

Hands-On Tutorial – H01

Introduction to Socket Programming

Penulis : ATA, RR, RF
Versi : 1 (20240917-0800)





Riwayat Versi

Setiap “**Versi**” yang dimaksudkan pada riwayat ini dan dijadikan rujukan utama bagi dokumen ini memuat perubahan yang bersifat substantif sehingga perlu diketahui oleh pemangku kepentingan dokumen ini. Dokumen dapat memiliki perubahan non-substantif yang tidak tercatat pada riwayat ini namun tetap tercatat pada riwayat versi yang dikelola Office pada salinan asli dokumen ini.

Riwayat versi ini diurutkan secara kronologis terurut dari versi paling akhir pada baris pertama hingga versi paling awal pada baris terakhir.

Versi	Tanggal dan Waktu	Halaman	Perubahan
1	20240917-0800	Semua	Rilis pertama

Daftar Isi

 Riwayat Versi	2
 Daftar Isi	3
 Informasi Umum	4
 Ekspektasi Hasil Pembelajaran	4
 Prasyarat	4
 Deskripsi	5
[50 Poin] Hands-On Sample Program	5
Persiapan : Direktori Kerja	7
The Very Basics : Connectionless Socket (UDP)	9
We Need to Talk : Connection-Oriented Socket (TCP)	14
[30 Poin] Reflection 1 : Connectionless vs Connection-Oriented Socket	17
A Quicker Way: Connection-Oriented Socket (QUIC)	19
Agreeing Upon a Language: Encoding – Decoding	26
[20 Poin] Reflection 2: QUIC Internalization	29
[50 Poin] Hands-on Simple Project	30
Format Pesan	31
Perilaku yang Diharapkan	32
Spesifikasi Teknis	33
Eksekusi Program	33
Rubrik Penilaian	34
 Informasi Pengumpulan Berkas	34
 Peraturan	36
Keterlambatan	36
Plagiarisme	36

Hands-On Tutorial – H01

Introduction to Socket Programming

Informasi Umum

Tipe Tugas	: Individu
Batas Waktu Pengumpulan	: Sabtu, 21 September 2024 pukul 17.00 WIB (SCeLE)
Format Penamaan Berkas	:
- Laporan	: H01_[NPM].pdf (Contoh: H01_2206144111.pdf)
- Go Workspace Archive	: H01_[NPM].zip (Contoh: H01_2206144111.zip)
Tautan Kerangka Laporan	: Klik Di Sini
Tautan Kerangka Workspace Go	: Klik Di Sini

Ekspektasi Hasil Pembelajaran

Students are expected to **implement (C3)** simple socket programs using Go.

Prasyarat

Sebelum mengerjakan Hands-on Tutorial ini, peserta diharapkan untuk dapat memenuhi prasyarat sebagai berikut:

1. Peserta telah memenuhi ekspektasi hasil pembelajaran yang ada pada **A01a, A01c, dan A01d**.
2. Peserta telah memiliki konsep dasar mengenai Application Layer, Transport Layer, dan Socket Programming
3. Peserta telah menyiapkan dua buah mesin virtual Google Compute Engine sesuai dengan spesifikasi yang diberikan pada **A01a**.
4. Peserta telah memasang bahasa pemrograman Go versi 1.23.1 di kedua mesin virtual yang telah disiapkan.
5. Peserta telah menentukan kode unik masing-masing yang akan digunakan sebagai nomor *port* unik untuk komunikasi aplikasi yang akan dikembangkan pada Hands-on Tutorial ini dengan ketentuan sebagai berikut (sesuai dengan apa yang sudah dikenalkan di **A01a**):
 1. Ambil 4 digit terakhir NPM Anda. Sebagai contoh, jika NPM Anda adalah 2206123456, maka angka yang dipilih adalah 3456.
 2. Ubah setiap angka nol (0) yang berada di depan angka bukan nol dengan angka sembilan (9). Sebagai contoh, jika NPM Anda adalah 2206000020, dengan 4 digit terakhirnya adalah 0020, maka nomor yang Anda dapatkan adalah 9920.

3. Jika nomor yang anda miliki adalah kurang dari 1025, maka tambahkan angka sembilan (9) di belakang nomor yang Anda miliki. Misalkan, jika NPM Anda adalah 2206001023, dengan 4 digit terakhirnya adalah 1023, maka nomor yang Anda dapatkan adalah 10239.
4. Nomor terakhir yang Anda dapatkan adalah nomor port unik Anda.

Deskripsi

[50 Poin] Hands-On Sample Program

Perhatian!

Bagian ini akan mencampurkan komponen yang tidak dinilai dan komponen yang akan dinilai. Komponen yang dinilai merupakan refleksi konstruktif dari pengalaman Hands-on yang telah kalian alami sebelum bagian refleksi tersebut. Jawaban refleksi perlu dijawab pada lembar laporan yang telah disediakan.

Setiap lapisan yang membangun sistem komunikasi di dalam jaringan memiliki perannya masing-masing. Pada lapisan Application Layer, kita telah berkenalan dengan mekanisme pertukaran pesan untuk mencapai tujuan tertentu yang diharapkan oleh pihak yang terlibat dalam komunikasi tersebut. Kemudian, lapisan Transport Layer di bawahnya memastikan bahwa kedua proses di masing-masing pihak yang berkomunikasi tersebut dapat saling menemukan satu sama lain dan mengantarkan pesan yang diminta. Masing-masing fungsi ini tentunya perlu dijembatani agar masing-masing lapisan tidak perlu terlalu “mencaplok” wilayah kerja lapisan lainnya. Jembatan antara Application Layer dan Transport Layer inilah yang kita kenal sebagai “socket”.

Socket sendiri dapat diartikan sebagai antarmuka antara aplikasi yang beroperasi di cakupan pengguna (*user space*) dengan layanan transportasi yang beroperasi di cakupan kernel (*kernel space*). Maka dari itu, pemrograman terhadap socket dapat diartikan sebagai aktivitas yang dilakukan untuk memberlakukan perilaku tertentu terhadap layanan transportasi sesuai dengan kebutuhan aplikasi yang beroperasi di atasnya. Beberapa fungsi yang diemban oleh sebuah socket dalam mengatur perilaku tersebut antara lain:

1. Alokasi dan pembubaran sumber daya terkait dengan komunikasi
2. Pemuatan dan pembongkaran pesan antara format yang dipahami aplikasi dan format yang dapat ditransmisikan melalui layanan transportasi

Pada Hands-on Tutorial ini, kita akan berkenalan dengan socket-socket yang dapat dibangun di dalam jaringan, aplikasi yang dapat kita kembangkan untuk memanfaatkan socket-socket tersebut, serta fungsi yang dapat kita capai dari pengembangan tersebut. Hands-on Tutorial ini menggunakan bahasa pengantar Go sehingga semua contoh yang diberikan akan menggunakan bahasa tersebut

dan peserta nantinya diharapkan dapat mengembangkan aplikasi yang memanfaatkan socket ini dengan menggunakan bahasa tersebut.

Persiapan : Direktori Kerja

Sila unduh kerangka direktori kerja Hands-On Tutorial 1 melalui [pranala ini](#)

Direktori kerja pemrograman untuk Hands-on Tutorial 1 terdiri dari **2 package utama**, yaitu **samples (Contoh)** dan **project (Proyek)**. Package Contoh berisi kode sumber siap pakai untuk aplikasi yang memanfaatkan layanan transportasi yang kita pelajari di mata kuliah ini serta kode sumber siap pakai untuk aplikasi yang memperkenalkan sistem *encoding-decoding* yang populer digunakan untuk menjembatani format pesan aplikasi dan layanan transportasi. Package Proyek berisi kerangka kode sumber yang nantinya digunakan oleh peserta untuk mengembangkan aplikasi sederhana yang memanfaatkan socket sesuai spesifikasi yang diberikan. Struktur direktori kerja yang diberikan adalah sebagai berikut:

Direktori	Berkas	Fungsi
H01-Source-Code/		Root dari direktori kerja
project/		Root dari package Proyek
publisher/	publisher.go	Kode sumber untuk aplikasi proyek (sisi Publisher / Klien)
subscriber/	subscriber.go	Kode sumber untuk aplikasi proyek (sisi Subscriber / Server)
utils/	utils.go	Kode sumber untuk aplikasi proyek (khusus untuk logika Encoder/Decoder dan Setup QUIC)
samples/		Root dari package Contoh
codec/	codec.go	Kode sumber contoh penerapan Encoder / Decoder
quic/		Root dari package contoh socket berbasis QUIC
client/	quicClient.go	Kode sumber contoh penerapan socket berbasis QUIC di sisi klien
server/	quicServer.go	Kode sumber contoh penerapan socket berbasis QUIC di sisi server
utils/	quicUtils.go	Kode sumber terkait dengan setup QUIC yang tidak perlu diperhatikan oleh peserta
tcp/		Root dari package contoh socket berbasis TCP
client/	tcpClient.go	Kode sumber contoh penerapan socket berbasis TCP di sisi klien

server/	tcpServer.go	Kode sumber contoh penerapan socket berbasis TCP di sisi server
udp/		Root dari package contoh socket berbasis UDP
client/	udpClient.go	Kode sumber contoh penerapan socket berbasis UDP di sisi klien
server/	udpServer.go	Kode sumber contoh penerapan socket berbasis UDP di sisi server

Keseluruhan direktori kerja ini membentuk suatu Workspace untuk bahasa pemrograman Go dan sudah dikonfigurasi dengan nama modulnya masing-masing sebagai berikut:

1) Contoh

- a) **Codec:** jarkom.cs.ui.ac.id/h01/samples/codec
- b) **QUIC:**
 - i) **Klien:** jarkom.cs.ui.ac.id/h01/samples/quic/client
 - ii) **Server:** jarkom.cs.ui.ac.id/h01/samples/quic/server
- c) **TCP:**
 - i) **Klien:** jarkom.cs.ui.ac.id/h01/samples/tcp/client
 - ii) **Server:** jarkom.cs.ui.ac.id/h01/samples/tcp/server
- d) **UDP:**
 - i) **Klien:** jarkom.cs.ui.ac.id/h01/samples/udp/client
 - ii) **Server:** jarkom.cs.ui.ac.id/h01/samples/udp/server

2) Proyek

- a) **Publisher / Klien:** jarkom.cs.ui.ac.id/h01/project/publisher
- b) **Subscriber / Server:** jarkom.cs.ui.ac.id/h01/project/subscriber

Untuk menjalankan salah satu modul, kalian dapat mengarahkan Terminal kalian ke *root directory* dari direktori kerja ini (di *folder H01-Source-Code*) dengan menjalankan instruksi berikut ini:

```
go run <nama modul>
# Contoh: go run jarkom.cs.ui.ac.id/h01/project/publisher
```

The Very Basics : Connectionless Socket (UDP)

bagaimana yang telah kita ketahui bersama, UDP memiliki karakteristik *connectionless* yang berarti tidak ada mekanisme yang diberlakukan oleh protokol ini untuk melakukan transaksi peran melalui kesepakatan komunikasi khusus. Hal ini membuat protokol ini memiliki layanan yang sangat sedikit dan bahkan tidak banyak hal yang ditambahkan olehnya dari apa yang sudah ditawarkan oleh lapisan Network Layer di bawahnya. Hal ini juga menjadikan pemrograman *socket* berbasis UDP yang paling mudah di antara tiga protokol yang akan kita bahas pada Hands-on Tutorial ini.

Yuk, mari kita berkenalan dengan contoh program *socket* yang memanfaatkan protokol UDP dengan cara berikut ini:

1. Siapkan dua buah jendela Command Prompt / Terminal / PowerShell / sejenisnya di **komputer lokal kamu**. Kamu boleh menggunakan tmux atau alat serupa **jika paham cara menggunakannya**.
2. Navigasi ke *root directory* dari direktori kerja H01 ini (folder h01-source-code).
3. Jalankan program *socket server* UDP di salah satu jendela dengan komando berikut ini:

```
go run jarkom.cs.ui.ac.id/h01/samples/udp/server
```

```
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/udp/server
UDP Server Socket Program Example in Go
Press Ctrl+C or Cmd+C to stop the program
[udp4] Listening on: 0.0.0.0:54321
[udp4] Creating receive buffer for next communication of size 2048
```

4. Jalankan program *socket client* UDP di jendela lainnya dengan komando berikut ini:

```
go run jarkom.cs.ui.ac.id/h01/samples/udp/client
```

```
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/udp/client
UDP Client Socket Program Example in Go
[udp4] Dialling from 127.0.0.1:44759 to 127.0.0.1:54321
[udp4] Creating receive buffer of size 2048
[udp4] Input message to be sent to server: ■
```

Berikan masukan berupa *string* apapun pada program klien dan tekan Enter. Perhatikan alur komunikasi yang terjadi antara program *server* dan *client* yang terjadi!

5. Hentikan eksekusi program *socket server* dengan menekan tombol Ctrl+C pada jendela yang menjalankan *socket server*

```
rafi muhammad@lubuntu-rf:~$ cd h01-source-code/
rafi muhammad@lubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/udp/server
UDP Server Socket Program Example in Go
Press Ctrl+C or Cmd+C to stop the program
[udp4] Listening on: 0.0.0.0:54321
[udp4] Creating receive buffer for next communication of size 2048
[udp4] Creating receive buffer for next communication of size 2048
[udp4] [Client: 127.0.0.1:44759] Received 10 bytes of message
[udp4] [Client: 127.0.0.1:44759] Message: halo halo

[udp4] [Client: 127.0.0.1:44759] Sending Response: HALO HALO
^Csignal: interrupt
rafi_muhammad@lubuntu-rf:~/h01-source-code$ █

-----
rafi_muhammad@lubuntu-rf:~$ cd h01-source-code/
rafi_muhammad@lubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/udp/client
UDP Client Socket Program Example in Go
[udp4] Dialling from 127.0.0.1:44759 to 127.0.0.1:54321
[udp4] Creating receive buffer of size 2048
[udp4] Input message to be sent to server: halo halo
[udp4] Sending message 'halo halo'
' to server
[udp4] Received 10 bytes of message from server
[udp4] Response from server: HALO HALO
rafi_muhammad@lubuntu-rf:~/h01-source-code$
```

Apakah kalian sudah melihat luaran yang serupa? Ini adalah salah satu contoh komunikasi antar-proses (*inter-process communication*) yang diwujudkan oleh layanan transportasi UDP. Meskipun saat ini kita masih menggunakan mesin yang sama, penggunaan layanan transportasi UDP berarti kita tetap menggunakan layanan jaringan (perhatikan luaran “Dialling” pada klien).

Sekarang, mari kita berkenalan sedikit mengenai barisan-barisan kode yang memungkinkan komunikasi yang sudah kalian lihat sebelumnya:

Server (sila buka berkas samples/udp/server/udpServer.go)

Kode	Fungsi
<pre>const (serverIP = "" serverPort = "54321" serverType = "udp4" bufferSize = 2048)</pre>	<p>Berikut adalah beberapa konstanta yang dipakai pada program ini, yaitu:</p> <ol style="list-style-type: none"> 1. serverIP: Menentukan di mana program ini harus mendengarkan pesan UDP. Kosong berarti server ini diminta untuk mendengarkan pesan UDP dari <i>interface</i> manapun. 2. serverPort: Menentukan nomor port yang digunakan untuk berkomunikasi dengan server ini. 3. serverType: UDP4 berarti kita menggunakan varian implementasi protokol UDP yang hanya beroperasi di IPv4 4. bufferSize: Menentukan ukuran buffer yang dimiliki oleh aplikasi ini yang digunakan untuk menerima pesan dari layanan transportasi

<code>listenAddress, err := net.ResolveUDPAddr(serverType, net.JoinHostPort(serverIP, serverPort))</code>	Pada bahasa pemrograman Go, umumnya kita perlu menggunakan struktur alamat yang spesifik untuk menghindari keambiguan. Baris kode ini akan membuat suatu struktur Alamat dengan tipe <i>server</i> yang kita inginkan (UDP di atas IPv4) yang mendengarkan pesan UDP pada IP dan Port yang kita tentukan di konstanta.
<code>socket, err := net.ListenUDP(serverType, listenAddress)</code>	Barisan ini membuat satu buah <i>socket</i> pada alamat UDP yang kita buat sebelumnya yang digunakan untuk mendengarkan pesan UDP yang masuk.
<code>defer socket.Close()</code>	Socket yang sudah dibuka direkomendasikan untuk ditutup. Pernyataan ini menyatakan bahwa kita memohon untuk menunda penutupan <i>socket</i> hingga fungsi <i>main</i> selesai dijalankan (baik itu selesai sempurna, diterminasi, maupun diberhentikan oleh error).
<code>for { ... }</code>	Ini adalah bentuk <i>infinite loop</i> yang disediakan oleh Go yang berarti kita akan terus mendengarkan pesan UDP sampai program diberhentikan.
<code>receiveBuffer := make([]byte, bufferSize)</code>	Barisan kode ini digunakan untuk membuat <i>buffer</i> pada aplikasi yang kita buat. Ini bukanlah <i>buffer</i> dari layanan transportasi. Nantinya, setiap kali kita mengambil pesan dari layanan transportasi, pesan tersebut akan diletakkan di sini.
<code>receiveLength, address, err := socket.ReadFromUDP(receiveBuffer)</code>	Barisan kode ini menunggu hingga ada pesan yang diantrikan di <i>buffer</i> layanan UDP dan membacanya untuk diletakkan di <i>buffer</i> yang sudah kita buat. Selain pesannya itu sendiri, kita juga akan mendapatkan informasi panjang pesan

	yang masuk (receiveLength) dan alamat pengirim pesan (address).
<code>go connectionHandler(socket, address, receiveBuffer, receiveLength)</code>	Barisan kode ini membuat sebuah <i>routine</i> baru yang memanggil fungsi <i>handler</i> sehingga penanganan pesan dapat dilakukan secara terpisah dari pemantauan pesan baru.
<code>func connectionHandler(socket *net.UDPConn, address *net.UDPAddr, receiveBuffer []byte, receiveLength int) { ... }</code>	
<code>message := string(receiveBuffer[:receiveLength])</code>	Dalam kasus ini, kita mengasumsikan bahwa metode komunikasi yang digunakan adalah UTF-8 String. Maka dari itu, konten <i>buffer</i> yang kita terima dapat kita konversi ke string.
	Perlu dicatat bahwa pesan yang diterima bisa jadi lebih pendek dari buffer yang kita alokasikan . Inilah fungsi dari <i>receiveLength</i> yang kita peroleh sebelumnya, yaitu untuk memotong pesan yang akan kita konversi sesuai panjang pesan yang diterima saja.
<code>response, err := logic(message)</code>	Pada umumnya, logika untuk memberikan luaran (<i>output</i>) berdasarkan suatu masukan pesan yang diterima (<i>input</i>) dilakukan pada modul atau metode berbeda, seperti di sini. Sila baca fungsi <i>logic</i> yang ada pada berkas yang sama untuk melihat logika yang berjalan untuk mendapatkan luaran yang diatur oleh aplikasi kita ini.
<code>_, err = socket.WriteToUDP([]byte(response), address)</code>	Setelah mendapatkan luaran yang akan dikirim kembali ke klien, baris kode ini akan meng- <i>cast</i> string kita menjadi sebuah <i>byte array</i> yang dapat diterima oleh layanan transportasi dan kemudian mengirimkannya melalui <i>socket</i> ke alamat

	klien yang sebelumnya mengirim pesan ini.
--	---

Client (sila buka berkas samples/udp/client/udpClient.go)

Kode	Fungsi
<pre>const (serverIP = "127.0.0.1" serverPort = "54321" serverType = "udp4" bufferSize = 2048)</pre>	<p>Kontennya serupa dengan Server, namun ada beberapa fungsi yang berbeda, yaitu:</p> <ol style="list-style-type: none"> 1. serverIP kali ini mendefinisikan server yang akan kita hubungi 2. serverPort kali ini mendefinisikan nomor port yang akan digunakan untuk “mencari” proses aplikasi server yang kita tuju
<pre>remoteUdpAddress, err := net.ResolveUDPAddr(serverType, net.JoinHostPort(serverIP, serverPort))</pre>	<p>Kontennya serupa dengan Server, namun kali ini mendefinisikan alamat UDP yang akan digunakan untuk menghubungi server.</p>
<pre>socket, err := net.DialUDP(serverType, nil, remoteUdpAddress)</pre>	<p>Klien juga akan membentuk socket untuk berkomunikasi dengan server. Bedanya, ketika Server menggunakan socket untuk mendengarkan pesan UDP, Klien menggunakan socket untuk menghubungi Server.</p>
<pre>defer socket.Close()</pre>	<p>Fungsinya sama seperti Server</p>
<pre>receiveBuffer := make([]byte, bufferSize)</pre>	<p>Fungsinya mirip seperti Server. Ini nantinya dipakai untuk menampung pesan balasan dari Server.</p>
<pre>message, err := bufio.NewReader(os.Stdin).ReadString('\n')</pre>	<p>Kode ini digunakan untuk memperoleh masukan dari Standard Input (masukan dari console yang kalian ketikkan)</p>

<code>_ , err = socket.Write([]byte(message))</code>	Setelah melakukan <i>type casting</i> pesan yang diterima dari Standard Input menjadi <i>byte array</i> , <i>socket</i> yang sudah dibuat akan mengirimkannya ke tujuan yang telah ditentukan.
<code>receiveLength, err := socket.Read(buffer)</code>	Klien akan menunggu hingga ada pesan yang masuk ke <i>buffer</i> layanan transportasi dan memasukkannya ke <i>buffer</i> aplikasi yang sudah dibuat. Selain pesannya itu sendiri, kita juga akan mendapatkan informasi panjang pesan yang masuk (<code>receiveLength</code>).
<code>response := string(buffer[:receiveLength])</code>	Fungsinya sama seperti di Server.

Jika kamu perhatikan, tidak banyak hal yang perlu kita lakukan untuk dapat melakukan komunikasi dengan protokol UDP. Hal ini tentunya sesuai dengan karakteristik protokol UDP yang diharapkan oleh aplikasi-aplikasi yang tidak membutuhkan banyak fitur-fitur dan mementingkan kecepatan dan kesederhanaan.

Lalu, bagaimana jika kita ingin menuntut layanan transportasi yang menerapkan *Reliable Data Transfer*? Tentu, kita membutuhkan protokol yang berbeda

We Need to Talk : Connection-Oriented Socket (TCP)

Berbeda dengan sebelumnya, *socket* yang dibangun di atas protokol layanan transportasi TCP akan mendapatkan akses ke beberapa fasilitas yang berkaitan dengan keandalan perpindahan data (*reliable data transfer*). Kalian seharusnya sudah memahami bahwa sifat utama dari sebuah komunikasi yang menggunakan layanan TCP adalah *connection-oriented* yang berarti setiap komunikasi yang terjadi harus membangun suatu koneksi di mana setiap komunikasi akan bergantung pada koneksi yang dibangun tersebut. Mari kita coba!

1. Siapkan dua buah jendela Command Prompt / Terminal / PowerShell / sejenisnya di **komputer lokal kamu**. Kamu boleh menggunakan tmux atau alat serupa **jika paham cara menggunakan**.
2. Navigasi ke *root directory* dari direktori kerja H01 ini (folder h01-source-code).
3. Jalankan program *socket server* TCP di salah satu jendela dengan komando berikut ini:

```
go run jarkom.cs.ui.ac.id/h01/samples/tcp/server
```

```
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/server
TCP Server Socket Program Example in Go
Press Ctrl+C or Cmd+C to stop the program
[tcp4] Listening on: 0.0.0.0:54321
```

4. Jalankan program *socket client* TCP di jendela lainnya dengan komando berikut ini:

```
go run jarkom.cs.ui.ac.id/h01/samples/tcp/client
```

```
rafi_muhammad81@ubuntu-rf:~$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/client
no required module provides package jarkom.cs.ui.ac.id/h01/samples/tcp/client: go.mod file not found
see 'go help modules'
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/client
TCP Client Socket Program Example in Go
[tcp4] Dialling from 127.0.0.1:40276 to 127.0.0.1:54321
[tcp4] Creating receive buffer of size 2048
[tcp4] Input message to be sent to server: |
```

Berikan masukan berupa *string* apapun pada program klien dan tekan Enter. Perhatikan alur komunikasi yang terjadi antara program *server* dan *client* yang terjadi!

5. Hentikan eksekusi program *socket server* dengan menekan tombol Ctrl+C pada jendela yang menjalankan *socket server*

Selayang pandang, mungkin kalian tidak melihat perbedaan yang signifikan antara program yang sebelumnya dengan program yang ini. Namun, coba kalian bandingkan luaran *server* TCP sebelum dan sesudah klien dijalankan (namun sebelum klien mengirimkan pesan)!

Sebelum

```
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/server
TCP Server Socket Program Example in Go
Press Ctrl+C or Cmd+C to stop the program
[tcp4] Listening on: 0.0.0.0:54321
```

Sesudah

```
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/server
TCP Server Socket Program Example in Go
Press Ctrl+C or Cmd+C to stop the program
[tcp4] Listening on: 0.0.0.0:54321
[tcp4] Receive connection from 127.0.0.1:40276
[tcp4] [Client: 127.0.0.1:40276] Creating receive buffer for connection of size 2048
```

Sudah melihat bedanya? Perhatikan baris kedua terakhir yang menyebutkan “**Receive connection from:**”. Meskipun kita belum mengirimkan pesan apapun, *Server* saat ini sudah *aware* bahwa ada klien yang ingin berkomunikasi. Mengapa hal ini bisa terjadi? Mari kita bedah sedikit kode sumbernya:

Catatan: Banyak metode socket di Go yang mirip antara protokol TCP maupun UDP. Sepertinya hal ini dilakukan untuk memenuhi prinsip Don't Repeat Yourself (DRY) dan

memberikan antarmuka yang konsisten bagi pengguna yang mungkin tidak terlalu peduli dengan layanan transportasi yang digunakan. Meskipun fungsi yang terlihat di kita terkesan sama, implementasinya bisa saja berbeda.

Server (sila buka berkas samples/tcp/server/tcpServer.go)

Kode	Fungsi
<pre>const (serverIP = "" serverPort = "54321" serverType = "tcp4" bufferSize = 2048)</pre>	Sama seperti UDP Server, dengan pengecualian tipe server yang kali ini bernilai tcp4 (Protokol TCP di atas IPv4).
<pre>listenAddress, err := net.ResolveTCPAddr(serverType, net.JoinHostPort(serverIP, serverPort))</pre>	Sama seperti UDP Server, namun kali ini digunakan untuk membentuk struktur alamat TCP IPv4.
<pre>socket, err := net.ListenTCP(serverType, listenAddress)</pre>	Sama seperti UDP Server
<pre>defer socket.Close()</pre>	Sama seperti UDP Server
<pre>for { ... }</pre>	<i>Loop</i> untuk mendengarkan komunikasi secara terus menerus
<pre>connection, err := socket.AcceptTCP()</pre>	Di Sini Letak Perbedaan TCP dan UDP Server kali ini bukan menunggu adanya pesan baru yang diterima, melainkan koneksi baru yang diminta oleh klien. Setelah menerima koneksi baru ini, socket baru akan dibuat untuk melayani koneksi baru ini.
<pre>go connectionHandler(connection)</pre>	Instruksi ini akan membuat <i>routine</i> baru untuk melayani koneksi baru ini. Hal ini membebaskan <i>main routine</i> untuk mendengarkan dan menerima koneksi baru.
<pre>receiveBuffer := make([]byte, bufferSize)</pre>	Sama seperti UDP Server
<pre>defer connection.Close()</pre>	Kali ini, ada socket lain yang perlu kalian tutup, yaitu socket yang melayani masing-masing koneksi yang dibuat. Kita juga akan menggunakan keyword <i>defer</i> agar

	penutupan ini dilakukan otomatis saat fungsi connectionHandler selesai dijalankan.
<code>receiveLength, err := connection.Read(receiveBuffer)</code>	Kali ini, kita perlu membaca pesan dari socket koneksi yang telah dibuat. Perhatikan bahwa informasi alamat pengirim kini sudah tidak dikembalikan!
<code>message := string(receiveBuffer[:receiveLength])</code>	Sama seperti UDP Server
<code>response, err := logic(message)</code>	Sama seperti UDP Server
<code>_, err = connection.Write([]byte(response))</code>	Ketika kita ingin menuliskan pesan untuk dikembalikan ke klien, kita tidak perlu menyebutkan alamat tujuannya.

Client (sila buka berkas samples/tcp/client/tcpClient.go)

Jika kalian perhatikan dengan seksama kode dari *socket client* berbasis TCP di Go, kalian hampir tidak akan menemukan perbedaan yang mencolok antara *client* berbasis UDP dan TCP, selain kata-kata UDP yang digantikan oleh TCP. Misalnya,

UDP Client:

```
socket, err := net.DialUDP(serverType, nil, remoteUdpAddress)
if err != nil {
log.Fatalln(err)
}
```

TCP Client:

```
socket, err := net.DialTCP(serverType, nil, remoteTcpAddress)
if err != nil {
log.Fatalln(err)
}
```

Meskipun prosedurnya cenderung sama (i.e., memanggil Kumpulan metode yang mirip dengan urutan yang mirip), hal yang akan terjadi di balik layar akan berbeda.

[30 Poin] Reflection 1 : Connectionless vs Connection-Oriented Socket

[10 Poin] R1-1. Personalisasi

Aplikasi yang beroperasi dalam jaringan tidak akan terasa dampaknya jika kita tidak menggunakan jaringan real dengan komunikasi antara sepasang *host* yang berbeda. Dalam kasus ini, misalnya kita menginginkan agar server UDP maupun TCP kita dijalankan di **VM1** yang kita miliki melalui **nomor port unik masing-masing** dan kita ingin agar server tersebut **dapat diakses oleh komputer lokal kita**. Perlu dicatat bahwa mesin GCP memiliki alamat IP publik dan alamat IP pribadi.

Modifikasi kode-kode sample yang diberikan agar kebutuhan di atas dapat tercapai! Selain modifikasi kode, jelaskan bagian kode mana yang kalian ubah dan alasan kalian dalam mengubah bagian tersebut!

[10 Poin] R1-2. Perbandingan Lalu Lintas

Pada bagian ini, kita akan mencoba mengobservasi lalu lintas dari masing-masing pasangan aplikasi yang kita punya dengan menggunakan Wireshark. Untuk mengumpulkan data bahan refleksi ini, lakukan langkah berikut:

1. Pindahkan **keseluruhan direktori kerja yang sudah diubah kodennya sesuai bagian R1-1 ke VM1**. Kalian dapat mengompres direktori kerjanya menjadi ZIP untuk diunggah ke VM1.
2. Siapkan satu buah jendela Command Prompt / Terminal / PowerShell / sejenisnya di **komputer lokal kamu**.
3. Siapkan satu buah jendela SSH di **VM1**.
4. Navigasi ke *root directory* dari direktori kerja H01 ini (folder h01-source-code) pada kedua jendela.
5. Jalankan aplikasi Wireshark di komputer lokal kamu.
6. Jalankan *capture* pada *interface* yang digunakan untuk berkomunikasi dengan internet.
7. Jalankan **UDP Server di VM1**, kemudian jalankan **UDP Client di komputer lokal kamu** dan berikan input sampai selesai. Hentikan eksekusi UDP Server jika sudah selesai.
8. Jalankan **TCP Server di VM1**, kemudian jalankan **TCP Client di komputer lokal kamu** dan berikan input sampai selesai. Hentikan eksekusi TCP Server jika sudah selesai.
9. Hentikan *capture* dan simpan berkas Wiresharknya. Berkas ini tidak akan dikumpulkan namun dibutuhkan untuk menjawab refleksi ini.

Setelah memperoleh data koneksi untuk kedua protokol ini, jawablah pertanyaan berikut ini:

Temukan perbedaan lalu lintas komunikasi yang terjadi antara klien-server TCP dengan klien-server UDP! Komunikasi apa yang ada di salah satu protokol dan tidak ada di protokol lawannya? Kaitkan dengan konsep Connectionless vs Connection-Oriented!

[10 Poin] R1-3. Bagaimana Jika Server Tidak Ada?

Pernahkah kamu mencoba membuka koneksi ke suatu layanan tertentu sedangkan layanan tersebut tidak ada atau sedang tidak menerima koneksi? Mari kita coba!

1. Pastikan UDP Server maupun TCP Server di VM1 kamu sudah dimatikan.
2. Siapkan dua buah jendela Command Prompt / Terminal / PowerShell / sejenisnya di **komputer lokal kamu**.
3. Navigasi ke *root directory* dari direktori kerja H01 ini (folder h01-source-code) pada kedua jendela.

4. Jalankan aplikasi Wireshark di komputer lokal kamu.
5. Jalankan *capture* pada *interface* yang digunakan untuk berkomunikasi dengan internet.
6. Jalankan **UDP Client di komputer lokal kamu pada jendela pertama.**
7. Jalankan **TCP Client di komputer lokal kamu pada jendela kedua.**
8. Tunggu sekitar 10 detik.
9. Hentikan *capture* dan simpan berkas Wiresharknya. Berkas ini tidak akan dikumpulkan namun dibutuhkan untuk menjawab refleksi ini.

Setelah memperoleh data koneksi untuk kedua protokol ini, jawablah pertanyaan berikut ini:

Apakah kedua aplikasi dapat menyadari bahwa Server tidak dapat dihubungi (memunculkan error)? Jika Ya, perhatikan hasil capture Wireshark dan coba identifikasi paket mana yang membantu aplikasi memunculkan error tersebut!

A Quicker Way: Connection-Oriented Socket (QUIC)

Pada bagian sebelumnya, kita telah berkenalan dengan 2 tipe *socket* yang mempergunakan layanan transportasi paling populer saat ini, yaitu TCP dan UDP. Kedua protokol tersebut memiliki keunggulan dan kelemahannya masing-masing dan akan cocok dengan kebutuhan-kebutuhan tertentu. Namun, ada beberapa skenario yang cukup sering ditemukan pada koneksi di internet yang menuntut adanya protokol baru. Tuntutan ini terutama datang dari protokol aplikasi “kesayangan” kita semua: HTTP.

HTTP yang selama ini mempergunakan protokol transportasi TCP mengalami kesulitan ketika menangani komunikasi berjumlah besar. Semakin kompleksnya situs web saat ini dengan asset-assetnya yang harus dimuat secara terpisah menimbulkan masalah yang disebut **Head-of-Line Blocking (HOL Blocking)**. HOL Blocking dan beberapa pertimbangan lain menuntun komunitas warga internet untuk membangun protokol baru, yaitu **QUIC**.

QUIC sebenarnya merupakan protokol yang dibangun di atas protokol UDP sehingga, *in some ways*, QUIC bisa dikatakan sebagai suatu program *socket*. Namun, perbedaan QUIC dengan aplikasi yang beroperasi di atas *socket* pada umumnya, QUIC itu sendiri memberikan layanan-layanan yang lazimnya dipegang oleh lapisan transportasi. Hal ini memperlihatkan salah satu karakteristik dari UDP, yaitu *bring-your-own-solution* untuk hal-hal seperti *reliable data transfer*, *congestion control*, *security*, dan lain-lain. QUIC memberikan antarmuka yang konsisten untuk menikmati layanan-layanan yang telah disebutkan sebelumnya dengan juga memberikan beberapa fitur baru atau perkembangan dari *transport* yang sudah ada.

Salah satu hal yang diperkenalkan pada QUIC adalah konsep **stream**. **Stream** adalah enkapsulasi paket yang memungkinkan beberapa paket dapat bertransaksi sekaligus melalui satu koneksi. Jika sebelumnya sepasang klien-server hanya bisa mengkomunikasikan satu pesan dalam satu waktu dalam satu koneksi, *stream* memungkinkan untuk memuat beberapa pesan sekaligus. Keunggulan yang dapat dicapai dari pendekatan ini bisa kalian gali lebih dalam dengan membaca dokumen standar terkait dengan QUIC yang dirilis oleh IETF.

Mari kita coba!

1. Siapkan dua buah jendela Command Prompt / Terminal / PowerShell / sejenisnya di **komputer lokal kamu**. Kamu boleh menggunakan tmux atau alat serupa **jika paham cara menggunakannya**.
2. Navigasi ke *root directory* dari direktori kerja H01 ini (folder h01-source-code).
3. Jalankan program *socket server QUIC* di salah satu jendela dengan komando berikut ini:

```
go run jarkom.cs.ui.ac.id/h01/samples/quic/server
```

```
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/quic/server
go: downloading github.com/quic-go/quic-go v0.38.1
go: downloading github.com/francoisq/gojay v1.2.13
go: downloading golang.org/x/net v0.10.0
go: downloading golang.org/x/sys v0.0.0
go: downloading golang.org/x/exp v0.0.0-20221205204356-47842c84f3db
go: downloading golang.org/x/crypto v0.4.0
QUIC Server Socket Program Example in Go
[udp4] Preparing UDP listening socket on 0.0.0.0:54321
2023/09/25 11:36:41 failed to sufficiently increase receive buffer size (was: 208 kib, wanted: 2048 kib, got: 416 kib). See https://github.com/quic-go/quic-go/wiki/UDP-Buffer-Sizes for details.
[quic] Listening QUIC connections on 0.0.0.0:54321
```

QUIC saat ini belum bergabung sebagai sebuah *standard library* di Go. Maka dari itu, program ini menggunakan *library* eksternal bernama quic-go. *Dependency* ini sudah diatur di go.mod yang kami berikan dan program akan otomatis mengunduhnya jika belum ada di sistem.

4. Jalankan program *socket client QUIC* di jendela lainnya dengan komando berikut ini:

```
go run jarkom.cs.ui.ac.id/h01/samples/quic/client
```

```
rafi_muhammad81@ubuntu-rf:~$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/client
no required module provides package jarkom.cs.ui.ac.id/h01/samples/tcp/client: go.mod file not found
  see 'go help modules'
rafi_muhammad81@ubuntu-rf:~$ cd h01-source-code/
rafi_muhammad81@ubuntu-rf:~/h01-source-code$ go run jarkom.cs.ui.ac.id/h01/samples/tcp/client
TCP Client Socket Program Example in Go
[tcp4] Dialling from 127.0.0.1:40276 to 127.0.0.1:54321
[tcp4] Creating receive buffer of size 2048
[tcp4] Input message to be sent to server: ■
```

Berikan masukan berupa *string* apapun pada program klien dan tekan Enter. Perhatikan alur komunikasi yang terjadi antara program *server* dan *client* yang terjadi!

5. Hentikan eksekusi program *socket server* dengan menekan tombol Ctrl+C pada jendela yang menjalankan *socket server*.

Perhatikan perbedaan luaran yang terjadi antara TCP dan UDP sebelumnya dengan QUIC! Ketika suatu pihak yang terlibat dalam komunikasi menggunakan QUIC, komunikasi tidak dapat dikirimkan begitu saja, melainkan harus dibungkus dalam suatu *stream* yang nantinya dapat diisi oleh pesan yang diinginkan. Mari kita coba bedah sedikit kode dari *socket* berbasis QUIC ini:

Server (sila buka berkas samples/quic/server/quicServer.go)

Kode	Fungsi
------	--------

<pre>const (serverIP = "" serverPort = "54321" serverType = "udp4" bufferSize = 2048 appLayerProto = "jarkom-quic-sample-minjar")</pre>	Mayoritas masih sama dengan UDP Server karena QUIC pada dasarnya adalah <i>wrapper</i> untuk UDP dengan tambahan fitur.
	Namun, perhatikan bagian appLayerProto . Pada QUIC, kalian dapat menegosiasikan protokol lapisan aplikasi yang digunakan di lapisan ini saat melakukan <i>handshake</i> . Sistem ini dinamakan Application Layer Protocol Negotiation (ALPN). Salah satu latar belakang adanya fitur ini adalah HTTP/1.1, 2, dan 3 yang <i>secure</i> sama-sama menggunakan port 443.
<pre>localUdpAddress, err := net.ResolveUDPAddr(serverType, net.JoinHostPort(serverIP, serverPort))</pre>	Masih sama dengan UDP Server.
<pre>socket, err := net.ListenUDP(serverType, localUdpAddress)</pre>	Untuk membuat sistem komunikasi melalui QUIC, kita perlu membuat <i>socket</i> UDP terlebih dahulu. Nantinya, QUIC akan memanfaatkan <i>socket</i> ini dan menambahkan layanan-layanan miliknya.
<pre>defer socket.Close()</pre>	Masih sama dengan UDP Server.
<pre>tlsConfig := &tls.Config{ Certificates: []tls.Certificate{ utils.GenerateTLSelfSignedCertificates(), }, NextProtos: []string{appLayerProto}, }</pre>	Salah satu hal yang menjadi fitur QUIC adalah mekanisme <i>security</i> yang sudah bawaan. Setiap kali kita membuka koneksi QUIC, kita perlu melakukan konfigurasi TLS.
	Kita akan mempelajari bagian ini lebih dalam pada Bab 6: Security.
<pre>listener, err := quic.Listen(socket, tlsConfig, &quic.Config{})</pre>	Bagian kode ini mempersiapkan <i>listener</i> berbasis QUIC yang akan

	mendengarkan koneksi berbasis QUIC melalui <i>socket UDP</i> yang telah dibuat.
	Konfigurasi QUIC yang akan kita gunakan adalah konfigurasi kosongan (<i>default</i>).
<code>defer listener.Close()</code>	Fungsinya mirip dengan penutupan <i>socket</i> , yaitu berhenti mendengarkan koneksi QUIC melalui <i>socket</i> ini setelah fungsi <i>main</i> selesai dijalankan.
<code>for { ... }</code>	
<code>connection, err := listener.Accept(context.Background())</code>	Karena QUIC adalah <i>connection-oriented protocol</i> , komunikasi antara klien yang datang pertama kali adalah permintaan membentuk koneksi. Namun, pada QUIC, proses ini tidak membuat <i>socket</i> baru, melainkan hanya menyediakan enkapsulasi.
<code>go handleConnection(connection)</code>	Seperti pada TCP, setiap koneksi perlu ditangani di <i>coroutine</i> berbeda agar tidak menghalangi <i>main coroutine</i> untuk terus mendengarkan koneksi baru.
<code>func handleConnection(connection quic.Connection) { ... }</code>	Fungsi ini akan menangani koneksi yang dioper oleh <i>main coroutine</i>
<code>stream, err := connection.AcceptStream(context.Background())</code>	Pada tahapan ini, server akan menunggu hingga diterima sebuah permintaan pembuatan <i>bidirectional stream</i> (<i>stream</i> di mana kedua belah pihak bisa mengirim pesan) dari kliennya.

<code>go handleStream(connection.RemoteAddr(), stream)</code>	Lagi-lagi kita perlu menangani <i>stream</i> ini secara terpisah di <i>coroutines</i> yang berbeda :D
<code>func handleStream(clientAddress net.Addr, stream quic.Stream) { ... }</code>	Fungsi inilah yang akan menerima permintaan untuk menangani <i>stream</i> pada koneksi yang sudah dibuka sebelumnya.
<code>_ , err := io.Copy(logicProcessorAndWriter{stream}, stream)</code>	Karena satu dan lain hal pada <i>library</i> QUIC yang dipakai, kita tidak dapat menggunakan metode pembacaan dan penulisan pada umumnya.
	Cara ini memanfaatkan karakteristik <i>stream</i> yang bisa “dibaca” (implementasi <i>io.Reader</i>) dan “ditulis” (implementasi <i>io.Writer</i>). Secara <i>vanilla</i> , solusi ini sebenarnya hanya “menyalin” hasil bacaan <i>stream</i> kembali ke <i>stream</i> tersebut. Namun, bagian <code>logicProcessorAndWriter{stream}</code> bisa melakukan pencegatan (<i>interception</i>) data yang akan disalin sehingga bisa diolah sebelum ditulis kembali.
<code>type logicProcessorAndWriter struct{ io.Writer }</code>	Agar dapat melakukan pencegatan data yang akan disalin, kita bisa membungkus <i>stream</i> yang menerapkan <i>interface</i> <i>io.Writer</i> dengan kelas lain yang bisa modifikasi perilaku “menulisnya”.
<code>func (lp logicProcessorAndWriter) Write(receivedMessageRaw []byte) (int, error) { ... }</code>	Fungsi inilah yang nanti akan memodifikasi perilaku “menulis” <i>stream</i> sehingga bisa kita olah terlebih dahulu.
<code>receivedMessage := string(receivedMessageRaw)</code>	Dalam hal ini, QUIC akan otomatis menyesuaikan panjang

	pesan sesuai konten yang diterima (tidak perlu dipotong).
<code>writeLength, err := lp.Writer.Write([]byte(response))</code>	Pada bagian ini, setelah kita mengolah masukan menjadi luaran, kita dapat memanggil “alat penulis” <i>stream</i> untuk menulis luaran yang kita harapkan.

Client (sila buka berkas samples/quic/client/quicClient.go)

Kode	Fungsi
<code>const (serverIP = "127.0.0.1" serverPort = "54321" serverType = "udp4" bufferSize = 2048 appLayerProto = "jarkom-quic-sample-minjar" sslKeyLogFileName = "ssl-key.log")</code>	Mayoritas sama dengan UDP Client, dengan beberapa perbedaan seperti: 1. appLayerProto juga ada di sini selain di QUIC Server. Di klien, fungsi dari variable ini adalah untuk mencoba bertanya apakah protokol yang kita tulis dapat dilayani oleh server. 2. sslKeyLogFileName dapat digunakan untuk mengekspor kunci TLS yang digunakan untuk mengenkripsi koneksi agar nantinya dapat dibaca oleh Wireshark.
<code>sslKeyLogFile, err := os.Create(sslKeyLogFileName) defer sslKeyLogFile.Close()</code>	Membuat berkas SSL Keylog File
<code>tlsConfig := &tls.Config{ InsecureSkipVerify: true, NextProtos: []string{appLayerProto}, KeyLogWriter: sslKeyLogFile,</code>	Pada konfigurasi TLS ini, kita meminta QUIC untuk mengabaikan verifikasi

<code>}</code>	sertifikat TLS (karena Server kita akan menggunakan sertifikat <i>self-signed</i>), memilih protokol yang kita tulis di appLayerProto untuk berkomunikasi dengan server, dan menulis kunci-kunci TLS yang diperoleh ke berkas yang sudah dibuat sebelumnya.
<code>connection, err := quic.DialAddr(context.Background(), net.JoinHostPort(serverIP, serverPort), tlsConfig, &quic.Config{})</code>	Pada bagian ini, kita akan memanggil server yang kita inginkan melalui layanan transportasi QUIC dengan konfigurasi TLS yang sudah kita buat sebelumnya.
<code>defer connection.CloseWithError(0x0, "No Error")</code>	Dalam kasus ini, kita akan membuat komunikasi ini selesai jika fungsi main selesai dieksekusi dengan men- <i>defer</i> penutupan koneksi dengan kode 0x0 (No Error). Hal ini diperlukan karena alasan penutupan koneksi di QUIC harus dibuat dengan jelas.
<code>receiveBuffer := make([]byte, bufferSize)</code>	Sama seperti UDP Client
<code>message, err := bufio.NewReader(os.Stdin).ReadString('\n')</code>	Sama seperti UDP Client
<code>stream, err := connection.OpenStreamSync(context.Background())</code>	Komando ini akan membuka <i>stream</i> antara klien dengan server untuk berkomunikasi.
<code>, err = stream.Write([]byte(message))</code>	Komando ini akan mengirimkan pesan yang sudah dikonversi menjadi sebuah <i>byte array</i> melalui <i>stream</i> yang sudah dibuat.
<code>receiveLength, err := stream.Read(receiveBuffer)</code>	Setelah mengirimkan pesan, klien akan menunggu sampai ada pesan yang

	diterima oleh <i>buffer</i> QUIC yang kemudian dibaca dan disimpan di <i>buffer</i> aplikasi klien.
	Informasi tentang panjang pesan yang diterima akan diperoleh pada hasil pengembalian nilai fungsi melalui variabel receiveLength .
<code>response := receiveBuffer[:receiveLength]</code>	Variabel <code>receiveLength</code> di atas kemudian dipergunakan untuk memotong pesan dari <i>buffer</i> agar sesuai dengan panjang pesan aslinya.

Agreeing Upon a Language: Encoding – Decoding

Dalam membentuk suatu protokol aplikasi, kita perlu mengakomodasi fakta bahwa layanan transportasi yang akan kita pakai **hanya akan menerima data berupa kumpulan bytes (bytes array)**, terlepas dari pesan yang ingin kita sampaikan. Hal ini menuntut aplikasi socket yang dikembangkan untuk dapat memiliki mekanisme agar kedua belah pihak dapat menyepakati cara untuk mengkonversi pesan aplikasi ke sebuah *bytes array* dan sebaliknya. Alat yang digunakan untuk mengimplementasikan mekanisme ini biasa disebut *codec*.

Ada dua pendekatan yang dapat digunakan dalam menyusun mekanisme ini dan protokol aplikasi dapat memilih salah satu atau bahkan keduanya, yaitu **String Encoding/Decoding (UTF-8)** dan **Binary Encoding/Decoding**. *String Encoding/Decoding* adalah mekanisme untuk mengkonversi pesan berbentuk *string* menjadi sebuah *bytes array* dengan bantuan *codec* UTF-8 di mana kita bisa membentuk pesan sebagai sebuah *string* dan langsung mengkonversi keseluruhannya memakai *codec* tersebut. Solusi ini dipakai secara penuh pada HTTP/1.1 dan secara parsial di beberapa protokol seperti DNS. Di sisi lain, *binary encoding* berarti kita perlu mengonversi struktur data yang kita miliki menjadi susunan angka biner yang dapat dibentuk sebagai *bytes array*.

Dalam pendekatan berbasis *string*, mayoritas bahasa pemrograman sudah memiliki cara untuk langsung mengkonversi suatu *string* menjadi sebuah *bytes array*. Dalam bahasa pemrograman Go, hal ini dapat dicapai dengan *type casting* sebagai berikut:

<code>[]byte (message)</code>

Di sisi lain, dalam melakukan *bytes encoding/decoding*, kita perlu melakukan manipulasi *bit* secara matematis untuk dapat mencapai format pesan yang diharapkan. Dalam hal ini, setiap bagian pesan akan memiliki representasi *bit*-nya masing-masing dengan panjang tertentu. Tentunya, jika pesan terdiri dari banyak komponen, terutama komponen-komponen pada *header*, manipulasinya akan semakin sulit. Maka dari itu, beberapa bahasa pemrograman seperti Go menyediakan mekanisme untuk dapat melakukan *serialization / deserialization* dari/ke objek dan *bytes array*.

Perhatikan kode **Codec** yang ada pada direktori **h01-source-code/samples/codec/codec.go!**

```
type MessageFixedSegment struct {
    Id          uint16
    Headers     uint8
    MessageLength uint8
}
```

Pada contoh *struct* di atas, kita memiliki struktur pesan yang terdiri dari tiga komponen. Masing-masing komponen memiliki tipe datanya masing-masing yang juga menunjukkan panjang dari komponen tersebut. Sebagai contoh, komponen Id bertipe data *unsigned integer* 16-bit (uint16) yang nantinya akan dimasukkan ke dalam *bytes array* sebagai 2 *bytes* berbeda. Bagaimana dengan urutannya? Jaringan secara umum menggunakan pengurutan ***Big-Endian***, sehingga *Most Significant Bit* (MSB) dari Id akan berada di sisi kiri (*byte* pertama).

```
sampleMessage := MessageFixedSegment{
    Id:           1234,
    Headers:      uint8(headers),
    MessageLength: uint8(len(message)),
}

fmt.Println(sampleMessage)

bytesBuffer := new(bytes.Buffer)
binary.Write(bytesBuffer, binary.BigEndian, sampleMessage)
...

finalBytes := bytesBuffer.Bytes()
```

Dengan adanya definisi *struct* seperti di atas, kita bisa menggunakan modul *binary* untuk “menulis” *bytes array* yang sesuai dengan urutan kemunculan komponen pada *struct* dan spesifikasi yang kita berikan.

Cara yang serupa juga dapat kita gunakan untuk “membaca” *bytes array* menjadi struktur data yang kita inginkan kembali.

```
var decodedFixedSegment MessageFixedSegment
bytesReader := bytes.NewReader(finalBytes)
binary.Read(bytesReader, binary.BigEndian, &decodedFixedSegment)
```

Lalu, bagaimana jika kita ingin menyusun data yang panjangnya tidak memiliki tipe data yang bersesuaian? *Unsigned Integer* pada Go hanya tersedia pada panjang 8, 16, 32, dan 64 bit saja. Di luar itu, kita perlu sedikit kreatif:

Untuk data yang hanya memiliki panjang 1 bit, tipe data boolean dapat dipakai.

Untuk data yang memiliki panjang yang tidak sejajar dengan *unsigned integer* atau *Boolean*, kita dapat mengambil panjang terdekat dan mengaturnya dengan *bitwise manipulation*. Misalnya, jika kita ingin memiliki 2 komponen data dengan panjang masing-masing 4 bit, maka kita dapat menggabungkan kedua komponen tersebut menjadi uint8 di mana komponen pertama kita *shift* sehingga menduduki 4 bit pertama. Contoh (dapat dilihat pada kode):

```
messageType := 0x1      // Assume length is 4 bits
messagePurpose := 0x5 // Assume length is 4 bits
headers := messageType<<4 + messagePurpose
```

Cara di atas dapat digunakan untuk panjang berapapun. Di sisi penerima, kita dapat menggunakan operasi modulo untuk “memotong” data dimulai dari belakang.

```
// Cara mengambil x bit paling belakang: data mod 2^x
// Misalnya, untuk mengambil 4 bit, kita lakukan modulo 16 (2^4)
messagePurpose = int(decodedFixedSegment.Headers) % 16
// Untuk “menghapus” bagian yang sudah kita baca untuk mendapatkan
// bagian lainnya, kita bisa lakukan bitwise shifting
messageType = int(decodedFixedSegment.Headers) >> 4
```

Pada beberapa kasus seperti yang dicontohkan pada program di atas, kita bisa saja memerlukan pemutuan data yang panjangnya tidak *fixed*. Misalnya, kita perlu mengirimkan pesan dari pengguna yang kita tidak tahu panjangnya sampai pengguna memasukkan pesan tersebut. Hal ini dapat kita akomodasi dengan melakukan *appending* terhadap data *bytes array* yang sudah kita buat sebelumnya.

```
bytesBuffer := new(bytes.Buffer)
binary.Write(bytesBuffer, binary.BigEndian, sampleMessage)
bytesBuffer.WriteString(message)

finalBytes := bytesBuffer.Bytes()
```

Namun, perlu diperhatikan bahwa kedua belah pihak tetap harus menyepakati panjang pesan tersebut agar prosedur konversi dapat berjalan dengan baik. Maka dari itu, biasanya panjang dari suatu komponen yang panjangnya bervariasi akan ditulis pada komponen lain **sebelum kemunculan pesan tersebut**. Dalam kasus ini, komponen *messageLength* yang panjangnya sudah *fixed* adalah komponen yang kita gunakan untuk menyimpan informasi ini.

```
sampleMessage := MessageFixedSegment{
    Id:          1234,
    Headers:     uint8(headers),
    MessageLength: uint8(len(message)),
}
```

Dengan adanya informasi ini, pihak yang menerima dapat mengetahui panjang dari komponen message ini dan mengatur proses *decoding* agar dapat membaca pesan dengan benar.

```
decodedMessage := make([]byte, decodedFixedSegment.MessageLength)
bytesReader.Read(decodedMessage)
```

Sila pelajari lebih lanjut kode yang telah diberikan.

[20 Poin] Reflection 2: QUIC Internalization

Modifikasi program QUIC Server dan QUIC Client yang diberikan sehingga dapat memenuhi kriteria berikut ini:

1. Kedua belah program harus bisa menyepakati bahwa protokol aplikasi yang akan digunakan adalah **jarkom-quic-sample-<nama panggilanmu>**. Contoh: jarkom-quic-sample-alice.
2. QUIC Server dan Client dimodifikasi agar dapat berkomunikasi melalui 2 *stream* sekaligus. Pesan yang dimasukkan oleh pengguna dikirimkan ke kedua *stream* tersebut secara paralel.

Jelaskan modifikasi kode yang kamu lakukan untuk memenuhi kriteria di atas dan lampirkan bukti eksekusi QUIC Server dan Client di **komputer lokal kamu** yang menunjukkan bahwa kriteria kedua benar-benar berhasil terpenuhi. Contoh luaran yang sesuai adalah sebagai berikut (perhatikan 2 *stream* ID yang berbeda):

```
[quic] [Client: 127.0.0.1:59774] Receive stream open request with ID 0
[quic] Receive message: abcde

[quic] Send message: ABCDE

[quic] [Client: 127.0.0.1:59774] Receive stream open request with ID 4
[quic] Receive message: abcde

[quic] Send message: ABCDE
```

[50 Poin] Hands-on Simple Project



Gambar 1. Layar PIDS pada Peron 1 Stasiun Cikoko LRT Jabodebek

Jika kalian pernah menggunakan transportasi umum, kalian mungkin sudah tidak asing dengan layar-layar yang ditempatkan di halte atau stasiun dan menampilkan informasi terkait dengan kedatangan atau keberangkatan armada transportasi umum pada halte atau stasiun tersebut. Layar-layar tersebut dapat menampilkan informasi dengan adanya suatu sistem yang dinamakan *Passenger Information Display System* (PIDS). Sistem ini memastikan bahwa pengguna layanan transportasi umum dapat memperoleh informasi dengan cepat dan akurat. Pada umumnya, sistem ini bisa menyediakan informasi seperti waktu kedatangan, jurusan, potensi keterlambatan, dan informasi lain yang dibutuhkan oleh pengguna.

Peran sistem ini semakin penting pada halte atau stasiun yang dilayani oleh lebih dari satu layanan transportasi umum, terutama bagi yang berbagi jalur yang sama. Sistem LRT Jabodebek Indonesia (Lin Cibubur dan Lin Bekasi) dan LRT Rapid Kuala Lumpur (Lin Ampang dan Lin Sri Petaling) merupakan salah satu kasus di mana beberapa stasiun pertama di lintasan dilayani oleh kedua layanan sekaligus secara bergantian. PIDS menjadi semakin penting untuk memastikan penumpang tidak salah menaiki kereta, seperti tidak sengaja menaiki kereta tujuan Bekasi kita ingin ke Cibubur.

Kebutuhan untuk menampilkan informasi dengan cepat dan akurat bagi penumpang ini berarti sistem PIDS perlu disokong dengan sistem komunikasi yang sederhana dan cepat. Maka dari itu, biasanya solusi sistem komunikasi yang digunakan dalam PIDS adalah perpaduan solusi populer (HTTP atau sejenis untuk berkomunikasi ke pusat kendali) dan solusi yang *custom* (untuk berkomunikasi dalam internal stasiun). Pada soal ini, kita akan mencoba memanfaatkan pemrograman *socket* untuk merancang sebuah sistem PIDS sederhana.

Pada soal ini, kalian diminta untuk mengimplementasikan sebuah protokol aplikasi **yang beroperasi di atas socket QUIC** yang dapat digunakan untuk mengoperasikan layar PIDS. Dalam protokol ini, ada dua pihak yang terlibat, yaitu:

1. *Node Kendali Stasiun* yang berperan sebagai *publisher* (klien). *Node* ini memiliki tugas untuk mengirimkan data kedatangan / keberangkatan ke layar-layar PIDS di stasiun tersebut.
2. *Node Layar PIDS* yang berperan sebagai *subscriber* (server). *Node* ini perlu mendengarkan pesan-pesan yang masuk dari *node* kendali stasiun dan membuat keputusan untuk menampilkan informasi tertentu sesuai dengan data yang diperoleh.

Dalam suatu protokol aplikasi, terdapat dua hal yang perlu diperhatikan, yaitu **format pesan** dan **perilaku yang diharapkan**.

Format Pesan

Sebuah paket yang akan ditransaksikan melalui protokol ini dinamakan **LRTPIDSPacket**. **Baik pesan yang dikirimkan (dari node kendali stasiun ke node layar) maupun balasannya (sebaliknya)** mengikuti format paket ini. Paket ini terdiri dari beberapa informasi yang diperlukan untuk mengambil tindakan pada *node* layar PIDS, yaitu untuk menampilkan pengumuman atau informasi tertentu. Informasi tersebut antara lain:

1. **Transaction ID** dengan ukuran 16 bit. Informasi ini digunakan untuk menjamin keunikan dari setiap transaksi pesan yang dilakukan antara *node* kendali stasiun dan *node* layar PIDS. Transaction ID disamakan antara paket yang dikirimkan oleh *node* kendali stasiun ke *node* layar PIDS maupun paket balasannya. Misalnya, jika *node* kendali stasiun mengirim pesan dengan ID 1, maka balasannya juga harus menggunakan ID 1.
2. **IsAck** dengan ukuran 1 bit. Bit ini di-set jika pesan merupakan balasan yang dikirim dari *node* layar PIDS ke *node* kendali stasiun. Jika pesan bukan merupakan balasan, bit ini tidak di-set.
3. **IsNewTrain** dengan ukuran 1 bit. Bit ini di-set jika pesan membawa informasi kereta baru untuk ditambahkan pada daftar jadwal kereta yang ditampilkan di PIDS.
4. **IsUpdateTrain** dengan ukuran 1 bit. Bit ini di-set jika pesan membawa informasi yang mengubah data kereta yang sudah ditampilkan di PIDS, seperti jika ada keterlambatan atau perubahan jurusan.
5. **IsDeleteTrain** dengan ukuran 1 bit. Bit ini di-set jika pesan membawa informasi untuk menghapus data kereta yang sudah ditampilkan di PIDS, seperti jika ada pembatalan perjalanan.
6. **IsTrainArriving** dengan ukuran 1 bit. Bit ini di-set jika pesan membawa informasi terkait kereta yang akan segera memasuki stasiun. Pada implementasi *real*-nya, pesan ini dipicu oleh sistem GPS atau sirkuit di rel yang “diinjak” oleh kereta di mulut stasiun yang menandakan kereta akan segera masuk.
7. **IsTrainDeparting** dengan ukuran 1 bit. Bit ini di-set jika pesan membawa informasi terkait kereta yang akan segera diberangkatkan. Pada implementasi *real*-nya, pesan ini dipicu oleh sistem di dalam kereta atau pusat kendali yang memutuskan bahwa kereta sudah masuk jadwal keberangkatan dan sudah siap diberangkatkan.

8. **TrainNumber** dengan ukuran 16 bit. Informasi ini berisi nomor perjalanan kereta yang unik untuk setiap perjalanan. Informasi ini nantinya dapat digunakan sebagai referensi ketika mengubah atau menghapus jadwal serta sebagai bahan referensi bagi pengguna layanan transportasi umum.
9. **DestinationLength** dengan ukuran 8 bit. Informasi ini memuat panjang *string* jurusan perjalanan kereta. Hal ini diperlukan karena jumlah huruf jurusan perjalanan tersebut bisa berbeda-beda, seperti Harjamukti (10 karakter) vs Jatimulya (9 karakter).
10. **Destination** dengan ukuran sesuai **DestinationLength**. Informasi ini memuat jurusan (tujuan akhir) perjalanan kereta yang dirujuk oleh paket ini.

Perilaku yang Diharapkan

Alur eksekusi program ini adalah sebagai berikut:

1. *Node* layar PIDS mendengarkan koneksi secara terus-menerus dari *node* kendali stasiun sampai diberhentikan.
2. Untuk mensimulasikan aksi pemicu yang dilakukan oleh sistem di dalam kereta saat kereta tiba atau berangkat, buatlah 2 buah paket **secara hard-coded** di aplikasi *node* kendali stasiun sebagai berikut:
 - a. Kereta dengan nomor perjalanan 42 tujuan Harjamukti tiba
 - b. Kereta dengan nomor perjalanan 42 tujuan Harjamukti berangkat
3. Paket A di-*encode* sebagai *bytes array* dan dikirimkan ke *node* layar PIDS.
4. Saat sampai di *node* layar PIDS, Paket A di-*decode* sehingga dapat dikembalikan menjadi objek aslinya.
5. *Node* layar PIDS menangani objek yang diterima dengan perilaku berikut ini:
 - a. Jika **IsTrainArriving** di-set, maka *handler* perlu mengembalikan pesan “Mohon perhatian, kereta tujuan [Tujuan pada Paket] akan tiba di Peron 1.”

Contoh: Mohon perhatian, kereta tujuan Harjamukti akan tiba di Peron 1.

- b. Jika **IsTrainDeparting** di-set, maka *handler* perlu mengembalikan pesan “Mohon perhatian, kereta tujuan [Tujuan pada Paket] akan diberangkatkan dari Peron 1.”
 - c. Kasus lainnya tidak perlu ditangani pada soal ini (dibiarkan saja).
6. Pesan yang dikembalikan oleh *handler* perlu dicetak di *standard output node* layar PIDS.
7. Setelah pesan dicetak, *node* layar PIDS perlu mengirimkan ACK yang berbentuk objek yang sama dengan yang diterima dari *node* kendali stasiun namun dengan bit **isACK** di-set.
8. ACK Paket A di-*encode* dan dikirimkan ke *node* kendali stasiun.
9. Saat sampai di *node* kendali stasiun, ACK Paket A di-*decode* sehingga dapat dikembalikan menjadi objek aslinya. Tidak ada yang perlu dilakukan untuk paket ini setelah di-*decode*.

10. Ulangi Langkah 3-9 untuk Paket B.
11. Tutup aplikasi di *node* kendali stasiun.

Spesifikasi Teknis

1. Aplikasi **wajib mengikuti template Publisher dan Subscriber yang sudah disediakan pada h01-source-code** karena sebagian hasil program ini akan dinilai secara otomatis oleh *unit tester*.
2. *Socket* yang dikembangkan **harus menggunakan QUIC**. Akan ada pengurangan poin jika tidak menggunakan QUIC.
3. **Aspek tipe data (struct) yang boleh diubah hanyalah tipe data komponennya (variabelnya saja)**. Dilarang menambahkan atau mengurangi variabel yang sudah ada. Kalian diperkenankan untuk membuat objek *Data Transfer Object* (DTO) jika diperlukan untuk mencegah modifikasi yang tidak perlu di tipe data yang sudah disepakati.
4. **Dilarang mengubah signature dari metode yang sudah disediakan** (nama, tipe masukan, tipe luaran).
5. Fungsi untuk *encode* dan *decode* harus ditempatkan di **utils.go**. Setiap fungsi **wajib hanya melakukan fungsi sesuai namanya (encode atau decode saja)**.
6. Fungsi untuk menangani objek pesan harus ditempatkan di fungsi **Handler** pada **subscriber.go**. Fungsi ini hanya ditugaskan untuk menerima masukan berupa objek yang sudah di-*decode* dan mengembalikan pesan yang harus ditampilkan sesuai konten informasi yang diterima.
7. Selain modul *quic-go* dan modul lain pada *h01-source code*, mohon untuk tidak menggunakan modul eksternal (yang mengharuskan adanya pengunduhan melalui go get).
8. Nama protokol aplikasi yang digunakan untuk keperluan ALPN adalah **Irt-jabodebek-[npm peserta]**. Contoh: Irt-jabodebek-2006142424
9. Konfigurasi IP dan *port* adalah sebagai berikut:
 - a. **Port**: Sesuai dengan nomor *port* unik masing-masing
 - b. **Node Kendali Stasiun**: VM1 milik masing-masing
 - c. **Node Layar PIDS**: VM2 milik masing-masing
 - d. **Gunakan Alamat IP Pribadi masing-masing VM untuk berkomunikasi**

Eksekusi Program

Node Kendali Stasiun

```
Go run jarkom.cs.ui.ac.id/h01/project/publisher
```

Node Layar PDIS

```
Go run jarkom.cs.ui.ac.id/h01/project/subscriber
```

Rubrik Penilaian

[20 Poin] Pengujian Mandiri

Sila laporkan hasil pengujian aplikasi yang kalian sudah rancang secara mandiri dengan mengikuti step berikut:

1. Pindahkan kode sumber ke VM1 dan VM2 milik kalian.
2. Siapkan satu jendela untuk SSH ke VM1 dan satu jendela untuk SSH ke VM2.
3. Jalankan aplikasi *node* layar PIDS di VM2.
4. Jalankan aplikasi *node* kendali stasiun di VM1.
5. Pastikan eksekusi sudah sesuai dengan alur yang diharapkan.
6. Buatlah tangkapan layar **yang menunjukkan kedua jendela SSH yang berjalan dalam satu gambar dan memuat informasi berikut ini:**
 - a. *Shell prompt* masing-masing VM
 - b. Baris komando untuk menjalankan masing-masing program
 - c. Luaran masing-masing program

[30 Poin] Pengujian dengan Autograder

Tim asisten dosen akan menilai pekerjaan kalian dengan menggunakan *auto grader* berbasis *unit testing*. Kalian hanya perlu memastikan kode yang dikumpulkan adalah benar-benar yang kalian kerjakan. Aspek yang dinilai adalah sebagai berikut:

1. Ketepatan pemilihan *socket*
2. Ketepatan atribut IP, Port, dan Protocol Name (ALPN)
3. Ketepatan fungsi Encoder
4. Ketepatan fungsi Decoder
5. Ketepatan fungsi Handler



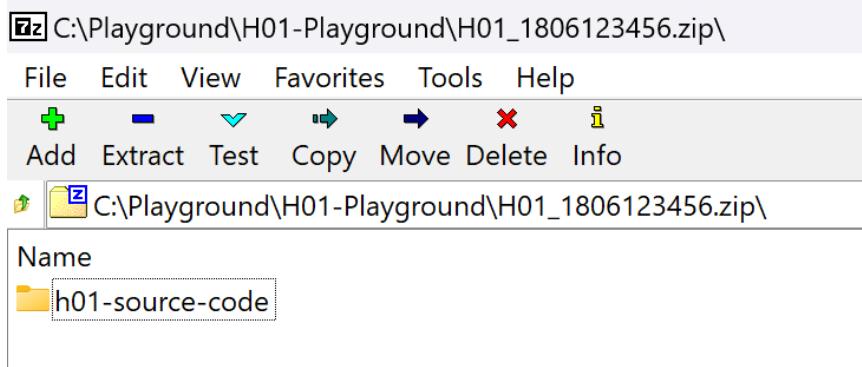
Informasi Pengumpulan Berkas

Berkas yang perlu kalian kumpulkan adalah sebagai berikut:

1. **Laporan** dengan *template* yang sudah disediakan dalam bentuk berkas PDF. Format penamaan laporan adalah **H01_[NPM].pdf** (Contoh: H01_2206123456.pdf).

Catatan: Mohon pastikan kembali bahwa tipe berkas yang diunggah adalah PDF. Kami kerap mengalami kesulitan ketika tipe berkas yang diunggah adalah DOCX atau bahkan PAGES yang memerlukan aplikasi khusus untuk membukanya.

2. **Direktori Kerja Kode H01 yang memuat semua kode hasil modifikasi terakhir dari semua bagian** secara utuh dalam bentuk arsip ZIP. Pastikan bahwa **konten terluar dari arsip tersebut adalah folder h01-source-code**. Format penamaan arsip adalah **H01_[NPM].zip** (Contoh: H01_2206123456.zip). Contoh konten arsipnya adalah sebagai berikut:



Peraturan

Keterlambatan

Anda diharapkan dapat mengumpulkan hasil pekerjaan yang dilakukan sebelum batas waktu pengumpulan. Jika terdapat kondisi di mana Anda terpaksa terlambat mengumpulkan hasil pekerjaan, terdapat jangka waktu tambahan di mana Anda masih diperbolehkan mengumpulkan hasil pekerjaan dengan konsekuensi tertentu. Jika X adalah durasi setelah batas waktu pengumpulan yang ditetapkan sampai waktu Anda mengumpulkan hasil pekerjaan, Anda akan menerima penalti nilai pekerjaan sebagaimana diatur pada peraturan berikut ini:

- $X < 10$ menit : Tidak ada penalti
- $10 \text{ menit} \leq X < 2 \text{ jam}$: 25% penalti
- $2 \text{ jam} \leq X < 4 \text{ jam}$: 50% penalti
- $4 \text{ jam} \leq X < 6 \text{ jam}$: 75% penalti
- $X \geq 6 \text{ jam}$: Cut-off (Pekerjaan anda tidak akan diterima)

Plagiarisme

Anda diperbolehkan berdiskusi tentang pekerjaan Anda dengan peserta kuliah lain atau pihak lainnya, namun Anda harus memastikan bahwa semua pekerjaan yang dikumpulkan adalah murni hasil pekerjaan Anda sendiri. Anda dilarang keras melakukan tindak plagiarisme atau kecurangan akademik lainnya. Menurut kamus daring Merriam-Webster, plagiarisme berarti:

- Mencuri dan mengklaim (ide atau kata orang lain) sebagai milik sendiri
- Menggunakan hasil (karya/pekerjaan orang lain) sebagai milik sendiri
- Melakukan pencurian literatur/sastra
- Merepresentasikan ulang sebuah ide/produk yang sudah ada sebagai sesuatu yang bersifat baru dan orisinal.

Tim pengajar memiliki hak untuk meminta klarifikasi terkait dugaan ketidakjujuran akademik, terutama plagiarisme, dan memberikan konsekuensi berupa pengurangan nilai hasil pekerjaan atau pencabutan nilai (nilai diubah menjadi nol) untuk hasil pekerjaan yang terkonfirmasi dikerjakan secara tidak jujur.