly increases as the number of connections increases from 1 to 500.
- The performance of CoTCP is the same for any number of connections.
- At 500 connections, the performance of TCP is similar to the performance of CoTCP and is close to ideal which are 5000 TPS and 100ms latency.

V. CONCLUSION AND FUTURE WORK

A. Conclusion

On this paper, we have presented our work on proposing a new a protocol based on TCP named as CoTCP, which allows sending and receiving multiple messages concurrently over a single TCP connection. Numerical results show several advancements from our work as below:

- CoTCP allows to send requests and receive responses asynchronously over the same connection; therefore, it helps to improve concurrency and reduce latency without having to open many connections.
- In case the server can handle requests and response to the client immediately, which rarely happens in real conditions; the performance of CoTCP is not better than sending requests and receiving responses sequentially using TCP.
- In case the server needs a certain amount of time to handle requests and response to the client, the performance of CoTCP is much better than the performance of TCP for small number of connections. These become comparable as the number of connections grows.
- The performance of CoTCP is less dependent on number of connections than the performance of TCP is.

B. Future Work

The benchmark results show improvements in concurrency of CoTCP compared to TCP. However, there is still limitation of CoTCP. It does not support multiplexing [16], [17] on a single connection. While a large message is being sent, other messages are blocked from being sent on the same connection. In the future, we will add multiplexing to CoTCP.

In order to achieve multiplexing on a single connection, a CoTCP message can be divided in multiple parts before being sent. Parts of multiple messages will be mixed together and will be sent over the same connection. The server will receive messages' parts and combine them into complete messages.

ACKNOWLEDGMENT

This work is supported by Switching Technology Research Center. The authors would also like to thank AMF team and all 5G core project members.

REFERENCES

[1] J. Postel (ed.), "Internet protocol - DARPA internet program protocol specification," RFC 791, USC/Information Sciences Institute, Sep. 1981.

[2] J. Postel (ed.), “Transmission control protocol - DARPA internet program protocol specification,” RFC 793, USC/Information Sciences Institute, Sep. 1981.

[3] E. Conrad, S. Misenar, and J. Feldman, “The basics of hacking and penetration testing,” 2nd ed., 2012.

[4] D. Kegel, "The C10K problem," May 8, 1999.

[5] D. DeJonghe, “Nginx cookbook,” 2nd ed., Oct. 28, 2020.

[6] R. Graham, ”The secret to 10 million concurrent connections - The kernel is the problem, not the solution,” 2013.

[7] M. Rotaru, “Scaling to 12 million concurrent connections: how MigratoryData did it,” Oct. 10, 2013.

[8] R. Rotaru, “How MigratoryData solved the C10M problem: 10 million concurrent connections on a single commodity server,” May 20, 2015.

[9] “Data plane development kit,” www.dpdk.org. https://www.dpdk.org (accessed: Jun. 22, 2022).

[10] L. Rizzo, “Netmap: a novel framework for fast packet i/o,” Luigi Rizzo, Università di Pisa, Italy, pp. 101–112, 2012.

[11] “Pf ring zero copy,” www.ntop.org. https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy (accessed: Jun. 22, 2022).

[12] A. A. A. Donovan, and B. W. Kernighan, “The Go programming language,” Oct. 26, 2015.

[13] E. Matthes, “Python crash course,” 2nd ed., May 2019.

[14] S. Isakova, D. Jemerov, “Kotlin in action,” Feb. 3, 2017.

[15] D. E. Knuth, “Fundamental algorithms,” 3rd ed., vol. 1, p. 229, 1997.

[16] J. Burke, “Multiplexing,” Nemertes Research, Aug. 2021.

[17] D. Cohen, “Multiplexing protocol,” IEN-90, USC/Information Sciences Institute, May 2, 1979.