Nama : Frankie Huang
NIM : 13521092

Seleksi Bagian A
Laboratorium Sistem Terdistribusi

I. Organisasi dan Arsitektur Komputer

1.a. C code

```c
int add_int32(int a, int b)
/**
 * Computes the addition of two 32-bit integer
 *
 * @param a      First integer
 * @param b      Second integer
 * @return int        Addition result
 */
{
    int sum = a ^ b;
    int carry = (a & b) << 1;

    if (carry == 0)
        return sum;

    return add_uint32(sum, carry);
}
```

1.b. “Pseudocode”

```c
// Buset kodenya panjang bener, intinya kurang lebih begini
unsigned add_float(float a, float b)
{
    // kasus NaN
    if (!(exponent(a) ^ 0xFF) && mantissa_a) return a;
    if (!(exponent(b) ^ 0xFF) && mantissa_b) return b;

    // Kasus Inf
    if (!(exponent(a) ^ 0xFF) && !(exponent(b) ^ 0xFF)) {
        if (sign(a) != sign(b)
            return NaN; // Inf-Inf = NaN
        else
            return a; // Inf atau -Inf
    }
    else if (!(exponent(a) ^ 0xFF)) return a;
    else if (!(exponent(b) ^ 0xFF)) return b;
```

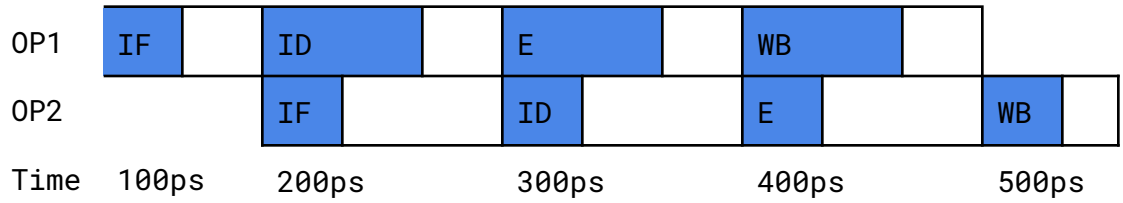
```
    // Kasus biasa, ini yang rada panjang
    int exponent_diff;

    // Cek float yang lebih besar
    bool abs_a_bigger;
    if (exponent(a) > exponent(b)) abs_a_bigger = true;
    else if (exponent(a) < exponent(b)) abs_a_bigger =
false;
    else if (mantissa(a) > mantissa(b)) abs_a_biger = true;
    else abs_a_bigger = false;

    // Lakukan operasi
    if (sign(a) == sign(b)) {
        // Penjumlahan
        sign(result) = sign(a)
    }
    else {
        // Pengurangan
        sign(result) = (abs_a_bigger ? sign(a) : sign(b));
    }

    // Normalize, lalu ubah ke float

    return (sign(result) << 31 | exponent(result) << 23 |
mantissa);
}
```

2. **Instruction pipeline** adalah teknik yang digunakan dalam CPU modern untuk mempercepat kinerja pengeksekusian *instruction* dengan membagi instruksi menjadi bagian-bagian yang lebih kecil menjadi:
   a. *Instruction fetch*, yaitu membaca instruksi dari PC;
   b. *Instruction decode*, yaitu menerjemahkan bytes menjadi instruksi;
   c. *Execute*, yaitu menjalankan instruksi; dan
   d. *Write-back*, yaitu menyimpan hasil perubahan ke dalam memori/register

   Bagian-bagian tersebut, umumnya disebut sebagai **instruction cycle** adalah teknik pembagian suatu instruksi menjadi bagian kecil yang berjalan secara sekuensial, sehingga memungkinkan instruksi dapat berjalan secara konkuren. Misalnya pada instruksi berikut dimana setiap bagian berjalan dalam rentang waktu 100ps,

```
movw $41, %ax   ; OP1
movw $42, %bx   ; OP2
```

Maka akan didapatkan *pipeline* diagram sebagai berikut



Waktu yang dibutuhkan untuk menjalankan sebuah bagian ditandai dengan warna biru. Dapat dilihat bahwa sebuah bagian akan menggunakan seluruh waktu yang dialokasikan (walaupun instruksi sudah selesai dijalankan) untuk mencegah terjadinya "tabrakan" antar instruksi. Namun, mungkin saja terdapat ketergantungan antar instruksi, seperti contoh berikut

```
movw $41, %ax
movw %ax, %bx
```

Jika kondisi seperti di atas muncul,terdapat dua cara untuk mengatasi ketergantungan tersebut, yaitu:

1. *Stalling*, yaitu dengan mengulang suatu operasi hingga ketergantungan telah hilang (misalnya dengan melakukan operasi *Instruction Decode* berulang kali).
2. *Forwarding*, yaitu dengan memasukkan instruksi NOP (x90) hingga ketergantungan telah hilang.

3.a. *Cache miss* adalah keadaan dimana suatu data yang ingin dibaca dari tidak berada pada cache, sehingga processor harus mengambil data yang diperlukan dari memori yang memerlukan waktu yang lebih lama.

3.b. Beberapa cara untuk mengurangi *cache miss*:

a. Membuat program yang memanfaatkan prinsip locality dengan baik.

b. Menggunakan *processor* dengan ukuran cache yang lebih besar.

c. Menggunakan algoritma dan struktur data yang optimal untuk memperbesar kemungkinan cache hit.
d. Menggunakan algoritma penggantian cache yang optimal, seperti LRU.

3.c. Prinsip *locality* adalah kecenderungan suatu program untuk mengakses elemen yang sama berulang kali (*temporal locality*) atau kecenderungan untuk mengakses elemen yang berdekatan dengan elemen yang diakses sebelumnya (*spatial locality*).

3.d. Misalnya pada tubes alstrukdat, yaitu membuat sebuah game yang memiliki fitur peta. Peta yang telah di-*load* pada awal tahap setup disimpan dalam suatu variabel agar dapat digunakan tiap loop. Dalam contoh ini, dimanfaatkan prinsip *temporal locality*, yaitu dimana variabel peta yang sering diakses disimpan pada cache pada awal permainan, sehingga pada iterasi berikutnya akan selalu terjadi cache hit.

4. General purpose register pada processor:

a. 16-bit

Register	Fungsi
ax	- Melakukan operasi aritmatik - Menyimpan return value
bx	Menyimpan *base address* dari suatu data di memori
cx	Mengontrol *loop* dengan mengecek boolean atau menghitung iterasi
dx	Melakukan manipulasi data dan penyimpanan data
si	Berfungsi sebagai *source pointer* untuk operasi string dan penyalinan data pada memori
di	Berfungsi sebagai *destination pointer* untuk operasi string dan penyalinan data pada memori

bp	Menyimpan alamat *base*/dasar dari stack
sp	Menyimpan alamat *top/head* dari stack dan mengatur *stack frame*

b. 32-bit

Register	Fungsi
eax	- Melakukan operasi aritmatik - Menyimpan return value
ebx	Menyimpan *base address* dari suatu data di memori
ecx	Mengontrol *loop* dengan mengecek boolean atau menghitung iterasi
edx	Melakukan manipulasi data dan penyimpanan data
esi	Berfungsi sebagai *source pointer* untuk operasi string dan penyalinan data pada memori
edi	Berfungsi sebagai *destination pointer* untuk operasi string dan penyalinan data pada memori
ebp	Menyimpan alamat *base*/dasar dari stack
esp	Menyimpan alamat *top/head* dari stack dan mengatur *stack frame*

c. 64-bit

Register	Fungsi
rax	- Melakukan operasi aritmatik - Menyimpan return value
rbx	Menyimpan *base address* dari suatu data di memori
rcx	- Mengontrol *loop* dengan mengecek boolean atau menghitung iterasi - Menyimpan nilai parameter keempat pada fungsi
rdx	- Melakukan manipulasi data dan penyimpanan data - Menyimpan nilai parameter ketiga pada fungsi

<table>
<tr><td>rsi</td><td>- Berfungsi sebagai source pointer untuk operasi string dan penyalinan data pada memori
- Menyimpan nilai parameter kedua pada fungsi</td></tr>
<tr><td>rdi</td><td>- Berfungsi sebagai destination pointer untuk operasi string dan penyalinan data pada memori
- Menyimpan nilai parameter pertama pada fungsi</td></tr>
<tr><td>rbp</td><td>Menyimpan alamat base/dasar dari stack</td></tr>
<tr><td>rsp</td><td>Menyimpan alamat top/head dari stack dan mengatur stack frame</td></tr>
<tr><td>r8</td><td>- Berfungsi sebagai register tambahan
- Menyimpan nilai parameter kelima pada fungsi</td></tr>
<tr><td>r9</td><td>- Berfungsi sebagai register tambahan
- Menyimpan nilai parameter keenam pada fungsi</td></tr>
<tr><td>r10</td><td rowspan="6">Register tambahan yang digunakan untuk membantu register lainnya untuk pengaksesan data</td></tr>
<tr><td>r11</td></tr>
<tr><td>r12</td></tr>
<tr><td>r13</td></tr>
<tr><td>r14</td></tr>
<tr><td>r15</td></tr>
</table>

5.a. C code

```c
int arraySum(int arr[], int length)
/**
 * Computes the sum of all array elements
 *
 * @param arr[]   Array of integers
 * @param length  Array length
 */
{
    int i = 0;
    int sum = 0;

add_element:
    sum += arr[i];
    i++;
    if (i < length)
        goto add_element;
    else
        goto finish;

finish:
    return sum;
}
```

5.b. C code

```c
int maxElement(int arr[], int length)
/**
 * Returns the max element of an array
 *
 * @param arr[]   Array of floats
 * @param length  Array length
 */
{
    int i = 1;
    int max = arr[0];

check:
    if (arr[i] > max)
        max = arr[i];
    i++;

    if (i < length)
        goto check;
    else
        goto finish;

finish:
    return max;
}

int minElement(int arr[], int length)
/**
 * Returns the min element of an array
 *
 * @param arr[]   Array of floats
 * @param length  Array length
 */
{
    int i = 1;
    int min = arr[0];

check:
    if (arr[i] < min)
        min = arr[i];
    i++;

    if (i < length)
        goto check;
    else
        goto finish;

finish:
    return min;
}
```

6. Terdapat dua jenis RAM yang umum digunakan, yaitu SRAM (Static RAM) dan DRAM (Dynamic RAM).

	SRAM	DRAM
Transistor	Menggunakan 6 transistor untuk menyimpan 1 bit	Menggunakan 1 transistor untuk menyimpan 1 bit
Kecepatan Akses	Pengaksesan jauh lebih cepat dibandingkan DRAM	10 kali lebih lambat dibandingkan SRAM
Tingkat Persistensi	Dapat menyimpan data yang sama selama daya terus mengalir	Perlu dilakukan *refresh* pada jangka tertentu untuk mencegah *data loss*
Kestabilan	Stabil terhadap gangguan eksternal	Sensitif terhadap gangguan eksternal
Harga	1000x lebih mahal dibanding DRAM	Jauh lebih murah dibandingkan SRAM
Konsumsi Daya	Menggunakan lebih banyak daya untuk menyimpan jumlah bit yang sama dibanding DRAM	Menggunakan daya yang lebih sedikit dibanding SRAM

7. Metode penyimpanan byte pada komputer, baik little endian maupun big endian tidak memiliki kelebihan signifikan antara satu sama lain. Alasan pemilihan *endianness* hanyalah bergantung pada aspek historis antara produser Intel (*little endian*) dan IBM & Oracle (*big endian*). Namun, terdapat beberapa argumen pendukung alasan *little endian* lebih baik digunakan, antara lain pada operasi penjumlahan dan pengurangan, operasi *fetch* akan dilakukan pada LSB, yaitu byte yang disimpan pertama pada sistem *little endian*, sehingga dapat dilakukan penjumlahan pada LSB sebelum byte lainnya telah di *fetch*.

8. Arsitektur komputer:

a. x86_64

Arsitektur x86_64 adalah arsitektur processor yang paling sering digunakan karena memiliki banyak fitur, sehingga membuat arsitektur ini menjadi sebuah arsitektur yang bersifat general.

Kelebihan:

- Mendukung banyak perangkat lunak dan sistem operasi
- Memiliki tingkat performa *single-thread* yang lebih baik dibanding arsitektur lainnya

Kekurangan:

- Menggunakan daya yang lebih tinggi dibandingkan arsitektur lainnya
- Lebih kompleks dan kurang efisien dalam manajemen daya pada arsitektur *mobile*

Desain ini membuat arsitektur x86_64 lebih cocok digunakan untuk komputer, laptop, serta server.

b. ARM

Arsitektur ARM merupakan salah satu jenis arsitektur RISC (Reduced Instruction Set Computer) yang dirancang untuk perangkat yang memprioritaskan efisiensi daya.

Kelebihan:

- Lebih hemat daya dibandingkan arsitektur lainnya
- Menyediakan jenis processor yang beragam yang menawarkan pilihan *low-performance* maupun *high-performance*

Kekurangan:

- Memiliki tingkat performa *single-thread* yang lebih buruk dibandingkan arsitektur lainnya
- Sulit untuk mendapatkan *software* yang kompatibel dengan arsitektur ARM
- Jarang digunakan untuk *high-performance computing*

Desain ini membuat arsitektur ARM lebih cocok digunakan pada perangkat *mobile*, IOT, serta perangkat lainnya yang tidak membutuhkan kinerja daya tinggi

c. MIPS

Arsitektur MIPS juga merupakan salah satu jenis arsitektur RISC yang juga dirancang untuk perangkat yang memprioritaskan efisiensi daya.

Kelebihan:

- Memiliki ISA yang lebih simpel dibandingkan arsitektur x86_64
- Dapat dikonfigurasi sehingga menggunakan daya rendah

Kekurangan:

- Sulit untuk mendapatkan *software* yang kompatibel dengan arsitektur MIPS
- Memiliki jenis processor dan *community support* yang jauh lebih sedikit dibandingkan arsitektur x86_64 dan ARM
- Jarang digunakan untuk *high-performance computing*

Penggunaan desain ini sudah mulai tergantikan oleh desain arsitektur ARM. Walaupun begitu, secara historis arsitektur ini sering digunakan pada perangkat *embedded*, *networking*, *signal processing*, dan *real-time*.

d. SPARC

Arsitektur SPARC juga merupakan salah satu jenis arsitektur SPARC yang dirancang untuk *high-performance computing*

Kelebihan:

- Dirancang untuk memanfaatkan *multi-threading* dan *paralel processing*
- Memiliki reliabilitas dan *error detection* yang baik
- Memiliki *support* terhadap fitur *dynamic threading* dan *high-bandwidth memory access*

Kekurangan:

- Sulit ditemui dan memiliki komunitas yang lebih kecil dibandingkan arsitektur x86_64 dan ARM
- Menggunakan konsumsi daya yang lebih tinggi dibandingkan arsitektur jenis lainnya
- Memiliki dukungan *software* yang lebih sedikit dibandingkan arsitektur x86_64 dan ARM

Desain ini membuat arsitektur SPARC sulit diimplementasikan pada *general-purpose products* dan lebih sering digunakan untuk perangkat yang memerlukan *high-performance computing*, seperti server.

II. Sistem Operasi

1. Terdapat beberapa metode yang digunakan ketika terjadi *write hit*, yaitu ketika alamat memori data yang ingin ditulis berada

pada *cache*. Namun, jika kita hanya mengubah data pada *cache*, akan terjadi *Data Inconsistency*, yaitu ketika data yang berada pada *cache* dan memori berbeda. Untuk mencegahnya, terdapat beberapa metode yang digunakan

a. *Write-through*, yaitu metode mengubah nilai pada *disk* dan *cache* secara bersamaan. Metode ini sebaiknya digunakan jika *data consistency* dan *data accuracy* merupakan prioritas utama dari sistem dan *performance* dapat diabaikan.
b. *Write-back*, yaitu metode mengubah nilai pada *disk* ketika data pada *cache* sudah tidak digunakan (*flushed*). Metode ini sebaiknya digunakan jika *performance* merupakan prioritas utama dari sistem dan *system crash* sangat jarang terjadi.
c. *Read ahead*, dimana *disk controller* akan membaca sektor selanjutnya pada *disk* yang mungkin diakses. Metode ini sebaiknya digunakan ketika terdapat data sekuensial yang ingin dibaca.
d. *No read ahead*, dimana *disk controller* tidak akan membaca sektor pada *disk*. Metode ini sebaiknya digunakan ketika data yang ingin dibaca tersebar secara *random*.
e. *Adaptive read ahead*, dimana *disk controller* akan memutuskan apakah sektor selanjutnya sebaiknya dibaca atau tidak. Metode ini sebaiknya digunakan ketika kasus data sekuensial dan data *random* sering terjadi.

2. Perbedaan

	Windows 10	Arch Linux
System Update	Merupakan OS dengan sistem **point release**	Merupakan OS dengan sistem **rolling release**
Installation Steps	Tahapan instalasi cukup **straightforward** dimana hampir seluruh pengaturan diatur otomatis	Tahapan instalasi cukup **rumit** dimana hampir seluruh tahapan harus diatur secara manual
OS Family	Berbasis pada keluarga OS **Windows**	Berbasis pada keluarga OS **Linux**
Target	Didesain untuk pengguna	Didesain untuk pengguna

Produk	**general** dan **enterprise**	yang menginginkan **full customization**
Package Manager	Menggunakan **Windows Store** sebagai *default package manager*	Menggunakan **pacman** sebagai *default package manager*
System Requirements	Memerlukan *system requirement* yang **high-resource** dibandingkan Arch	Dapat dikonfigurasi untuk berjalan pada sistem dengan **low-resource**
User Interface	Menggunakan **GUI** secara default	Menggunakan **TUI** secara default
Driver Support	Sebagian besar *device driver* dirancang agar berjalan dengan Windows dan akan terinstal secara **otomatis**	*Device driver* yang ingin digunakan mungkin harus diinstal secara **manual** terlebih dahulu sebelum dapat digunakan
User Permission	Mayoritas *user* adalah **administrator**	Mayoritas *user* adalah **non-superuser**
Terminal	Secara *default* menggunakan **powershell** atau **cmd**	Secara *default* menggunakan **bash**

Persamaan

- User Account, kedua OS dapat dikonfigurasi untuk digunakan oleh banyak *user*
- File System, kedua OS memberikan dukungan file system FAT32, NTFS, dan ext4
- File Permission, kedua OS dapat diatur untuk memberikan file permission yang berbeda untuk berbagai pengguna
- Daily Driver, kedua OS dapat dirancang untuk digunakan sebagai *daily driver OS* (penggunaan sehari-hari)
- Architecture Support, kedua OS mendukung *processor* dengan arsitektur 32-bit dan 64-bit
- Device, kedua OS dirancang untuk digunakan pada perangkat *desktop*

3. Bayangkan komputer itu adalah sebuah kelas, dimana di dalam sebuah kelas tersebut terdapat banyak murid. Namun, jika di kelas tersebut hanya terdapat murid, maka kegiatan pembelajaran tidak dapat berjalan dengan baik. Misalkan di dalam kelas terdapat 30 siswa dan 15 buku cerita, maka murid-murid akan saling bertengkar untuk bisa membaca buku itu. Untuk itu, diperlukan seseorang yang bertindak sebagai pemimpin dari kelas tersebut, yaitu seorang guru. Disini, guru dapat membagi barang-barang secara adil kepada siswa agar tidak terjadi perkelahian. Nah, disini kita bisa membayangkan seorang siswa sebagai perangkat lunak/*software* dan guru sebagai *resource manager*/*operating system*.

4.a. **RAM**

Ketika sebuah program dijalankan, kode akan dipindahkan menuju RAM agar dapat digunakan. Sistem operasi akan secara otomatis mengalokasikan ukuran memori yang dianggap sesuai untuk program tersebut.

4.b. **SSD/HDD**

Ketika sebuah program dijalankan, kode program akan dipindahkan dari SSD/HDD yang terkait menuju RAM. Kemudian, ketika terjadi operasi *read* atau *write* yang dilakukan program, maka data akan dibaca dari/ditulis di SSD/HDD yang terkait. Sistem operasi akan memanggil *syscall read()* atau *write()* untuk melakukan operasi tersebut.

4.c. **Processor**

Sesudah kode dipindahkan menuju RAM, processor akan melakukan operasi *decode* pada tiap instruksi. Selama waktu hidup program, sistem operasi akan berfungsi sebagai *resource manager* program tersebut dengan mengganti proses menggunakan *context switch* jika program sedang *idle*.

4.d. **Swap memory**

Jika seluruh RAM sudah digunakan ketika program ingin dijalankan/sedang berjalan, maka sistem operasi akan otomatis memindahkan data yang tidak digunakan menuju swap memory. Jika data tersebut ingin digunakan lagi, sistem operasi akan mengembalikan data tersebut menuju RAM dan memindahkan data lain jika RAM sudah penuh.

5. Secara literal, maksudnya adalah semua operasi *input-output* baik antar *hardware* maupun antar *process* disimpan dalam sebuah file yang terdapat di dalam direktori /dev/. Misalkan ketika kita mencolok sebuah *flashdisk*, maka OS Linux akan otomatis mendeteksi adanya sebuah *hardware* baru yang muncul pada direktori /dev/, seperti /dev/sda1. Kemudian, kita dapat melakukan operasi *mount* (seperti operasi *read* pada file biasa) terhadap file tersebut. File tersebut dapat kita tulis dan baca, seperti file pada umumnya, dan ketika sudah tidak digunakan, kita bisa melakukan operasi *unmount* (seperti operasi *close* pada file biasa). Contoh lainnya adalah ketika kita melakukan *pipe*, *process* yang melakukan *output stream* akan menulis hasil *output* ke dalam sebuah file dan akan dibaca oleh *process* lain dengan *input stream*.

6. **NTFS (New Technology File System)** adalah *successor* dari exFAT) yang dikembangkan oleh Microsoft yang memiliki fitur-fitur *protection*, *multi-user access control, logging*, dll. yang dirancang untuk sistem operasi yang memiliki fitur keamanan.
   FAT32 (File Allocation Table 32) merupakan *successor* dari FAT16 yang dirancang untuk mengatasi keterbatasan FAT16 dengan mengubah ukuran *chunck* menjadi 32 bit. Berbeda dengan NTFS, FAT32 sendiri menyimpan data langsung pada storage tanpa enkripsi.

   Perbedaan

	NTFS	FAT32
Struktur	Kompleks	Simpel
Ukuran file maximum	16TB	4GB
Enkripsi	Terenkripsi dengan Encrypting File System (EFS)	Tidak ada
Fault Tolerance	Otomatis	Tidak ada

Compression	Mendukung	Tidak mendukung
Kecepatan akses	Relatif tinggi dibanding filesystem lainnya	Rendah
Level akses beragam	Mendukung	Tidak ada
Konversi antar filesystem	Tidak mendukung	Mendukung

III. Jaringan Komputer

1. Enkripsi **data at rest** bertujuan untuk **mengamankan data statik** yang tersimpan pada sebuah *storage* sedangkan enkripsi **data in transit** bertujuan untuk mencegah pihak lain membaca data ketika sedang berpindah.

Perbandingan enkripsi Data at Rest

	Full Disk Encryption	File Based Encryption
Scope	Seluruh data pada *storage*	Sebagian *file* yang dipilih
Fungsi	Mencegah pencurian data jika *storage* hilang, dicuri, atau diakses oleh pihak lain	Menjaga *file/folder* yang dipilih
Manajemen Kunci	Umumnya menggunakan satu kunci untuk mengenkripsi seluruh data	Dapat menggunakan banyak kunci untuk berbagai *file/folder*

Perbandingan enkripsi Data in Transit

	Secure Socket Layer (SSL)	Secure Shell (SSH)
Autentikasi	Menggunakan sistem	Tidak ada

	username/password	
Port Number	443	22
Fungsi	Mengenkripsi komunikasi antara browser dan server	Mengenkripsi komunikasi antara dua perangkat antar internet
Sistem Enkripsi	Menggunakan gabungan antara algoritma enkripsi *symmetric* dan *asymmetric encryption*	Menggunakan algoritma enkripsi *symmetric key*
Jenis	*Security protocol*	*Cryptographic network protocol*

2. Algoritma tersebut berhubungan dengan bagaimana cara TCP mengatur jumlah paket *in transit* untuk mencegah terjadinya *congestion collapse*, yaitu keadaan dimana sebuah *host* mengirim paket sebanyak-banyaknya yang menyebabkan terjadinya *congestion* (keadaan dimana sebuah paket di-*drop*) dan akan mengirimkan kembali paket tersebut, yang mengakibatkan lebih banyak *congestion*. Kondisi *congestion* dapat diketahui dengan melihat apakah terdapat sebuah paket yang di-*drop* atau terjadinya *timeout*. Terdapat variabel yang penting untuk diketahui sebelum memahami algoritma ini, yaitu

```
MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
EffectiveWindow = MaxWindow - (LastSentByte - LastByteACKed)
```

Penjelasan variabel-variabel di atas adalah sebagai berikut:
- CongestionWindow : Batas maksimum jumlah data yang berada dalam *transit* dalam suatu waktu
- AdvertisedWindow : Jumlah data yang dapat diterima oleh *receiver* dalam suatu waktu
- LastSentByte : Byte terakhir yang dikirim oleh *sender*
- LastByteACKed : Byte terakhir yang telah diterima (*acknowledged*) oleh receiver
- EffectiveWindow : Jumlah efektif data yang dikirim oleh *sender*

Algoritma:

a. **Additive Increase/Multiplicative Decrease**

Sesuai namanya, algoritma ini bekerja dengan menambah nilai EffectiveWindow secara additive dan mengurangi nilai EffectiveWindow secara multiplicative. Misalkan nilai *CongestionWindow* dapat menampung 16 paket; dengan menggunakan algoritma ini nilainya akan menjadi 8 paket jika terjadi *congestion*. Sebaliknya, setiap kali suatu paket berhasil diantar, nilai congestion window akan berubah menjadi

Increment = MSS × (MSS / CongestionWindow)
CongestionWindow += Increment

Algoritma ini baik digunakan ketika jumlah paket yang ingin dikirim mendekati jumlah kapasitas yang tersedia.

b. **Slow Start**

Algoritma ini bekerja dengan mengirimkan satu paket pada awal transmisi. Setiap kali sebuah paket berhasil dikirim (ACKed), *sender* akan mengirimkan dua kali jumlah paket yang dikirim sebelumnya. Proses ini akan terus berjalan hingga *sender* mendeteksi terjadinya *congestion*. Kemudian, jumlah paket yang dikirim berkurang secara *multiplicative* dan dilanjutkan penambahan jumlah paket secara *additive*.

```
// Pseudocode algoritma Slow Start

int CongestionWindow = 1;
bool CongestionState = 0;

while (sendingData and not CongestionState)
{
    bool status = send(data);

       if (status == 1) { // Data ACKed
           CongestionWindow *= 2;
       }
       else { // Congestion occurs
           CongestionState = 1;
           CongestionWindow /= 2;
       }
}

while (sendingData)
{
```

```
        bool status = send(data);

        if (status == 1) { // Data ACKed
            CongestionWindow += 1;
        }
        else { // Congestion occurs
            CongestionWindow /= 2;
        }
}
```

Algoritma ini dirancang untuk menutupi kelemahan algoritma ADMI, yaitu lambatnya pengiriman data pada awal transmisi.

c. **Fast Retransmit and Fast Recovery**
Algoritma ini bekerja dengan menggunakan algoritma *heuristik*. Berbeda dengan kedua algoritma sebelumnya yang mendeteksi *congestion* dengan melihat adanya *timeout*, algoritma ini mendeteksi adanya *congestion* jika *sender* menerima *duplicate ACKs* (umumnya 3). *Duplicate ACKs* dapat terjadi karena *receiver* akan selalu mengembalikan nilai *ACK* terbesar yang seluruh nilai di bawahnya telah di *acknowledge*. Bagian ini yang disebut sebagai *fast retransmit* akan mempercepat waktu pengiriman karena tidak bergantung pada waktu *timeout* yang dapat memakan waktu yang banyak. Kemudian, setelah *congestion* terdeteksi, *sender* akan membagi dua nilai paket yang dikirim dan tetap berada pada tahap *fast recovery*, bukan *slow start*.

3. OSI Reference Model mendefinisikan pembagian fungsionalitas *network* menjadi 7 layer/tingkatan. Model ini bukanlah sebuah tingkatan *protokol* yang digunakan seluruh jaringan di dunia, melainkan sebuah *reference model* untuk sebuah tingkatan protokol. Metode pengiriman data antar *layer* adalah dengan melakukan proses *enkapsulasi* dari satu *layer* ke *layer* di bawahnya. Setelah tiba di perangkat tujuan, *layer* terendah akan melakukan proses *dekapsulasi* dan mengirimkannya ke *layer* di atasnya. Proses dekapsulasi dimulai dengan membaca *header* dari data yang dikirim untuk mengetahui jenis operasi yang harus dilakukan. Hal ini akan diulangi hingga seluruh proses *dekapsulasi* telah selesai dan data akan dibaca sesuai dengan protokol yang telah ditetapkan. *Layer* yang terdapat pada model ini dimulai dari tingkat terendah yaitu

a. **Physical**
   PDU : Bit, Symbol
   Fungsi : Melakukan transmisi antara *raw bits* antar *hardware*
b. **Data Link**
   PDU : Frame
   Fungsi : Mengelompokkan bit menjadi sebuah agregat yang lebih besar bernama *frames* dan sebaliknya.
c. **Network**
   PDU : Packet
   Fungsi : Melakukan dan mengatur *addressing*, *routing*, dan *traffic control* antar jaringan
d. **Transport**
   PDU : Message
   Fungsi : Menyediakan cara bagi *end-host* untuk mengirimkan data dari asal hingga tujuan
e. **Session**
   PDU : Data
   Fungsi : Menggabungkan beberapa *transport messages* menjadi satu
f. **Presentation**
   PDU : Data
   Fungsi : Mengatur format data yang ingin dikirimkan
g. **Application**
   PDU : Data
   Fungsi : Protokol level tinggi yang dapat dibaca dan diterjemahkan oleh aplikasi

4.a. DHCP Server adalah sebuah perangkat yang bertugas untuk menyimpan informasi mengenai konfigurasi protokol DHCP. DHCP adalah sebuah protokol *client/server* yang secara otomatis akan menyediakan IP host beserta dengan IP address yang berkaitan dan konfigurasi lainnya yang berhubungan, seperti **subnet mask** dan **default gateway**.

4.b. Tugas dan mekanisme umum dari DHCP Server adalah:
- Menyediakan IP host beserta IP address beserta konfigurasi lainnya yang berhubungan kepada DHCP *client*

- Menyimpan informasi mengenai setiap *host* yang terhubung dengan DHCP *server*
- Mengontrol seluruh koneksi *host* yang terhubung dengan DHCP *server*

4.c. Keempat jenis *message* yang digunakan oleh DHCP adalah:
- DHCPDISCOVER, bertipe **broadcast**. Digunakan oleh *client* untuk mengecek apakah terdapat DHCP *server* yang tersedia.
- DHCPOFFER, bertipe **unicast**. *Server* akan merespons DHCPDISCOVER dari *client*
- DHCPREQUEST, bertipe **broadcast** atau **unicast**. *Client* akan meminta sebuah *IP address* yang disediakan oleh DHCP *server*
- DHCPACK, bertipe **unicast**. *Server* akan menyetujui *request* dari *client* dan memberikan parameter konfigurasi jaringan kepada *client*.

5. Pada metode transmisi *packet switching* digunakan strategi bernama *store-and-forward*, dimana setiap *node* dari *store-and-forward network* akan menerima paket (dalam konteks ini merupakan bagian dari data lengkap) dan kemudian mengirimkannya lagi ke *node* lain hingga tiba di *node* tujuan.
   Pada metode transmisi *circuit switching*, *circuit-switched-network* akan pertama menetapkan jalur/sirkuit yang akan dipilih, kemudian aliran bit akan dikirimkan dari *source node* menuju *destination node*.
   Pada metode transmisi *message switching*, kembali digunakan strategi *store-and-forward*, namun data akan dikirim secara utuh. Setelah data diterima dalam suatu *node*, data akan terus dikirimkan ke *node* lain hingga tiba di *node* tujuan.

6.a. Setiap perangkat yang terhubung dengan internet memiliki sebuah *IP address*, yaitu sebuah alamat unik yang menandakan perangkat tersebut dalam sebuah jaringan internet. Ketika ingin mengakses sebuah data di internet, perangkat kita akan mengirimkan sebuah *request* yang akan dikirimkan ke ISP yang kita gunakan. ISP tersebut kemudian akan menggunakan DNS Server untuk mengubah *link* yang kita minta menjadi *IP address* dari *server* yang menyimpan data tersebut dan mengirimkan *request* yang kita inginkan. Setelah sampai, *server* akan

merespons *request* kita dan akan mengembalikannya ke ISP yang kita gunakan. ISP kemudian akan mengirimkan data tersebut kembali ke kita. Selama proses pengiriman dan penerimaan data, dilakukan proses enkapsulasi dan dekapsulasi data.

6.b. Karena ISP harus mengeluarkan biaya untuk membuat, memonitor, dan melakukan *maintenance* infrastruktur yang mereka miliki. Selain itu, *bandwidth* yang kita gunakan tersusun atas kumpulan bit yang menggunakan listrik.

6.c. Router dipasang di rumah kita agar koneksi internet dapat dicapai di rumah kita.

6.d. Router berfungsi sebagai *extender*, dimana satu *IP address* yang dimiliki oleh *router* dapat digunakan oleh berbagai perangkat melalui protokol DHCP. Selain itu, router juga memiliki *firewall* yang berguna sebagai fitur *security* tambahan dan juga dapat memonitor penggunaan internet tiap perangkat yang terhubung.

IV. Sistem Paralel dan Terdistribusi

1. Hirarki memori pada CUDA terdiri atas 5 tingkatan, yaitu:
   - **Register**: Merupakan memori dengan kecepatan akses tertinggi dan dapat diakses langsung oleh *thread*.
   - **Local memory**: Merupakan memori yang terdapat pada tiap *thread* yang digunakan ketika seluruh *register* telah digunakan atau pada saat dimana *register* tidak dapat digunakan.
   - **Shared memory**: Merupakan memori yang dapat diakses oleh seluruh *threads* dalam *block* yang sama.
   - **Global memory**: Merupakan memori dengan kecepatan terendah dan ukuran terbesar.
   - **Constant memory**: Merupakan memori yang mirip dengan *global memory* namun bertipe *read-only* dan berukuran lebih kecil.

1.a. Tidak bisa, karena *local memory* hanya dapat diakses oleh *thread* yang bersangkutan.

1.b. Tidak bisa, karena *shared memory* hanya dapat diakses oleh *thread* yang berada pada *block* yang sama.

1.c. Jika *warp* yang bersangkutan berada pada *block* yang sama, maka *shared memory* dapat diakses.

1.d. Ya, karena seluruh *thread* yang berjalan dapat mengakses *global memory*.

2. **The network is reliable** adalah sebuah *fallacy* yang mengasumsikan bahwa sebuah *network* akan selalu *available* dan *data* akan selalu tiba. Namun pada kenyataannya, bisa saja terjadi *outages*, *delays*, dan *packet loss*. Hal ini dapat menyebabkan berbagai masalah, seperti *data corruption*, *lost connection*, dan *degraded performance*.
Topology doesn't change adalah sebuah *fallacy* yang mengasumsikan bahwa topologi dari sebuah *network* tidak pernah berubah. Namun pada kenyataannya, sebuah *network* akan berubah seiring waktu dengan penambahan atau pengurangan *node* yang ada. Hal ini akan menyebabkan berbagai masalah, seperti *routing loops*, *partition*, dan *lost connection*.

3.a. **MPI_Send** adalah fungsi *blocking* yang digunakan untuk mengirimkan data ke *process* lain. Bentuk dari fungsi ini adalah

```
MPI_Send(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_COMM comm
)
```

Fungsi ini akan mengirimkan data yang tersimpan dalam `buffer` sepanjang `count` dengan tipe `datatype` yang ditujukan kepada suatu *process* `dest` dengan `tag` yang berfungsi untuk menandakan *pesan* tersebut melalui tipe komunikasi `comm`.

MPI_Recv adalah fungsi *blocking* yang digunakan untuk menerima data dari *process* lain. Bentuk dari fungsi ini adalah

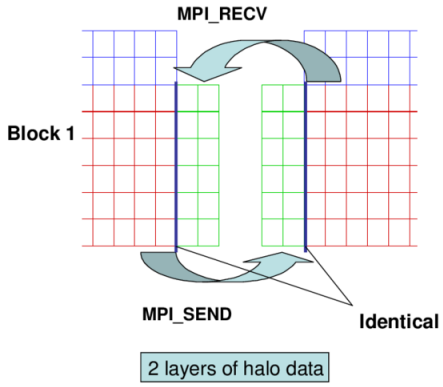
```
MPI_Recv(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
```

```
    MPI_COMM comm,
    MPI_Status *status
)
```

Fungsi ini akan menyimpan data yang akan disimpan dalam `buffer` sepanjang `count` dengan tipe `datatype` yang berasal dari process `source` dengan `tag` yang berfungsi untuk menandakan pesan tersebut melalui tipe komunikasi `comm` dan akan mengembalikan tipe penerimaan `status`.

Kedua fungsi ini bekerja sama untuk mengirimkan dan menerima data dari satu *process* ke *process* lainnya. *Process* pengirim akan memanggil fungsi MPI_Send dan *process* penerima akan memanggil MPI_Recv.

Berikut adalah illustrasi dari cara kerja kedua fungsi di atas



3.b. **MPI_Bcast** adalah fungsi yang digunakan untuk menyebarkan data yang sama ke *process* lainnya. Bentuk dari fungsi ini adalah

```
MPI_Bcast(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_COMM comm
)
```

Fungsi ini akan menyebarkan data yang tersimpan pada `buffer` sepanjang `count` dengan tipe `datatype` yang dikirimkan oleh *process root* dengan tingkatan `rank` melalui tipe komunikasi

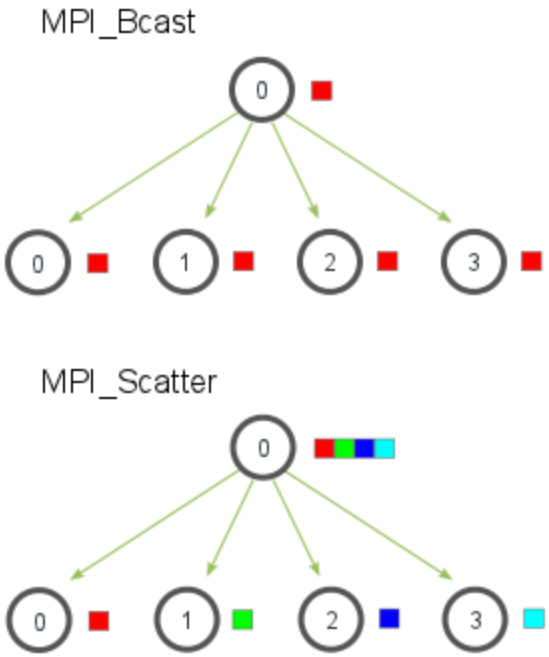
`comm`. Kemudian, seluruh *process* lainnya akan menerima data yang sama, yaitu `buffer`.

MPI_Scatter adalah fungsi yang digunakan untuk menyebarkan data yang berbeda yang disimpan dalam sebuah array ke *process* lainnya. Bentuk dari fungsi ini adalah

```
MPI_Scatter(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int root,
    MPI_COMM comm
)
```

Fungsi ini akan menyebarkan data yang tersimpan dalam `sendbuf`, dimana setiap *process* akan menerima data berbeda yang berukuran `sendcount` dengan tipe `sendtype` yang dikirimkan oleh *process* root dengan tingkatan `rank` melalui tipe komunikasi `comm`. Jenis data yang diterima oleh *process* lain ditentukan oleh rank dari process penerima. Kemudian, *process* penerima akan menyimpan data ke dalam `recvbuf` berukuran `recvcount` dengan tipe data `recvtype`.

Berikut adalah illustrasi dari cara kerja kedua fungsi di atas



3.c. **MPI_Reduce** adalah fungsi yang digunakan untuk melakukan suatu operasi *reduce* (penambahan, perkalian, dsb.) dari berbagai elemen yang berada pada *process* yang berbeda. Kemudian, fungsi ini akan mengembalikan hasilnya ke suatu *process* yang telah ditetapkan. Bentuk dari fungsi ini adalah

```
MPI_Reduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm
)
```

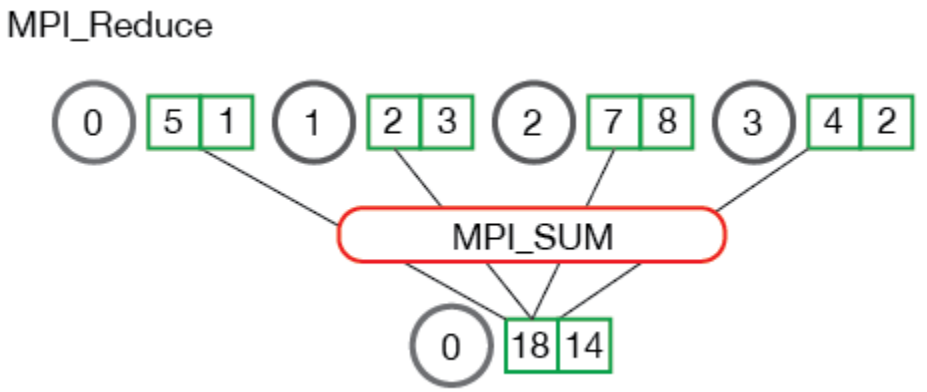
Fungsi ini akan menerima data `sendbuf` sepanjang `count` dengan tipe data `datatype` dari seluruh *process*. Kemudian, dilakukan operasi `Op` yang akan disimpan pada *process* `root` dalam variabel `recvbuf` melalui tipe komunikasi `comm`.

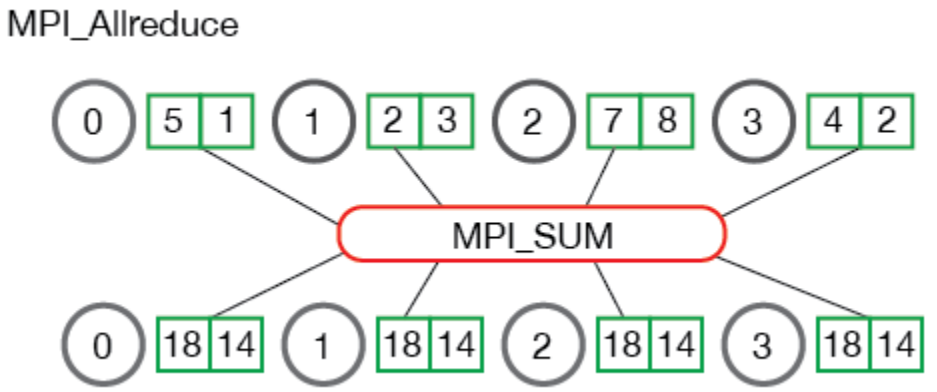
MPI_Allreduce adalah fungsi yang digunakan untuk melakukan suatu operasi *reduce* (penambahan, perkalian, dsb.) dari berbagai elemen yang berada pada *process* yang berbeda. Kemudian, fungsi ini akan mengembalikan hasilnya ke seluruh process yang bersangkutan. Bentuk dari fungsi ini adalah

```
MPI_Allreduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm
)
```

Fungsi ini akan menerima data `sendbuf` sepanjang `count` dengan tipe data `datatype` dari seluruh *process*. Kemudian, dilakukan operasi `Op` yang akan disimpan dalam variabel `recvbuf` pada seluruh *process* yang bersangkutan melalui tipe komunikasi `comm`.

Berikut adalah illustrasi dari cara kerja kedua fungsi di atas





3.d. **MPI_Gather** adalah fungsi yang digunakan untuk menggabungkan berbagai elemen yang berada pada *process* yang berbeda. Kemudian, fungsi ini akan menyimpan hasil penggabungan ke

suatu *process* yang telah ditetapkan. Bentuk dari fungsi ini adalah

```
MPI_Gather(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int root,
    MPI_COMM comm
)
```

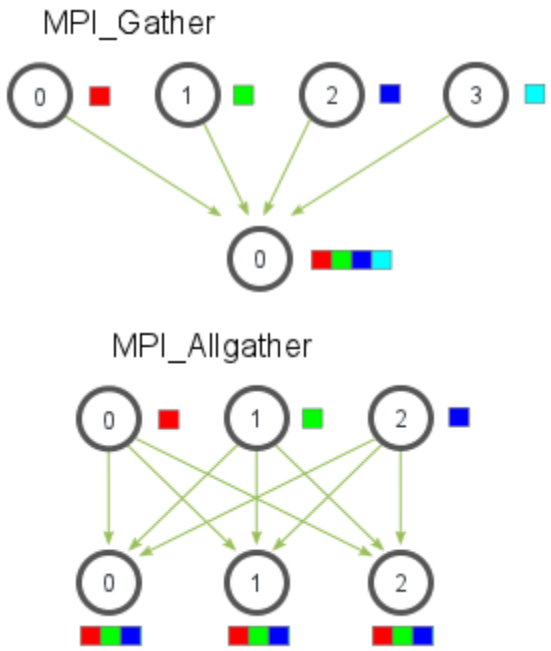
Fungsi ini akan menerima data `sendbuf` sepanjang `sendcount` dengan tipe data `sendtype` dari seluruh *process*. Kemudian, hasil penggabungan akan disimpan pada *process* `root` dalam variabel `recvbuf` sepanjang `recvcount` dengan tipe data `recvtype` melalui tipe komunikasi `comm`.

MPI_Allgather adalah fungsi yang digunakan untuk menggabungkan berbagai elemen yang berada pada *process* yang berbeda. Kemudian, fungsi ini akan menyimpan hasil penggabungan ke seluruh *process* yang bersangkutan. Bentuk dari fungsi ini adalah

```
MPI_Allgather(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    MPI_COMM comm
)
```

Fungsi ini akan menerima data `sendbuf` sepanjang `sendcount` dengan tipe data `sendtype` dari seluruh *process*. Kemudian, hasil penggabungan akan disimpan pada setiap *process* yang bersangkutan dalam variabel `recvbuf` sepanjang `recvcount` dengan tipe data `recvtype` melalui tipe komunikasi `comm`.

Berikut adalah illustrasi dari cara kerja kedua fungsi di atas



4. **CAP Theorem** adalah sebuah teorema yang menyatakan bahwa suatu sistem terdistribusi hanya dapat memenuhi dua dari tiga hal berikut:
   - *Consistency*, setiap replika data memiliki nilai yang sama dan menyimpan data hasil operasi *write* paling baru
   - *Availability*, setiap *client* yang melakukan *request* akan selalu mendapatkan respons yang valid, tanpa pengecualian
   - *Partition Tolerance*, sistem dapat terus berjalan jika terjadi sebuah *partition*, yaitu terjadinya koneksi yang hilang atau *delayed* antara dua *node* yang berbeda

   Terdapat tiga jenis sistem yang mungkin dibuat, yaitu:
   - **CA system**, yaitu sistem yang mengutamakan *consistency* dan *availability* di seluruh *node*. Namun, jika terjadi *partition* maka sistem ini tidak dapat digunakan karena tidak bisa menjamin *fault tolerance*. Contoh sistem ini adalah PostgreSQL.
   - **CP system**, yaitu sistem yang mengutamakan *consistency* dan *partition tolerance*. Jika sebuah *partition* atau *node* mengalami *failure*, *node* tersebut akan dimatikan dan operasi *write* dilarang hingga *node* tersebut bisa digunakan kembali. Contoh sistem ini adalah MongoDB.

- **AP system**, yaitu sistem yang mengutamakan *availability* dan *partition tolerance*. Jika sebuah *partition* mengalami *failure*, *node* tersebut masih dapat digunakan, namun data yang terdapat pada *node* tersebut tidak akan menyimpan data terbaru. Contoh sistem ini adalah Apache Cassandra.

5.a. **Database replication** adalah suatu proses menyalin data dari database *source* ke database lainnya untuk memastikan semua *database* mempunyai akses ke data yang sama.

5.b. CockroachDB merupakan salah satu contoh *CP system*, yaitu sistem yang mengutamakan *consistency* dan *partition tolerance*. Hal ini berarti bahwa jika terdapat suatu *server down*, maka sistem ini dijamin akan dapat memberikan data yang tepat kepada *requester*.

5.c. CockroachDB menggunakan **RAFT Protocol** untuk menentukan data pada seluruh *node* memiliki nilai yang sama. Protokol ini mempunyai 3 komponen utama, yaitu:
- **Leader election**: Sebuah *leader* akan dipilih dari *nodes* yang ada. *Leader* bertanggung jawab untuk menerima *request*, mereplikasikannya ke *node* lain, dan membuat keputusan.
- **Log replication**: *Leader* akan menyimpan *log* yang berisi semua *transaction* dan membagikannya ke *node* lain. Ketika *node* lain menerima *log*, *node* tersebut akan mengubah *state*-nya agar sesuai dengan *log* yang diberikan.
- **Safety**: Protokol ini menjamin seluruh *node* memiliki *state* yang sama dengan menggunakan sistem *voting* sebelum suatu keputusan diambil.

6.a. Algoritma Consensus adalah kelompok algoritma yang digunakan untuk menjamin kesamaan data pada sistem atau proses yang terdistribusi

6.b. Ketiga algoritma consensus yang sering digunakan:

a. **Algoritma Paxos**

Algoritma ini memiliki 3 aktor, yaitu:
- *Proposer* bertugas untuk menerima *value* dari *client* dan akan mempersuasi *acceptor* untuk menerima *value* mereka.
- *Acceptor* bertugas untuk menerima value dari *proposer* dan mengumumkan hasil penerimaan.

- *Learner* bertugas untuk mengumumkan hasil akhir *consensus*.

Pada sistem, ketiga bagian ini *non-eksklusif*, sehingga suatu *node* dapat menjadi tiga aktor sekaligus. Algoritma *paxos* memiliki dua fase, yaitu

i. *Prepare phase*

Setiap *proposer* memilih *value* dan mengirimkannya pada *acceptor*. Tidak semua *acceptor* harus menerima *value* tersebut, namun mayoritas *node* harus menerima *value* tersebut. Kemudian, terdapat tiga kasus yang mungkin terjadi pada *acceptor*:

- Jika value yang diterima lebih besar dari *value* terbesar yang dimiliki, *acceptor* akan mengabari *accept* kepada *proposer* pengirim
- Jika *acceptor* sudah pernah menerima *value* yang lebih besar, *acceptor* akan mengabari *accept* kepada *proposer* pengirim, serta menyertakan value yang dimiliki
- Jika *acceptor* menerima value yang lebih rendah, maka *acceptor* akan mengabaikan *value* tersebut

ii. *Commit phase*

Setelah fase *prepare* terdapat tiga kasus yang mungkin terjadi pada *proposer*:

- Jika *proposer* mendapatkan respons *accept*, *proposer* akan mengirimkan *value* yang dimilikinya lagi kepada *acceptor*
- Jika *proposer* tidak mendapatkan respons, maka *proposer* akan menyimpulkan bahwa value yang dimiliki tidak cukup tinggi dan akan menggantinya
- Jika *proposer* mendapatkan *respons accept* dari mayoritas *acceptor*, maka *proposer* akan mengabari *learner* bahwa konsensus telah dicapai

Pada sisi *acceptor*, terdapat dua kasus yang mungkin terjadi:

- Jika value *request* yang diterima lebih tinggi, *acceptor* akan membalas *accepted* kepada *proposer* dan mengubah valuenya
- Jika value yang diterima lebih rendah, maka *acceptor* akan mengabaikannya

Contoh kasus algoritma paxos pada sistem dengan 5 node dengan sintaks State = { (Node, ID, Promise=(ID, User)) }

- Alice ingin mengakses sistem dengan mengirimkan request pada node 3. Node 3 akan membangkitkan ID 1.
  - *Initial phase*

```
State = {
    (1, X, X),
    (2, X, X),
    (3, 1, X),
    (4, X, X),
    (5, X, X)
}
```

  - *Prepare phase*

```
State = {
    (1, X, (1, Alice)),
    (2, X, (1, Alice)),
    (3, 1, (1, Alice)),
    (4, X, (1, Alice)),
    (5, X, (1, Alice))
}
```

  - *Commit phase*

```
State = {
    (1, 1, (1, Alice)),
    (2, 1, (1, Alice)),
    (3, 1, (1, Alice)),
    (4, 1, (1, Alice)),
    (5, 1, (1, Alice))
}
```

- Alice ingin mengakses sistem dengan request pada node 5. Node 5 akan membangkitkan ID 5. Namun, node 2 dan 3 mengalami gangguan.
  - *Initial phase*

```
State = {
    (1, 1, (1, Alice)),
    X (2, 1, (1, Alice)),
    X (3, 1, (1, Alice)),
    (4, 1, (1, Alice)),
    (5, 5, (1, Alice))
}
```

  - *Prepare phase*

```
State = {
    (1, 1, (5, Alice)),
    X (2, 1, (1, Alice)),
```

```
        X (3, 1, (1, Alice)),
        (4, 1, (5, Alice)),
        (5, 5, (5, Alice))
    }
```

- *Commit phase*

```
    State = {
        (1, 5, (5, Alice)),
        X (2, 1, (1, Alice)),
        X (3, 1, (1, Alice)),
        (4, 5, (5, Alice)),
        (5, 5, (5, Alice))
    }
```

Mayoritas dicapai dengan ID 2 dengan 2 node mengalami gangguan.

- Sistem diperbaiki dan seluruh node bekerja kembali. Bob ingin mengakses sistem dengan request ke node 2. Node 2 akan membangkitkan ID 3.
  - *Initial phase*

```
    State = {
        (1, 5, (5, Alice)),
        (2, 3, (3, Bob)),
        (3, 1, (1, Alice)),
        (4, 5, (5, Alice)),
        (5, 5, (5, Alice))
    }
```

  - *Prepare phase*

```
    State = {
        (1, 5, (5, Alice)),
        (2, 3, (3, Bob)),
        (3, 1, (3, Bob)),
        (4, 5, (5, Alice)),
        (5, 5, (5, Alice))
    }
```

  - *Commit phase*

```
    State = {
        (1, 5, (5, Alice)),
        (2, 3, (3, Bob)),
        (3, 3, (3, Bob)),
        (4, 5, (5, Alice)),
        (5, 5, (5, Alice))
    }
```

Mayoritas tidak dicapai, akses bob ditolak.

- Sistem akan mengulangi permintaan akses, node 2 akan membangkitkan ID 7. Namun, karena pemilik ID mayoritas telah diketahui merupakan Alice, Bob tidak akan mendapatkan akses.
  - *Initial phase*
    ```
    State = {
        (1, 5, (5, Alice)),
        (2, 7, (7, Alice)),
        (3, 3, (3, Bob)),
        (4, 5, (5, Alice)),
        (5, 5, (5, Alice))
    }
    ```
  - *Prepare phase*
    ```
    State = {
        (1, 5, (7, Alice)),
        (2, 7, (7, Alice)),
        (3, 3, (7, Alice)),
        (4, 5, (7, Alice)),
        (5, 5, (7, Alice))
    }
    ```
  - *Commit phase*
    ```
    State = {
        (1, 7, (7, Alice)),
        (2, 7, (7, Alice)),
        (3, 7, (7, Alice)),
        (4, 7, (7, Alice)),
        (5, 7, (7, Alice))
    }
    ```
- Alice ingin mengakses dengan request ke node 3. Node 3 akan membangkitkan ID 10. Namun, setelah mengirimkan *node* 1 dan 2 melakukan *commit*, *node 3* mengalami gangguan.
  - *Initial phase*
    ```
    State = {
        (1, 7, (7, Alice)),
        (2, 7, (7, Alice)),
        (3, 7, (10, Alice)),
        (4, 7, (7, Alice)),
        (5, 7, (7, Alice))
    }
    ```
  - *Prepare phase*
    ```
    State = {
        (1, 7, (10, Alice)),
        (2, 7, (10, Alice)),
    ```

```
        (3, 10, (10, Alice)),
        (4, 7, (10, Alice)),
        (5, 7, (10, Alice))
    }
```

- *Commit phase*

```
    State = {
        (1, 10, (10, Alice)),
        (2, 10, (10, Alice)),
        X (3, 10, (10, Alice)),
        (4, 7, (10, Alice)),
        (5, 7, (10, Alice))
    }
```

- Bob datang kembali dan ingin mengakses sistem dengan request ke node 4. Node 4 akan membangkitkan ID 9. Namun, setelah diketahui bahwa *commit* Alice belum dilakukan pada seluruh *node*, *node* yang telah melakukan *commit* akan memberitahukan akses kepada seluruh *node* lainnya dan akses akan dilarang untuk bob.

Algoritma ini bekerja karena jika suatu permintaan akses pernah di *commit*, maka permintaan tersebut pernah menjadi sebuah mayoritas pada *prepare phase*. Hal ini membuat sistem tahan terhadap gangguan ketika *node* yang meminta akses mengalami gangguan.

b. **Algoritma Raft**

Algoritma ini bekerja dengan membagi waktu menjadi term dimana setiap kali terjadi *voting*, nilai term akan bertambah dimana setiap *term* akan dimulai dengan voting. Masing-masing *node* pada sistem dapat berupa *follower*, *candidate*, atau *leader*. Setiap *follower* dapat menjadi *candidate* dan setiap *follower* dapat melakukan *election timeout* yang bersifat *random*. Ketika voting terjadi, *follower* akan melakukan *timeout*, *follower* akan berubah menjadi *candidate,* memilih dirinya sendiri, dan meminta *vote request* ke *follower* lainnya. Kemudian, jika *follower* belum pernah menerima *vote request* pada *term* saat ini, maka *follower* akan menerima *request* tersebut dan menambah nilai *term* miliknya. *Candidate* yang menerima *vote* mayoritas akan menjadi *leader*. *Leader* kemudian akan mengirimkan *heartbeat*, yaitu pesan berkala untuk mengecek apakah *follower* mengalami gangguan. Jika *follower* tidak mendapatkan *heartbeat* secara berkala, maka

dapat dipastikan bahwa *leader* mengalami gangguan. Kemudian, voting akan dilakukan kembali dengan *node* yang masih aktif. Contoh kasus algoritma raft pada sistem dengan 5 node dengan sintaks State = { (Node, ID) } dan Vote = { Yes/No }. Node 1 akan bertindak sebagai leader

- Iterasi 1: Alice ingin mengakses sistem dengan ID 1.
  - *Initial state*

    ```
    State = {
        (1, 1),
        (2, X),
        (3, X),
        (4, X),
        (5, X)
    }
    ```
  - *Voting state*

    ```
    Vote = {
        Yes,
        Yes,
        Yes,
        Yes,
        Yes
    }
    ```
  - *Result state*

    ```
    State = {
        (1, 1),
        (2, 1),
        (3, 1),
        (4, 1),
        (5, 1)
    }
    ```
- Iterasi 2: Alice ingin mengakses sistem dengan ID 3. Node 2 dan 3 mengalami gangguan.
  - *Initial state*

    ```
    State = {
        (1, 3),
        X (2, 1),
        X (3, 1),
        (4, 1),
        (5, 1)
    }
    ```
  - *Voting state*

    ```
    Vote = {
        Yes,
    ```

```
        -,
        -,
        Yes,
        Yes
    }
```

- *Result state*

```
    State = {
        (1, 3),
        X (2, 1),
        X (3, 1),
        (4, 3),
        (5, 3)
    }
```

- Iterasi 3: Sistem diperbaiki, Alice ingin mengakses sistem dengan ID 5. Node 1 dan 4 mengalami gangguan dan node 5 dipilih sebagai *leader* karena memiliki ID tertinggi. Sebagai *leader*, node 5 akan melakukan *heartbeat* kepada node lainnya
  - *Initial state*

```
    State = {
        X (1, 3),
        (2, 3),
        (3, 3),
        X (4, 3),
        (5, 5)
    }
```

  - *Voting state*

```
    Vote = {
        -,
        Yes,
        Yes,
        -,
        Yes
    }
```

  - *Result state*

```
    State = {
        X (1, 3),
        (2, 5),
        (3, 5),
        X (4, 3),
        (5, 5)
    }
```

Algoritma ini bekerja karena *leader* akan melakukan *heartbeat* agar nilai pada *node* lainnya selalu sama dengan *node* leader sebelum dilakukan voting. Ketika *leader* mengalami gangguan, maka akses akan diberikan kepada *node* lain yang memiliki ID tertinggi, sehingga mencegah terjadinya akses oleh pihak lain.

c. **Algoritma Zab**

Algoritma ini bisa dikatakan merupakan pertengahan antara algoritma Paxos dan Raft. Cara kerja algoritma ini adalah sebagai berikut

i. *Leader Election*

Sebelum operasi dapat dilakukan, Zab akan memilih satu *leader* diantara *node* yang tersedia. *Leader* berperan untuk melakukan *message broadcast* kepada *node* lainnya.

ii. *Message Broadcasting*

Ketika terdapat suatu permintaan akses, *leader* akan membuat sebuah *message* dengan ID yang unik dan mengirimkannya ke *node* lainnya. *Message* dikirimkan pada suatu FIFO *channel* yang memastikan *message* tiba sesuai urutan.

iii. *Acknowledgments and Quorums*

Setelah *message* diterima, setiap *node* dapat mengirimkan ACK kepada *leader*. Jika *leader* telah menerima pesan ACK dan memenuhi kuorum, maka *leader* akan tahu bahwa *message* diterima dan akan bersiap mengirimkan *message* selanjutnya.

iv. *Handling Features*

Ketika *leader* sedang mengalami gangguan, proses pemilihan *leader* baru akan dilakukan

Contoh kasus algoritma zab pada sistem dengan 5 node dengan sintaks State = { (Node, ID, Promise=(ID, Vote={ Yes/No })) }. Node 1 akan bertindak sebagai leader

- Iterasi 1: Alice ingin mengakses sistem dengan ID 1.
  - *Initial state*

```
State = {
    (1, 1, (X, X)),
    (2, X, (X, X)),
    (3, X, (X, X)),
    (4, X, (X, X)),
```

```
    (5, X, (X, X))
}
```

- *Broadcasting phase*

```
State = {
    (1, 1, (X, X)),
    (2, X, (1, Yes)),
    (3, X, (1, Yes)),
    (4, X, (1, Yes)),
    (5, X, (1, Yes))
}
```

- *Commit phase*

```
State = {
    (1, 1, (X, X)),
    (2, 1, (1, Yes)),
    (3, 1, (1, Yes)),
    (4, 1, (1, Yes)),
    (5, 1, (1, Yes))
}
```

- Iterasi 2: Alice ingin kembali mengakses sistem dengan ID 3. Node 2 dan 3 mengalami gangguan.
  - *Initial state*

```
State = {
    (1, 3, (X, X)),
    X (2, 1, (X, X)),
    X (3, 1, (X, X)),
    (4, 1, (X, X)),
    (5, 1, (X, X))
}
```

  - *Broadcasting phase*

```
State = {
    (1, 3, (X, X)),
    X (2, 1, (X, X)),
    X (3, 1, (X, X)),
    (4, 1, (3, Yes)),
    (5, 1, (3, Yes))
}
```

  - *Commit phase*

```
State = {
    (1, 3, (X, X)),
    X (2, 1, (X, X)),
    X (3, 1, (X, X)),
    (4, 3, (3, Yes)),
    (5, 3, (3, Yes))
}
```

- Iterasi 3: Sistem diperbaiki, Alice ingin mengakses sistem dengan ID 5. Node 1 dan 4 mengalami gangguan dan node 5 dipilih sebagai *leader* karena memiliki ID tertinggi.
  - *Initial state*
    ```
    State = {
        X (1, 3, (X, X)),
        (2, 1, (X, X)),
        (3, 1, (X, X)),
        X (4, 3, (X, X)),
        (5, 5, (X, X))
    }
    ```
  - *Broadcasting phase*
    ```
    State = {
        X (1, 3, (X, X)),
        (2, 1, (5, Yes)),
        (3, 1, (5, Yes)),
        X (4, 3, (X, X)),
        (5, 5, (X, X))
    }
    ```
  - *Commit phase*
    ```
    State = {
        X (1, 3, (X, X)),
        (2, 5, (5, Yes)),
        (3, 5, (5, Yes)),
        X (4, 3, (X, X)),
        (5, 5, (X, X))
    }
    ```
- Iterasi 4: Sistem diperbaiki, Alice ingin mengakses sistem dengan ID 7. Setelah *leader* mengirimkan *commit* kepada *node* 1, *node* 5 mengalami gangguan.
  - *Initial state*
    ```
    State = {
        (1, 3, (X, X)),
        (2, 5, (X, X)),
        (3, 5, (X, X)),
        (4, 3, (X, X)),
        (5, 7, (X, X))
    }
    ```
  - *Broadcasting phase*
    ```
    State = {
        (1, 3, (7, Yes)),
        (2, 5, (7, Yes)),
    ```

```
        (3, 5, (7, Yes)),
        (4, 3, (7, Yes)),
        (5, 7, (X, X))
    }
```

- *Commit phase*

```
    State = {
        (1, 7, (7, Yes)),
        (2, 5, (7, Yes)),
        (3, 5, (7, Yes)),
        (4, 3, (7, Yes)),
        X (5, 7, (X, X))
    }
```

- Iterasi 5: Setelah beberapa saat, *leader* baru (*node 1*) akan mendeteksi bahwa terdapat *commit* yang belum di *acknowledge*.
  - *Initial state*

```
    State = {
        (1, 7, (X, X)),
        (2, 5, (7, Yes)),
        (3, 5, (7, Yes)),
        (4, 3, (7, Yes)),
        X (5, 7, (X, X))
    }
```

  - *Commit phase*

```
    State = {
        (1, 7, (X, X)),
        (2, 7, (7, Yes)),
        (3, 7, (7, Yes)),
        (4, 7, (7, Yes)),
        X (5, 7, (X, X))
    }
```

Algoritma ini bekerja dengan alasan yang serupa dengan algoritma Paxos dan Raft. Jika *leader* mengalami gangguan, maka *node* yang memiliki ID tertinggi akan menjadi *leader*. Kemudian karena setiap *commit* memerlukan kuorum, maka dapat dipastikan *node* yang memiliki ID tertinggi memiliki *data* terbaru.

7. Skalabilitas adalah kemampuan suatu sistem untuk meng*handle* peningkatan kerja dengan menambah *resource* baru. Terdapat dua jenis skalabilitas, yaitu

- *Vertical scaling* dengan menambah *resource* baru (menambah RAM, meng*upgrade* CPU, dll.) dalam sistem
- *Horizontal scaling* dengan menambah *node* baru (menambah *server*) dalam sistem

Strategi untuk meningkatkan skalabilitas antara lain:

- *Replication*, dengan menyalin data ke *server* lain, data akan dapat diakses jika satu *server* mengalami kerusakan
- *Load Balancing*, yaitu membagi tugas secara merata antar *node* dalam sistem untuk mencegah suatu *server* mengalami *overload*
- *Caching*, yaitu menyimpan data yang sering diakses dalam memori untuk mengurangi akses data pada disk