BAB IV
PERANCANGAN DAN IMPLEMENTASI

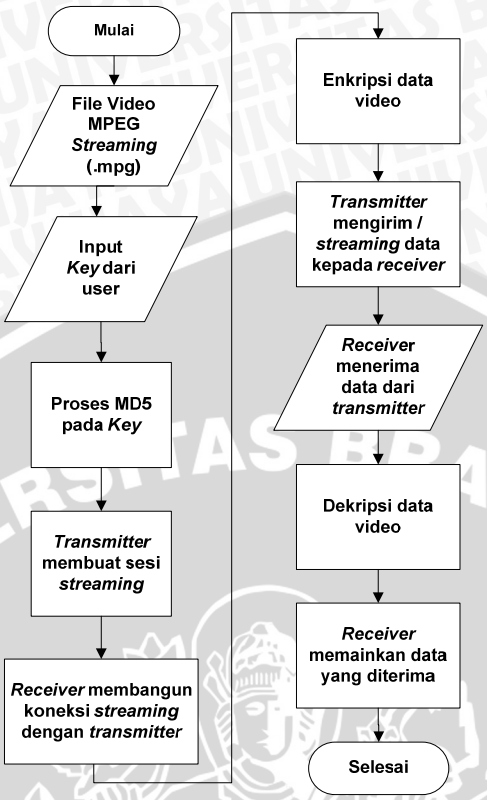
Perancangan dan implementasi aplikasi kriptografi video *streaming* menggunakan algoritma *video encryption algorithm* (VEA) yang dikerjakan dengan beberapa tahap yakni meliputi membangun koneksi antara *server* dan *client*, mengirim dan menerima *file* atau berkas antara *server* dan *client*, proses *enkripsi* pada saat pengiriman atau *streaming* dari *server*, proses dekripsi pada *client*. Pembuatan secara bertahap tersebut agar memudahkan penganalisaan aplikasi pada setiap bagian maupun aplikasi secara keseluruhan.

4.1 Perancangan Secara Umum

Perancangan aplikasi global merupakan tahap awal sebagai acuan dalam perancangan aplikasi yang akan dibuat. Perancangan ini didahului dengan pendefinisian kegiatan pengguna dalam menggunakan program kriptografi video *streaming* menggunakan algoritma *video encryption algorithm* (VEA) yang digunakan meliputi diagram alir umum sistem, diagram alir enkripsi VEA, cara kerja aplikasi, dan konfigurasi perangkat keras.

4.1.1 Diagram Alir Umum Sistem

Sistem enkripsi video *streaming* ini terdiri dari beberapa komponen yang dapat digambarkan secara umum dengan model seperti pada Gambar 4.1 :



Gambar 4.1 Diagram alir umum sistem

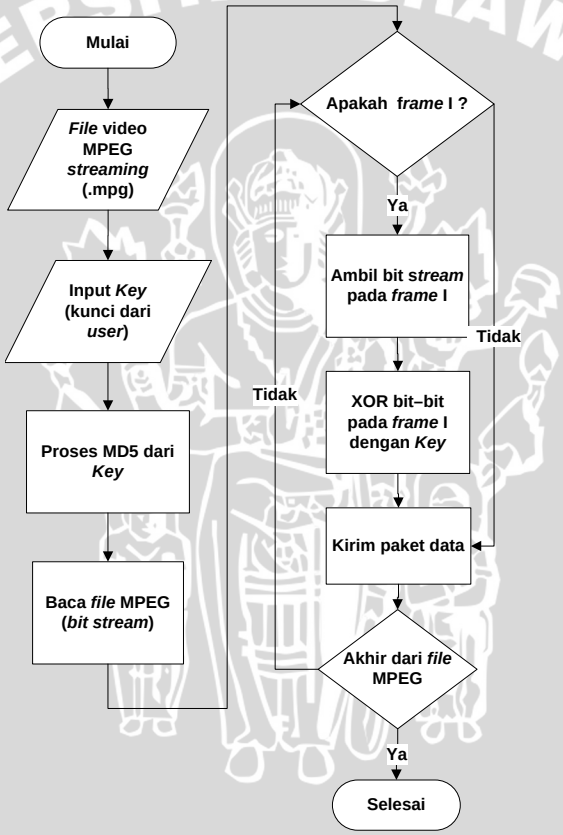
Keterangan dari diagram alir pada Gambar 4.1 :

1. Mendapatkan *file* video MPEG-1 dan MPEG-2 dengan ekstensi .mpg yang akan di-*streaming*-kan.
2. Mendapatkan *private key* dari *user* yang akan digunakan untuk proses enkripsi dan dekripsi.
3. Melakukan proses hash pada *key* dengan algoritma MD5 yang bertujuan untuk memperkuat enkripsi data video.
4. *Transmitter* atau *server* membuat sesi *streaming* supaya *receiver* atau *client* dapat membangun koneksi sehingga *transmitter* dapat mengirim data *streaming*.
5. *Receiver* melakukan proses koneksi *streaming* dengan *transmitter*.
6. Setelah terbangun koneksi antara *transmitter* dengan *receiver*, maka dilakukan proses pengiriman data. Namun data yang dikirim akan dienkripsi terlebih dahulu.
7. Proses pengiriman atau *streaming* data dari *transmitter* pada *receiver*.

8. *Receiver* menerima data *streaming* dari *transmitter*.
9. Data yang diterima akan didekripsi oleh *receiver* untuk mendapatkan data awal.
10. Setelah didapat data video original dari proses dekripsi, maka *receiver* memainkan data video MPEG tersebut.

4.1.2 Diagram Alir Enkripsi VEA

Diagram alir dari enkripsi VEA ditunjukkan pada Gambar 4.2:



Gambar 4.2 Diagram alir enkripsi VEA

Berikut penjelasan dari diagram alir dari Gambar 4.2 :

1. Mendapatkan *file* video MPEG-1 dan MPEG-2 dengan ekstensi .mpg yang akan di-*streaming*-kan dan dienkripsi.

2. Mendapatkan *key* (kunci) dari masukan user yang akan digunakan untuk proses enkripsi.
3. Melakukan proses MD5 dari *key* yang telah didapat yang betujuan untuk memperkuat *key* ataukunci.
4. Membaca *file* video MPEG dengan menggunakan operasi *file* atau berkas.
5. Mengecek apakah frame yang sedang dibaca adalah *frame* I. Jika benar, maka *bit-bit* pada *frame* Iakan di-XOR-kan dengan *bit-bit key* dari hasil proses MD5. Kemudian melakukan proses pengiriman paket atau *streaming*pada*receiver*. Jika hasil pengecekan dari frame I adalah salah, maka *bit- bit* data tersebut langsung dilakukan proses pengiriman data atau *streaming*.
6. Dilakukan pengecekan lagi yakni apakah sudah sampai akhir dari *file* atau *end of file* (EOF). Jika hasil pengecekan tidak, maka dilakukan proses pada poin 5, jika hasil pengecekan benar, maka proses selesai.

4.1.3 Cara Kerja Aplikasi

Aplikasi kriptografi video *streaming* menggunakan *video encryption algorithm* (VEA) terdiri dari dua elemen penting yakni *server* dan *client*. *Server* adalah program aplikasi yang menyediakan layanan berupa *file* video yang akan di-*streaming*-kan, mengirimkan video *streaming*, dan enkripsi. *Client* adalah program aplikasi yang meyediakan layanan untuk menerima video *streaming*, memainkan video tersebut atau sebagai player, dan dekripsi. *Server* dan *Client* untuk berkomunikasi memanfaatkan protokol TCP/IP. Komunikasi tersebut selain digunakan untuk saling berkomunikasi tetapi juga digunakan untuk *streaming* video. Karena jumlah *client* memungkinkan lebih dari satu, maka *server* untuk dapat membangun koneksi dengan semua *client* menggunakan multi-*thread*.

Sebelum melakukan koneksi antara *server* dan *client*, yang pertama kali yang dilakukan adalah melakukan persiapan dari sisi *server* dahulu yakni menyedikan *file* video mpeg yang akan di-*streaming*, memasukan *port* sebagai acuan untuk berkonesi, memasukan kunci atau key yang akan digunakan sebagai kunci enkripsi. Setelah semua sudah siap, maka *server* siap membuka koneksi TCP atau *listening* pada *client* yang akan berkoneksi.

Pada sisi *client* untuk dapat berkoneksi dengan *server* memerlukan parameter sebagai berikut memasukan IP *address* dan *port* dari *server*, memasukan kunci atau *key* enkripsi. Setelah semua parameter yang diperlukan sudah lengkap, *client* bisa mencoba membangun koneksi dengan *server*. Saat koneksi telah terbangun, *server* akan mengirimkan informasi pada client mengenai informasi video seperti resolusi, besar *file* video, dan sebagainya. Kemudian *client* meminta *streaming* dari *server*, lalu *server* akan memberikan data *streaming* pada *client* sesuai yang diminta. Namun sebelum data tersebut dikirim akan dilakukan pengecekan terlebih dahulu apakah data *streaming* tersebut mempunyai *frame* I, jika mempunyai *frame I* maka data tersebut akan dilakukan enkripsi dengan cara meng-XOR-kan dengan hasil MD5 (dimana hasil MD5 tersebut merupakan kunci yang telah dimasukan/didefinisikan sebelumnya oleh *user* atau pengguna kemudian kunci tersebut dienkripsi oleh fungsi MD5).

Setelah dienkripsi lalu dikirim, jika data tadi tidak mempunyai frame I maka data tersebut langusung dikirimkan. Kemudian *clien*t yang menerima data *streaming* tersebut akan mendekripsi paket tersebut lalu akan men-*decoding* data tersbut menjadi gambar lalu ditampilkan di layar (*player*).

4.1.4 Konfigurasi Perangkat Keras

Aplikasi kriptografi video *streaming* menggunakan algoritma *video encryption algorithm* (VEA) ini bias berjalan dengan baik apabila *hardware* sudah terkonfigurasi. *Hardware* disini meliputi komputer *server* dan komputer *client*, dimana telah terhubung sesuai dengan protokol TCP/IP pada kasus saat ini penulis menggunakan jaringan lokal seperti pada Gambar 4.3

Gambar 4.3 Konfigurasi *server* dan *client* pada jaringan lokal

(**Sumber** : Perancangan)

4.2 Perancangan Perangkat Lunak

Perancangan perangkat lunak dibangun dengan menggunakan bahasa pemrograman C#. Karena aplikasi ini terdiri dari dua buah elemen yakni aplikasi sisi *server* dan aplikasi sisi *client*, maka pada perancangan *software* ini terdapat dua perancangan. Dengan rancangan yang memiliki spesifikasi sebagai berikut :

1. Pada sisi *server* antara lain:
   - Mampu membangun koneksi atau hubungan dengan *client* menggunakan TCP/IP.
   - Mampu menangani atau *handling* tiap-tiap *client* dengan multi-*thread.*
   - Mampu menerima masukan parameter berupa berkas mpeg-1 (.mpg) dan mpeg-2 (mpeg).
   - Mampu melakukan proses enkripsi data video dengan kunci yang telah didefinisikan/ditentukan oleh pengguna (*user*) sebelumnya.
   - Mampu mentransfer atau mengirimkan data yang telah dienkripsi pada *client*.
2. Pada sisi *client* antara lain :
   - Mampu menerima koneksi dari *server* dengan TCP/IP.
   - Mampu menerima parameter yang dikirim oleh *server*.
   - Mampu melakukan proses dekripsi data video yang telah dienkripsi dengan memasukkan kunci yang sama.
   - Mampu menampilkan video (*player*).

4.2.1 Perancangan Sisi *Server*

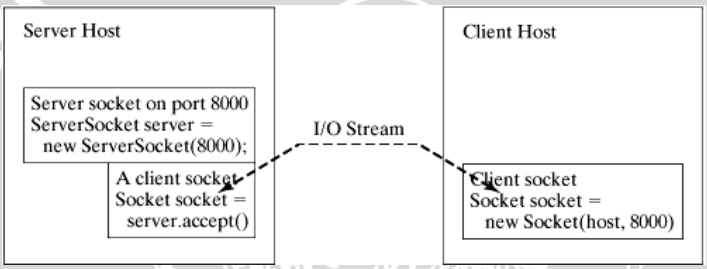
Aplikasi sisi *server* yang di beri nama “VeaTube Server” merupakan aplikasi yang akan memberi layanan (*servis*) pada setiap *client.* Perancangan sisi *server* harus mempunyai kemampuan seperti yang disebutkan pada subbab 4.2

4.2.1.1 Perancangan Membangun Koneksi dengan *Client*

Pada perancangan tahap ini dijelaskan alur atau proses dari koneksi *server* dengan *client*. Karena bahasa pemrograman yang dipakai untuk membuat aplikasi ini adalah C# maka akan digunakan *class-class* yang tersedia pada DotNet C# yaitu *class* TcpListener untuk menangani masalah koneksi atau

protocol TCP/IP. Pada *class* Socket menyediakan program untuk berkomunikasi melalui *socket. Socket* merupakan titik akhir dari koneksi logikal antara dua *host* dan digunakan untuk mengirim dan menerima data. Pemrograman jaringan biasanya melibatkan sebuah *server* dan banyak *client*.

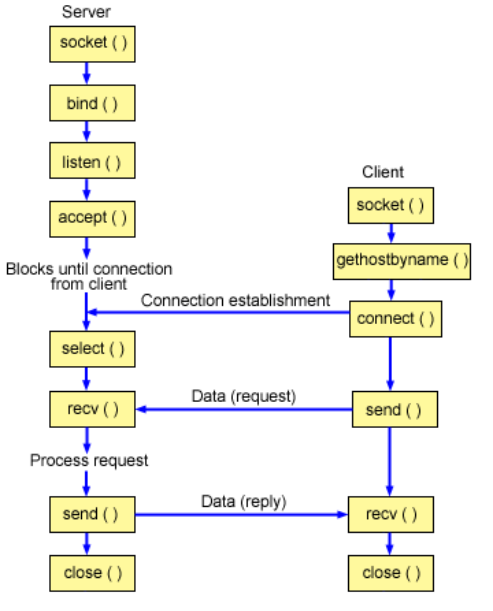
Kemudian *client* mengirim permintaan ke *server* dan *server* merespon permintaan tersebut. Sekali saja koneksi sudah terbentuk yaitu koneksi antara *client* dan *server*, maka komunikasi sudah dapat berjalan melalui *socket* tersebut. Untuk membangun sebuah *server*, maka yang diperlukan adalah membuat sebuah *server socket* dan memberi sebuah alamat *port* pengenal, dimana *port* tersebut digunakan untuk mendengarkan koneksi oleh *server* dan digunakan untuk acuan membangun koneksi oleh *cilent* seperti pada Gambar 4.4



Gambar 4.4 Skema sebuah *server socket* dan sebuah *client*
(**Sumber** : Liang, 2004:816)

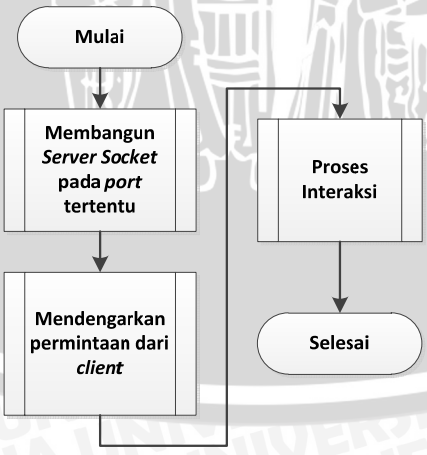
Skema pada Gambar 4.4 merupakan gambaran dari sebuah *server socket* dengan *port listener* 8000. Sehingga *server* akan mendengarkan dan menunggu koneksi pada *port* 8000. Pada pihak *client* untuk bisa berkoneksi dengan *server* tersebut harus memasukan alamat IP *address server* berserta *port*-nya yakni 8000. Pada perancangan kali ini akan digunakan *binding* alamat *port* 2323 untuk *server* s*ocket*.

Pembahasan pada subbab ini merupakan *server*, *server* ini yang akan bertugas untuk membangun *server socket*. Apabila *server socket* telah siap mendengarkan permintaan koneksi pada suatu *port*, dimana nilai *port t*ersebut didapat dari masukan *client* (*user*) misal *port* 2323, maka *server* tersebut mendengarkan dan menerima koneksi dari *client*. Ilustrasi *server socket* ditunjukkan pada Gambar 4.5



Gambar 4.5 Ilustrasi komunikasi *server socket* dengan *client* (**Sumber :**http://publib.boulder.ibm.com/infocenter/iseries /v7r1m0/topic/rzab6/rxab6502 .gif)

Pada Gambar 4.5 setelah *server* mendapat permintaan untuk membangun koneksi dari *client*, maka proses *handshaking* telah terbentuk dan siap melakukan proses komunikasi untuk mengirim data maupun menerima data. Diagram alir dari proses membangun koneksi dengan *client* ditunjukkan pada Gambar 4.6



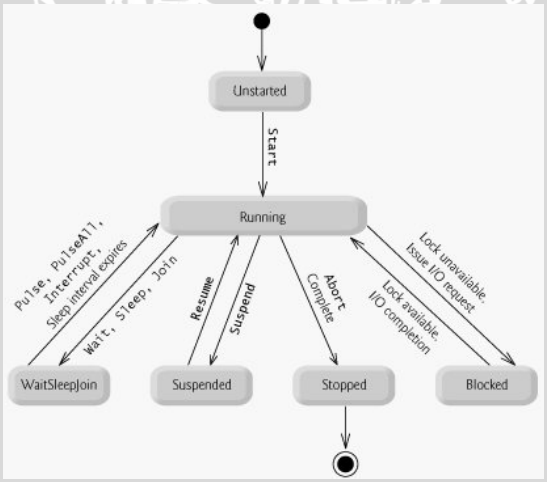
Gambar 4.6 Diagram alir pembangunan koneksi dengan *client*

Berikut penjelasan dari diagram alir dari Gambar 4.6 :

1. Membangun atau membuat *server socket* pada *port* tertentu sebagai acuan c*lient* untuk bisa berkoneksi.
2. Mendengarkan apakah ada permintaan untuk berkoneksi dari *client* dengan sebuah class Thread.
3. Melakukan proses interaksi yang telah terbangun antar *server* dan *client* yang ditangani oleh sebuah class Thread.

4.2.1.1.1 Perancangan Penanganan *Request* Koneksi dari *Client* dengan Menggunakan Sebuah *Thread*

Pada perancangan tahap ini akan dijelaskan alur proses dari penanganan *request* koneksi dari *client* dengansebuah *thread*. Pada *thread* ini berisi seluruh proses interaksi dengan *client*. Dalam membuat *thread* pada C# sudah disediakan sebuah *class* untuk *thread* dengan menggunakan namespace System.Threading, dimana hanya mendeklarasikan proses apa saja yang harus dijalankan pada objek tersebut. Class Thread pada C# mempunyai siklus hidup seperti pada Gambar 4.7

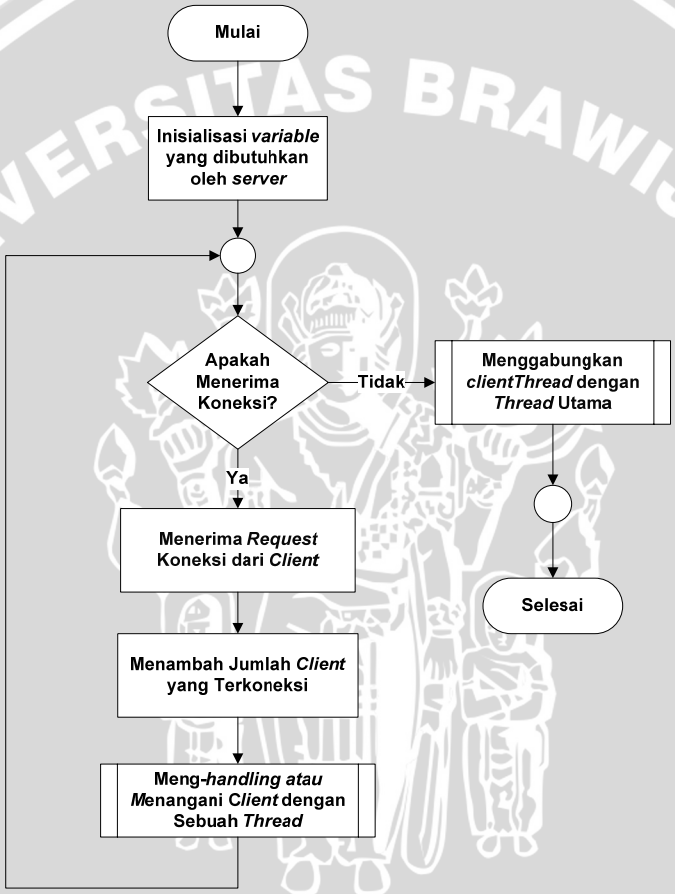


Gambar 4.7 Siklus *thread* pada C#

(**Sumber** :http://flylib.com/books/2/255/1/html/2/images/15fig01.jpg)

Sesuai siklus pada Gambar 4.7 setelah proses pembuatan objek *thread* baru, *thread* tersebut akan melakukan proses sesuai yang telah dideklarasikan sebelumnya baik itu menunggu (*sleeping*) maupun menjalankan suatu proses. Setelah selesai melakukan proses *thread* tersebut akan berhenti atau *terminated*

dengan cara menggabungkannya dengan *thread* utama. Pada perancangan aplikasi ini, proses yang harus dijalankan oleh *thread* adalah menangani interaksi dengan *client* dengan kemampuan antara lain mampu membangun koneksi dengan *client*, mengirimkan parameter dan *file* pada *client*. Diagram alir dari proses penanganan *request* koneksi dari *client* dengan menggunakan sebuah *thread* ditunjukkan pada Gambar 4.8



Gambar 4.8 Diagram alir penanganan *request* koneksi dari *client*

Berikut penjelasan dari diagram alir dari Gambar 4.8:

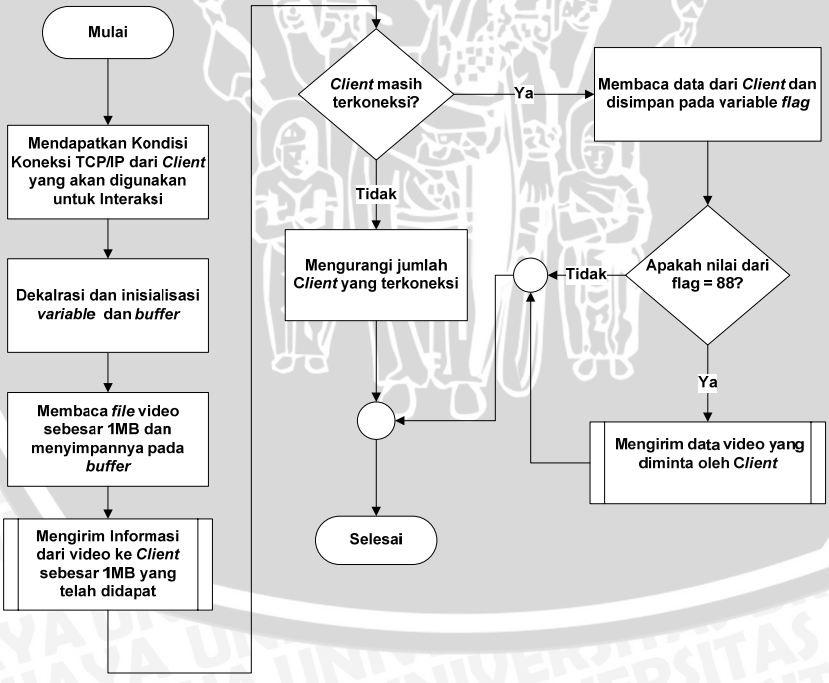
1. Mendeklarasikan variabel yaitu variabel yang diberi nama clientThread yang akan dibutuhkan oleh *server* pada proses ini.
2. Pada tahap ini dilakukan proses perulangan terus menerus untuk menunggu *request* koneksi dari *client*. Jika ada *client* yang terkoneksi, maka akan lanjut

pada proses berikutnya. Jika tidak bisa terhubung, maka menghentikan clientThread atau menggabungkan clientThread dengan *thread* utama.

3. Jika proses koneksi dengan *client* bisa terhubung, maka akan melakukan penambahan jumlah *client* yang telah terkoneksi.
4. Setelah membuat *thread* baru untuk menangani *client* tersebut, maka kembali melakukan *monitoring* atau pengawasan apakah ada *client* yang ingin berkoneksi atau tidak.

4.2.1.2 Perancangan Penanganan *Client* dengan Sebuah *Thread*

Pada subbab ini akan dijelaskan proses *handling* atau penanganan *client*. Penanganan yang dimaksud pada hal ini adalah dengan sebuah *thread*, *server* dapat menerima dan mengirim informasi (parameter) dari dan pada *client* serta *server* juga dapat mengirim data video pada *client.* Diagram alir untuk perancangan proses *handling* atau penanganan *client* secara umum ditunjukkan pada Gambar 4.9



Gambar 4.9 Diagram alir penanganan *client* dengan sebuah *Thread*

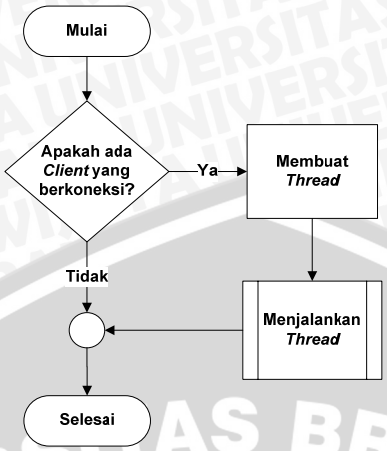
Berikut penjelasan dari diagram alir dari Gambar 4.9:

1. Mendapatkan kondisi koneksi TCP/IP dari *client* yang akan digunakan untuk interaksi.
2. Mendeklarasikan dan menginisialisasi *variable* dan *buffer* yang akan dibutuhkan pada proses ini.
3. Membaca *file* video sebesar 1MB dan menyimpannya pada *buffer*.
4. Mengirim informasi dari video ke *client* sebesar 1MB yang telah didapat.
5. Pada tahap ini dilakukan proses perulangan terus-menerus untuk menungguapakah *client* masih terkoneksi apa tidak. Jika *client* masih terkoneksi, makaakan lanjut pada proses berikutnya. Jika *client* tidak terhubung (*disconnect*), maka akan mengurangi jumlah *client* yang terkoneksi.
6. Jika proses koneksi dengan *client* masih terhubung, maka akan membaca data dari *client* dan disimpan pada variabel *flag*.
7. Kemudian mengecek lagi apakah nilai dari *flag* adalah 88. Jika *flag* bernilai 88, maka akan mengirim data video yang diminta oleh c*lient*. Sedangkan jika bernilai *flag* bukan 88, maka tidak dapat mengirim data video yang diminta *client* dan menunggu *client* tidak terhubung (*disconnect*).

4.2.1.2.1 Perancangan Penanganan Banyak *Client* dengan Multi-*Thread*

Pada perancangan tahap ini akan dijelaskan alur atau proses dari penanganan tiap-tiap *client* oleh *server*. Pada dasarnya multi-*Thread* terdiri dari *thread -thread* yang mempunyai suatu tugas berupa proses yang harus dijalankan secara bersamaan. Dalam cakupan ini, proses tersebut adalah menangani sebuah *client* dan seluruh interaksi prosesnya. Karena satu *thread* dirancang untuk menangani satu *client*, maka untuk dapat menangani banyak *client* dengan cara *thread* tersebut diperbanyak sehingga tercipta *thread-thread* yang identik sehingga dapat menangani banyak *client*. Pada implementasinya untuk memperbanyak *thread* akan menggunakan *array* atau larik sehingga akan terbentuk *array of thread*.

Hasil dari proses pembuatan *array of thread* maka terbentuk multi-*thread* seperti pada diagram alir yang ditunjukkan Gambar 4.10



Gambar 4.10 Diagram alir multi-*thread* pada penanganan banyak *client*

Berikut penjelasan diagram alir Gambar 4.10 :

1. Setiap ada *client* yang ingin berkoneksi maka akan dibuatkan *thread* baru untuk menangani *client* tersebut, dimana tugas dari *thread* tersebut sudah dijelaskan pada subbab 4.2.1.1.1
2. Menjalankan proses *thread* tersebut sesuai dengan proses yang telah dideklarasikan sebelumnya.

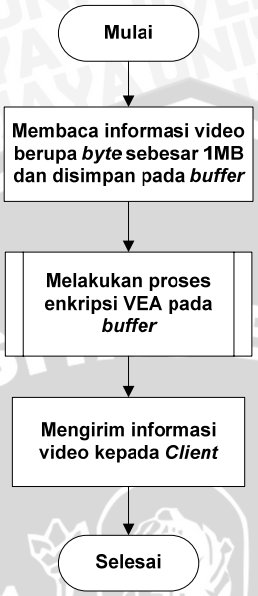
4.2.1.3 Perancangan Proses Interaksi dengan *Client*

Pada subbab ini akan dijelaskan mengenai proses interaksi antara *server* dengan *client*. Interaksi yang dimaksud pada hal ini adalah *server* dan *client* dapat menerima dan mengirim informasi (parameter) maupun data video.

4.2.1.3.1 Perancangan Mengirimkan Informasi pada *Client*

Pada sisi *client* membutuhkan informasi dari video berupa lebar, tinggi, tipe format video, video *timebase* untuk menyiapkan *player* dan memainkan video tersebut. Informasi tersebut sebesar 1MB *byte* yang diambil dari *file* video tersebut. Kemudian informasi tersebut dienkripsi menggunakan algoritma VEA dan dikirim pada *client*. Setelah informasi sebesar 1MB tersebut diterima oleh *client*, maka informasi tersebut akan diambil dan diolah menjadi informasi yang dibutuhkan oleh *display* dan *play* untuk memainkan video tersebut.

Diagram alir untuk perancangan pengiriman informasi pada *client* ditunjukkan pada Gambar 4.11



Gambar 4.11 Diagram alir perancangan pengiriman informasi pada *client*

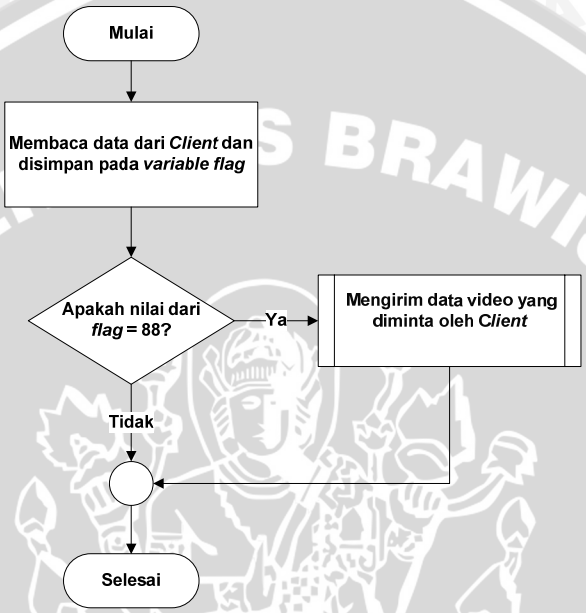
Berikut penjelasan diagram alir Gambar 4.11 :

1. Membaca informasi video berupa *byte* sebesar 1MB dan kemudian disimpan dalam *buffer*.
2. Melakukan enkripsi informasi video yang tersimpan dalam *buffer* tersebut dengan menggunakan algoritma VEA.
3. Kemudian mengirim informasi video sebesar 1MB yang telah dienkripsi pada *client*.

4.2.1.3.2 Perancangan Penerimaan Permintaan Video dan Pengiriman Data *Streaming*

Pada subbab ini akan dijelaskan mengenai proses penerimaan permintaan video oleh *client* pada *server*. Kemudian *server* mengirimkan data *streaming* video pada *client*. Alasan dilakukan proses penerimaan permintaan video dari *client* dimaksudkan agar mengetahui bahwa *client* sudah siap dalam menyiapkan *player* sesuai dengan informasi yang didapat sebelumnya yang sudah dijelaskan

pada subbab 4.2.1.3.1 dan siap untuk melakukan proses *streaming*. Sehingga untuk dapat mengetahui kesiapan dari *client* tersebut maka harus dibuat sebuah *flag* yang dikirim oleh *client* yang digunakan sebagai sebuah penanda bahwa *client* sudah siap. Diagram alir untuk perancangan penerimaan permintaan video dari *client* ditunjukkan pada Gambar 4.12

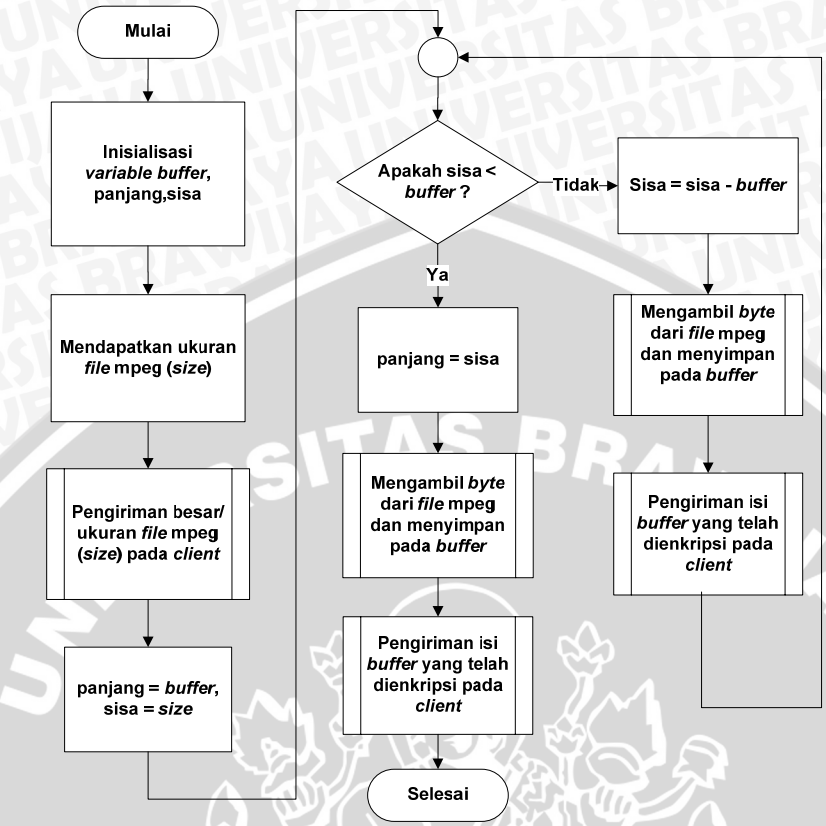


Gambar 4.12 Diagram alir penerimaan permintaan video dari *client*

Berikut penjelasan diagram alir Gambar 4.12 :

1. Membaca data dari *client* dan disimpan padavariabel *flag* untuk penanda.
2. Mengecek apakah nilai dari *flag* adalah 88. Jika iya, maka *server* akan mengirim data video yang diminta oleh *client*, dimana data video tersebut sebesar 1MB yang telah dienkripsi dengan algoritma VEA sebelum dikirim. Jika tidak, dimana *flag* bernilai *null* atau tidak bernilai 88, maka akan *disconnect* atau selesai.

Setelah *server* menerima sebuah *flag* berupa penanda yang dikirim oleh *client* maka *server*akan mengirim data *streaming* video tersebut pada *client.* Diagram alir untuk pengiriman data *streaming* video pada *client* yang ditunjukkan pada Gambar 4.13



Gambar 4.13 Diagram alir pengiriman data *streaming* pada *client*

Berikut penjelasan diagram alir Gambar 4.13 :

1. Pendeklarasian *variable buffer* untuk menyimpan *byte* dari *file* video, variabel panjang digunakan untuk parameter seberapa panjang *byte* yang harus dibaca dan disimpan ke dalam variable *buffer*. Sedangkan variabel sisa digunakan untuk menyimpan hasil dari nilai pengurangan ukuran *file* video total dikurangi oleh *buffer* (1MB).
2. Mendapatkan ukuran (*size*) dari *file* video yang sudah dikurangi *buffer* sebesar 1MB.
3. Mengirimkan informasi berupa besar atau ukuran (*size*) *file* video pada *client* supaya mengetahui ukuran *file* video yang akan diterima.
4. Pemberian nilai variabel panjang sebesar *buffer* dan sisa sebesar *size*.
5. Dilakukan pengecekan apakah sisa lebih kecil dari *buffer*. Jika iya, maka *end of file* (EOF) telah tercapai dan akan menjalankan proses pemberian nilai pada

variabel panjang sebesar sisa. Kemudian mengambil *byte* sebesar panjang menggunakan operasi *file C#* dari *class* FileStream, dimana nama objeknya adalah fileReader. Kemudian *byte-byte* tersebut dienkripsi dengan algoritma VEA dan disimpan pada *buffer*. Langkah berikutnya adalah mengirim isi dari *buffer* yang telah terenkripsi pada *client*.

6. Jika hasil pengecekan tidak, maka akan dilakukan proses pengurangan variabel sisa dikurangi *buffer*. Hal ini bertujuan untuk memberitahukan seberapa besar sisa ukuran *file* video yang harus dibaca dan dikirim. Kemudian mengambil *byte* sebesar panjang menggunakan operasi *file C#* dari *class* FileStream, dimana nama objeknya adalah fileReader. Kemudian *byte-byte* tersebut dienkripsi dengan algoritma VEA dan disimpan pada *buffer*. Langkah berikutnya adalah mengirim isi dari *buffer* yang telah terenkripsi pada *client.*

4.2.1.4 Perancangan Mengenkripsi Data *Frame* I

Pada perancangan ini dijelaskan mengenai enkripsi video mpeg menggunakan algortima VEA. Berdasarkan teori dari algoritma VEA bahwa melakukan enkripsi video mpeg hanya pada data *frame* I. Sehingga langkah awal yang harus dilakukan adalah mencari dimana letak setiap data *frame* I yang terdapat video mpeg kemudian melakukan enkripsi pada setiap *byte*-nya dengan menggunakan kunci yang telah didapat.

Sesuai standart internasional atau ISO bahwa *file* format video mpeg-1 dan video mpeg-2 mempunyai susunan struktur hierarki video, seperti pada subab 2.1.2 dan pada Gambar 2.1 terlihat bahwa susunan *header* mpeg terbagi menjadi *sequence, gop, picture, slice,* dan *macroblock*. Untuk membedakan dan mengetahui *layer header* tersebut terdapat penanda atau pengenal yang disebut dengan *start code header* mpeg.

Start code terdiri dari 4 *byte* penanda, dimana nilai pada 3 *byte* awal adalah *fix* yakni 00 00 01 kemudian pada *byte* terakhir nilai sesuai dengan id dari *header* tersebut. Misalnya untuk *start code* pada *sequence header* adalah 00 00 01 B3. Jadi id *header* untuk *sequence* adalah B3, untuk *start code* dari *picture header* adalah 00 00 01 00. Dan untuk id *header* yang selengkapnya terdapat pada Tabel 2.2 tentang *start code table* yang sudah dijelaskan sebelumnya pada bab I.

Karena tipe *frame* itu terdapat pada *layer* atau daerah *picture*, maka untuk mendapatkan data dari *layer picture* harus mengetahui dimana letak dari *layer picture* tersebut, dengan cara mencari setiap *start code* dari *picture header*. *Start code* dari *picture header* adalah 00 00 01 00 sehingga sebelum melakukan enkripsi pada data *layer picture* maka dilakukan pencarian *layer picture header* terlebih dahulu.

Terdapat 4 jenis tipe *frame* pada mpeg-1 dan mpeg-2 yakni tipe *frame* I, P, B, D. Untuk mendapatkan *frame* I dari 4 banyak tipe *frame*, maka harus mengetahui struktur dari *picture header* tersebut karena tipe *frame* tersebut berada didalam *picture header*. Berikut ini tabel *picture header*.

byte 4								byte 5								byte 6								byte 7							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
temperal sequence number								frame type 1=I, 2=P 3=B, 4=D								VBV delay								---							

Tabel 4.1 Struktur *Picture Header*

(**Sumber** : http://dvd.sourceforge.net/dvdinfo/mpeghdrs.html)

Dari Table 4.1 didapat bahwa jumlah *picture header* adalah 4 *byte* dan jika ditambahkan dengan *start code picture header* menjadi 8 *byte*. Sehingga untuk melakukan pencarian *start code* beserta *picture header* dibutuhkan 8 *byte buffer* untuk menyimpan *byte-byte* tersebut dimana pada perancangan ini diberi nama bufferFlag. Nilai bufferFlag dengan cara mengambil *byte* per-*byte* dari *buffer* utama yang menyimpan *byte-byte* dari *file* video mpeg kemudian dilakukan pengecekan nilai untuk untuk menemukan *start code* dari *picture header*.

Setelah melakukan pengecekan dan menemukan *start code* untuk *picture header* hal yang dilkukan berikutnya adalah melakukan pengecekan untuk mengetahui *frame* atau *picture* data tersebut bertipe apa. Untuk mengetahui tipe dari *frame* tersebut dengan cara melakukan perhitungan sederhana yakni mengambil nilai dari 3 *bit* pada *byte* ke 5 di index 3, 4, 5 pada *picture header*.

Untuk mendapatkan 3 *bit* sebagai berikut :

1. Melakukan operasi AND pada nilai *byte* ke -5 dengan nilai heksa 38.

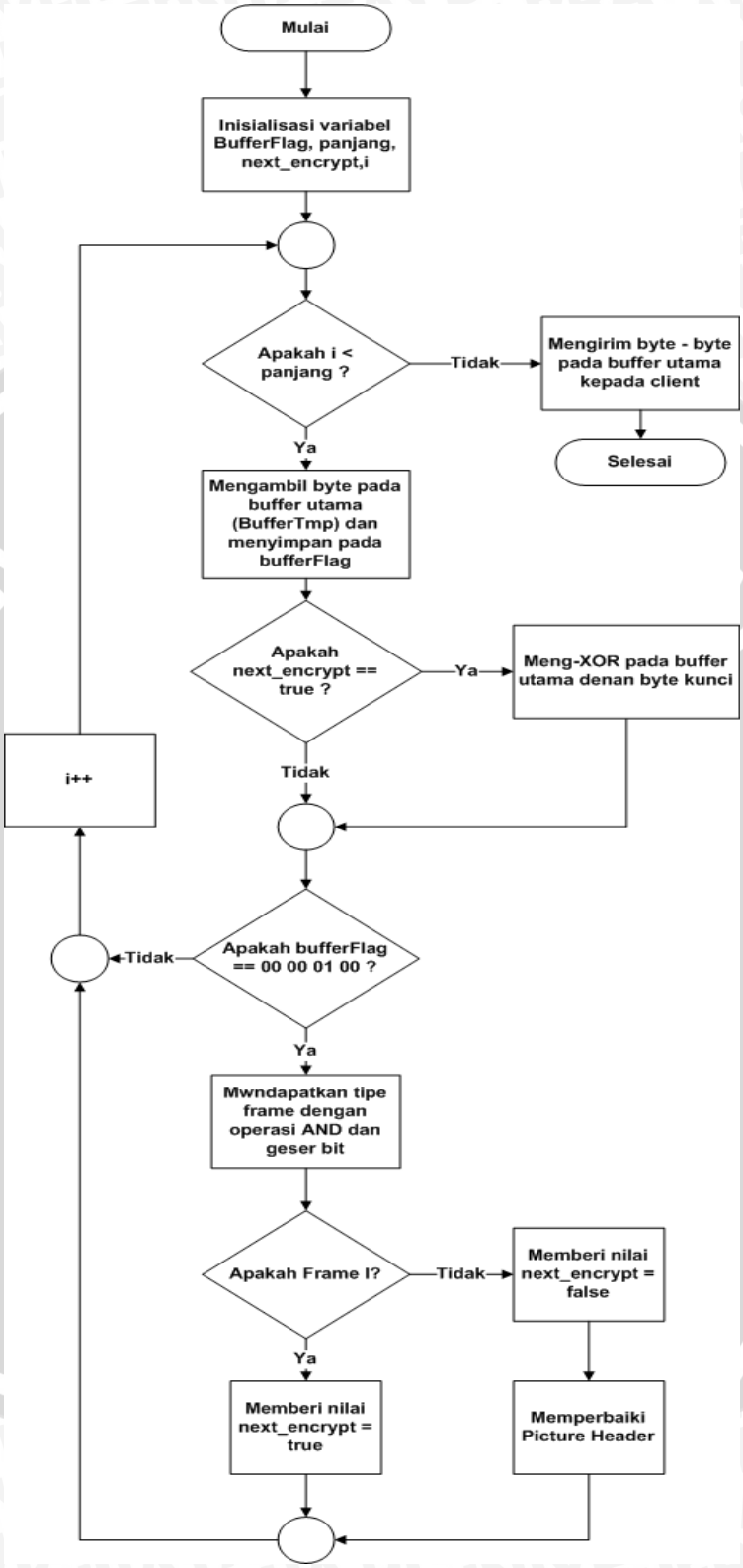
2. Melakukan operasi geser *bit* ke arah kanan sebanyak 3.
3. Melakukan pengecekan untuk mendapatkan tipe *frame* dari hasil perhitungan. Dari Table 4.1 *picture header* pada *byte* ke -5 dapat diketahui bahwa :
   i. Jika nilainya 1 berarti tipe *frame* I
   ii. Jika nilainya 2 berarti tipe *frame* P
   iii. Jika nilainya 3 berarti tipe *frame* B
   iv. Jika nilainya 4 berarti tipe *frame* D

Berikut contoh untuk perhitungannya, misalnya pada kondisi *start code picture header* yang ditemukan sekarang dengan pada *byte* ke-5 mempunyai nilai 0F heksa atau dalam 15 desimal.

0F pada bilangan biner adalah	0000 1111
38 pada bilangan biner adalah	0011 1000
dilakukan operasi AND menjadi	0000 1000
dilakukan geser *bit* ke kanan 3	0000 0001

Setelah mendapatkan nilai akhir dari perhitungan, maka dilakukan pengecekan. Jika nilai akhirnya didapat nilai 1 (nilai dalam desimal), maka *frame* yang ditemukan sekarang adalah bertipe I.

Proses berikut setelah menemukan bahwa *picture header* bertipe *frame* I adalah melakukan proses enkripsi pada *picture* data tersebut dan berhenti melakukan proses enkripsi saat menemukan *start code picture header* berikutnya. Sesuai teori dari algoritma VEA bahwa melakukan enkripsi pada *picture* data dengan cara meng-XOR-kan pada *byte-byte* yang ada *picture* data yang bertipe *frame* I tersebut dengan *byte-byte* kunci. Berikut diagram alir untuk enkripsi yang ditunjukkan pada Gambar 4.14



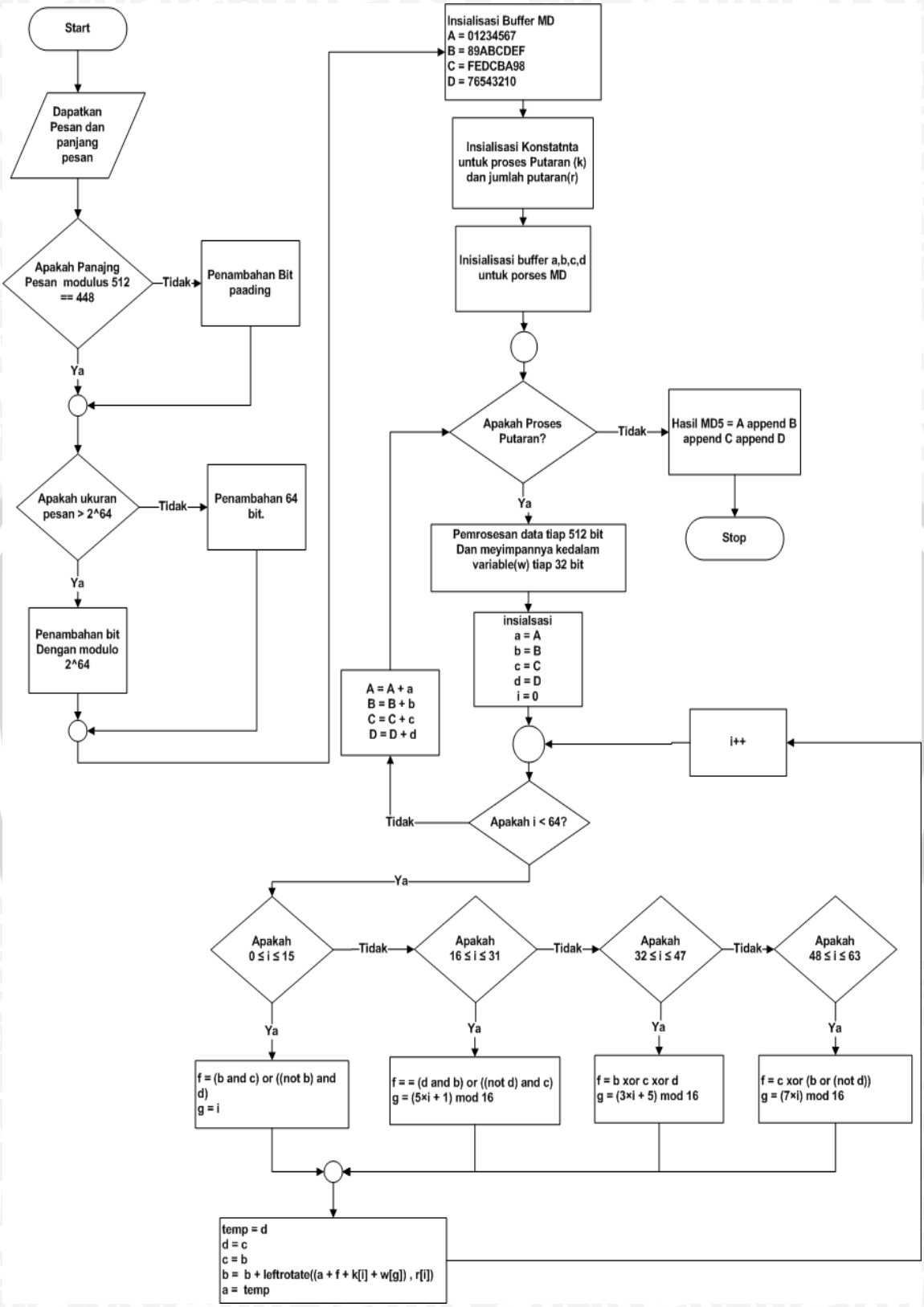
Gambar 4.14 Diagram alir enkripsi VEA pada *frame* I

Penjelasan diagram alir pada Gambar 4.14 :

1. Mendeklarasikan variabel yang akan digunakan seperti bufferFlag,panjang, next_encrypt, i.
2. Melakukan pengecekan apakah i lebih kecil dari panjang atau besar *buffer*, jika tidak berarti semua *byte* pada *buffer* sudah terenkripsi dan mengirimkan *byte-byte* pada *buffer* pada *client*.
3. Jika iya, maka mengambil nilai dari *buffer* utama kemudian dimasukan pada bufferFlag.
4. Dilakukan pengecekan apakah kondisi next_encrypt adalah *true*, jika iya *byte-byte* pada *buffer* utama akan di-XOR-kan dengan *byte-byte* dari hasil MD5.
5. Jika tidak dilakukan pengecekan pada bufferFlag apakah nilainya 00 00 01 00, jika tidak kembali ke poin 2 dan menambah index i.
6. Jika iya, maka melakukan perhitungan untuk mendapatkan tipe *frame*.
7. Melakukan pengecekan apakah tipe *frame* adalah I. Jika iya memberi nilai pada next_encrypt adalah *true* dan kembali pada poin 2 dan menambah index i.
8. Jika tidak, memberi nilai pada next_encrypt adalah *false*.
9. Memperbaiki *picture header* dan kembali ke poin 2 juga menambah index i.

4.2.1.5 Perancangan Fungsi Umum MD5

Perancangan MD5 pada subbab ini berdasarkan pada dasar teori, dimana juga mengacu pada *standart* internasional yang sudah ditetapkan yakni rfc132. Hasil dari MD5 ini berupa *byte-byt*e yang akan digunakan sebagai kunci dalam proses enkripsi dan dekripsi. *Byte-byte* dari kunci tersebut di-XOR-kan dengan *byte-byte* pada *frame* I pada *file* video mpeg. Berikut diagram alir untuk MD5 yang ditunjukkan pada Gambar 4.15



Gambar 4.15 Diagram alir fungsi MD5

Berikut penjelasan diagram alir Gambar 4.15 :

1. Mendapatkan pesan dan panjang pesan yang akan diolah.
2. Mengecek apakah panjang pesan tersebut jika dimoduluskan dengan 512 nilainya adalah 448. Jika tidak, maka akan dilakukan penambahan *bit padding* agar sisa hasil baginya (modulus) adalah 448. Karena yang akan diproses sebesar 512 *bit* atau kelipatannya (L*512). Jika iya, maka akan melakukan proses selanjutnya.

   Misal panjang pesan 564, maka akan ditambah *bit padding* sebanyak 396 *bit*. Karena (564 + 396) mod 512 = 448.
3. Mengecek apakah panjang pesan > 2^64. Jika tidak, maka tinggal menambahkan 64 *bit* dengan nilainya sesuai dengan besar pesan. Jika iya, maka nilai dari 64 *bit* tersebut adalah hasil dari besarnya pesan dimoduluskan dengan 2^64.

   Misal besarnya (2^65), maka besar nilai dari 64 *bit* yang akan ditambahkan tersebut adalah 32. Karena (2^65+32) mod 2^64 = 32.

   Setelah tahap ini panjang dari input MD sudah 512 atau kelipatnya. Misal 960+64 = 1024.
4. MD5 membutuhkan 4 buah penyangga (*buffer*) yang masing-masing panjangnya 32 *bit*. Total panjang penyangga adalah 4 * 32 = 128 *bit*. Keempat penyangga ini menampung hasil proses dan hasil akhir. Dengan nilai awal yakni A = 01234567, B = 89ABCDEF, C = FEDCBA98, D = 76543210.
5. MD5 juga membutuhkan konstanta yang akan digunakan dalam proses putaran sebagai bilangan penambah, dimana nilai ini didapat dari persamaan T[i] = 2^32 * abs(sin(i)), dimana i adalah iterasi 0 sampai 63.

   Selain konstanta pengali juga dibutukan jumlah putaran tiap-tiap iterasi (r).
6. Inisialisasi *buffer* untuk proses MD yakni a, b, c, d.
7. Mulai melakukan proses pengolahkan pesan, dimana tiap operasi diambil 512 *bit*. Dimana nilai 512 *bit* tersebut dsimpan dalam *vaiable* w tiap 32 *bit* dalam bentuk *array* (w[]).
8. Pemberian nilai dari *buffer* yakni a = A , b = B , c =C , d = D.
9. Melakukan iterasi atau perulangan dari 0 – 63, dan disini mulai melakukan putaran-putaran sesuai operasi dasar MD5.

10. Jika i masih dalam range 0 – 15 (atau disebut putaran dasar pertama), maka akan melakukan operasi :
    f = (b and c) or ((not b) and d)
    g= i
11. Jika i masih dalam range 16 – 31 (atau disebut putaran dasar kedua), maka akan melakukan operasi :
    f = (d and b) or ((not d) and c)
    g= (5*I +1) mod 16
12. Jika i masih dalam range 32 – 47 (atau disebut putaran dasar ketiga), maka akan melakukan operasi :
    f = (b xor c) xor d
    g= (3*I + 5) mod 16
13. Jika i masih dalam range 48 – 63 (atau disebut putaran dasar keempat), maka akan melakukan operasi :
    f = c xor (b or (not d))
    g= (7*1) mod 16
14. Setelah melalui proses diatas dilakukan proses pertukaran nilai dan proses pengubah nilai sebagai berikut:
    temp = d
    d = c
    c = b
    b = b + leftrotate((a + f + k[i] + w[g]) , r[i])
    a = temp
    dimana fungsi leftrotate didefinisikan sebagai berikut :
    leftrotate (x, c)
    {
    return (x << c) or (x >> (32-c));
    }
15. Proses diatas dilakukan selama proses iterasi. Jika iterasi sudah, maka nilai yang berada pada *buffer* dipindah ke *buffer* utama yakni :
    A = A + a
    B = B + b

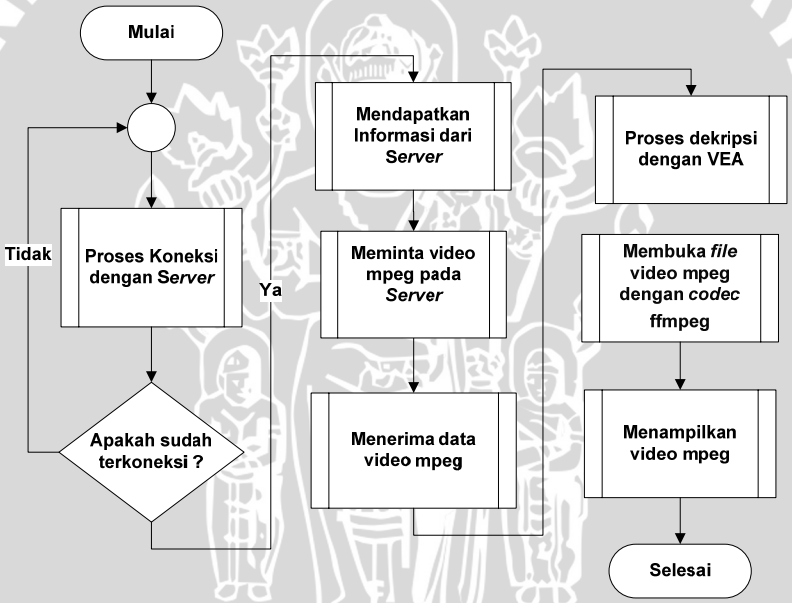
C = C + c

D = D + d

16. Jika masih ada 512 *bit* berikutnya, maka proses diatas diulangi. Jika tidak, maka hasil dari proses tersebut akan digabungkan atau *append* (+).
    Hasil_MD5 = A + B + C + D, sehingga terbentuk hasil dari proses MD.

4.2.2 Perancangan Sisi *Client*

Aplikasi sisi *client* yang di beri nama "VeaTube Client" merupakan aplikasi yang akan meminta layanan (*servis*) pada *server*. Perancangan sisi *client* harus mempunyai kemampuan seperti yang disebutkan pada subbab 4.2 dan diagram alir untuk perancangan *client* yang ditunjukkan pada Gambar 4.16



Gambar 4.16 Diagram alir perancangan *client*

Berikut penjelasan dari diagram alir dari Gambar 4.16:

1. Membangun koneksi dan memasukkan parameter berupa alamat ip dan *port* dari *server* yang telah ditentukan agar *client* dapat berkoneksi.
2. Pada tahap ini dilakukan proses perulangan terus menerus untuk menunggu apakah *client* sudah terkoneksi dengan *server* apa tidak. Jika sudah terkoneksi, maka *client* akan mendapatkan informasi video sebesar 1MB yang dikirim oleh

server, dimana informasi video sebesar 1MB tersebut seperti lebar, tinggi, video *timebase*, jenis *codec* dan lain sebagainya yang telah dienkripsi sebelumnya oleh *server*. Jika belum terkoneksi, maka *client* akan menunggu sampai terkoneksi dengan *server*.

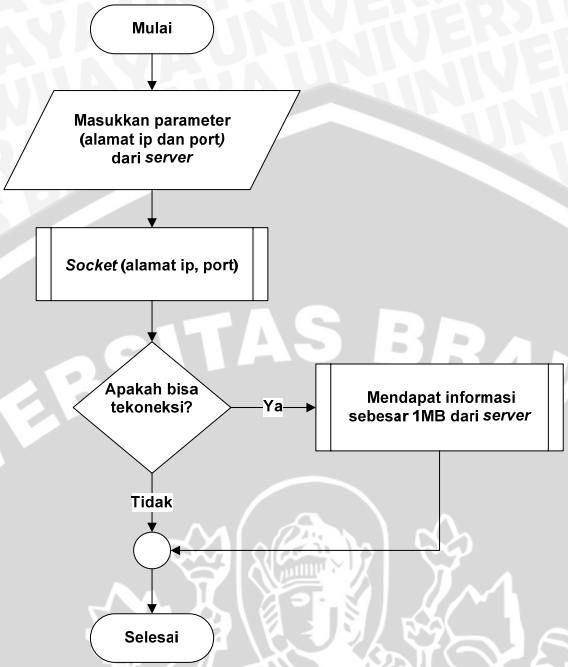
3. Setelah *client* mendapatkan informasi tersebut, kemudian *client* meminta video pada *server* dengan memberikan sebuah variable *flag* dengan nilai 88 sebagai penunjuk atau penanda.
4. Kemudian *client* menerima data video mpeg yang diminta, dimana video tersebut sudah dienkripsi sebelumnya oleh *server*.
5. Ketika *server* mengirimkan *file* video mpeg yang terenkripsi, *client* akan melakukan proses dekripsi *file* video mpeg menggunakan algoritma VEA.
6. Setelah itu *client* membaca atau membuka *file* video mpeg yang telah didekripsi tersebut dengan menggunakan *codec* FFMPEG dan untuk *byte-byte*-nya diolah untuk mendapatkan *frame*, kemudian *frame* tersebut dijadikan gambar dalam bentuk bitmap, kemudian ditampilkan sesuai dengan waktunya sehingga menjadi video.
7. Menampilan video mpeg tersebut menggunakan *player* yang telah dibuat.

4.2.2.1 Perancangan Membangun Koneksi dengan *Server*

Pada perancangan ini dijelaskan mengenai alur atau proses dari *client* membangun koneksi dengan *server*. Karena *client* berkoneksi dengan *server* dengan *protoco*l TCP/IP maka *client* membutuhkan parameter berupa IPaddress dan juga *port* dari *server*. Parameter tersebut dimasukan secara manual oleh *user* karena alamat *server* bisa berubah-ubah atau tidak tetap.

Setelah mendapatkan alamat IP dan *port* dari *user*, selanjutnya melakukan proses koneksi atau *handshaking* dengan *server*. Karena bahasa pemrograman yang digunakan adalah Dotnet C#, maka fungsi-fungsi *socket* sudah tersedia tinggal digunakan dan hanya memasukan parameter IP dan *por*t. Kemudian setelah mencoba melakukan proses koneksi, *client* akan mendapatkan hasil dari percobaan koneksi tersebut. Jika berhasil koneksi, maka *client* akan mendapatkan kiriman *byte* sebesar 1MB yang akan digunakan sebagai informasi video. Jika tidak berhasil, maka *client* akan memberitahu proses koneksi gagal.

Berikut diagram alir dari proses pembangunan koneksi dengan *server* yang ditunjukkan pada Gambar 4.17



Gambar 4.17 Diagram alir membangun koneksi dengan *server*

Berikut penjelasan dari diagram alir dari Gambar 4.17:

1. Memasukkan parameter berupa alamat IP dan *port* dari *server* yang telah ditentukan agar *client* dapat berkoneksi.
2. Mencoba membangun koneksi dengan *socket* dengan parameter IP address yakni alamat IP dari *server* dan *port* tujuan yakni *port* yang terdapat pada *server socket*.
3. Kemudian proses pengecekan apakah *client* sudah terkoneksi dengan *server*. Jika bisa terkoneksi, maka *client* akan mendapatkan informasi video sebesar 1MB yang dikirim oleh *server*. Jika tidak terkoneksi, maka *client* akan menunggu sampai kondisi telah terkoneksi.

4.2.2.2 Perancangan Proses Interaksi dengan *Server*

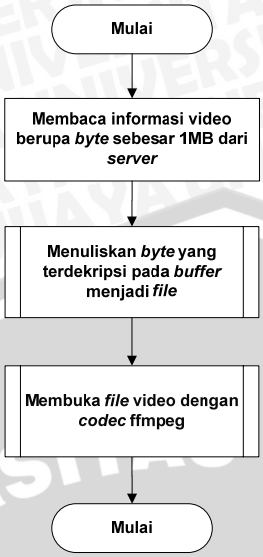
Pada subbab ini akan dijelaskan mengenai proses interaksi antara *client* dengan *client*. Interaksi yang dimaksud pada hal ini adalah *server* dan *client* dapat menerima dan mengirim informasi (parameter) maupun data video.

4.2.2.2.1 Perancangan Menerima Informasi dari *Server*

Pada subbab ini dijelaskan mengenai informasi yang diterima *client* yang dikirim oleh *server*. Setelah *client* berhasil berkoneksi dengan *server*, maka *server* akan mengirimkan *byte* sebesar 1 MB yang diambil dari *file* video yang akan di-*streaming*-kan, dimana *byte-byte* yang dikirim tersebut sudah dienkripsi dengan algoritma VEA. Setelah *client* menerima byte sebsar 1 MB yang dsimpan dalam *buffer*, proses beriktunya yakni mendekripsi *byte* dalam *buffer* tersebut dengan menggunakan algotima VEA. Setelah didekripsi *byte-byte* dalam *buffer* tersebut akan ditulis menjadi *file* yang bernama vea***.tmp.

Karakter "***" dalam nama *file* temporer berisi waktu pada saat kapan *client* menuliskan *file* tersebut. Hal ini dilakukan untuk menghindari kesamaan nama *file* temporer jika terdapat lebih dari satu sesi *streaming*. Karena *file* tersebut temporer atau sementara maka setelah program ditutup *file* temporer tersebut akan dihapus agar tidak menjadi *file* sampah. Kemudian *file* tersebut akan dibaca menggunakan fungsi-fungsi dari FFMPEG untuk mendapatkan informasi yang dibutuhkan seperti tinggi, lebar, format video, video *timebase*.

Berikut diagram alir dari proses penerimaan informasi dari *server* yang ditunjukkan pada Gambar 4.18



Gambar 4.18 Diagram alir penerimaan informasi dari *server*

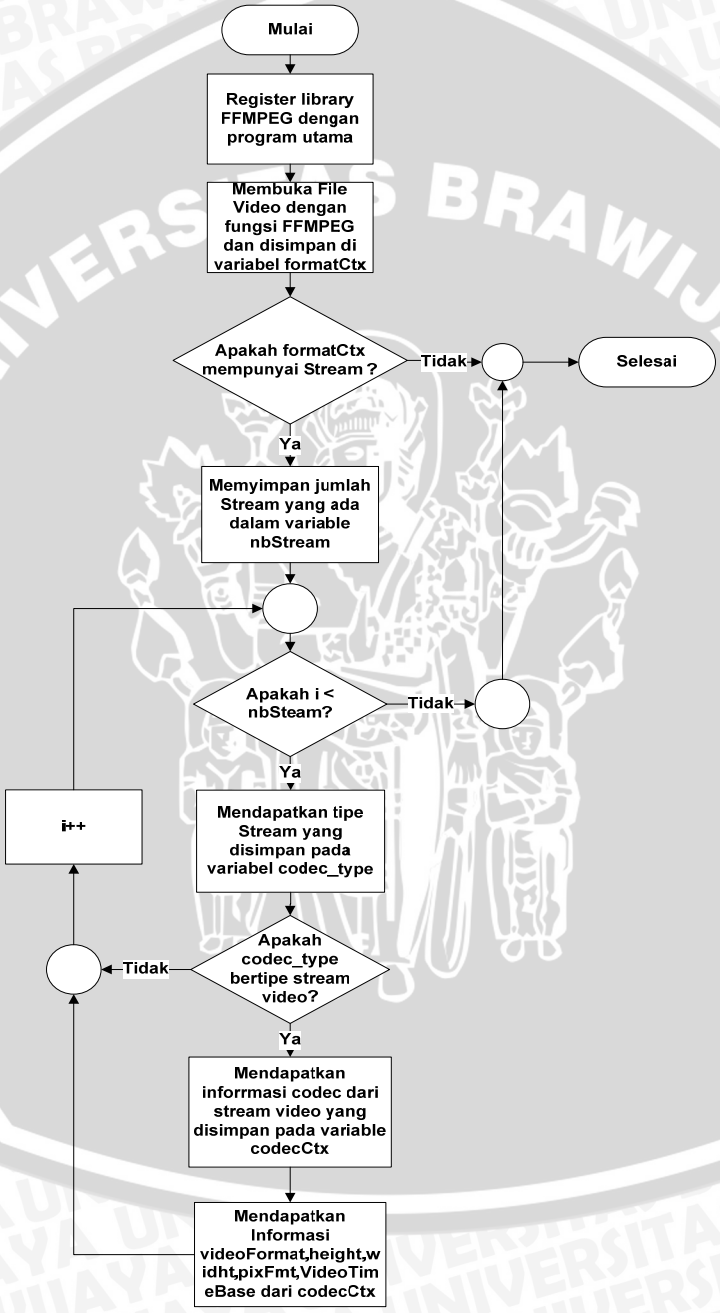
Berikut penjelasan dari diagram alir dari Gambar 4.18:

1. Membaca informasi video berupa *byte* sebesar 1MB dari *server*, dimana informasi video tersebut berupa ukuran video seperti lebar, tinggi, video *timebase*, jenis *codec* yang digunakan, dan lain sebagainya.
2. Menuliskan *byte* yang telah didekripsi yang tersimpan pada *buffer* menjadi *file* untuk dapat diolah.
3. Membuka *file* video dengan menggunakan *codec* FFMPEG utuk mendapatkan informasi yang telah disebutkan pada poin ke-1 yang akan berguna untuk proses menampilkan video.

Pada diagram alir Gambar 4.18 terdapat proses atau fungsi mendapatkan informasi dengan membaca *file* menggunakan *library* FFMPEG, untuk penjelasanya sebagai berikut. Proses pertama adalah meregisterkan *library* FFMPEG dengan program utama supaya semua *codec* dari *library* tersebut bisa digunakan pada program tersebut. Sehingga pada proses inilah pemanggilan *library* dari FFMPEG yang pertama kali yang kemudian hanya mempergunakan fungsi-fungsi dan *codec*-nya saja tanpa harus didefinisikan lagi. Setelah mendapatkan informasi, yang akan proses berikutnya membuatkan *form display*

untuk tempat menampilkan gambar dimana propertinya sesuai dengan informasi yang telah didapat.

Berikut diagram alir untuk membaca *file* video mpeg dengan *library* FFMPEG untuk mendapatkan informasi yang ditunjukkan pada Gambar 4.19



Gambar 4.19 Diagram alir pembacaan *file* video mpeg dengan *library* FFMPEG

Berikut penjelasan diagram alir pada Gambar 4.19 :

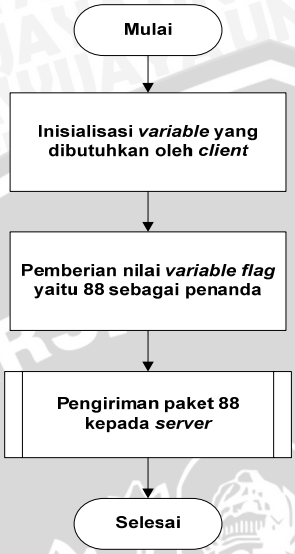
1. Melakukan register *codec* yang ada di *library* FFMPEG pada program utama..
2. Membuka *file* video mpeg dengan dengan fungsi FFMPEG dan menyimpannya pada variabel formatCtx.
3. Melakukan pengecekan apakah pada formatCtx mempunyai *stream*. Jika tidak maka akan memberitahu bahwa *file* tersebut tidak mempunyai *stream*.
4. Jika iya, menyimpan jumlah *stream* yang ada dari *file* tersebut. *Stream* disni bisa berupa video *stream*, audio *stream*.
5. Melakukan pengecekan apakah index i lebih kecil dari nbStream, dimana nbStream merupakan variabel yang menyimpan jumlah *stream*. Jika tidak, maka selesai.
6. Jika iya, maka mendapatkan tipe *stream* dan disimpan pada variabel codec_type.
7. Dilakukan pengecekan lagi apakah codec_type bertipe *stream* video. Jika tidak, maka menambah index i dan kembali ke poin ke-5.
8. Jika iya, maka melakukan proses yang mendapatkan informasi *codec* dari *stream* yang disimpan pada variabel codecCtx.
9. Mendapatkan informasi seperti video *timebase*, *height*, wodht, pixFmt dari codecCtx. Kemudian menambah index i dan kembali pada poin ke-5.

4.2.2.2.2 Perancangan Pengiriman Permintaan Video dan Menerima Data *Streaming*

Setelah terbangun koneksi dengan *server*, *client*akan melakukan proses interaksi dengan *server*. Interaksi tersebut antara lain mengirimkan permintaan data video berupa *flag* dan menerima data *streaming* berupa *byte*, dimana *byte-byte* tersebut akan ditulis dalam *file* yang akan dimainkan menjadi video menggunakan *library* FFMPEG.

Proses yang akan dilakukan setelah berkoneksi dan mendapatkan informasi dari *server* adalah *client* mengirim permintaan data video. Pengiriman permintaan ini berupa *flag* dari *byte* yang bernilai 88. *Client* mengirimkan *flag* tersebut pada *server* digunakan sebagai tanda bahwa *client* telah siap menerima

data *streaming* dari *server*. Berikut diagram alir untuk mengirim permintaan data video pada *server* yang ditunjukkan pada Gambar 4.20



Gambar 4.20 Diagram alir pengiriman permintaan video pada *server*

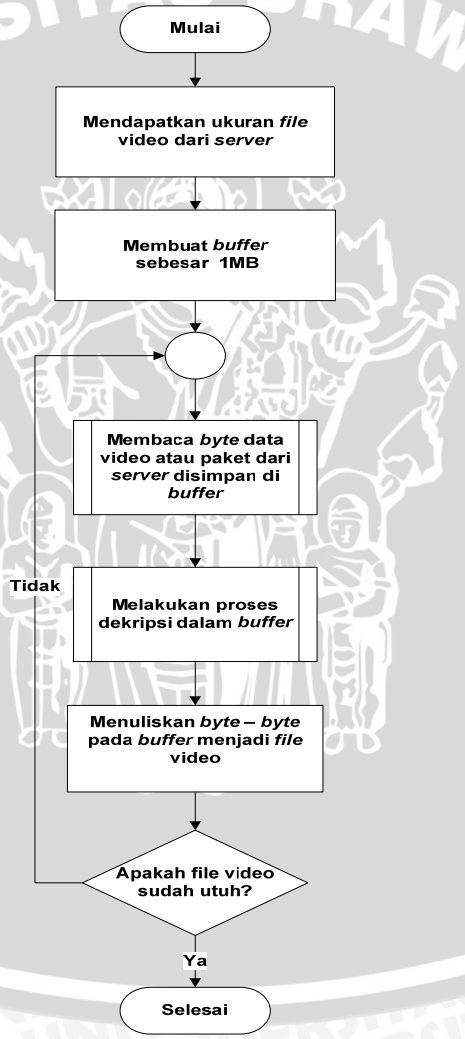
Berikut penjelasan diagram alir Gambar 4.20 :

1. Inisialisasi variabel yang dibutuhkan oleh *client* pada proses ini yaitu seperti *buffer*, *flag*, panjang dan sebagainya.
2. Pemberian nilai variable *flag* yaitu 88 oleh *client* sebagai penanda atau petunjuk untuk meminta paket data.
3. Kemudian *client* melakukan proses pengiriman paket 88 tersebut pada *server* agar *server* dapat mengetahui permintaan paket data video oleh *client.*

Setelah *client* mengirimkan *flag*pada*server*, *server*akan merespon dengan mengirimkan data *streaming* berupa *byte-byte* dari *file* video mpeg, dimana *byte-byte* tersebut sudah dienkrpsi oleh *server* dengan algoritma VEA. Namun sebelum mengirimkan data video tersebut, *server* terlebih dahulu akan mengirimkan ukuran atau *size* dari file video. Hal tersebut berguna agar *client* bisa mengetahui kapan berhenti menerima data *byte* dari *server*.Setiap kali menerima data dari *server*, *byte-byte* tersebut disimpan pada *buffer*, dimana ukuran kapasitas maksimal dalam satu kali penyimpanan adalah 1 MB. Kemudian pada *buffer*

dilakukan proses dekripsi menggunakan algoritma VEA. Setelah dilakukan proses dekripsi *byte-byte* pada *buffer* tersebut, kemudian ditulis menjadi *file* video dan digabungkan dengan *file* sebelumnya yang sudah dibuat pada saat proses menerima infomasi pada subbab 4.2.2.2.1 yakni *file* video vea***.tmp.

Proses tersebut berulang sampai ukuran atau *size* dari *file* video vea***.tmp sama dengan ukuran *file* video asli, dimana ukuran videonya telah diterima *client* sebelum menerima data video. Berikut diagram alir untuk menerima data *streaming* dari *server* yang ditunjukkan pada Gambar 4.21



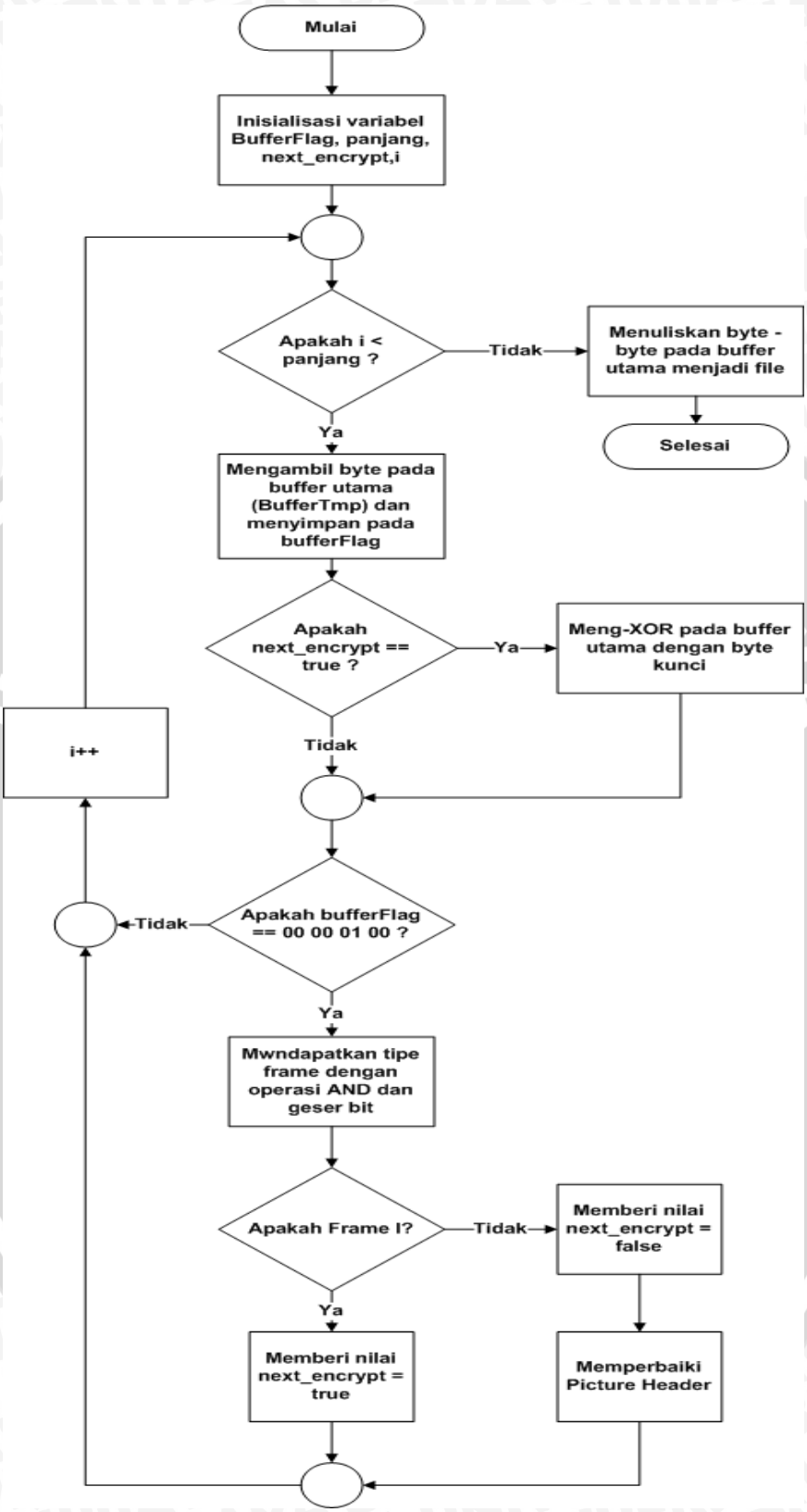
Gambar 4.21 Diagram alir penerimaan data *streaming* dari *server*

Berikut ini penjelasan diagram alir Gambar 4.21 :

1. Mendapatkan ukuran *file* video dari *server*.
2. Membuat *buffer* sebesar 1MB untuk menyimpan *byte-byte* informasi video dan paketdata video yang dikirim oleh *server*.
3. Membaca data video melalui *network stream*kemudian disimpan pada *buffer* yang telah dibuat.
4. Melakukan proses dekripsi dari *byte-byte* data video yang berada di dalam *buffer* tersebut.
5. Menuliskan *byte-byte* data video yang telah didekripsi yang berada di dalam *buffer* menjadi *file* video yang bernama vea***.tmp.
6. Pada tahap ini dilakukan proses perulangan terus menerus untuk mengecek apakah ukuran *file* video sama dengan ukuran video yang asli. Jika sama, maka proses selesai. Jika tidak sama, maka kembali pada proses penerimaan *byte* data video dari *server* pada poin 3.

4.2.2.3 Perancangan Mendekripsi Data *Frame* I

Pada perancangan dekripsi dengan algorima VEA pada *frame* I yang terdapat pada *client* sebenarnya sama persis dengan perancangan enkripsi data frame I yang terdapat *server* pada subbab 4.2.1.4. Perbedaannya terletak pada keluarannya, jika pada *server* setelah dilakukan enkripsi maka *byte-byte* pada *buffer*akan dikirim pada*client* melalui *network stream*. Sedangkan pada *client* setelah dilakukan dekripsi *byte-byte* yang terdapat pada *buffer*akan dituliskan menjadi *file* temporer atau sementara yang bernama “vea***.tmp”. Berikut diagram alir dekripsi VEA pada *frame* I yang ditunjukkan pada Gambar 4.22



Gambar 4.22 Diagram alir untuk mendekripsi data *frame* I

Berikut penjelasan diagram alir Gambar 4.22 :

1. Mendeklarasikan variabel yang akan digunakan seperti bufferFlag, panjang, next_encrypt, i.
2. Melakukan pengecekan apakah i lebih kecil dari panjang atau besar *buffer*. Jika tidak, maka berarti semua *byte* pada *buffer* sudah terdekripsi dan menuliskan *byte-byte* pada *buffer* menjadi *file*.
3. Jika iya, maka mengambil nilai dari *buffer* utama kemudian dimasukan pada bufferFlag.
4. Dilakukan pengecekan apakah kondisi next_encrypt adalah *true*. Jika iya, maka *byte-byte* pada *buffer* utama akan di-XOR-kan dengan *byte-byte* dari hasil MD5.
5. Jika tidak, maka dilakukan pengecekan pada bufferFlag apakah nilainya 00 00 01 00. Jika tidak, maka kembali ke poin 2 dan menambah index i.
6. Jika iya, maka melakukan perhitungan untuk mendapatkan tipe *frame*.
7. Melakukan pengecekan apakah tipe *frame* adalah I. Jika iya, maka memberi nilai pada next_encrypt adalah *true* dan kembali pada poin 2 dan menambah index i.
8. Jika tidak, maka memberi nilai pada next_encrypt adalah *false*.
9. Memperbaiki *picture header* dan kembali ke poin 2 juga menambah index i.

4.2.2.4 Perancangan Penanganan *Update Streaming* dengan Sebuah *Thread*

Pada subbab ini dijelaskan mengenai penanganan proses *update streaming* menggunakan *thread*. Proses ini terjadi saat *user* menekan tombol *play*, dimana setelah tombol *play* ditekan akan membuat *thread*. *Thread* tersebut betugas untuk berinteraksi dengan *server* yang tujuannya untuk meng-*update streaming*. Untuk proses yang dijalankan oleh *thread* ini sudah dijelaskan pada subbab 4.2.2.2 dan berikut untuk diagram alir penanganan *update streaming* dengan sebuah *thread* yang ditunjukkan pada Gambar 4.23

Gambar 4.23 Diagram alir penanganan *update streaming* dengan *thread*

Berikut penjelasan diagram alir Gambar 4.23 :

1. Proses pembuatan *thread* yang berfungsi untuk meng-*updatestreaming* video yang dikirim oleh *server*pada *client*.
2. Kemudian *thread* tersebut menjalankan fungsi updateStreaming(), dimana isi dari fungsi ini sudah dijelaskan pada subbab 4.2.2.2.2

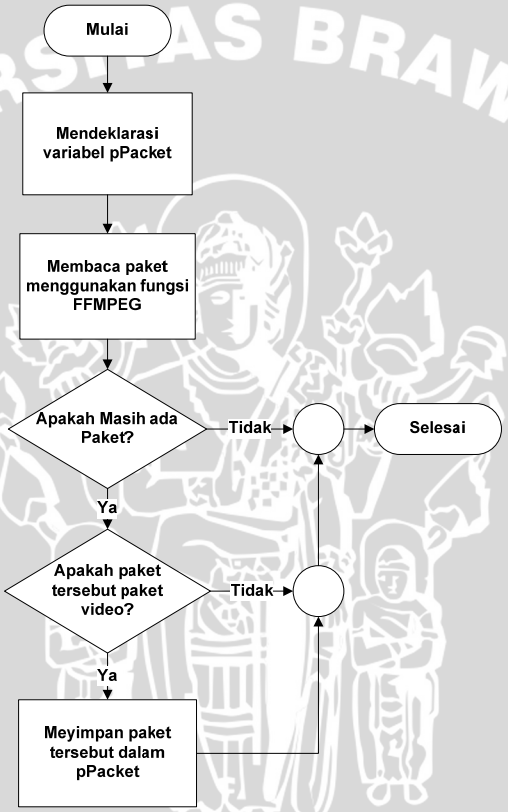
4.2.2.5 Perancangan Penanganan *Update* Video dengan Sebuah *Thread*

Pada perancangan ini sama seperti pada subab 4.2.2.4 bahwa saat *user* menekan tombol *play* akan membuat *thread*, dimana akan bertugas meng-*update* video atau bertugas sebagai *player* video untuk membaca *file* video mpeg menjadi gambar yang ditampilkan berdasarkan waktu sehingga menjadi video yang bisa dinikmati.

Hal terpenting dari video adalah gambar yang akan ditampilkan dan juga waktu yang digunakan sebagai acuan untuk menampilkan gambar tersebut. Pada perancangan ini digunakan fungsi-fungsi dari *library* FFMPEG untuk mendapatkan kedua hal tersebut. Pada *file* video mpeg yang akan dibaca oleh FFMPEG diolah dan disimpan menjadi format data milik FFMPEG, dimana format tersebut pada perancangan ini diberi nama formatCtx. Untuk data video dan data audio disimpan dalam bentuk paket-paket data sehingga untuk

mendapatkan data video baik itu gambar dan waktu, maka harus dilakukan pembacaaan paket-paket dari formatCtx tersebut.

Paket video untuk perancangan ini disebut pPacket, dimana pPacket didapat dengan cara membaca paket-paket melalui formatCtx. Saat mendapatkan sebuah paket akan dilakukan pengecekan apakah paket tersebut milik paket video. Jika iya, maka paket tersebut akan disimpan pada pPacket. Berikut diagram alir untuk pembacaan paket video yang ditunjukkan pada Gambar 4.24



Gambar 4.24 Diagram alir pembacaan paket video

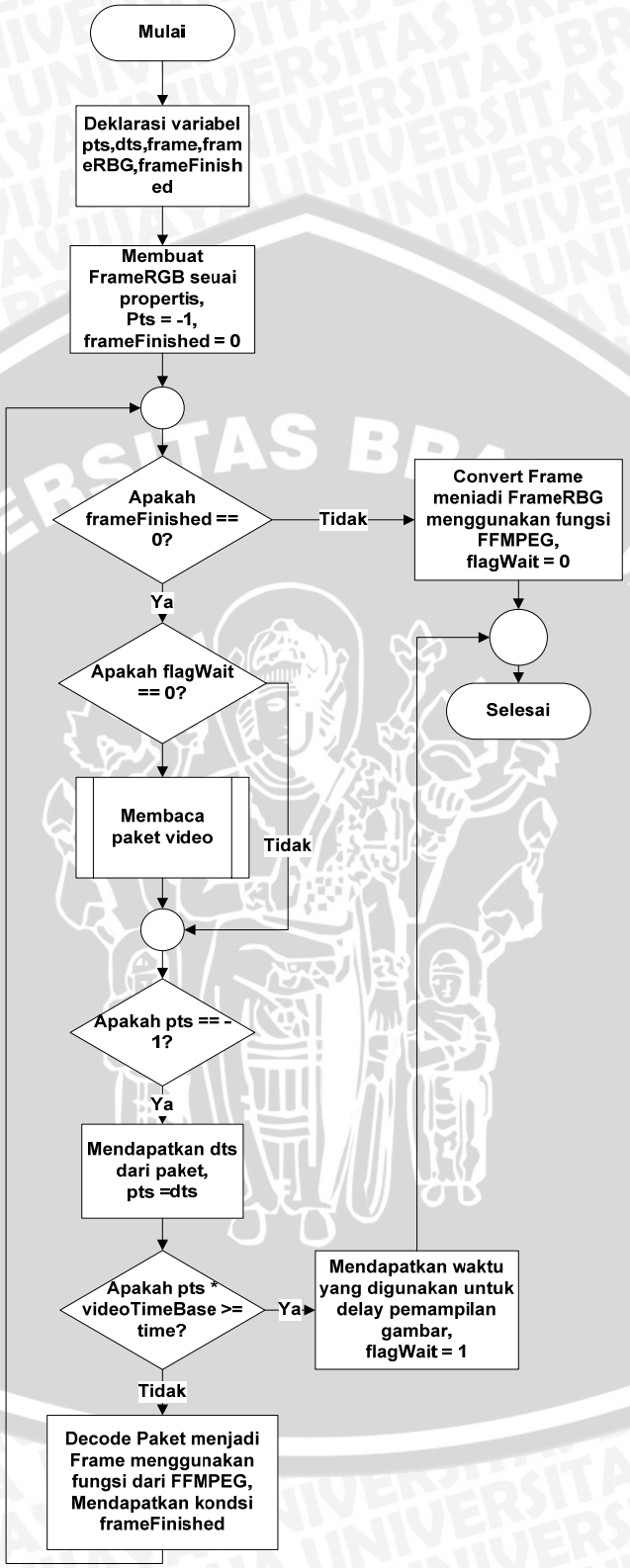
Berikut penjelasan diagram alir Gambar 4.24 :

1. Mendeklarasikan variabel pPaket yang akan digunakan untuk menyimpan paket video dari formatCtx.
2. Membaca paket menggunakan fungsi dari FFMPEG.
3. Melakukan pengecekan apakah ada paket. Jika tidak ada paket, maka selesai.

4. Jika ada paket, maka melakukan pengecekan lagi. Apakah paket tersebut merupakan paket video. Jika tidak, maka selesai.
5. Jika iya, maka meyimpan paket video tersebut pada pada pPacket.

Setelah mendapatkan paket video, proses berikutnya adalah membentuk *frame* yang tersimpan pada tiap-tiap paket. Proses pembentukan *frame* pada FFMPEG dilakukan dengan cara mendapatkan *frame partial* yang tersimpan pada paket video yang kemudian diproses menjadi *frame* yang utuh. Untuk mengubah paket video menjadi *frame* disebut *decoding* video, dimana dari FFMPEG sudah tersedia fungsi untuk men-*decoding* yakni fungsi DecodeVideo(). Saat men-*decoding* paket menjadi *frame* terdapat variabel yang digunakan sebagai penunjuk bahwa *frame* tersebut sudah utuh atau belum, variabel itu disebut frameFinished. Dimana saat nilai dari frameFinsished bernilai 0, maka *frame* tersebut belum utuh. Kemudian saat *frame* tersebut sudah utuh, maka *frame* tersebut bisa digunakan. Namun *frame* tersebut bertipe pixel format YUV sedangkan pada peracangan ini akan digunakan pixel format RBG sehingga harus dilakukan proses perubahan pixel format. Perubahan pixel format tersebut menggunakan fungsi yang terdapat pada FFMPEG yang bernama Convert().

Setelah *frame* sudah siap, proses beriktunya adalah mendapatkan waktu penampilan *frame* tersebut. Waktu penampilan dibedakan menjadi 2 yaitu PTS (*presentation time stamp*) merupakan waktu penampilan *frame* berdasarkan waktu penampilan yang sudah tercatat pada paket *frame* tersebut. Sedangkan DTS (*decoding time stamp*) merupakan waktu penampilan *frame* berdasarkan waktu *decoding* dari *frame* tersebut. Namun pada perancangan ini, nilai pts yang asli tidak dipakai tetapi nilai pts-nya diambil dari nilai dts. Penggunaan dts lebih aman daripada menggunakan pts asli karena pada format video mpeg terdapat *frame* yang bertipe P dan B, dimana *frame* tersebut saat di-*decode* membutuhkan *frame* lain sehingga jika menggunakan pts asli akan tidak sesuai waktunya. Jadi, nilai pts didapat dari dts-nya. Pada proses penampilan *frame*, jika waktu pts lebih cepat daripada waktu *timer*, maka untuk *thread* yang menjalankan proses ini akan tidur sampai waktu penampilan benar. Berikut diagram alir untuk mendapatkan *frame* video yang ditunjukkan pada Gambar 4.25

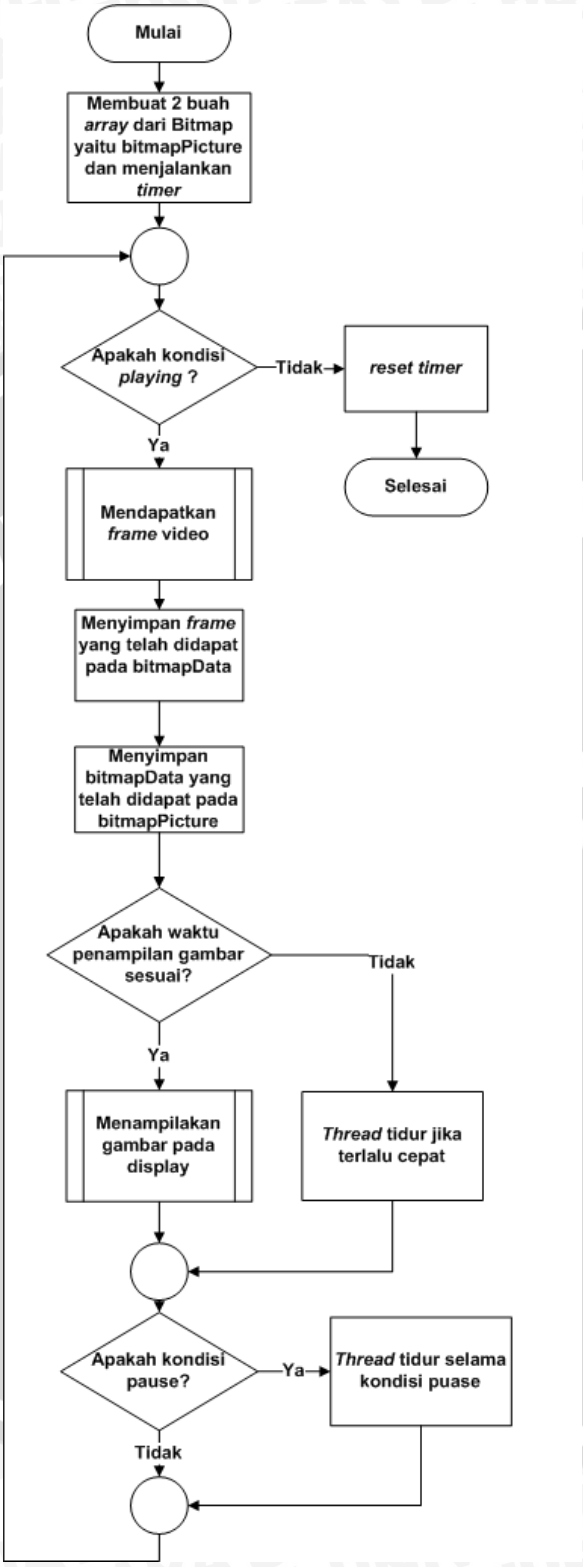


Gambar 4.25 Diagram alir mendapatkan *frame* video

Berikut penjelasan diagram alir Gambar 4.25 :

1. Mendeklarasikan variabel pts, dts, *frame*, frameRGB, frameFinished.
2. Menginisialisasi variabel frameRGB sesuai dengan *properties*, pts = -1, dan frameFinished = 0
3. Melakukan pengecekan apakah nilai dari frameFinish adalah 0. Jika tidak, maka *frame* telah untuh dan *frame* tersebut akan di-*convert* menjadi frameRBG yang menggunakan fungsi dari FFMPEG. Memberi nilai flagWait = 0.
4. Jika iya, maka mengecek apakah flagWait adalah 0. Jika iya, maka membaca dan mendapatkan paket video. Jika tidak, maka menuju ke poin 5.
5. Mendapatkan dts dari paket video dan disimpan pada variabel pts.
6. Mengecek apakah pts * *videoTimeBase* lebih besar dari waktu sekarang. Jika iya, maka akan dilakukan *delay* untuk penampilan gambar yaitu dengan cara *thread* akan tidur. Dan memberi nilai flagWait = 1.
7. Jika tidak, maka men-*decode* paket video yang telah didapat menjadi *frame*, dan mendapatkan kondisi frameFinished.
8. Kembali ke poin 2.

Setelah mendapatkan 2 bagian yang terpenting dari video yakni *frame* dan pts. Proses beriktunya adalah membuat fungsi untuk menampilkan *frame* tersebut pada waktu yang tepat, proses terdapat pada fungsi yang diberi nama videoUpdater(). Pada perancangan ini format gambar yang digunakan adalah bitmap sehingga *frame* yang telah didapat akan disimpan ke dalam format bitmap data yang diberi nama bitmapData, kemudian bitmap data tersebut akan dijadikan gambar yang berformat bitmap yang diberi nama bitmapPicture. Untuk teknik penampilan menggunakan dua buah bitmapPicture yang dibuat dari *array* bitmap. Hal ini bertujuan untuk *dual buffering*, dimana terdapat dua variabel bitmapPicture yang dipakai secara bergantian agar proses penampilan gambar lebih cepat. Berikut diagram alir dari penanganan *update* video yang dijalankan dengan sebuah *thread* yang ditunjukkan pada Gambar 4.26



Gambar 4.26 Diagram alir penanganan *update* video dengan sebuah *thread*

Berikut penjelasan diagram alir Gambar 4.26 :

1. Membuat 2 buah *array* bitmap yang diberi nama bitmapPicture dan menjalankan *timer*.
2. Mengecek apakah kondisi playing. Jika tidak, maka me-*reset timer* dan selesai.
3. Jika iya, maka mendapatkan *frame* video dari hasil olahan FFMPEG.
4. Menyimpan *frame* menjadi format bitmap data yang diberi nama bitmapData.
5. Menyimpan bitmapData menjadi format gambar bitmap dengan nama bitmapPicture.
6. Melakukan pengecekan apakah waktu penampilan gambar sudah sesuai. Jika tidak, maka *thread* akan tidur (*sleep*) apabila waktu telalu cepat.
7. Jika ya, maka menampilkan BitmpatPicture pada FormDisplay.
8. Mengecek apakah kondisi *pause*. Jika iya, maka *thread* akan tidur (*sleep*) selama kondisi *pause*.
9. Jika tidak, maka kembali ke poin 2.

4.3 Implementasi Sistem

Setelah tahap perancangan kemudian tahap selanjutnya adalah tahap implementasi dimana implementasi ini merupakan proses transformasi hasil perancangan perangkat lunak yang telah dibuat ke dalam kode (*coding*) sesuai dengan *syntax* dari bahasa pemrograman yang digunakan.

4.3.1 Lingkungan Implementasi

Aplikasidibuat dengan menggunakan bahasa pemrograman C#.Sistem diimplementasikan dengan menggunakan spesifikasi sebagai berikut:

1. Perangkat Keras (komputer)
   - Komputer *Server* :
     - Jumlah PC = 1
     - *Processor* = Intel Core 2 Duo E7500
     - *Memory* = 1024MB DDR3
     - VGA = VGA intel GMA X4500 *Graphics*
     - *Motherboard* = *mainboard* FOXCONN ETON
     - *Size Harddisk* = 320GB

- Komputer *Client* :
  - Jumlah PC = 3
  - *Processor* = Intel Core 2 Duo E7500
  - *Memory* = 1024MB DDR3
  - VGA = VGA intel GMA X4500 *Graphics*
  - *Motherboard* = *mainboard* FOXCONN ETON
  - *Size Harddisk* = 320GB

2. Perangkat Lunak
   - Komputer *Server* :
     - Sistem Operasi = *Windows* XP *service pack* 3
     - Bahasa Pemrograman = C#
     - *Engine* = *Microsoft Visual Studio* 2008 versi DotNet 3.5
   - Komputer *Client* :
     - Sistem Operasi = *Windows* XP *service pack* 3
     - Bahasa Pemrograman = C#
     - *Engine* = *Microsoft Visual Studio* 2008 versi DotNet 3.5

4.3.2 Implementasi Proses Koneksi

Pada implementasi proses koneksi untuk membuat *server socket* atau TCPListening pada *server* menggunakan *library* Dotnet C# dari *namespace* System.Net.Sockets dan System.Net. untuk menangani fungsi jaringan. Pada *client* untuk membangun koneksi dengan *server* juga menggunakan namespace System.Net.Sockets. Berikut ini merupakan bagian dari *script* C#.

- Untuk *socket* pada *server*

Mencoba membangun *server socket* atau TCP Listening dengan parameter ip *address* dari *server* tersebut dan *port* yang telah dimasukan oleh *user* dimana *default port*-nya adalah 2323. Membuat *server socket* pada *port* 2323,

```
tcpListener = new TcpListener(IPAddress.Any, portAddress);
tcpListener.Start();
```

membuat *buffer* untuk proses input dan output pada *stream* untuk berkomunikasi dengan *client*,

```
NetworkStream clientStream = tcpClient.GetStream();
```

menyimpan nilai variabel jika ada permintaan berkoneksi,

```
TcpClient clients = tcpListener.AcceptTcpClient();
```

- Untuk *socket* pada *client*

Supaya bisa berkomunikasi dengan *server*, maka *client* harus mengetahui alamat ip dan *port* dari *server* sehingga *user* harus memasukan ip dan *port server* secara manual. Mendapatkan ip dan *port* dari *user*,

```
IPServer = IPAddress.Parse(textBoxIPServer.Text);
portServer = int.Parse(textBoxPortServer.Text);
```

setelah mendapatkan ip dan *port* akan dilakukan proses membangun koneksi dengan *server*,

```
serverAddress = new IPEndPoint(IPServer, portServer);
clientConnection = new TcpClient();
clientConnection.Connect(serverAddress);
```

setelah terkoneksi dengan *server* maka membuat *buffer input* dan *output* untuk komunikasi,

```
clientStream = clientConnection.GetStream();
```

4.3.3 Implementasi Penanganan *Request* Koneksi dari *Client* dengan *Thread*

Dalam implementasi *thread* pada C# bisa menggunakan *library* dari *namespace* System.Threading dengan *class* Thread. Pada *thread* ini berisi proses menangani atau *handle* dari *request* koneksi dari *client* seperti pada perancangan. Untuk mendeklarasi *class* Thread sebagai berikut.

```
Thread clientThread = null;
```

kemudian untuk menginisialisasi objeknya dan menjalankan *thread* sebagai berikut,

```
threadListener = new Thread(new ThreadStart(listeningForClient));
threadListener.IsBackground = true;
```

```
threadListener.Start();
```

Dari pseudocode diatas bahwa objek dari *class* Thread yang diberi nama threadListener tersebut akan menjalankan fungsi yang bernama listeningForCLient. Untuk potongan fungsi dari listeningForClient sebagai berikut,

```
private void listeningForClient()//dijalankan oleh threadListener
{
        Thread clientThread = null;

        try
   {
        while (true) //melakukan looping
     {
        TcpClient clients = tcpListener.AcceptTcpClient();

        clientThread  =  new  Thread  (new  ParameterizedThreadStart
        (handlingClient)); //membuat thread baru

        clientThread.Start(clients);
        jumlahClient++; //menambah jumlah client yang berkoneksi

        if (richTextBoxStatus.InvokeRequired) //jika perlu invoke
        {
        richTextBoxStatus.Invoke(newupdateStatusInvoker(updateStatu)
        ,"Client Berhasil berkoneksi, jumlah Client yang terkoneksi
        = " + jumlahClient + Environment.NewLine);  }
        else  //jika tidak perlu invoke
        {
        richTextBoxStatus.Text = "Client Berhasil berkoneksi, jumlah
        Client   yang   terkoneksi   =   "   +   jumlahClient   +
        Environment.NewLine;                                    }
      }
    }
        catch (Exception ex)
        {
        MessageBox.Show(ex.Message);
        if (clientThread != null)
          {

            clientThread.Join();} //menggabungkan clientThread
                                  dengan Thread Utama/induk program
          }
}
```

4.3.4 Implementasi Menangani *Client* dengan *Thread*

Seperti yang sudah dijelaskan pada perancangan pada subbab 4.2.1.2 bahwa penanganan *client* yang sudah terkoneksi akan ditangani oleh sebuah *thread* yang bernama clientThread, clientThread dibuat oleh threadListener disaat ada *client* yang berhasil berkoneksi dengan *server*. Setiap ada *client* yang meminta

koneksi, maka *server* akan menerimanya dan kondisi TCP/IP disimpan pada variabel *client*, dimana variabel *client* tersebut merupakan objek dari *class* TcpClient. Berikut *script* C# untuk penerimaan *client* dan pembuatan objek clientThread dari *class* Thread.

```
TcpClient client = tcpListener.AcceptTcpClient();
```

untuk membuat clientThread setelah ada *client* yang terkoneksi dan menjalankan *thread* tersebut seperti berikut,

```
clientThread = new Thread(new ParameterizedThreadStart
(handlingClient));
clientThread.Start(clients);
```

Dari pseudocode di atas bahwa clientThread menjalankan proses fungsi yang bernama handlingClient. Fungsi tersebut berisi proses untuk menangai *client*, seperti interaksi dengan *client*, mengirimkan data video pada *client*, dan menerima *request* data video dari *client*. Untuk isi fungsi dari handlingClient sebagai berikut.

```
private void handlingClient(object client)//Proses Interaksi
 {
   TcpClient tcpClient = (TcpClient)client;//mendapatkan kondisi
    koneksi dari client
   NetworkStream clientStream = tcpClient.GetStream();
   FileStream fileReader = new FileStream  //membaca File
  (textBoxFilePath.Text,FileMode.Open, FileAccess.Read);
   byte[] bufferTmp = new byte[1048576]; //membuat array byte
   byte[] bufferPattern = new byte[8] { 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xff, 0xff };  //pencari startCode MPEG (global)
   bool flagEncrypt = false;
   byte[] flag = new byte[1]; //sebagai flag apakah client meminta
    paket
   byte[] bufferLong = new byte[8]; //menyimpan nilai sementara
    yang akan dikirim ke client
   int indexHash = 0;  //pengindex hash
   int byteReaded; //penunjuk seberapa besar byte yg diterima
   fileReader.Read(bufferTmp, 0, bufferTmp.Length);
   offset = bufferTmp.Length  //mendapatkan dan menyimpan posisi
    pointer pembaca file

   Stopwatch timerEnkripsi = new Stopwatch();
   timerEnkripsi.Start();
   double timeStart = timerEnkripsi.ElapsedMilliseconds / 1000;
   sentStreaming(clientStream, bufferTmp, fileReader, ref
    bufferPattern, ref flagEncrypt, ref indexHash, ref totalFrame,
    ref totalFrameI);
   double timeEnd = timerEnkripsi.ElapsedMilliseconds / 1000;
   double timeTotal = timeEnd - timeStart;
```

```
while (true)
   {
      try
       {
        //Membaca data berupa informasi yang dikirim oleh client
        byteReaded = clientStream.Read(flag, 0, flag.Length);
          if (byteReaded == 0)  //jika nilai byteReaded adalah 0
                                maka client disconnect dengan server
        {
              jumlahClient--;
            if (richTextBoxStatus.InvokeRequired
              {  richTextBoxStatus.Invoke(new
updateStatusInvoker(updateStatus), "Client telah terDisConnect,
jumlah Client yang terkoneksi = " + jumlahClient + Environment.
NewLine);
              }
            else
              {
                 richTextBoxStatus.Text = "Client telah
terDisConnect, jumlah Client yang terkoneksi = " + jumlahClient +
Environment.NewLine;
              }
                break;
         }
            if (flag[0] == 88)
          {
           //mengirim ukuran file/fileSize
           bufferLong=BitConverter.GetBytes(fileSize);//mengubah
           nilai fileSize menjadi byte-byte

           clientStream.Write(bufferLong,0,bufferLong.Length);//
           mengirim nilai fileSize bufferLong
           timeStart = timerEnkripsi.ElapsedMilliseconds/1000;

           sentStreaming(clientStream, bufferTmp, fileReader,
           ref bufferPattern, ref flagEncrypt, ref indexHash,
           ref totalFrame, ref totalFrameI); //melakukan proses
           streaming
           timeEnd = timerEnkripsi.ElapsedMilliseconds / 1000;
           timeTotal += (timeEnd - timeStart);

             if (veaChecked == 0)
              {
               richTextBoxStatus.Invoke(new
updateStatusInvoker(updateStatus), "Total Time Without VEA = " +
timeTotal + " Second" + Environment.NewLine); }
              else
               {
                richTextBoxStatus.Invoke(new
updateStatusInvoker(updateStatus), "Total Time Encryption = " +
timeTotal + " Second" + Environment.NewLine);
                richTextBoxStatus.Invoke(new
updateStatusInvoker(updateStatus), "Total Frame Mpeg = " +
totalFrame + Environment.NewLine);
                richTextBoxStatus.Invoke(new
updateStatusInvoker(updateStatus), "Total Frame I= " + totalFrameI
+ Environment.NewLine);   }
           }
       }
```

```
            catch (Exception ex)
            {
            MessageBox.Show(ex.Message);
            return;  }
    }
}
```

4.3.5 Implementasi Menangani Banyak *Client* dengan Multi-*Thread*

Multi-*thread* pada kontek ini tidak dibuat secara ekspilisit dengan menggunakan *array of thread* tetapi secara implisit. Implisit dsini maksudnya adalah setiap clientThread yang telah dibuat oleh threadListener jika setiap ada *client* yang meminta koneksi, maka setiap clientThread yang terbentuk secara tidak langsung akan membentuk multi-*thread* dari clientThread. Berikut ini potongan dari *source code*untuk membentuk multi-*thread* dari clientThread.

```
clientThread = new Thread(new ParameterizedThreadStart
(handlingClient));
clientThread.Start(clients);
```

4.3.6 Implementasi Mengirimkan Informasi pada *Client*

Pengiriman informasi merupakan proses yang tidak boleh dilewatkan karena informasi tersebut akan digunakan oleh *client* untuk melakukan proses membuat *player* dari video, dimana informasi itu didapat dari *byte-byte* sebesar 1 MB dari *file* video dan disimpan pada *buffer*. Kemudian pada *buffer* dilakukan enkripsi VEA dan proses selanjutya dikirim pada *client*. Berikut ini merupakan bagian dari *script* C# sesuai dengan perancangan pada subbab 4.2.1.3.1 yang telah dijelaskan sebelumnya.

```
private void sentInformation(NetworkStream clientStream, byte[]
bufferTmp, FileStream FileReader, ref byte[] bufferPattern, ref
bool flagEncrypt, ref int indexHash, ref int totalFrame,ref int
totalFrameI)
      {
       if (veaChecked == 0)
          {
             clientStream.Write(bufferTmp, 0, bufferTmp.Length);
          }
       else
          {
             enkripsi(clientStream, bufferTmp, hash,
             bufferTmp.Length, ref bufferPattern, ref flagEncrypt,
             ref indexHash, ref totalFrame,ref totalFrameI);
          }
      }
```

4.3.7 Implementasi Penerimaan Permintaan Video dan Pengiriman Data *Streaming* pada *Client*

Proses interaksi pada *server* dengan *client* sesuai pada perancangan terdiri dari 2 bagian penting yakni penerimaan permintaan video dan pengiriman data video. Kedua proses tersebut terdapat pada satu fungsi yakni pada fungsi handlingClient yang sudah dijelaskan secara umum pada subbab 4.3.4

Namun berikut implementasi secara khusus pada setiap bagian. Berikut ini potongan program untuk bagian mendapatkan penerimaan permintaan video.

```
byteReaded = clientStream.Read(flag, 0, flag.Length);
```

Penjelasan pseudocode diatas adalah clientThread membaca masukan data dari *client* melalui *network stream* dan menyimpan *byte* yang diterima dalam variabel *flag* dan menyimpan jumlah *byte* yang diterima dalam variabel byteReaded. Setelah mendapatkan input atau data yang dikirm oleh *client*, maka akan dilakukan pengecekan, apakah jumlah *byte* yang diterima tidak nol. Hal ini dilakukan untuk mengetahui bahwa data yang terima tersebut ada nilainya, karena jika jumlah *byte* yang diterima tersebut adalah 0 maka itu berarti *client* telah melakukan putus koneksi atau *disconnect* dengan *server*. Berikut ini merupakan bagian dari *script* pada C#.

```
if (byteReaded == 0)
      {
         jumlahClient--;
         if (richTextBoxStatus.InvokeRequired)
         {
           richTextBoxStatus.Invoke(new
           updateStatusInvoker(updateStatus), "Client telah
           terDisConnect, jumlah Client yang terkoneksi = " +
           jumlahClient + Environment.NewLine);
          }
         else
          {
           richTextBoxStatus.Text = "Client telah terDisConnect,
           jumlah Client yang terkoneksi = " + jumlahClient +
           Environment.NewLine;
          }
         break;
       }
```

Kemudian jika jumlah *byte*-nya tidak sama dengan 0 akan dilakukan pengecekan, apakah isi dari variabel *flag* adalah 88. Jika iya, maka itu berarti

client meminta data *streaming* video dan *server* akan mengirimkan data video yang diminta oleh *client*. Berikut ini merupakan bagian dari *script* pada C#,

```
if (flag[0] == 88)
      {
        bufferLong = BitConverter.GetBytes(fileSize);
        clientStream.Write(bufferLong, 0, bufferLong.Length);
        sentStreaming(clientStream, bufferTmp, fileReader, ref
bufferPattern, ref flagEncrypt,ref indexHash);
      }
```

4.3.8 Implementasi Enkripsi Data *Frame* I

Implementasi Algoritma Enkripsi VEA pada *file* video MPEG sesuai dengan teori dan perancangan bahwa yang akan dilakukan enkripsi hanya pada data *picture frame* I. Enkripsi yang dilakukan adalah setiap pengambilan data *byte* sebesar 1 MB (kecuali sisa pada bagian akhir) yang disimpan pada *buffer*. Pada *buffer* dilakukan pengecekan untuk mencari *frame* I dengan cara mengecek setiap *start code header*, dimana untuk menentukan *start code* digunakan *buffer* yang diberi nama bufferFlag. Kemudian setelah menemukan *frame* I akan dilakukan enkrpsi VEA dengan kunci yang telah didapat dari hasil MD5. Untuk membedakan kondisi saat melakukan enkripsi atau tidak dilakukan pemberian nilai pada variabel *next_encrypt*, jika saat kondisi enkripsi bernilai *true* dan jika saat kondisi tidak mengenkripsi bernilai *false*. Untuk melakukan pengecekan pada bufferFlag dalam mencari *start code* dan pemberian nilai *flag* enkripsi sebagai berikut,

```
if (bufferFlag[0] == 0x00 && bufferFlag[1] == 0x00 &&
bufferFlag[2] == 0x01)
{
      if (bufferFlag[3] == 0x00)
        {
         totalFrame++;   //menambah jumlah frame
         coding_type = ((int)bufferFlag[5] & 0x38) >> 3;

         if (coding_type == 1)// jika frame i
             {
               totalFrameI++;  //menambah jumlah frame I
               next_encrypt = true;
               flagEncrypt = true;
             }
}
```

meng-XOR-kan nilai *byte* pada *buffer* dengan *byte* MD5 sebagai berikut,

```
if (next_encrypt == true || flagEncrypt == true)
```

```
{
  bufferTmp[i] = (byte)((int)bufferFlag[7] ^ hash[indexHash %
   hash.Length]);
  indexHash++;
}
```

untuk fungsi keseluruhan enkripsi VEA pada C# sebagai berikut,

```
private void enkripsi(NetworkStream clientStream, byte[]
bufferTmp, byte[] hash, int panjang, ref byte[] bufferPattern, ref
bool flagEncrypt, ref int indexHash, ref int totalFrame,ref int
totalFrameI)
{
 byte[] bufferFlag = new byte[8];//pencari startCode MPEG (lokal)
 bool next_encrypt = false; //penunjuk enkripsi (lokal)
 int shift = 0; //sebagai index geser saat bertemu start code
 int coding_type; //menyimpan type frame (I,P,B)
  for (int k = 0; k < bufferPattern.Length; k++ )
    {
      bufferFlag[k] = bufferPattern[k];
    }

  for (int i = 0; i < panjang; i++)
  {
    bufferFlag[0] = bufferFlag[1]; //mengeser nilai yang ada di
                                    bufferFlag
    bufferFlag[1] = bufferFlag[2];
    bufferFlag[2] = bufferFlag[3];
    bufferFlag[3] = bufferFlag[4];
    bufferFlag[4] = bufferFlag[5];
    bufferFlag[5] = bufferFlag[6];
    bufferFlag[6] = bufferFlag[7];
    bufferFlag[7] = bufferTmp[i]; //mengambil nilai yg ada
   bufferTmp dan dsimpan pada bufferFlag[7]

    if (next_encrypt == true || flagEncrypt == true) //flag
   enkripsi bernilai true maka dilakukan proses enkripsi
    {
      bufferTmp[i] = (byte)((int)bufferFlag[7] ^ hash[indexHash %
      hash.Length]);
      indexHash++; //menambahkan index hash
    }
    if (bufferFlag[0] == 0x00 && bufferFlag[1] == 0x00 &&
       bufferFlag[2] == 0x01)
      {
        if (bufferFlag[3] == 0x00)
        {
          totalFrame++;  //menambah jumlah frame
          coding_type = ((int)bufferFlag[5] & 0x38) >> 3;

          //mendapatkan type frame
          if (coding_type == 1) //jika frame i
            {
                totalFrameI++;  //menambah jumlah frame I
                next_encrypt = true;
                flagEncrypt = true;
            }
```

```
                else
                    {
                        next_encrypt = false;
                        flagEncrypt = false;
                        if (i < 7)
                        {
                            shift = i;
                        }
                       else
                        {
                            shift = 7;
                        }

                        for (int m = shift; m >= 0; m--)  //for yang
                        digunakan agar start code tidak ikut di-Xor-kan
                            {
                                bufferTmp[i - m] = bufferFlag[7 - m];
                            }
                        }
                }
            }
        }

         clientStream.Write(bufferTmp, 0, panjang);
            for (int l = 0; l < bufferPattern.Length; l++)
            {
              bufferPattern[l] = bufferFlag[l];
            }
}
```

4.3.9 Implementasi MD5

MD5 yang dibuat pada implementasi ini mengikuti *standart* dari internasional seperti pada rfc1321 dan sesuai perancangan pada subbab 4.2.1.5 dan untuk menghitung jumlah *bit padding* yang ditambahkan berdasarkan modulus 64 dan memperbaharui jumlah bit,

```
index = (int)((uint)(this.count[0] >> 3) & 0x3f);
padLen = (index < 56) ? (56 - index) : (120 - index);
HashCore(PADDING, 0, padLen);
```

untuk menambahkan panjang *bit padding*,

```
HashCore(bits, 0, 8);
```

untuk *padding*-nya sebagai berikut,

```
static private byte[] PADDING = new byte[]
{0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

menginisialisasi *magic byte* yang merupakan konstantayang sudah disepekati oleh internasionaldari MD5 dan merupakan suatu ketetapan,

```
state[0] = 0x67452301;
state[1] = 0xefcdab89;
state[2] = 0x98badcfe;
state[3] = 0x10325476;
```

untuk fungsi dasar MD5 sebagai berikut,

```
static private uint F(uint x, uint y, uint z)
{
return (((x) & (y)) | ((~x) & (z)));
}
static private uint G(uint x, uint y, uint z)
{
return (((x) & (z)) | ((y) & (~z)));
}
static private uint H(uint x, uint y, uint z)
{
return ((x) ^ (y) ^ (z));
}
static private uint I(uint x, uint y, uint z)
{
return ((y) ^ ((x) | (~z)));
}
static private uint ROTATE_LEFT(uint x, byte n)
{
return (((x) << (n)) | ((x) >> (32 - (n))));
}
```

bagian *transform* atau putaran pada MD5 sebagai berikut,

```
private void Transform(byte[] block, int offset)
{
uint a = state[0], b = state[1], c = state[2], d = state[3];
uint[] x = new uint[16];
  Decode(x, 0, block, offset, 64);

// putaran 1
  FF(ref a, b, c, d, x[0], S11, 0xd76aa478); /* 1 */
  FF(ref d, a, b, c, x[1], S12, 0xe8c7b756); /* 2 */
  FF(ref c, d, a, b, x[2], S13, 0x242070db); /* 3 */
  FF(ref b, c, d, a, x[3], S14, 0xc1bdceee); /* 4 */
  FF(ref a, b, c, d, x[4], S11, 0xf57c0faf); /* 5 */
  FF(ref d, a, b, c, x[5], S12, 0x4787c62a); /* 6 */
  FF(ref c, d, a, b, x[6], S13, 0xa8304613); /* 7 */
  FF(ref b, c, d, a, x[7], S14, 0xfd469501); /* 8 */
  FF(ref a, b, c, d, x[8], S11, 0x698098d8); /* 9 */
  FF(ref d, a, b, c, x[9], S12, 0x8b44f7af); /* 10 */
  FF(ref c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
  FF(ref b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
  FF(ref a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
  FF(ref d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
  FF(ref c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
  FF(ref b, c, d, a, x[15], S14, 0x49b40821); /* 16 */
```

```
// putaran 2
  GG(ref a, b, c, d, x[1], S21, 0xf61e2562); /* 17 */
  GG(ref d, a, b, c, x[6], S22, 0xc040b340); /* 18 */
  GG(ref c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
  GG(ref b, c, d, a, x[0], S24, 0xe9b6c7aa); /* 20 */
  GG(ref a, b, c, d, x[5], S21, 0xd62f105d); /* 21 */
  GG(ref d, a, b, c, x[10], S22, 0x2441453); /* 22 */
  GG(ref c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
  GG(ref b, c, d, a, x[4], S24, 0xe7d3fbc8); /* 24 */
  GG(ref a, b, c, d, x[9], S21, 0x21e1cde6); /* 25 */
  GG(ref d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
  GG(ref c, d, a, b, x[3], S23, 0xf4d50d87); /* 27 */
  GG(ref b, c, d, a, x[8], S24, 0x455a14ed); /* 28 */
  GG(ref a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
  GG(ref d, a, b, c, x[2], S22, 0xfcefa3f8); /* 30 */
  GG(ref c, d, a, b, x[7], S23, 0x676f02d9); /* 31 */
  GG(ref b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

// Putaran 3
  HH(ref a, b, c, d, x[5], S31, 0xfffa3942); /* 33 */
  HH(ref d, a, b, c, x[8], S32, 0x8771f681); /* 34 */
  HH(ref c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
  HH(ref b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
  HH(ref a, b, c, d, x[1], S31, 0xa4beea44); /* 37 */
  HH(ref d, a, b, c, x[4], S32, 0x4bdecfa9); /* 38 */
  HH(ref c, d, a, b, x[7], S33, 0xf6bb4b60); /* 39 */
  HH(ref b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
  HH(ref a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
  HH(ref d, a, b, c, x[0], S32, 0xeaa127fa); /* 42 */
  HH(ref c, d, a, b, x[3], S33, 0xd4ef3085); /* 43 */
  HH(ref b, c, d, a, x[6], S34, 0x4881d05); /* 44 */
  HH(ref a, b, c, d, x[9], S31, 0xd9d4d039); /* 45 */
  HH(ref d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
  HH(ref c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
  HH(ref b, c, d, a, x[2], S34, 0xc4ac5665); /* 48 */

// Putaran 4
  II(ref a, b, c, d, x[0], S41, 0xf4292244); /* 49 */
  II(ref d, a, b, c, x[7], S42, 0x432aff97); /* 50 */
  II(ref c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
  II(ref b, c, d, a, x[5], S44, 0xfc93a039); /* 52 */
  II(ref a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
  II(ref d, a, b, c, x[3], S42, 0x8f0ccc92); /* 54 */
  II(ref c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
  II(ref b, c, d, a, x[1], S44, 0x85845dd1); /* 56 */
  II(ref a, b, c, d, x[8], S41, 0x6fa87e4f); /* 57 */
  II(ref d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
  II(ref c, d, a, b, x[6], S43, 0xa3014314); /* 59 */
  II(ref b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
  II(ref a, b, c, d, x[4], S41, 0xf7537e82); /* 61 */
  II(ref d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
  II(ref c, d, a, b, x[2], S43, 0x2ad7d2bb); /* 63 */
  II(ref b, c, d, a, x[9], S44, 0xeb86d391); /* 64 */

  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;
```

```
for (int i = 0; i < x.Length; i++)
    x[i] = 0;
}
```

untuk hasil outputnya didapat dari mengubah *byte* menjadi *char* menggunakan fungsi encode sebagai berikut,

```
private static void Encode(byte[] output, int outputOffset, uint[]
input, int inputOffset, int count)
{
int i, j;
int end = outputOffset + count;
for (i = inputOffset, j = outputOffset; j < end; i++, j += 4)
  {
    output[j] = (byte)(input[i] & 0xff);
    output[j + 1] = (byte)((input[i] >> 8) & 0xff);
    output[j + 2] = (byte)((input[i] >> 16) & 0xff);
    output[j + 3] = (byte)((input[i] >> 24) & 0xff);
    }
   }
```

4.3.10 Implementasi Menerima Informasi dari *Server*

Pada sisi *client* setelah behasil berkoneksi dengan *server*, *client* akan menerima informasi dari *server* seperti pada perancangan yang telah dijelaskan pada subbab 4.2.2.2.1 dan pada implementasi proses ini terdapat pada fungsi yang diberi nama getInformation(). Berikut *source code* dari C# untuk fungsi getInformation.

```
public void getInformation()//fungsi untuk mendapatkan informasi
dari video seperti lebar,tinggi,jenis codec
{
   byte[] buffer = new byte[1048576];
   clientStream.Read(buffer, 0, buffer.Length);  //membaca byte-
    byte yang dikirim oleh server
   fileNameTemp = "C:\\vea_" + DateTime.Now.ToString
  ("dd_MM_yyyy_hh_mm_ss") + ".tmp";  //membuat nama vea temporary
  dengan acuan waktu
   fileWriter = new FileStream(fileNameTemp, FileMode.Create);
    //membuat file dengan namanya sesuai fileNameTemp

   if (veaChecked == 0)  //mengecek apakah tidak menggunakan vea
    {
    fileWriter.Write(buffer, 0, buffer.Length);
    }
   else
    {
    dekripsi(buffer, hash, buffer.Length); //melakukan proses
     dekripsi
    }
```

```
    clientVideoDecoder(fileNameTemp); //mengirimkan byte-byte yang
     ada di bufferTmp kepada Client
    timeEnd = (double)timerDekripsi.ElapsedMilliseconds / 1000;
    timeTotal = timeEnd - timeStart;
    richTextBoxStatus.Text += "Video Format= " + VideoFormat +
     Environment.NewLine;
    richTextBoxStatus.Text += "Width video = " + width +
     Environment.NewLine;
    richTextBoxStatus.Text += "Height video = " + height +
     Environment.NewLine;
    richTextBoxStatus.Text += "Timebase video = " + videoTimeBase +
     Environment.NewLine;
}
```

4.3.11 Implementasi *Update Streaming* dari *Server*

Seperti pada perancangan saat tombol *play* oleh *user*, *client* akan mengirimkan permintaaan data *streaming* berupa *flag* pada *server* untuk meminta data *streaming*. Setelah *server* menerima *flag*, *server* akan mengirimkan data *streaming* dan *client* menerima data tersebut sampai sampai *end of file*. Proses meng-*update streaming* terdapat pada fungsi yang diberi nama updateStreaming berikut *source code* pada C#.

```
public void updateStreaming(object state)//fungsi update streaming
{
byte[] flag = new byte[1]; //byte array yg digunakan sebagai flag
byte[] bufferLong = new byte[8]; //untuk menyimpan nilai sementara
sebesar Long(8 byte) dari server
byte[] bufferInt = new byte[4]; //untuk menyimpan nilai sementara
sebesar int(4 byte) dari server

int panjang = 1048576;
int byteReaded; //untuk membaca dan menghitung byte yang diterima

flag[0] = 88;
clientStream.Write(flag, 0, flag.Length); //mengirim paket 88 pada
server sebagai penunjuk meminta paket dari server

clientStream.Read(bufferLong, 0, bufferLong.Length); //membaca
paket dari server yang disimpan pada bufferLong

fileSize = BitConverter.ToInt64(bufferLong, 0); //nilai-nilai byte
yang dismpan pada bufferLong diubah menjadi nilai desimal

byte[] buffer = new byte[panjang];  //membuat buffer sebesar 1 MB
dari byte array

fileWriter.Position = fileWriter.Length; //mengarahkan pointer
penulis fileWriter ke posisi terakhir

// membaca data atau paket dari server
while ((byteReaded = clientStream.Read(buffer, 0, panjang)) != 0)
  {
   if (veaChecked == 0)//mengecek apakah tidak menggunakan vea
```

```
        {
          fileWriter.Write(buffer, 0, byteReaded);
        }
      else
        {
         dekripsi(buffer, hash, byteReaded); //proses dekripsi
        }
       if (fileWriter.Length == fileSize) //jika ukuran file sama
          dengan ukurang fileSize dari video maka keluar dari while
        {
        break;
        }
    }
     timeEnd = (double)timerDekripsi.ElapsedMilliseconds / 1000;
     timeTotal += (timeEnd - timeStart);
     if (veaChecked == 0)
       {
         richTextBoxStatus.Invoke(new
         updateStatusInvoker(updateStatus2), "Total Time Without VEA
         = " + timeTotal + " Second" + Environment.NewLine);
       }
     else
       {
         richTextBoxStatus.Invoke(new
         updateStatusInvoker(updateStatus2), "Total Time Decryption
         = " + timeTotal + " Second" + Environment.NewLine);
         richTextBoxStatus.Invoke(new
         updateStatusInvoker(updateStatus2), "Total Frame Mpeg = " +
         (totalFrame + 1) + Environment.NewLine);
         richTextBoxStatus.Invoke(new
         updateStatusInvoker(updateStatus2), "Total Frame I = " +
         (totalFrameI + 1) + Environment.NewLine);
        }
    fileWriter.Close();
    fileWriter.Dispose();  //mendelete object fileWriter
}
```

4.3.12 Implementasi Dekripsi Data *Frame* I

Implementasi dekripsi VEA pada dasarnya sebagian besar sama dengan enkripsi pada *server* tetapi disesuaikan dengan keadaan dari *client*. Implementasi proses dekripsi terdapat pada fungsi yang bernama dekripsi. Berikut ini adalah *source code* dari C#.

```
private void dekripsi(byte[] bufferTmp, byte[] hash, int panjang)
{ //Fungsi dekripsi video

bool next_encrypt = false;
int shift = 0;
int coding_type;
for (int i = 0; i < panjang; i++)
  {
     bufferFlag[0] = bufferFlag[1];
     bufferFlag[1] = bufferFlag[2];
     bufferFlag[2] = bufferFlag[3];
     bufferFlag[3] = bufferFlag[4];
```

```
            bufferFlag[4] = bufferFlag[5];
            bufferFlag[5] = bufferFlag[6];
            bufferFlag[6] = bufferFlag[7];
            bufferFlag[7] = bufferTmp[i];

             if (next_encrypt == true || flagEncrypt == true)
                 {
                   bufferTmp[i] = (byte)((int)bufferFlag[7] ^
                   hash[indexHash % hash.Length]);
                   indexHash++;
                 }

              if (bufferFlag[0] == 0x00 && bufferFlag[1] == 0x00 &&
                  bufferFlag[2] == 0x01)
                 {
                    if (bufferFlag[3] == 0x00)
                   {
                       totalFrame++;   //menambah jumlah frame
                       coding_type = ((int)bufferFlag[5] & 0x38) >> 3;
                       if (coding_type == 1)//if frame i
                        {
                         totalFrameI++;  //menambah jumlah frame I
                         next_encrypt = true;
                         flagEncrypt = true;
                        }
                      else
                     {
                       next_encrypt = false;
                       flagEncrypt = false;

                       if (i < 7)
                       {
                         shift = i;
                       }
                       else
                       {
                          shift = 7;
                       }

                       for (int m = shift; m >= 0; m--)
                       {
                         bufferTmp[i - m] = bufferFlag[7 - m];
                       }
                     }
                   }
                 }
            }
            fileWriter.Write(bufferTmp, 0, panjang);
        }
```

4.3.13 Implementasi Video *Codec* FFMPEG

Implementasi pada subab ini akan dijelaskan mengenai pemakaian *library* FFMPEG yang akan digunakan sebagai *codec* untuk membaca informasi video dan memainkan *file* video. Seperti pada perancangan bahwa *library* yang

digunakan untuk membaca *file* video MPEG adalah FFMPEG. Akan tetapi FFMPEG merupakan *library* yang dibuat atau di-*compile* menggunakan bahasa pemrograman C++ bukan DotNet. Jadi bagi C# DotNet *library* tersebut merupakan *library native* atau asing. Sehingga untuk bisa menggunakan *library* FFMPEG maka diperlukan penjembatan agar antara C# Dotnet dengan *library* FFMPEG tersebut dapat berkomunikasi. Penjembatan yang dimaksud tersebut adalah *library* yang bernama AvDnCpp.dll, dimana AvDnCpp merupakan singkatan dari Audio Video Dotnet dan dibuat dari bahasa C++ Dotnet.

Dengan adanya *library* AvDnCpp.dll sehingga *library* FFMPEG bisa digunakan pada C# Dotnet. Untuk bisa menggunakan *library* AvDnCpp.dll tersebut dengan cara menambahkan referensi pada *project* C# tersebut dan memanggilnya, berikut *script* untuk mengimport *library* AvDnCpp.dll.

```
using AvDn;
```

Dengan menggunakan *syntax* "using" sudah bisa menggunakan semua class dan fungsi pada *library* tersebut. Sedangkan *library* dari FFMPEG yang digunakan dalam program ini hanya 3 *library* yakni avformat-50.dll, avcodec-51.dll, avutil-49.dll. Pada Tiap-tiap *library* tersebut mempunyai fungsi yang berbeda-beda. Untuk *library* avformat-50.dll mempunyai fungsi untuk mengenali format video, dimana akan mendapatkan informasi tiap format video yang dikenali. Untuk *library* avcodec-51.dll berfungsi sebagai *codec* untuk memainkan video karena pada *library* ini tersimpan *codec* dari setiap format video yang dikenali oleh *library* FFMPEG. Sedangkan untuk *library* avutil-49.dll berisi utilitas atau fungsi pembantu yang berguna saat mempergunakan *library* avformat-50.dll dan avcodec-51.dll, misalnya terdapat fungsi untuk meregisterkan semua *codec* pada program. Memang pada dasarnya ketiga *library* tersebut saling terkait sehingga menjadi *library* yang bernama FFMPEG. Dan terdapat syarat yang penting untuk menggunakan semua *library-library* tersebut yakni tempat dari *library* tersebut harus sama atau satu lokasi dengan *file executable* yang akan menggunakannya.

Seperti pada perancangan *client*, setelah mendapatkan informasi berupa byte sebesar 1MB dan ditulis menjadi *file*, proses yang dilakukan berikutnya adalah membaca *file* video tersebut dengan *codec* yang terdapat pada *library*

FFMPEG untuk mendapatkan informasi-informasi yang penting dari video tersebut sebelum memainkannya. Informasi-informasi tersebut seperti lebar, tinggi, dan video *timebase*.

Proses mengolah informasi tersebut terdapat pada fungsi yang bernama clientVideoDecoder. Pada fungsi tersebut juga terdapat proses yang berfungsi untuk meregisterkan semua *codec* dari *library* FFMPEG dengan program utama yakni dengan perintah AvDnUtils.RegisterAll(). Berikut *source code* dari fungsi clientVideoDecoder.

```
public void clientVideoDecoder(String fileName)//Fungsi men-decode
file video
{
   AvDnUtils.RegisterAll();
         formatCtx = new AvDnFormatContext(fileName);
         if (formatCtx.FindStreamInfo() < 0)
          {
           throw new Exception("Couldn't find stream info");
          }
         videoStream = -1;
         int nbStream = formatCtx.NbStreams;
        for (int i = 0; i < nbStream; i++)
        {
          int codec_type = formatCtx.GetCodecTypeForStream(i);
          if (codec_type == Av.CODEC_TYPE_VIDEO)
          {
            videoStream = i;
       codecCtx = formatCtx.GetCodecContextForStream(videoStream);
           if (codecCtx.OpenDecoder() < 0)
            {
              throw new Exception("couldn't open decoder");
            }
            if (codecCtx.getCodecID() == 1)
            {
               VideoFormat = "MPEG-1";
            }
            else if (codecCtx.getCodecID() == 2)
            {
              VideoFormat = "MPEG-2";
            }
            else
            {
              VideoFormat = "Unknown Codec";
            }

            width = codecCtx.Width;
            height = codecCtx.Height;
            pixFmt = codecCtx.PixFmt;
        videoTimeBase = formatCtx.getTimeBaseStream(videoStream);
          }
       }
}
```

4.3.14 Implementasi Video *Updater*

Pada implementasi ini terdapat sebuah *thread* yang bertugas untuk menjalankan proses untuk meng-*update* video atau menampilkan dari *file* video yang diolah oleh *library* FFMPEG menjadi gambar yang bisa dilihat sesuai dengan waktu penampilan sehingga menjadi video yang bisa dinikmati. *Thread* yang digunakan pada bagian ini tidak seperti *thread* yang dibuat secara manual oleh pemrogram tetapi dikelola secara automatis oleh DotNet sehingga pemrogram hanya memberitahu fungsi mana yang harus dikerjakan, kemudian DotNet langsung mengatur *thread* tersebut secara otomatis yaitu menggunakan class ThreadPool. Kekurangannya apabila dikerjakan secara otomatis yaitu kita kurang bisa leluasa mengontrolnya seperti secara manual. Fungsi yang akan dijalankan oleh tersebut bernama videoUpdater, dimana fungsi tersebut berisi proses mengolah video dan menampilkannya. Berikut *syntax* pada C# untuk membuat *thread* otomatis.

```
ThreadPool.QueueUserWorkItem(videoUpdater);
```

Sesuai pada peranacangan fungsi videoUpdater berisi proses untuk mendapatkan setiap *frame* dan menampilakannya sesuai waktu yang tepat. Berikut fungsi dari videoUpdater dari C#,

```
public void videoUpdater(Object state) //Fungsi meng-update video
{
 int detik = 0;
 int menit = 0;
 int jam = 0;
 Bitmap[] bitmapPicture = new Bitmap[2]; //membuat 2 buah array
 dari class Bitmap yang berfungsi untuk penampilan gambar dan
 dual buffering
 bitmapPicture[0] = new Bitmap(width,height,PixelFormat.
 Format24bppRgb);
 bitmapPicture[1] = new Bitmap(width, height, PixelFormat.
 Format24bppRgb);
 int target = 0; //variable yg digunakan sebagai pemilih kapan
 menggunakan bitmap[0] atau bitmap[1]

  try
    {
   while (playing)  //looping selama kondisi masih playing
     {
           double time =(double)timer.ElapsedMilliseconds/1000.0;
           detik = (int)time;
           menit = detik / 60;
           detik = detik - menit * 60;
           jam = menit / 60;
           menit = menit % 60;
```

```
if (displayVideo.InvokeRequired) //mengecek apakah perlu
melakukan Invoke
 {
    displayVideo.Invoke(new
   updateStatusInvoker(updateStatus),"Display Video =
   "+((jam<10)?"0"+jam.ToString():jam.ToString())+":"+((men
   it<10)?"0"+menit.ToString():menit.ToString())+":"+((deti
   k<10)?"0"+detik.ToString():detik.ToString()));
  }
else
  {
   richTextBoxStatus.Text = "Display Video =
   "+((jam<10)?"0"+jam.ToString():jam.ToString())+":"+((men
   it<10)?"0"+menit.ToString():menit.ToString())+":"+((deti
   k<10)?"0"+detik.ToString():detik.ToString());
  }

BitmapData bitmapData = bitmapPicture[target].LockBits(new
Rectangle(0, 0, bitmapPicture[target].Width,
bitmapPicture[target].Height), ImageLockMode.WriteOnly,
PixelFormat.Format24bppRgb); //membuat bitmap data dari
class BitmapData sesuai kondisi dan ukuran bitmapPicture
yang berguna sebagai isi dari bitmap tersebut

playing = nextVideoFrame(bitmapData.Scan0, Av.PIX_FMT_BGR24,
ref time); //mendapatkan frame video (bitmap data) dan waktu
penampilan

bitmapPicture[target].UnlockBits(bitmapData); /dari frame
video yang telah didapat dijadikan gambar yang disimpan pada
bitmap yang siap ditampilkan

if (time == 0) //jika waktu sudah benar maka menampilkan
               picture
{
 //melakukan invoke dengan cara delegasi sebuah fungsi, hal
 ini dibutuhkan karena melakukan proses yang berbeda thread
 displayVideo.pictureBoxVideo.Invoke((voidDelegate)delegate
 {
  displayVideo.pictureBoxVideo.Image= bitmapPicture[target];
  //menampilkan bitmapPicture ke pictureBoxVideo yang ada d
  form displayVideo

  target = (target == 0) ? 1 : 0;  //mengecek untuk
  pergantian target atau index dari array
 });
}

if (playing && time > 0.005) /tidak melakukan proses jika
waktunya terlalu cepat
      {
         Thread.Sleep((int)(time * 1000.0)); //thread tidur
       }
 while (paused)
       {
          Thread.Sleep((int)((1.0 / 20.0) * 1000));//thread
    tidur
       }
```

```
            }

            timer.Reset();  //me-reset Timer
          }
catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
Finally
  {
            bitmapPicture[target].Dispose(); //menghapus objek
  }
}
```

Pada fungsi videoUpdater terdapat fungsiNextVideoFrame, dimana fungsi tersebut berfungsi untuk mendapatkan *frame* dan waktu penampilannya atau pts berdasarkan hasil olahan dari *library* FFMPEG. Berikut ini merupakan *source code*-nya.

```
//fungsi untuk mendapatkan frame video dan mengubah menjadi gambar
yang akan siap dtampilkan
public bool nextVideoFrame(IntPtr target, int desiredFormat, ref
double time)
{
      long pts = -1; //pts (presentation timestamp) merupakan
       referensi waktu untuk penampilan dari sebuah frame

      long dts;  //dts (decoding timstamp) merupakan referensi
      waktu untuk men-decode dari sebuah frame

      //Alokasikan frame
      AvDnFrame frame = new AvDnFrame(); //variable frame untuk
      membangun sebuah frame dari packet yg telah diterima

      AvDnFrame frameRGB = new AvDnFrame();//variable frame yg
      menyimpan frame yang sudah jadi dan sesuai format RGB

      frameRGB.FillPicStruct(target, desiredFormat, width, height
      ); //mengalokasikan buffer frameRGB yang telah dbuat dan
      memberikan ukuran juga pixel format yang akan dbentuk

      int frameFinished = 0;  //variable penanda frame sudah
      selesai dbentuk atau belum

      while (frameFinished == 0) //looping terus selama frame
      belum selesai dibentuk
        {
         if (flagWait == 0)//mengecek apakah sedang tidak menunggu
          {
            if (vPacket == null) //mengecek apakah paket belum di-
                                    inisialisasi
            {
              vPacket = new AvDnPacket();
            }
```

```
                lock (locker)  //menge-lock proses agar isi dalam
                scope ini tidak boleh diakses oleh thread lain
                {
                  while (videoPacketQueue.Count < 1) //mengecek apakah
                  queue kosong. Jika iya, maka meminta paket lagi
                  {
                   if (!readPacket())  //mengecek jika saat request
                    paket false atau gagal, maka keluar
                    {
                      return false;
                    }
                   }
                   vPacket = videoPacketQueue.Dequeue(); //mengambil
                    paket dari Queue dan menyimpanya pada vPacket
                  }
              }
              if (pts == -1)
              {
                 pts = vPacket.GetPts;  //mendapatkan nilai pts

                 dts = vPacket.GetDts;   //mendapatkan nilai dts

                 if (dts != AvDnUtils.StandartPts())//mengecek apakah
                 dts-nya ada nilainya. Jika ada, maka nilai yg dipakai
                 adalah dts untuk pts
                 {
                    pts = dts;
                 }
                 else
                 {
                   pts = 0;
                 }

                 if (pts * videoTimeBase >= time)//mengecek apakah waktu
                 penampilanya terlalu cepat,jika iya tidak melakukan
                 proses selanjutnya
                 {
                   time = (pts * videoTimeBase) - time;  //mendapatkan
                   delay untuk menunggu agar waktu penampilannya tepat

                   flagWait = 1;  //memberi penada untuk menunggu
                   return true;
                 }
                  time = 0;//tidak ada delay karena waktunya sudah benar

              }
             //melakukan proses decoding dari paket yg telah didapat
             sehingga menjadi frame utuh. Jika sudah utuh, maka flag
             frameFinised akan diisi
             codecCtx.DecodeVideo(frame, vPacket, out frameFinished);
        }
         //jika frame sudah jadi,maka dilakukan peng-convert-an dari
         frame raw menjadi frame RGB
         frameRGB.Convert(desiredFormat, frame, pixFmt, width, height);

         flagWait = 0;  //menge-nol-kan flagwait sebagai tanda sudah
                        tidak menunggu
         return true;
    }
```

Pada fungsi NextVideoFrame terdapat fungsi readPacket, fungsi ini berfungsi untuk mendapatkan paket data video dari hasil olahan *library* FFMPEG, dimana paket tersebut digunakan untuk membangun atau membentuk *frame* video. Berikut fungsi dari readPacket pada C#.

```
//fungsi untuk membaca paket-paket dari frame yg ada di video
public bool readPacket()
{
   AvDnPacket pPacket = new AvDnPacket();//membuat object baru
   untuk membaca paket
   if (formatCtx.ReadFrame(pPacket) < 0) //membaca paket dari
   frame yang disimpan dalam pPacket dan jika hasil membaca
   paketnya kurang dari 0, maka tidak ada paket yangg dibaca
     {
       return false;
     }
   if (pPacket.StreamIndex == videoStream //mengecek apakah paket
       tersebut untuk data video
     {
      videoPacketQueue.Enqueue(pPacket); //memasukan paket
      tersebut pada queue
     }
return true;
}
```

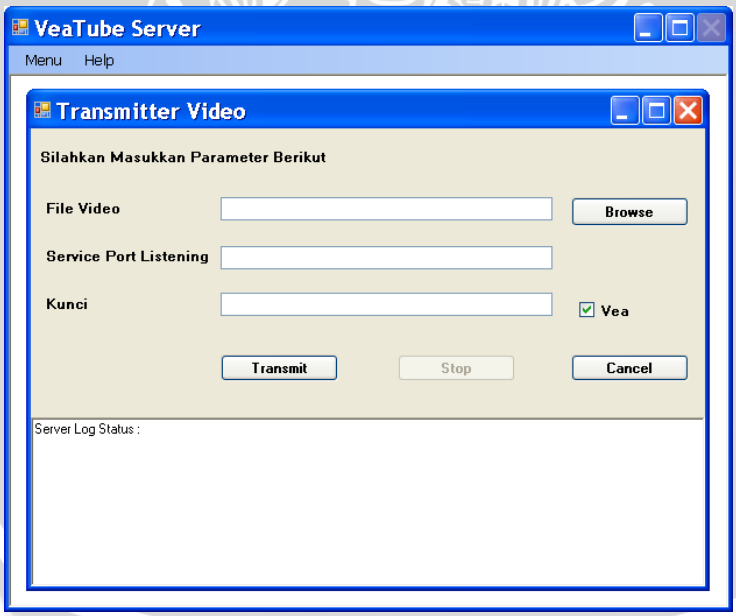
4.4 Implementasi *Interface* (Antarmuka)

Pada aplikasi VeaTube ini memiliki 2 tampilan utama yaitu tampilan utama untuk aplikasi *server* dan tampilan utama untuk aplikasi *client*. Tampilan utama untuk aplikasi *server* dapat dilihat pada Gambar 4.27 dan tampilan utama untuk aplikasi *server* memiliki satu internal *form* yaitu *transmitter* video yang ditunjukkan pada Gambar 4.28

Tampilan utama untuk aplikasi *client* dapat dilihat pada Gambar 4.29 dan kedua tampilan utama aplikasi *server* dan *client* hanya memiliki satu tampilan utama dan dibuat sesederhana mungkin agar *user* tidak mengalami kebingungan sewaktu menggunakannya. Tampilan utama untuk aplikasi *client* memiliki dua internal *from* yaitu *receiver* video yang ditunjukkan pada Gambar 4.30 dan *display* video yang ditunjukkan pada Gambar 4.31

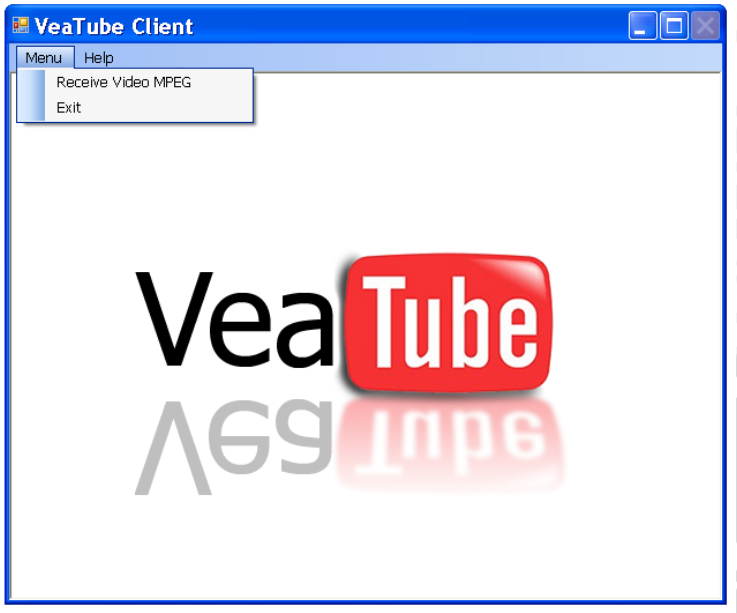
Gambar 4.27 Tampilan utama untuk aplikasi *server*

(**Sumber** : Perancangan)



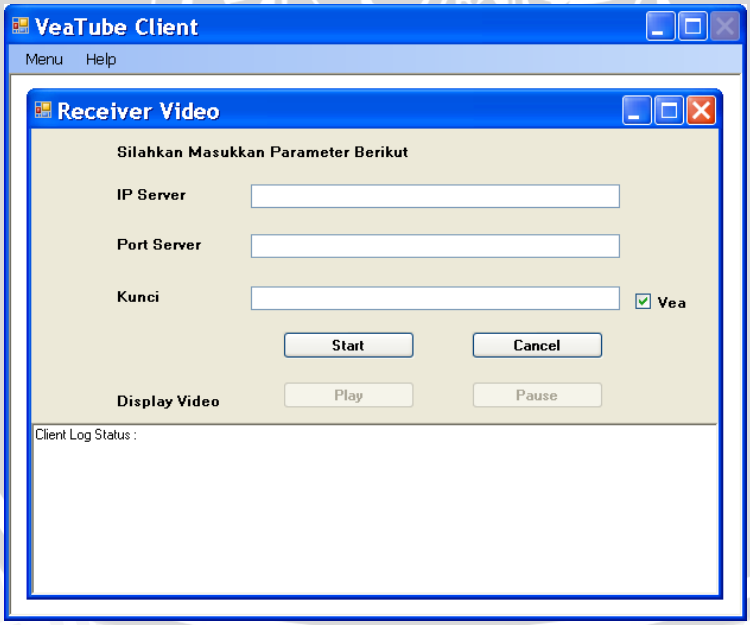
Gambar 4.28 Tampilan *transmitter* video

(**Sumber** : Perancangan)



Gambar 4.29 Tampilan utama untuk aplikasi *client*

(**Sumber** : Perancangan)



Gambar 4.30 Tampilan *receiver* video

(**Sumber** : Perancangan)

Gambar 4.31 Tampilan *display* video

(**Sumber** : Perancangan)