BAB 2 LANDASAN KEPUSTAKAAN

Dalam bab landasan kepustakaan ini akan dibahas tentang kajian pustaka beserta dasar teori yang mendukung dan menjadi acuan dalam penelitian. Kajian pustaka akan membahas tentang analisis penelitian pembanding yang sudah ada, sedangkan dasar teori membahas tentang teori-teori yang dibutuhkan untuk membantu penyusunan penelitian.

2.1 Kajian Pustaka

Kajian pustaka digunakan untuk perbandingan analisa terkait penelitian yang sudah dilakukan dengan penelitian yang akan dilakukan. Dalam penelitian ini ada beberapa jurnal atau skripsi yang digunakan untuk melakukan perbandingan tersebut. Perbandingan tersebut disajikan dalam tabel 2.1.

Tabel 2.1 Kajian Pustaka

No	Judul Jurnal	Persamaan	Perbedaan	
			Penelitian Terdahulu	Rencana Penelitian
1	Sugiri, Theo M B. 2016. Analisis Perbandingan Unjuk Kerja TCP Reno Dan Vegas Pada Jaringan *Wired.*	Melakukan analisis TCP Vegas. Dengan parameter *throughput, packet loss* dan *delay.*	Analisis TCP Reno dan Vegas pada omnet++. Parameter yang diukur *throughput, packet loss* dan *delay.* Model jaringan *wired*.	Analisis TCP Vegas dan TCP New Reno pada NS2. Menggunakan antrian *Droptail* dan *Random Early Detection.* Parameter yang diukur, *packet delivery ratio, throughput, delay* dan *packet loss.* Model jaringan kabel.
2	Kurniawan, Yoppi. 2016. Analisis Perbandingan Unjuk Kerja TCP Reno Dan TCP New Reno Pada *Router Droptail* Dan *Random Early Detection.*	Melakukan analisis TCP New Reno pada NS2. Dengan parameter *throughput, packet loss* dan *delay.*	Analisis perbandingan TCP Reno dan TCP New Reno pada NS-2. Parameter yang diukur *throughput, delay, packet loss* dan *congestion window*. Model jaringan *wired.*	
3	Manibuy, Kukuh R A. 2017. Analisis Perbandingan Unjuk Kerja TCP Tahoe Dan New Reno Pada Jaringan *Wireless* dan *Wired.*	Melakukan analisis TCP New Reno. Dengan parameter *throughput, delay* dan *packet loss.*	Analisis Perbandingan TCP Tahoe dan New Reno pada omnet++. Parameter yang diukur *throughput, delay* dan *packet loss*. Model jaringan *wireless dan wired.*	
4	Albareki, Mustofa A M. 2008. Comparison	Melakukan analisis TCP Vegas. Dengan	Analisis perbandingan TCP Tahoe, TCP Reno, TCP Vegas, TCP Veno,	

	Between Transmission Control Protocol Schemes On Wireless Environment.	parameter *throughput* dan *delay*.	TCP Sack dan TCP Freeze. Model jaringan *wired, wireless* dan Heterogen.	
5	Chhabra, A., Dhiman, A., & Joshi, M. 2014. Performance Evaluation Of Variants Of TCP Based On Buffer Management Over *WiMAX.*	Melakukan analisis TCP New Reno. Dengan parameter *throughput, delay dan packet loss*.	Analisis perbandingan performa TCP New Reno, TCP Westwood dan TCP Cubic. Model jaringan *WiMAX.*	

Setelah mengetahui beberapa penelitian yang telah dilakukan, penelitian ini fokus terhadap analisis perbandingan kinerja TCP Vegas dan TCP New Reno menggunakan antrian *Random Early Detection* dan *Droptail.*

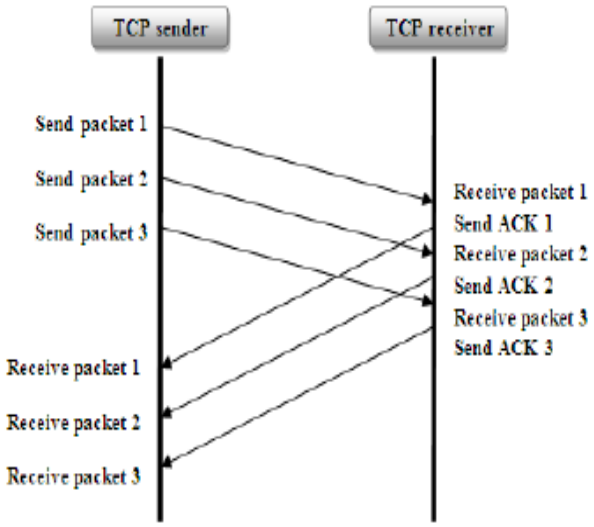
2.2 Dasar Teori

Dasar teori digunakan untuk mendukung penelitian tentang analisis kinerja. Beberapa teori untuk mendukung penelitian ini, diantaranya adalah sebagai berikut:

2.2.1 *Transmission Control Protokol* (TCP)

Transmission Control Protokol (TCP) merupakan dasar bahasa komunikasi dan protokol berbasis *conection-oriented* yang berada pada lapisan *transport.* TCP terdiri dari kumpulan peraturan dan prosedur untuk mengendalikan komunikasi yang melintasi *link*. Banyak varian TCP telah dimodifikasi dan dikembangkan untuk lebih baik dalam berkomunikasi. Sebagian besar versi TCP saat ini sudah memiliki algoritma *congestion control* dalam jaringan untuk memperbaiki *throughput* jaringan (Qureshi dkk, 2009). Akhir-akhir ini, TCP telah berkembang dengan pesat dalam mengirimkan data melalui *high-speed link*. Selanjutnya, TCP yang dirancang untuk jaringan kabel juga dapat digunakan di jaringan nirkabel karena memiliki beberapa fitur seperti *flow control, reliability* dan *congestion control*. TCP mampu menyesuaikan ukuran *congestion window* untuk meningkatkan kinerja, tapi itu juga bisa menyebabkan penurunan kinerja TCP itu sendiri (Henna, 2009).

Setiap TCP sumber mengatur ukuran *congestion window* menggunakan mekanisme *congestion control* dan TCP secara dinamis mengatur *window size*, tergantung pada paket *acknowledgment* (ACK) atau *packet loss*. Gambar 2.1 mengilustrasikan contoh dari tiga paket yang dikirim oleh TCP sumber lalu diterima oleh TCP tujuan dan TCP tujuan mengirimkan ACK tiap paket kembali ke TCP sumber.



Gambar 2.1 Ilustrasi pengiriman dan penerimaan paket pada TCP

Sumber : Ghassan Akram Abed (2013)

Banyak varian TCP telah dikembangkan untuk jaringan dan aplikasi yang berbeda. pada TCP Tahoe, Reno dan Vegas, algoritma *congestion control* memungkinkan *window size* bertambah satu segmen di setiap RTT. Kenaikan ini berhenti saat *window size* mencapai titik *congestion*. Karena itu *congestion control* pada TCP adalah faktor penting terhadap kinerja beberapa data yang dikirimkan dalam suatu jaringan. Ada empat tahap dalam TCP *congetion control* yaitu *slow start, congestion avoidance, fast retransmit* dan *fast recovery.*

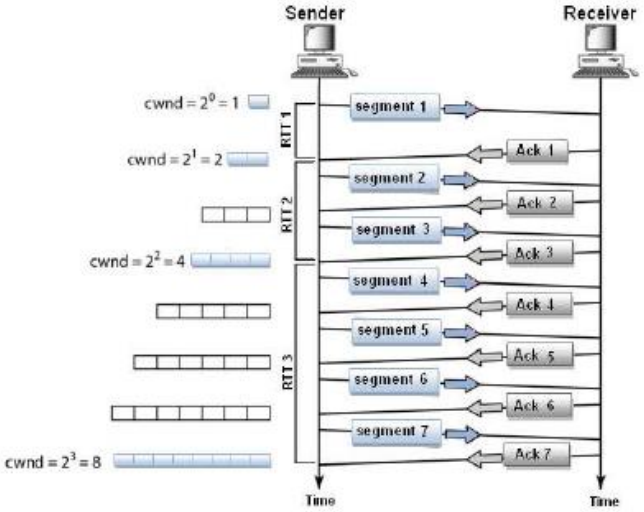
2.2.2 Slow Start

Tahap ini merupakan tahap awal dimana TCP sumber mengatur kecepatan pengiriman data ke TCP tujuan. Proses *slow start* baru dimulai ketika *acknowledgement* diterima dari TCP tujuan. Oleh karena itu, tingkat transmisi dari TCP sumber sepenuhnya tergantung pada *acknowledgement* yang dikirim oleh TCP tujuan. Meskipun *slow start* merupakan bagian dari *congestion control*, secara signifikan bisa mempengaruhi perilaku dan kinerja jaringan. Tujuan mekanisme *slow start* adalah membantu TCP sumber untuk menyesuaikan *bandwidth* yang ada di jaringan, sehingga jumlah segmen yang bisa masuk dalam jaringan menjadi banyak (Wang dkk, 2000).

Proses *slow start* menggunakan *congestion window* dengan satu *segment size* dan dengan setiap ACK yang diterima, ukuran *congestion window* bertambah satu segmen. Alasan ini membuat *congestion window* meningkat secara eksponensial dan segmen ditambahkan ke jaringan untuk setiap RTT (Cavendish dkk, 2009). Seperti ditunjukkan pada Gambar 2.2 saat koneksi dibuat, *congestion window* diatur ke satu segmen di awal dan meningkat satu segmen untuk masing-masing segmen ACK ketika berhasil diterima (Allcock dkk, 2005).

Slow start dimulai dengan satu segmen untuk congestion window (congestion window = 1) dan dengan setiap ACK yang berhasil diterima, *congestion window* dinaikkan satu (*congestion window* = *congestion window* + 1). C*ongestion*

window akan bertambah secara eksponensial untuk setiap RTT dan menduplikasi ukuran *congestion window* (*congestion window* = 2 * *congestion window*) sampai terjadi *congestion*. Dalam hal ini, TCP menuju ke tahap berikutnya, yaitu *congestion avoidance.*

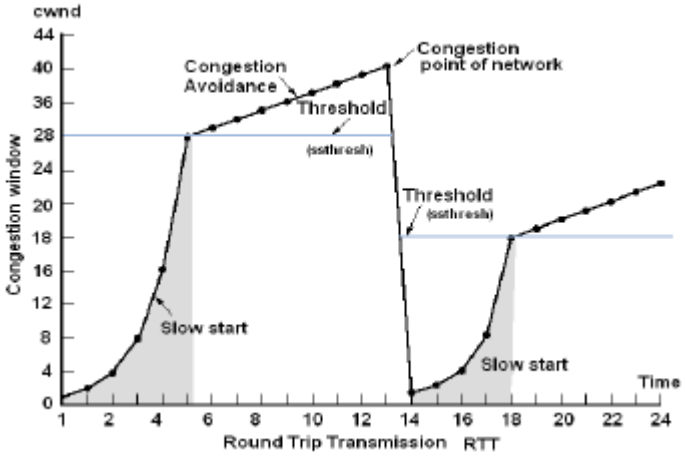


Gambar 2.2 Fase *Slow Start*

Sumber : Ghassan Akram Abed (2013)

2.2.3 Congestion Avoidance

Tahap ini merupakan tahap kedua, dimana mekanisme *congestion avoidance* memperbaiki masalah kegagalan jaringan dan digunakan dalam implementasi TCP. Setelah nilai *ssthresh* tercapai, TCP memperlambat laju peningkatan *window size* dan saat tingkat transmisi TCP melebihi kapasitas *link* pada jaringan maka *packet loss* terjadi. TCP segera mendeteksi *packet loss* ini dan mengurangi ukuran *congestion window* hampir setengah dari nilai saat ini dan kemudian memasuki fase *congestion avoidance*. Gambar 2.3 mengilustrasikan kombinasi operasi dan hubungan antara *slow start* dan *congestion avoidance*.



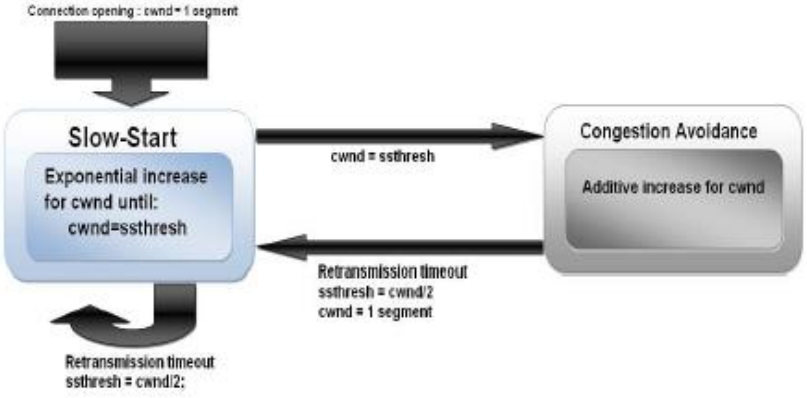
Gambar 2.3 Kombinasi Operasi *Slow Start* dan *Congestion Avoidance*

Sumber : Ghassan Akram Abed (2013)

Mekanisme *congestion avoidance* diimplementasikan saat ukuran *congestion window* lebih besar dari *ssthresh* (Sarolahti, 2007). Biasanya, *congestion window* tumbuh pada fase *slow start* dan *window* akan meningkat sampai melebihi *ssthresh*. Ketika *congestion window* lebih besar *ssthresh*, TCP mencapai tahap *congestion avoidance* dan pada tahap ini, tujuan utamanya adalah menjaga *throughput* tetap tinggi dan menghindari *congestion*.

Selanjutnya, jika TCP menemukan segmen yang hilang, hal itu menandakan bahwa *congestion* telah terjadi pada jaringan. Sebagai tindakan perbaikan, TCP menurunkan laju aliran dengan cara menurunkan *congestion window*. Namun, jika *congestion window* berkurang, TCP kembali ke fase *slow start*. Mekanisme *congestion window* memaksa TCP untuk menstabilkan kondisi berada di bawah keadaan *packet loss* yang dapat diterima dan meningkatkan *congestion window* dalam volume konstan per RTT.

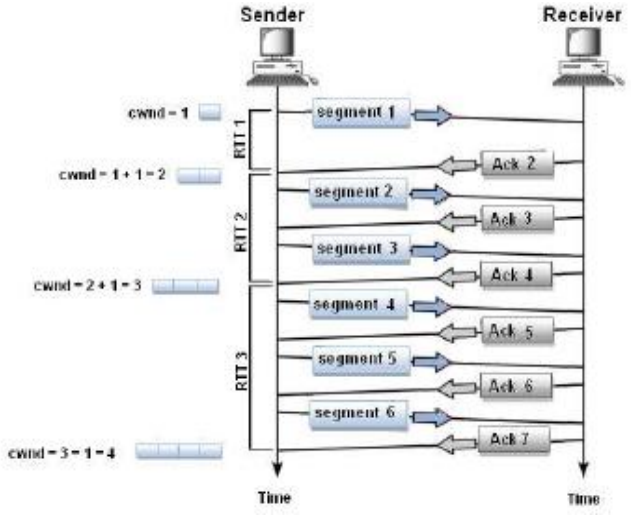
Jika *congestion* terjadi, TCP memperlambat tingkat pengiriman data di jaringan dan kembali ke *slow start*. Hal ini menjelaskan kedua mekanisme dijalankan bersama (Parziale, 2006). Dalam fase s*low start,* jika menjatuhkan satu atau beberapa paket karena *congestion*, *congestion avoidance* digunakan untuk memperlambat laju pengiriman. Akibatnya, TCP sumber langsung mengurangi ukuran *congestion window* sampai setengah dari ukuran *window* saat ini. Walaupun demikian, saat *congestion* terdeteksi, *congestion window* diatur ulang ke satu segmen, yang secara otomatis memaksa TCP sumber untuk memasuki tahap *slow start* seperti ditunjukkan pada Gambar 2.3.



Gambar 2.4 Hubungan *Slow Start* dan *Congestion Avoidance*

Sumber : Ghassan Akram Abed (2013)

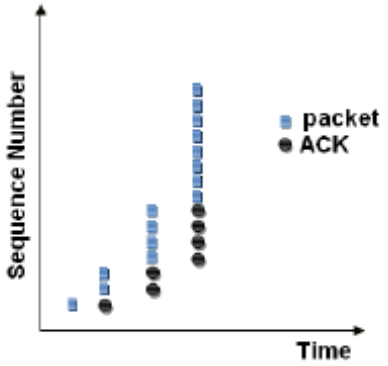
Seperti ditunjukkan pada gambar 2.4 saat *congestion window* > *ssthresh*, sinyal TCP masuk mode *congestion avoidance*, sedangkan jika *congestion window* = *ssthresh*, *slow start* atau *congestion avoidance* bisa digunakan (Bansal, 2005). Pada gambar 2.2 ukuran *congestion window* dua kali lipat untuk setiap RTT yang menghasilkan kenaikan *congestion window* secara eksponensial. Kenaikan ini berlanjut sampai *congestion window* mencapai atau melebihi nilai *ssthresh.*



Gambar 2.5 Fase *Congestion Avoidance*

Sumber : Ghassan Akram Abed (2013)

Gambar 2.2 mengilustrasikan aktivitas TCP *congestion window*. Pada awalnya, TCP sumber menggandakan ukuran *congestion window* untuk setiap RTT sampai ukuran *congestion window* menjadi lebih besar dari *ssthresh*. Lalu, *congestion window* itu meningkat satu segmen untuk setiap RTT seperti yang ditunjukkan pada Gambar 2.5.



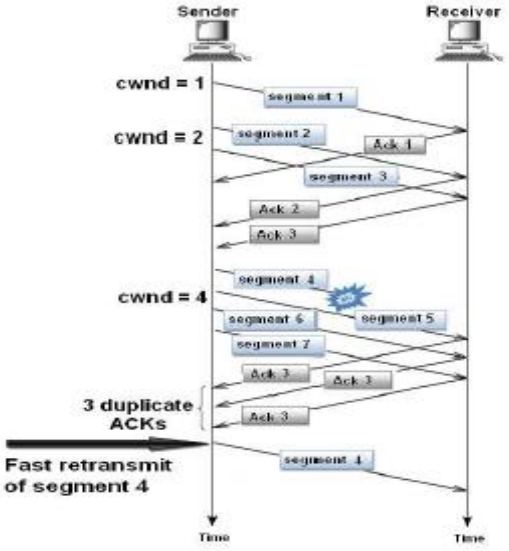
Gambar 2.6 Duplikat *Congestion Window* Untuk Setiap RTT

Sumber : Ghassan Akram Abed (2013)

2.2.4 Fast Retransmit dan Fast Recovery

Tahap ini merupakan tahap ketiga, dimana pada dasarnya *slow start* dan *congestion avoidance* adalah untuk mengatur *throughput* setelah terjadi *packet loss*. *Fast retransmit* dan *fast recovery* dapat mempercepat koneksi *recovery* tanpa membatasi *congestion avoidance*. *Fast retransmit* dan *recovery* membedakan *segment loss* dengan menduplikasi *acknowledgement* (Olsen, 2003). Mekanisme *fast recovery* menyesuaikan pengiriman segmen baru sampai TCP sumber menerima *non*-duplikat *acknowledgement*. Sekali mengalami *segment loss,* TCP tujuan akan mempertahankan pengiriman *acknowledgement* dan menentukan nomor pesanan yang dapat diprediksi berikutnya.

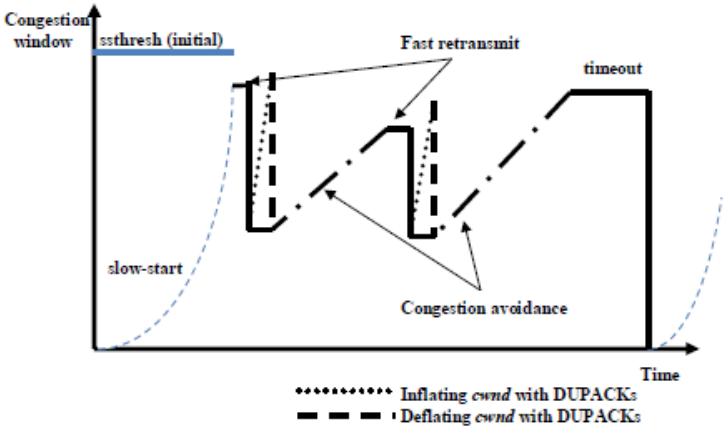
Jika terdapat satu *segment loss*, TCP akan membuat ACK untuk segmen selanjutnya. Hal tersebut memungkinkan pengirimnya menerima duplikat ACK. Dalam *fast retransmit,* TCP menduplikasi ACK untuk mengirim ulang segmen. Dalam *fast recovery*, ketika *segment loss*, TCP menjaga kecepatan pengiriman yang ada tanpa kembali ke *slow start*. *Fast retransmit* dan *fast recovery* dilakukan untuk memecahkan masalah terkait *packet loss* tanpa menunggu RTO. Mekanisme *fast retransmit* pertama kali diformulasikan di TCP Tahoe (Wang dkk, 2000). Hal tersebut berdasarkan konsep pengiriman kembali *unacknowledgemen segment* setelah menerima tiga duplikat ACK, saat hal itu terjadi ukuran *congestion window* kembali ke satu segmen dan mulai dari *slow start*. Tujuan menerima tiga duplikat ACK adalah untuk mengirim segmen yang hilang karena *congestion* (Ho dkk, 2005). Mekanisme *fast retransmit* diilustrasikan pada Gambar 2.8.



Gambar 2.7 Mekanisme *Fast Retransmit*

Sumber : Ghassan Akram Abed (2013)

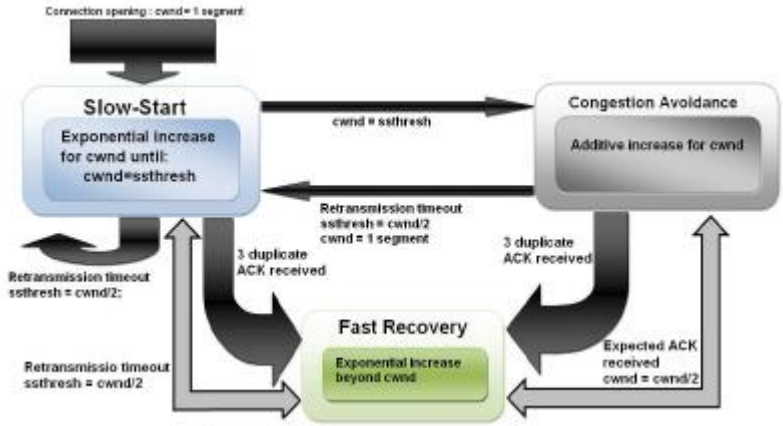
Gambar 2.9 menunjukkan skenario *fast recovery* dari *congestion window* dan menjelaskan hubungan antara *timing, slow start* dan *congestion avoidance*.



Gambar 2.8 Mekanisme *Fast Recovery*

Sumber : Ghassan Akram Abed (2013)

Mekanisme *fast recovery* diterapkan dalam TCP Reno (Floyd, 2004), hal tersebut untuk menggantikan fase *slow start* dan *congestion avoidance* yang dapat mengurangi ukuran *congestion window* menjadi setengah. Mekanisme *congestion control* TCP Reno terdiri dari empat prosedur yaitu: *slow start, congestion avoidance, fast retransmit* dan *fast recovery.* Dua tahap pertama digunakan oleh TCP sumber untuk mengatur jumlah paket yang dimasukkan ke dalam *bandwidth* dan dua prosedur terakhir digunakan untuk memulihkan diri dari *packet loss* tanpa menunggu RTO (Abrahamsson dkk, 2002). Gambar 2.10 menjelaskan interaksi *slow-start, congestion avoidance* dan *fast recovery* .



Gambar 2.9 Interaksi *Slow Start, Congestion Avoidance*, dan *Fast Recovery*

Sumber : Ghassan Akram Abed (2013)

Algoritma *fast recovery* dikembangkan di TCP New Reno yang ditujukan saat terjadi *multiple packet loss*, yang akan meningkatkan *throughput* saat banyak *packet loss* dalam *single window*. Oleh karena itu, perbedaan antara Reno dan New Reno adalah Reno menunggu RTO untuk setiap paket kemudian menuju fase *slow start*, sementara New Reno meningkatkan proses *recovery.* Padahal, kinerja Reno terpengaruh ketika banyak *packet loss* dalam *window* yang sama (Subedi dkk, 2008).

2.2.5 TCP Vegas

TCP Vegas adalah hasil modifikasi dari TCP Reno. Dimana TCP Vegas lebih pada *packet delay* dari pada *packet loss*. Tujuan dari TCP Vegas yaitu mendeteksi *congestion* sebelum *packet loss* terjadi dan ketika *packet loss* terjadi, pengiriman paket akan berubah menjadi *linear*. Dalam prosesnya TCP vegas memiliki prinsip dasar *congestion avoidance, fast retransmit* dan *slow start* (Brakmo dan Peterson, 2006).

2.2.5.1 *Congestion Avoidance*

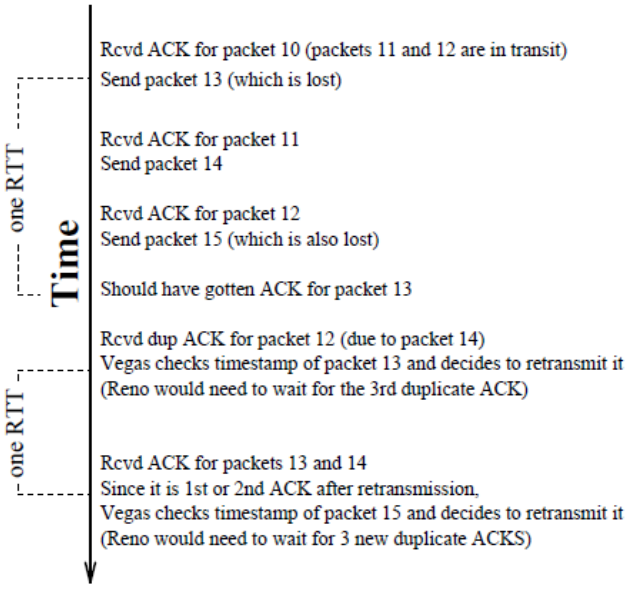
Dalam fase ini dilakukan perhitungan RTT dalam jaringan. Dengan langkah-langkah sebagai berikut:

1. Menentukan *base* RTT, dimana *base* RTT merupakan RTT terkecil yang diukur selama koneksi.

2. Menghitung *expected rate*, dimana *expected rate* merupakan transmisi yang diharapkan. *Expected rate = congestion window / Base* RTT.
3. Menghitung *actual rate. Actual rate = congestion window* / RTT.
4. Menghitung perbedaan (*diff*) antara *expected rate* dan *actual rate*. Dengan hasil tersebut *congestion window* akan dirubah untuk mengatasi *congestion* dan untuk menghindari *packet loss.*
   diff = expected-actual.
5. Menentukan parameter *alfa* dan *beta*, dimana *alfa* merupakan batas ambang bawah dan *beta* merupakan batas ambang atas.
   - Jika *diff < alfa*, *congestion window* + 1.
   - Jika *diff > beta*, *congestion window – 1.*

2.2.5.2 *Fast Retransmission*

Dalam fase *fast retransmission* TCP Vegas merekam waktu dari setiap data yang di kirimkan. Ketika ACK diterima, waktu direkam dan mulai melakukan perhitungan RTT dari waktu itu. Selanjutnya TCP Vegas menggunakan perhitungan RTT tersebut untuk melakukan *retransmission* dengan dua situasi. Contoh mekanisme retransmission dapat dilihat pada Gambar 2.11.



Gambar 2.10 Ilustrasi Mekanisme Fast Retransmission

Sumber : Brakmo dan Peterson (2006)

1. Ketika menerima duplikat ACK, TCP Vegas menghitung waktu terbaru dan waktu tercatat, jika lebih besar dari *timeout* maka akan dilakukan *retransmission* tanpa menunggu adanya tiga duplikat ACK.

2. Ketika tanpa ada duplikat ACK, TCP Vegas akan melihat interval waktu sejak data pertama dikirim, jika lebih besar dari *timeout* maka akan dilakukan *retransmission*.

Setiap *packet loss* yang terjadi sebelum *congestion window* turun terakhir kali, tidak dapat dijadikan acuan terjadinya *congestion. Congestion window* akan turun seperempat ketika terjadi kondisi *retransmission* tersebut.

2.2.5.3 *Slow start*

Dalam fase *slow start* ini, kenaikan pengiriman paket data terjadi secara eksponensial hanya untuk dua RTT yang berbeda. Ketika *actual rate* kurang dari *expected rate* maka pengiriman data akan berubah menjadi *linear.*

2.2.6 TCP New Reno

TCP New Reno dikembangkan dari TCP Reno, dengan mengabaikan fase *slow start* ketika terjadi *congestion* dengan adanya tiga duplikat *acknowledgement*, yag dilanjutkan ke fase *fast recovery*. Dalam prosesnya TCP New Reno memiliki prinsip dasar *slow start, congestion avoidance, fast retrasmission* dan *fast recovery* (Lar dkk, 2011).

2.2.6.1 Slow Start

Dalam fase *slow start* dilakukan proses pengiriman paket data pada jaringan. Pertama yang dikirim hanya satu paket data dan menunggu ACK dari TCP tujuan, setelah itu paket data yang dikirimkan meningkat secara eksponensial. Kenaikan secara eksponensial ini akan berhenti jika terjadi *packet loss* karena ACK belum diterima.

2.2.6.2 *Congestion Avoidance*

Dalam fase *congestion avoidance*, TCP berusaha menghindari *congestion.* Di fase ini, *congestion window* akan naik secara *linear* dan ketika TCP sumber menerima tiga duplikasi *acknowledgement* dari TCP tujuan, nilai *congestion window* akan turun menjadi setengah.

2.2.6.3 *Fast Retransmission*

Dalam fase *fast retransmission*, dilakukan peningkatan untuk mengurangi waktu tunggu TCP sumber sebelum mengirim ulang paket yang hilang. TCP sumber menggunakan *timer* yang digunakan untuk mengetahui segmen yang hilang. Jika *acknowledgement* dari suatu paket tidak diterima dalam kurun waktu tertentu, TCP sumber berasumsi bahwa paket tersebut hilang saat proses pengiriman, selanjutnya TCP sumber akan melakukan pengiriman ulang segmen yang hilang tersebut. *Fast retransmission* didasari oleh duplikat *acknowledgement*, dimana proses *fast retransmission* dapat dijelaskan sebagai berikut:

1. Saat TCP sumber mengirim paket dengan *sequence number* 1 ke TCP tujuan, TCP tujuan akan mengirimkan *acknowledgement* ke TCP sumber disertai *sequence number* 2 yang berarti TCP sumber sudah menerima *sequence number* 1 dan meminta paket dengan *sequence number* 2.

2. Setelah TCP sumber mengirim paket dengan *sequence number* 2, diasumsikan paket tersebut hilang, dan TCP tujuan meminta paket dengan *sequence number* 3 dan 4.
3. Selanjutnya TCP sumber mengirim paket dengan *sequence number* 3 dan 4, lalu TCP tujuan hanya mengirimkan *acknowledgement* disertai paket dengan *sequence number* 2 dan meminta paket dengan *sequence number* 5.
4. Selanjutnya TCP sumber mengirim paket dengan *sequence number* 5, lalu TCP tujuan hanya mengirim *acknowledgement* disertai paket dengan *sequence number* 2.
5. Karena TCP sumber sudah menerima 3 duplikat *acknowledgement* dengan *sequence number* 2, maka TCP sumber menilai paket dengan *sequence number* 2 telah hilang dan segera melakukan proses *retrasnmission.*

2.2.6.4 *Fast Recovery*

Dalam fase *fast recovery*, dilakukan pengembangan oleh TCP New Reno terhadap TCP Reno. Dalam fase ini TCP Reno hanya mampu mengatasi *single error* pada paket data, namun jika terjadi *multiple error* TCP Reno akan gagal dan akan kembali ke fase *slow start*. Sedangkan dalam fase ini, TCP New Reno dapat menangani *multiple error* pada paket data. Namun jika paket data *error* yang berderet lebih dari dua maka terjadi *timeout* dan akan kembali ke fase *slow start*. Dalam fase ini dapat menjaga *throughput* tetap tinggi ketika terjadi *congestion.*

2.2.7 Antrian

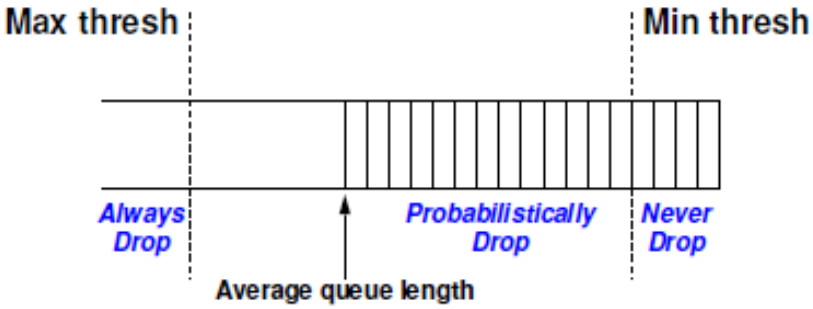
Antrian merupakan paket yang sedang menunggu untuk dilayani. Antrian memiliki artian beberapa paket yang datang dan masuk dalam *buffer* menunggu untuk dilayani dan diproses kemudian ditransmisikan. Dalam antrian yang padat dibutuhkan manajemen dalam antrian tersebut untuk memilih mana paket yang akan diterima*,* ditandai dan di *drop* atau di transmisikan. Dalam penelitian ini digunakan dua manajemen antrian yaitu *Random Early Detection* dan *Droptail*.

2.2.7.1 *Random Early Detection (RED)*

Menurut Sungur, A (2015) *Random Early Detection* adalah mekanisme antrian yang bisa melakukan *drop* paket sebelum kapasitas *buffer* penuh atau sebelum terjadi *congestion* bahkan kelumpuhan pada jaringan dengan cara menandai paket untuk di *drop* secara *random. Random Early Detection* juga merupakan manajemen antrian yang memanfaatkan mekanisme TCP *congestion control* untuk menjaga antrian dengan paket data serendah mungkin. Tidak seperti *Droptail, Random Early Detection* mampu menggunakan algoritmanya untuk memilih secara *random* paket data mana yang akan di *drop* sebagai tanda terjadi *congestion*. Bila *average queue size* mencapai *threshold* yang telah ditentukan, *Random Early Detection* memberitahukan *congestion* dengan menjatuhkan paket di *router*. Dalam *Random Early Detection* terdapat parameter yang akan dijelaskan pada sub-bab dibawah ini.

1. Parameter *Random Early Detection*

Parameter adalah faktor yang menentukan keberhasilan mekanisme *Random Early Detection*. Parameter utama yang digunakan untuk menghitung probabilitas *packet drop* adalah *min thresh, max thresh, avg* dan *p.* Pertama, *min thresh* dan *max thresh* harus ditentukan terlebih dahulu untuk menggunakan *Random Early Detection*. Keberhasilan parameter terletak pada menentukan *average queue size* (*avg*), nilai *min thresh* dan *max thresh*. Penggunaan *link* secara berlebihan harus dihindari untuk mencegah jatuhnya terlalu banyak paket. Jika *avg < min thresh* maka paket akan dilayani. Jika *avg > max thresh* maka semua paket yang masuk akan di *drop*. Jika *min thresh < avg <max thresh,* maka paket di *drop* dengan probabilitas *p*. Ilustrasi *Random Early Detection* dapat dilihat pada Gambar 2.12.



Gambar 2.11 Antrian *Random Early Detection*

Sumber : Bill Cheng (2012)

Parameter yang diatur dalam *Random Early Detection* harus mengakomodasi *bandwidth* yang bervariasi. Agar *Random Early Detection* memberikan kinerja jaringan yang optimal, aturan berikut harus diterapkan ketika mengatur parameter dengan berbagai kondisi lalu lintas yaitu sebagai berikut:

1. *Average queue size* harus dihitung dengan menetapkan w_q, paling sedikit 0,001.

2. Untuk memaksimalkan kinerja jaringan, *min thresh* harus diatur cukup tinggi sehingga *average queue size* tidak terlalu rendah. Dengan jaringan yang banyak peningkatan *traffic* data, *average queue size* yang terlalu rendah akan menyebabkan antrian menjadi terlalu cepat padat.

3. Ukuran *buffer* antara *min thresh* dan *max thresh* jangan terlalu besar sehingga kemungkinan untuk menandai atau men *drop* paket yang masuk tidak terlalu tinggi.

2. Menentukan Parameter W_q

W_q adalah rata-rata pergerakan eksponensial yang digunakan *Random Early Detection* untuk menghitung *average queue size* dan *q* adalah ukuran antrian saat itu. Rata-rata yang dihitung harus mencerminkan ukuran antrian rata-rata saat ini dan harus dijaga tetap dibawah *max thresh*. Mengatur parameter w_q yang terlalu tinggi atau terlalu rendah bisa langsung mempengaruhi cara *avg* merespon

perubahan ukuran antrian sebenarnya. Jika w_q terlalu tinggi, maka algoritma akan menjadi tidak berguna, karena *congestion* sementara tidak akan terdeteksi, oleh karena itu deteksi *congestion* akan terlambat dan kinerja akan sama dengan *Droptail*. Jika w_q terlalu rendah, maka tahap awal *congestion* tidak akan terdeteksi, diperkirakan *average queue size* merespons terlalu lambat terhadap *congestion* sementara. W_q direkomendasikan untuk diatur setidaknya 0,001 dalam jaringan nyata dan 0,002 pada NS-1 dan NS-2 dengan batas atas 0.0042 oleh karena itu: $0.001 \leq wq \leq 0.0042$.

3. Menentukan *Min Thresh* Dan *Max Thresh*

Perbedaan antara *min thresh* dan *max thresh* harus cukup besar untuk memungkinkan jumlah paket yang dikirim cukup banyak sebelum di *drop*. Jika perbedaan antara *min thresh* dan *max thresh* terlalu kecil, maka *congestion* akan terlambat untuk terdeteksi dan antriannya akan mencapai atau hampir mencapai ukuran *buffer* maksimum, hal itu akan seperti *Droptail. Min thresh* harus diatur dengan menghitung *delay* antrian tertinggi dan mengkalikan dengan *bandwidth*. *Throughput* akan turun jika *min thresh* diatur terlalu kecil dan jika diatur terlalu tinggi maka *delay* akan turun. Jika transmisi data antar *link* lambat, maka perbedaan antara *min thresh* dan *max thresh* menjadi lebih tinggi lagi. Disarankan *min thresh* harus di atur setidaknya lima paket atau lima kali rata-rata ukuran paket dalam satuan *byte*. Disarankan *max thresh* tidak diatur lebih tinggi dari 0,1 seperti pengaturan *default* pada NS-2.

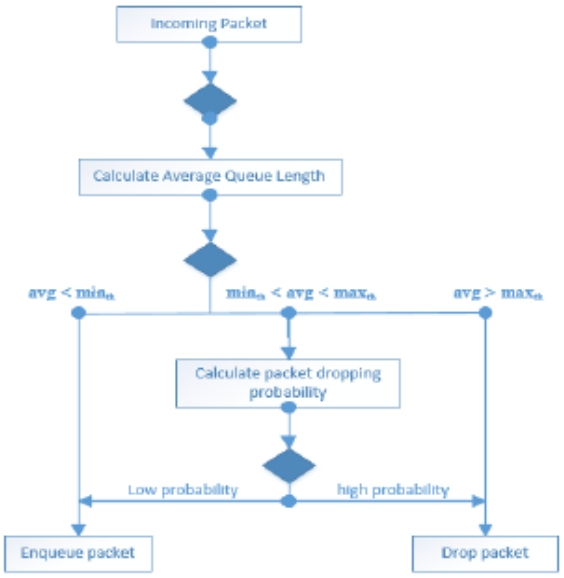
4. Menentukan *Average Queue Lenght*

Average queue size dihitung dari setiap setiap paket yang datang. Yang digunakan untuk menghitung *average queue size* adalah *exponential weighted moving average* (w_q) (EWMA). Perhitungan *weighted moving average* ini dihitung berdasarkan *average queue lenght* dari pada panjang antrian saat itu karena memberikan gambaran yang lebih baik tentang *congestion*. Jika sumber memperlambat transmisi paket pada perhitungan yang dilakukan dengan menggunakan panjang antrian saat itu, diketahui bahwa antrian bisa dengan cepat menjadi kosong dan penuh lagi. Jika diasumsikan *average queue size* adalah nol dan meningkat oleh paket *L* dengan setiap paket yang datang.

5. Algoritma *Random Early Detection*

Ada dua sub-algoritma yang terdapat dalam algoritma *Random Early Detection* yang bekerja untuk mengendalikan *average queue size*. Untuk menghindari peningkatan *traffic* data, bagian pertama dari algoritma ini diperlukan untuk menghitung *average queue size*. *Average queue size* dihitung saat antrian kosong dengan membuat asumsi bagaimana caranya banyak paket yang bisa ditransmisikan selama waktu *idle*. Bagian kedua dari algoritma ini digunakan untuk menandai paket secara *random* diantara *min thresh* dan *max thresh.* Saat rata-rata lebih besar dari atau sama dengan *max thresh* dan jika probabilitas *packet drop* yang dihitung tinggi maka paket akan di *drop* dan transmisi paket data akan diperlambat, sebaliknya jika rata-rata lebih rendah atau sama dengan *min thresh* dan probabilitas *packet drop* yang dihitung rendah maka

paket tidak akan di *drop*. Gambar 2.13 menunjukkan diagram algoritma *Random Early Detection*.



Gambar 2.12 Algoritma *Random Early Detection*

Sumber : Asli Sungur (2015)

Algoritma *Random Early Detection* dapat ditunjukkan dengan mengukur paket dalam antrian. Pada Gambar 2.13 menunjukkan algoritma *Random Early Detection* yang dapat dijabarkan sebagai berikut.

Keterangan:
- *avg* : *average queue size*
- *time* : waktu saat ini
- *count* : jumlah paket sejak terakhir *packet drop*
- w_q : *queue weight*
- *minthresh* : batas bawah antrian
- *maxthresh* : batas atas antrian
- max_p : nilai maksimum probabilitas *packet drop* p_b
- p_a : probabilitas *packet drop* saat ini
- p_b : probabilitas *packet drop*
- *q* : *queue size* saat ini
- f : fungsi linear waktu
- *m* : jumlah paket kecil

A) *Average queue size* adalah nol.

B) Antrian dalam keadaan *idle* (kosong).

C) Hitung *average queue size* dengan jumlah paket *L* yang datang dengan persamaan 2.1.

$$avg_L = L+1+ (1-w_q)^{L+1}-1 / w_q \quad (2.1)$$

Jika antriannya tidak kosong maka,

1) Gunakan persamaan 2.2 untuk menghitung *average queue size avg.*

$$avg = (1 - w_q)\, avg + w_q q \quad (2.2)$$

Jika antriannya kosong (*idle*) maka,

2) Perkirakan jumlah paket *m* yang bisa ditransmisikan saat periode *idle* dengan menggunakan persamaan 2.3.

$$m = f(time\text{-}q_{time}) \quad (2.3)$$

3) Setelah periode *idle,* antrian menghitung *average queue size,* jika ada paket *m* datang gunakan persamaan 2.4.

$$\mathrm{avg} = (1\text{-}\mathrm{w_q})^{\mathrm{m}}\, \mathrm{avg} \quad (2.4)$$

Jika *min thresh < avg < max thresh* maka,

4) Tingkatkan jumlah paket yang di *drop* sejak terakhir paket yang di *drop.*
5) Hitung probabilitas *drop pa.*
6) Hitung probabilitas paket yang ditandai *pb* dari 0 sampai max_p dan *avg* dari *min thresh* sampai *max thresh*, menggunakan persamaan 2.5.

$$\mathrm{P_b} = \mathrm{max_p}\,(\mathrm{avg} - \mathrm{min_{th}})/(\mathrm{max_{th}} - \mathrm{min_{th}}) \quad (2.5)$$

7) Hitung probabilitas *drop* dengan menggunakan hasil *pb* dari persamaan 2.6.

$$\mathrm{Pa} = \mathrm{Pb}/(1 - \mathrm{count} * \mathrm{Pb}) \quad (2.6)$$

Jika probabilitas drop paket yang dihitung di langkah 7 rendah maka,

8) *Enqueue* paket dan jangan *drop.*

Jika probabilitas *drop* paket yang dihitung persamaan 2.6 tinggi dan mendekati *max thresh* maka,

9) *Drop* paket secara acak dan hitung paket sejak *drop* terakhir, lalu *reset* menjadi 0.

Jika *avg > max thresh* maka,

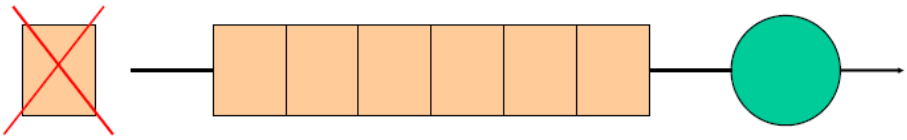
10) *Drop* semua paket yang masuk dan tetapkan jumlah paket sejak *drop* terakhir ke nol.

Jika *avg < min thresh* maka,

11) *Enqueue packet.*

2.2.7.2 *Droptail*

Menurut Kumar dkk (2012), Algoritma FIFO digunakan pada *Droptail* dengan mekanisme paket yang datang pertama akan dilayani dan ditransmisikkan pertama, namun ketika *buffer* penuh maka paket yang datang akan di *drop*. Hal itu artinya, *Droptail* akan melakukan *drop* pada paket yang datang sampai *buffer* memiliki ruang untuk paket baru. Ilustrasi manajemen antrian *Droptail* dapat dilihat pada Gambar 2.14.



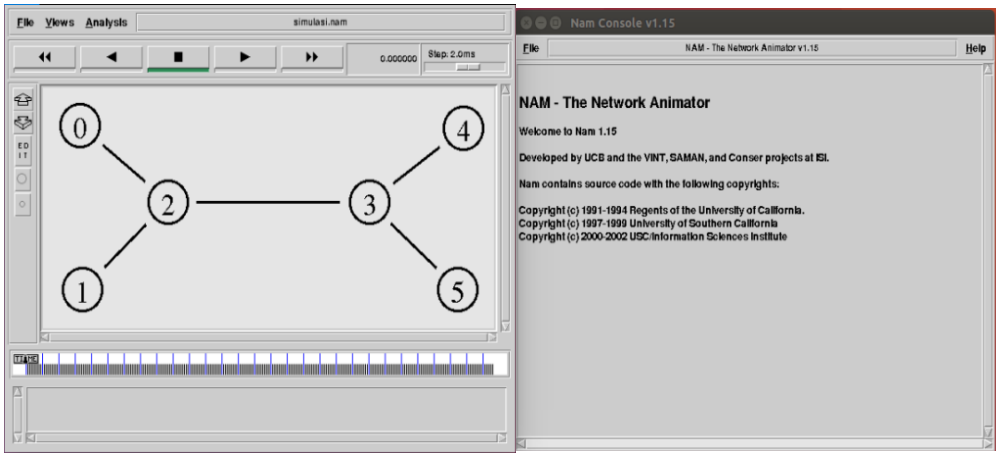
Gambar 2.13 Antrian *Droptail*

Sumber : Peterson dan Davie (2007)

Dalam gambar 2.14 menjelaskan antrian *Droptail* yang menggunakan mekanisme FIFO, dimana jika *buffer* penuh paket data yang datang akan di *drop.*

2.2.8 *Network Simulator* 2

Network Simulator 2 (NS-2) adalah *software* yang dibuat untuk keperluan penelitian. NS-2 merupakan *software* yang bersifat *open source* dibawah *Gnu Public Licence* (GPL), hal tersebut menyebabkan *software* ini dapat diunduh dengan gratis melalui *website* NS-2 (The Network Simulator, 2017). NS-2 menggunakan C++ dan Tcl/Otcl sebagai bahasa pemrogramannya. Dimana C++ sebagai *library* dan Tcl/Otcl sebagai *script* simulasi yang di buat oleh pengguna. Visualisasi dari simulasi menggunakan *Network Animator (nam*). Sedangkan hasil dari simulasi merupakan *file trace* berekstensi *.tr. File* tersebut yang dianalisa untuk mendapatkan parameter hasil. Menganalisa *file trace* tersebut dapat dilakukan dengan menggunaan *script* berekstensi *.wkt* untuk memudahkan dalam analisis hasil.



Gambar 2.14 *Interface* NAM

Pada gambar 2.15 memperlihatkan tampilan dari *network animator*. Dimana setelah kita merancang suatu simulasi dalam *file* berekstensi *.tcl*, maka hasilnya dapat dilihat pada *network animator.*

Type Identifier	Time	Source Node	Destination Node	Packet Name	Packet Size	Flags	Flow ID	Source Address	Destination Address	Sequence Number	Packet Unique ID

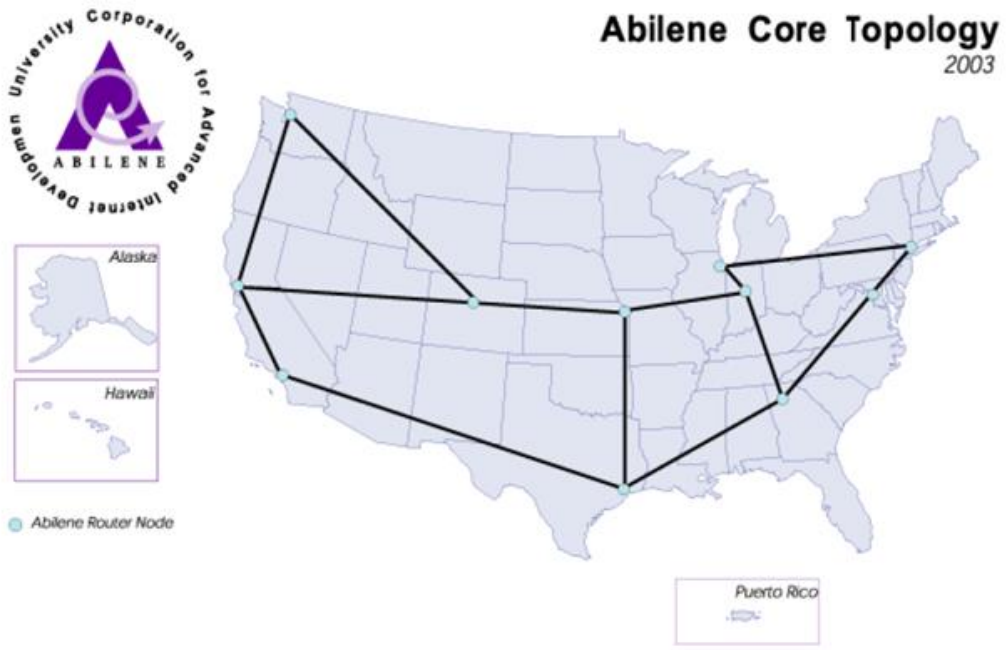
Gambar 2.15 *File Trace*

Sumber : Teerawat Issariyakul dan Ekram Hossain (2012)

Pada gambar 2.16 memperlihatkan kolom-kolom hasil *record* yang ada pada *file trace*, dimana didalamnya terdapat *type identifier, time, source node* dan lain sebagainya yang nanti akan di analisa lebih lanjut untuk mendapatkan hasil sesuai parameter hasil yang telah ditentukan.

2.2.9 Topologi Abilene

Topologi *Abilene* merupakan suatu topologi *backbone* dengan kinerja tinggi yang di ciptakan oleh *Internet Community* dengan *Network Operations Center* (NOC) di Universitas Indiana pada akhir tahun 1990. Namun saat ini topologi *Abilene* sudah tidak digunakan lagi, karena pada tahun 2007 telah ditingkatkan menjadi *Internet2 Network*. *Abilene* di ambil dari salah satu nama stasiun yang ada di kota Kansas, karena topologi *Abilene* adalah implementasi dari rute kereta api yang terdapat di Amerika. Tujuan di buatnya topologi *Abilene* yaitu untuk mencapai 10 Gbps konektivitas antar *node* dan 100 Mbps konektivitas antar *host* dan *node*. *Abilene* merupakan *private network* yang diperuntukkan bagi penelitian serta pendidikan, namun biasanya tidak hanya itu saja, karena akses alternatif akan diberikan ke banyak sumber daya melalui *public network*.



Gambar 2.16 Topologi *Abilene*

Sumber : Jehn-Ruey Jiang (2014)

2.2.10 Quality Of Service (*QoS*)

Kinerja sebuah jaringan dapat bervariasi akibat beberapa masalah seperti *packet delivery ratio, throughput, delay, packet loss* yang dapat membuat efek pada suatu jaringan. Kinerja suatu jaringan dapat diukur baik dan buruknya dengan *Quality of Service* (*QoS*). *QoS* merupakan suatu teknologi pada jaringan komputer yang diterapkan untuk memberi layanan yang maksimal untuk semua pengguna jaringan komputer(Parasian, 2014). Dalam penelitian ini akan menguji parameter QoS yaitu *packet delivery ratio, throughput, delay dan packet loss.*

2.2.10.1 *Packet Delivery Ratio*

Packet delivery ratio merupakan perbandingan jumlah paket yang diterima *node* tujuan dan jumlah paket yang dikirimkan oleh *node* sumber dalam kurun waktu tertentu. Persamaan 2.7 untuk mencari *Packet delivery ratio*.

$$Packet\ Delivery\ Ratio = \frac{paket\_diterima}{paket\_dikirim} x\ 100\% \quad (2.7)$$

2.2.10.2 *Throughput*

Throughput merupakan banyaknya paket yang tiba atau diterima tujuan dalam interval waktu tertentu. Efektifitas dan efisiensi kinerja protokol dan kinerja jaringan diukur menggunakan *throughput*. Persamaan 2.8 untuk mencari *throughput*.

$$Throughput = \frac{paket\_berhasil\_diterima}{jumlah\_waktu\_digunakan} x\ ukuran\ paket \quad (2.8)$$

2.2.10.3 *Delay*

Delay merupakan waktu yang dibutuhkan suatu paket saat dikirimkan dari *node* sumber ke *node* tujuan. *Delay* dapat dipengaruhi oleh jarak, waktu proses yang lama, media fisik bahkan *congestion* jaringan. Persamaan untuk mencari *Delay* adalah sebagai berikut:

$$Delay = \frac{waktu\_paket\_[i]\_diterima - waktu\_paket\_[i]\_dikirim}{jumlah\_paket\_dikirim} \quad (2.9)$$

Keterangan : i = nomor paket yang berhasil diterima

2.2.10.4 *Packet Drop*

Packet drop merupakan paket yang terbuang saat proses trasmisi dari *node* sumber ke tujuan. Terjadi karena adanya *congestion* sehingga *buffer* penuh dan manajemen antrian yang berada pada *router* melakukan *drop*. Persamaan 3.0 untuk mencari *Packet drop*.

$$Packet\ Drop = \frac{paket\_dikirim - paket\_diterima}{paket\_dikirim} x\ 100\% \quad (3.0)$$