

PENGATURAN PROSES DALAM SISTEM TERDISTRIBUSI

Yustina Sri Suharini

ABSTRACT

Beyond its advantages in load sharing, communication performance, availability, utilizing special capabilities; distributed processes has its nature problems. Distributed global state is a major problem due to unsynchronized local clock and time lag. Anyway, distributed global state has to be defined since it is the key-factor to avoid deadlock and starvation in one side and force mutual exclusion at the other side. Many algorithms have been dedicated to solve these problems. This paper discuss process migration, distributed global state, distributed mutual exclusion, and distributed queue along with its algorithms.

1. MIGRASI PROSES

Salah satu komponen yang penting dalam sistem terdistribusi adalah migrasi proses. Migrasi merupakan pengalihan sebagian keadaan proses dari satu mesin ke mesin yang lain. Konsep migrasi memiliki dua karakteristik penting yaitu adanya kemampuan untuk mengambil alih sebuah proses dalam suatu mesin, kemudian mengaktifkan proses tersebut pada mesin lain.

Berikut ini adalah beberapa alasan mengapa migrasi proses perlu dilakukan.

- *Load Sharing*
Pemindahan proses dari sebuah mesin yang bebannya padat ke mesin lain yang bebannya ringan akan meningkatkan performansi sistem secara keseluruhan.
- *Communication Performance*
Proses-proses yang berinteraksi secara intensif dapat dipindahkan agar mengurangi biaya komunikasi selama mereka berinteraksi. Pemindahan proses kadang lebih menguntungkan dibandingkan dengan pemanggilan proses secara berulang-ulang.
- *Availability*
Perpindahan dibutuhkan oleh proses-proses yang berlangsung lama, untuk menjaga agar proses tetap berjalan apabila terjadi kesalahan atau gangguan pada mesin tertentu.
- *Utilizing Special Capabilities*
Proses dapat dipindahkan ke sebuah sistem yang mempunyai karakteristik khusus, yang mana karakteristik tersebut tidak dipunyai oleh sistem lain.

Topik-topik yang penting untuk dibicarakan dalam mendesain fasilitas migrasi antara lain adalah siapa yang menginisiasi migrasi, berapa besar proses yang dimigrasikan, dan bagaimana *messages* dan *signals* berperan di dalamnya.

1.1. Inisiasi Migrasi

Siapa yang melakukan inisiasi migrasi adalah tergantung pada tujuan migrasi. Jika tujuan migrasi adalah untuk keseimbangan sistem secara keseluruhan, maka kapan migrasi dilakukan tergantung pada beberapa modul dalam sistem operasi yang sedang memonitor beban sistem global. Sebuah modul akan bertanggung jawab atas pengambilalihan proses atau pensinyalan proses yang akan dimigrasi. Untuk menentukan di mana proses akan dimigrasikan, modul tersebut berhubungan dengan modul-modul sejenis dalam sistem lain sedemikian sehingga pola beban pada sistem lain dapat diketahui. Jika tujuan migrasi adalah mendapatkan sumber daya tertentu, maka proses dapat langsung memigrasi diri sendiri tanpa harus diatur oleh modul sistem.

1.2. Apa yang Dimigrasi ?

Apabila suatu proses dimigrasi, maka proses yang berada dalam sistem asal akan dihapus, dan proses tersebut diciptakan pada sistem tujuan. Dengan kata lain, migrasi merupakan suatu perpindahan proses, dan bukan penyalinan proses dari sistem asal ke sistem tujuan. Bagian proses yang dimigrasi adalah *process image*, yang berisi paling tidak *process control blok*. Di samping itu link-link di antara proses itu dan proses-proses yang lain, seperti *passing messages* dan *passing signals*, harus diperbaharui (*update*). Tugas untuk memindahkan *process control blok* dan memperbaharui link merupakan tanggung jawab sistem operasi. Gambar 1 memperlihatkan sebuah contoh keadaan sistem sebelum dan sesudah pelaksanaan migrasi.

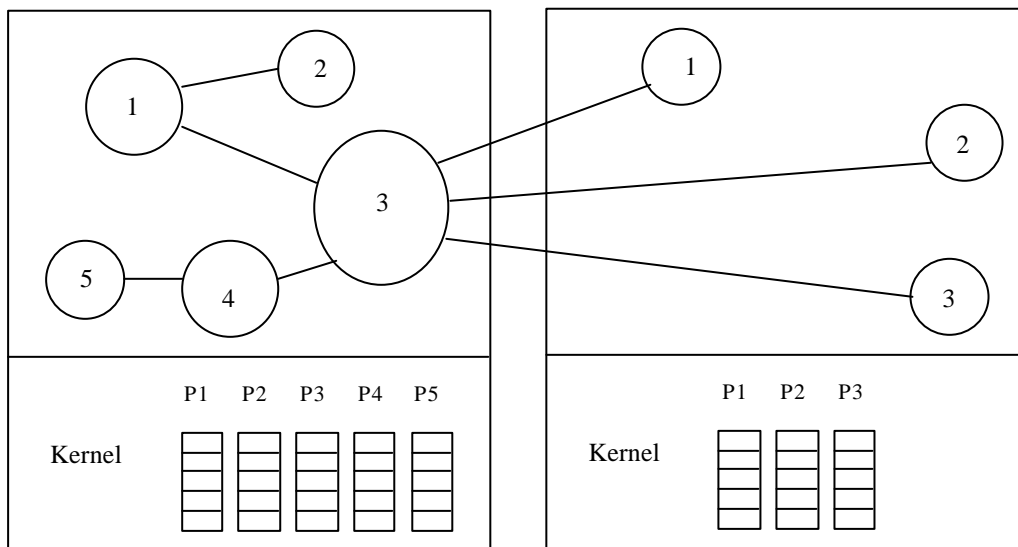
1.3. Migrasi Diri Sendiri

Urutan peristiwa yang terjadi pada saat suatu proses memutuskan untuk bermigrasi (memigrasi dirinya sendiri) adalah sebagai berikut.

1. Ketika suatu proses memutuskan untuk bermigrasi proses memilih mesin tujuan dan mengirim *message* ke mesin tersebut. *Message* membawa sebagian dari *process image* dan informasi tentang open file.
2. Di sisi penerima, kernel membuat proses anak (*child*), kemudian memberikan informasi yang telah diterima kepada proses anak.
3. Proses baru yang telah tercipta mulai mengidentifikasi data, lingkungan, argument, informasi stack, sebanyak yang dibutuhkan. Teks program disalin atau dijadikan sistem file yang global.
4. Sebagai akhir dari migrasi, proses asli diberi sinyal tanda selesai migrasi. Proses akan mengirim *message* terakhir kepada proses baru, kemudian menghapus dirinya sendiri.

1.4. Migrasi oleh Modul Sistem Global

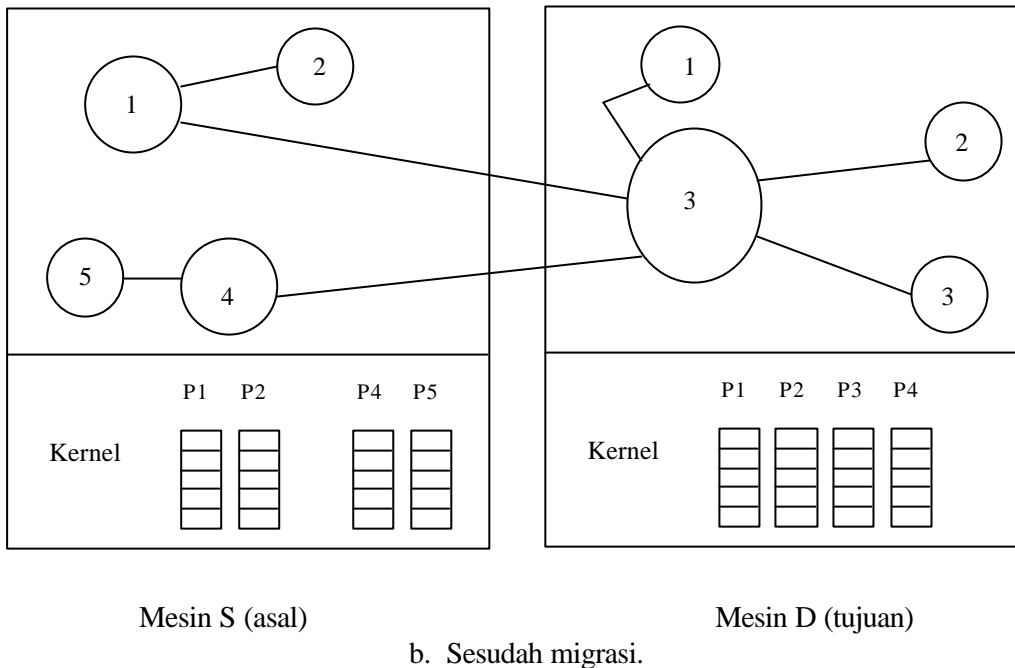
Aspek lain migrasi yang juga penting antara lain berhubungan dengan keputusan tentang migrasi. Pada beberapa kasus, keputusan seperti itu dibuat oleh entitas tunggal. Sebagai contoh, jika tujuan migrasi adalah mencapai keseimbangan beban sistem, maka sebuah modul akan memantau beban relatif pada berbagai mesin dan membuat migrasi.



Mesin S (asal)

Mesin D (tujuan)

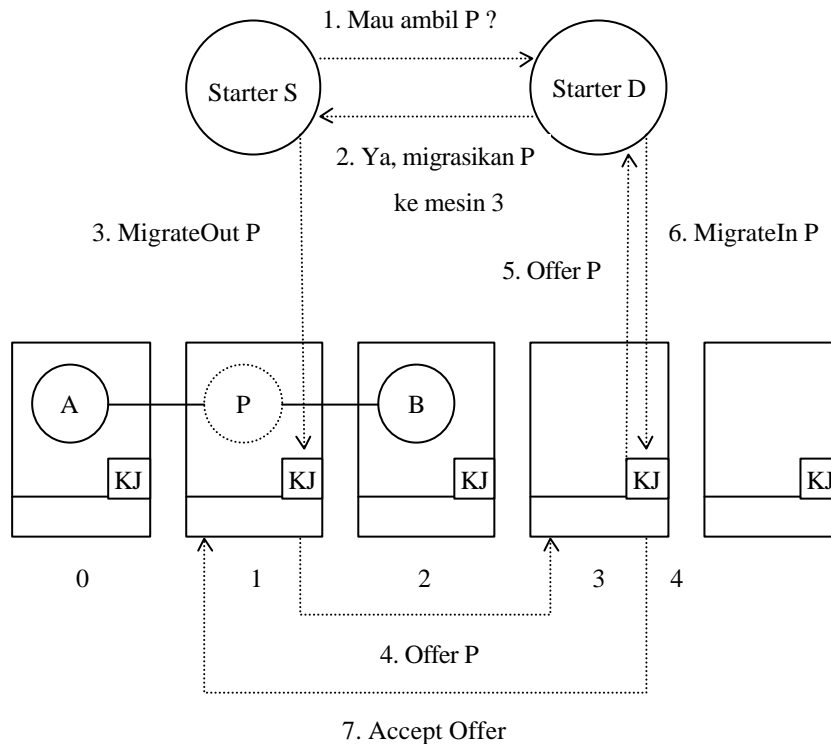
a. Sebelum migrasi.



Gambar 1. Keadaan sistem sebelum dan sesudah migrasi.

Berikut ini merupakan langkah-langkah yang terjadi jika migrasi yang dilakukan bertujuan untuk menjaga keseimbangan beban sistem secara keseluruhan.

1. Starter yang mengontrol sistem asal (S) memutuskan bahwa proses (P) akan dimigrasi ke sistem tujuan (D). Starter S mengirim message berisi permintaan transfer ke starter D.
 2. Jika starter D disiapkan untuk menerima proses, maka ia mengirim balik *acknowledgment* positif.
 3. Starter S memberitahukan migrasi tersebut kepada kernel S melalui *service call*.
 4. Kernel S menawarkan pengiriman proses ke D. Penawaran berisi data P.
 5. D bisa menolak atau menerima tawaran dari kernel S. D menolak apabila ukuran proses tidak sesuai dengan kemampuannya. Apabila D menerima maka ia meneruskan tawaran tersebut kepada starternya.
 6. Starter D akan membuat keputusan, keputusan itu kemudian diberikan kepada D.
 7. D menyediakan sumber daya yang dibutuhkan untuk mencegah *deadlock* dan masalah *flow-control* yang lain, dan kemudian mengirim sinyal terima kepada S.
- Gambar 2 disajikan untuk memperjelas langkah-langkah negosiasi migrasi.



Gambar 2. Negosiasi migrasi.

1.5. Pengusiran (*Eviction*)

Negosiasi proses memperbolehkan sistem tujuan (S) untuk menolak tawaran migrasi. Di samping itu sistem tujuan juga bisa mengusir proses yang telah dimigrasikan kepadanya. Kasus seperti itu bisa terjadi misalnya bila sebuah *workstation* tiba-tiba *down*, padahal dia sudah menerima migrasi beberapa proses dari workstation lain.

Berikut adalah hal-hal penting yang merupakan komponen mekanisme pengusiran.

1. Monitor pada masing-masing *node* mendeteksi aktivitas *workstation's console*, sehingga bisa memberikan inisiasi pengusiran kepada masing-masing proses asing yang migrasi.
2. Jika proses diusir, dia akan dimigrasi balik kepada sistem asal atau ke sistem lain yang memungkinkan.
3. Semua proses yang ditandai dengan label pengusiran akan di-*suspend* secepatnya.
4. *Entire address space* untuk proses yang diusir dikirim balik ke *home node*.

1.6. Perpindahan Preemptive dan Nonpreemptive

Migrasi proses *preemptive* merupakan perpindahan proses yang dieksekusi secara parsial, minimum proses tersebut telah selesai dibuat dan bisa jadi sebagian proses telah

dieksekusi. Migrasi proses semacam ini membutuhkan transfer keadaan (*state*) proses.

Sedangkan migrasi proses *nonpreemptive* adalah perpindahan proses yang mana proses telah selesai dibuat tetapi sama sekali belum dieksekusi. Tentu saja migrasi proses jenis ini lebih sederhana, karena tidak membutuhkan transfer keadaan proses.

Pada kedua jenis transfer di atas, informasi tentang lingkungan di mana proses-proses akan dieksekusi, harus ditransfer ke *remote node*, yaitu ke sistem yang dituju. Informasi ini meliputi direktori aktif di mana *user* sedang bekerja dan hak-hak khusus yang dimiliki oleh proses yang bersangkutan.

Untuk keseimbangan beban sistem, biasa digunakan migrasi proses *nonpreemptive*, karena migrasi tsb mempunyai keuntungan dalam hal menghindari migrasi yang *full-blown*, yaitu proses dimigrasi secara total (100%), tanpa memperhitungkan besarnya proses. Salah satu kelemahan skema migrasi *nonpreemptive* adalah jika ada perubahan distribusi beban yang mendadak, sistem tsb. kurang reaktif.

2. KEADAAN GLOBAL TERDISTRIBUSI (*Distributed Global State*)

Semua topik konkurensi seperti *mutual exclusion*, *deadlock*, dan *starvation*, juga terdapat dalam sistem yang terdistribusi. Tidak adanya keadaan global (*global state*) untuk seluruh sistem menyebabkan :

1. Desain sistem terdistribusi menjadi sangat rumit.
2. Tidak mungkin sistem operasi dan proses manapun dalam sistem tersebut bisa mengetahui keadaan sistem secara keseluruhan.

Sebuah proses hanya bisa mengetahui keadaan semua proses dalam sistem lokal, dengan mengakses *process control block* di memori. Proses tersebut dapat mengetahui keadaan proses lain yang jauh dari sistem lokal (*remote process*) melalui *messages*, yang menunjukkan keadaan suatu proses beberapa saat yang lalu.

Keterlambatan waktu (*time lag*) yang terjadi pada sistem terdistribusi merupakan hambatan utama bagi topik-topik yang berhubungan dengan konkurensi.

Masalah yang timbul adalah *global state* yang sesungguhnya tidak dapat ditentukan karena keterlambatan waktu pada saat pemindahan *message*. State atau keadaan sebuah proses adalah urutan dari *message* yang telah dikirim/diterima melalui *channel* yang sesuai untuk proses tersebut. *Channel* bisa dianggap sebagai jalur transfer pesan yang mempunyai satu arah, arah tersebut menuju ke proses yang dituju. *Channel* timbul di antara dua proses jika proses-proses tersebut bertukar *messages*. Jika dua proses saling menukarkan *messages*, maka dibutuhkan dua *channel*. *Global state* yang dimaksud merupakan kombinasi keadaan seluruh proses, atau keadaan global. Pendefinisian sebuah keadaan global dapat dilakukan dengan mengumpulkan seluruh catatan (*snapshot*) dari semua proses. *Snapshot* dapat diidentikkan dengan sebuah catatan singkat tentang keadaan setiap proses. Setiap *snapshot* terdiri atas daftar seluruh *messages* yang dikirim dan diterima melalui seluruh *channel* mulai dari *snapshot* terakhir. Sedangkan *distributed snapshot* adalah kumpulan *snapshot*, satu dari tiap proses yang ada pada sistem terdistribusi.

Dikehendaki *distributed snapshot* mencatat keadaan global yang konsisten, yaitu keadaan global yang mana setiap kondisi proses yang mencatat *message* yang diterima, akan mengirim *message* lain yang menyatakan bahwa *message* telah diterima.

Untuk mengetahui keadaan global sistem terdistribusi, dikembangkan algoritma untuk *Distributed Snapshot*. Algoritma mengasumsikan bahwa *message* yang dikirim sesuai dengan urutan pengiriman, dan tidak ada *message* yang hilang. Algoritma menggunakan kontrol *message* yang dinamakan **marker**. Beberapa proses menginisiasi algoritma dengan jalan mencatat state-nya dan mengirim sebuah marker pada semua *channel* keluar sebelum *message*

berikutnya dikirim. Setiap proses p yang menerima *marker* dari proses q , mengikuti langkah-langkah berikut.

1. p mencatat seluruh kondisi lokal sp .
2. p mencatat keadaan channel masukan dari q ke p
3. p meneruskan *marker* ke semua proses di sekitarnya melalui seluruh channel keluaran.

Langkah-langkah tersebut harus dilakukan secara atomik, artinya tidak ada *message* yang dikirim / diterima oleh p sampai ketiga langkah dilakukan. Apabila setelah mencatat *state*-nya ternyata p menerima *marker* dari *channel* lain misalnya dari *channel* r , maka p akan mencatat keadaan *channel* dari r ke p sebagai urutan *message* yang telah diterima p dari r mulai dari p mencatat sp sampai dengan p menerima *marker* dari r . Algoritma berhenti apabila sebuah proses telah menerima semua *marker* dari seluruh *channel* masukan.

Hasil pengamatan [ANDR90] tentang algoritma tersebut.

1. Setiap proses boleh mengawali algoritma dengan mengirim sebuah *marker*.
2. Algoritma akan berhenti pada saat tertentu jika setiap *message* termasuk *marker* didistribusikan pada saat-saat tertentu.
3. Algoritma ini adalah algoritma terdistribusi : setiap proses bertanggung jawab untuk mencatat keadaan dirinya sendiri dan keadaan seluruh *channel* yang masuk.
4. Jika seluruh *state* sudah dicatat (algoritma telah selesai pada semua proses), global state konsisten yang diperoleh dari algoritma ini dapat di-*assemble* pada setiap proses dengan menghitung seluruh proses yang :
 - mengirim data keadaan yang tercatat pada setiap *channel* keluaran
 - meneruskan data keadaan yang menerima setiap *channel* keluaran

Pada saat inisiasi dapat dihitung proses yang ada untuk menentukan keadaan globalnya.

5. Algoritma ini tidak mempengaruhi atau dipengaruhi oleh algoritma terdistribusi yang lain yang mana proses-proses sedang berpartisipasi di dalamnya.

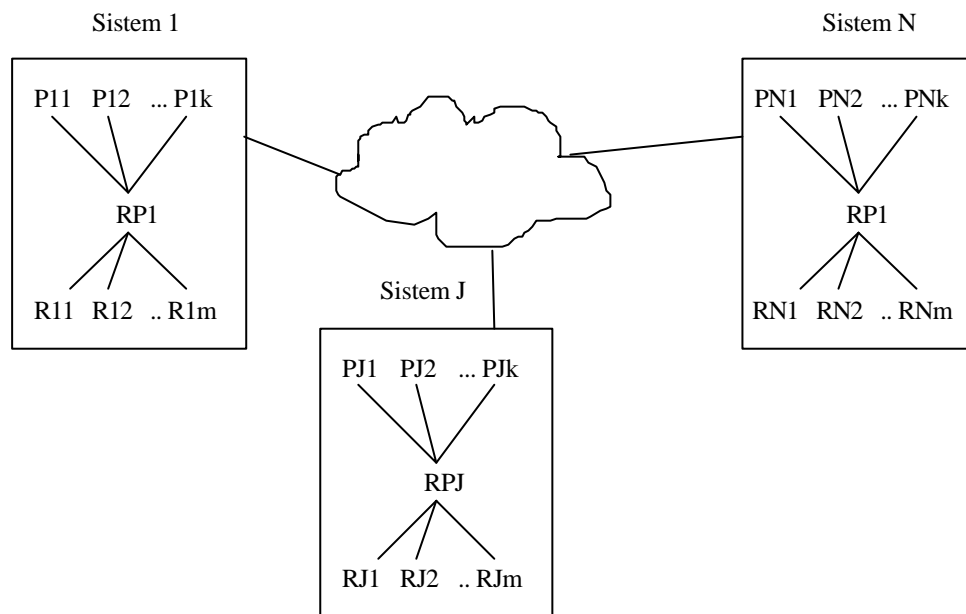
Algoritma tersebut merupakan alat yang fleksibel dan dapat digunakan untuk mengadaptasi setiap algoritma terpusat pada lingkungan terdistribusi karena dasar dari setiap algoritma terpusat adalah pengetahuan tentang keadaan global sistem.

3. DISTRIBUTED MUTUAL EXCLUSION

Pada saat dua proses atau lebih bersaing untuk mendapatkan sumber daya sistem, mau tidak mau harus ada suatu mekanisme untuk memaksa keadaan *mutual exclusion*. Dalam satu waktu hanya ada satu proses yang diijinkan berada pada seksi kritis. Contoh yang baik misalnya penggunaan sumber daya printer.

Keberhasilan konkurensi di antara banyak proses tergantung pada kemampuan untuk menentukan seksi kritis dan memaksakan *mutual exclusion*. Setiap fasilitas yang mendukung *mutual exclusion* harus mempunyai syarat-syarat berikut.

1. *Mutual exclusion* harus dipaksakan. Dalam satu waktu hanya ada satu proses yang diijinkan berada pada seksi kritis.
2. Proses yang berhenti (*halt*) pada seksi non kritis tidak boleh mengganggu proses lain.
3. Sebuah proses yang memerlukan akses ke seksi kritis tidak boleh menunggu dalam waktu yang tidak terbatas : tidak boleh ada *deadlock* atau *starvation*.
4. Pada saat tidak ada proses yang berada dalam seksi kritis, setiap proses yang hendak masuk ke seksi kritis harus diijinkan masuk ke seksi kritis tanpa tunda waktu.
5. Tidak ada asumsi yang dibuat untuk kecepatan proses relatif atau jumlah prosesor.
6. Sebuah proses berada pada seksi kritis hanya pada waktu yang tertentu.



Gambar 3. Model mutual exclusion pada sistem terdistribusi.

Gambar 3 merupakan salah satu model *mutual exclusion* pada konteks terdistribusi. Algoritma yang dibuat untuk mewujudkan keadaan mutual exclusion pada sistem jaringan dapat berupa algoritma terpusat atau algoritma terdistribusi.

3.1. Algoritma Mutual Exclusion Terpusat

Dalam sebuah algoritma yang terpusat, satu buah node akan digunakan sebagai node kendali dan akses kendali untuk seluruh seluruh obyek yang digunakan bersama. Jika suatu proses memerlukan akses ke sumber daya kritis, proses tersebut mengirimkan sebuah

permintaan ke proses pengendali sumber daya lokalnya. Proses pengendali sumber daya lokal mengirimkan *message* permintaan ke *node* kendali. Jika sumber daya ternyata bisa dipakai, maka *node* kendali akan mengirim balik *message* yang berisi ijin penggunaan sumber daya. Jika proses telah selesai menggunakan sumber daya, ia akan mengirimkan *release message* ke *node* kendali.

Algoritma terpusat mempunyai dua sifat utama :

1. Hanya *node* kendali yang menentukan alokasi sumber daya.
2. Semua info yang dibutuhkan terkonsentrasi pada *node* kendali, termasuk identitas dan status alokasi setiap sumber daya.

Pendekatan terpusat bersifat langsung dan mudah dilihat. Namun apabila terjadi kegagalan pada *node* kendali, maka mekanisme *mutual exclusion* tidak dapat dijalankan sama sekali. Di samping itu setiap alokasi sumber daya dan dealokasinya memerlukan pertukaran *message* dengan *node* kendali. Ini dapat mengakibatkan *bottle-neck* pada *node* kendali.

3.2. Algoritma Mutual Exclusion Terdistribusi

Karena permasalahan yang timbul pada algoritma terpusat, maka dikembangkan suatu algoritma terdistribusi. Algoritma terdistribusi mempunyai karakteristik sebagai berikut.

1. Semua *node* memiliki jumlah informasi yang sama, yaitu sejumlah rata-ratanya.
2. Setiap *node* hanya memiliki gambaran parsial dari sistem keseluruhan, dan harus bisa membuat keputusan berdasarkan informasi yang hanya sebagian itu.
3. Setiap *node* memiliki tanggung jawab yang sama untuk keputusan akhir.
4. Setiap *node* mengeluarkan usaha yang sama dalam mencapai putusan akhir, yaitu sejumlah rata-ratanya.
5. Kegagalan satu *node* pada umumnya tidak mengakibatkan kegagalan sistem keseluruhan.
6. Muncul kondisi di mana sistem tidak mempunyai clock yang berlaku umum untuk setiap sistem, yang mana clock tersebut dibutuhkan untuk mengatur waktu setiap *event*.

Algoritma terdistribusi mensyaratkan bahwa semua informasi yang diketahui oleh setiap *node* dikomunikasikan ke setiap *node* yang lain. Namun demikian, pada saat-saat tertentu beberapa informasi tersebut akan berada dalam kondisi transit dan belum mencapai *node* yang lain. Jadi, karena keterlambatan waktu dalam komunikasi, informasi dari *node* biasanya tidak lengkap dan terbaru, dalam hal ini hanyalah berupa informasi sebagian.

Karena keterlambatan komunikasi di antara sistem, tidak mungkin mempertahankan sebuah clock agar dapat digunakan untuk seluruh sistem. Di samping itu, sangat sulit menggunakan sebuah clock sentral dan mempertahankan seluruh clock lokal sinkron dengan clock sentral, karena selama pemakaian akan terjadi pergeseran di antara clock lokal yang beragam yang akan mengakibatkan ketidaksinkronan.

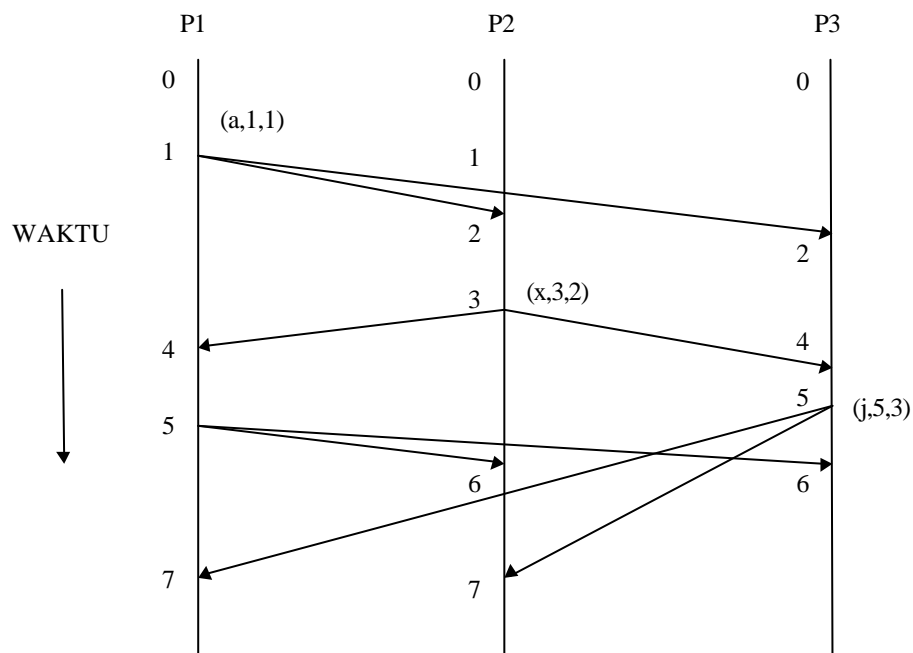
Adanya penundaan dalam komunikasi dan tidak adanya clock yang seragam, membuat mekanisme *mutual exclusion* sulit dikembangkan dalam sistem terdistribusi, dibandingkan dengan sistem terpusat.

3.3. Pengurutan Kejadian dalam Sistem Terdistribusi

Dasar operasi dari kebanyakan algoritma terdistribusi adalah pengurutan kejadian secara temporer. Misal diinginkan untuk menyatakan bahwa kejadian a dalam sistem i terjadi sebelum atau sesudah kejadian b dalam sistem j, dan diinginkan kejelasan urutan kejadian seluruh sistem pada jaringan, maka ada dua hambatan utama yang dihadapi. Pertama, mungkin terjadi

penundaan di antara sistem lain, kedua tidak adanya sinkronisasi yang mengakibatkan pembacaan clock di sebuah sistem dan sistem yang lain berbeda-beda.

Lamport [LAMP78] mengusulkan pengurutan kejadian tanpa menggunakan clock fisik. Dalam suatu sistem terdistribusi, cara interaksi proses-proses adalah melalui *message*. Oleh karena itu sebuah kejadian lokal dapat diidentikkan dengan sebuah *message*. Sebagai contoh sebuah proses dapat mengirimkan *message* jika akan memasuki atau meninggalkan seksi kritis. Untuk menghilangkan ambiguitas, kejadian didefinisikan sebagai pengiriman *message*. Jadi jika suatu proses mengirimkan *message*, maka sebuah kejadian dicatat.



Gambar 4. Contoh operasi algoritma stempel waktu.

Skema stempel waktu bertujuan untuk mengurutkan kejadian yang berisi transmisi-transmisi *message*. Setiap sistem i pada jaringan mempunyai pencacah lokal C_i , yang berfungsi sebagai clock. Setiap kali sistem mentransmisikan sebuah *message*, sistem tersebut menambah nilai clock dengan 1. *Message* dikirim dalam bentuk :

$$(m, T_i, i)$$

di mana :

m : isi *message*

T_i : stempel waktu untuk *message* ini, sama dengan C_i

i : identifikasi numerik dari lokasi / situs tertentu.

Pada saat *message* diterima, sistem yang menerima, j , mengatur clocknya menjadi satu lebih banyak daripada nilai maksimum. Clock pada saat itu dan stempel waktu yang datang adalah :

$$C_j = 1 + \text{maksimum} [C_j, T_i]$$

Pada setiap situs, pengurutan situs dilakukan sebagai berikut. Untuk *message* x dari situs i dan y dari situs j , x dikatakan mendahului y jika salah satu kondisi di bawah ini terpenuhi :

1. Jika $T_i < T_j$ atau
2. Jika $T_i = T_j$ dan $i < j$

Pada Gambar 4, tiap *message* dikirim dari satu proses ke proses lain. Jika beberapa proses tidak dikirim dengan cara ini, beberapa situs tidak menerima seluruh pesan dalam sistem, dan karenanya tidak mungkin seluruh situs memiliki urutan *message* yang sama.

4. ANTRIAN TERDISTRIBUSI (*Distributed Queue*)

Salah satu pendekatan yang diusulkan untuk menciptakan keadaan mutual exclusion terdistribusi didasarkan pada konsep antrian terdistribusi. Ada tiga algoritma yang disajikan sebagai alternatif penyelesaian persoalan mutual exclusion dengan konsep antrian.

4.1. Algoritma Tiga Pesan

Algoritma didasarkan pada asumsi berikut.

1. Sebuah sistem terdistribusi terdiri atas N *node*, masing-masing bernomor 1 sampai N . Tiap *node* berisi sebuah proses yang membuat permintaan untuk akses secara *mutual exclusion* ke sumber daya mewakili proses-proses yang lain. Proses ini juga menjadi penengah bagi permintaan-permintaan yang datang pada saat yang bersamaan.
2. *Message* yang dikirim dari satu proses ke proses yang lain diterima sesuai dengan urutan pengirimannya.
3. Tiap *message* diteruskan ke tujuannya dalam sejumlah waktu yang tertentu.
4. Jaringan terhubung penuh, artinya tiap proses dapat mengirim *message* secara langsung ke proses yang lain tanpa membutuhkan proses perantara dalam pengirimannya.

Tiga jenis *message* yang digunakan dalam algoritma ini adalah :

- (request, T_i, i) : sebuah permintaan untuk akses ke sumber daya yang dibuat oleh P_i .
- (reply, T_j, j) : P_j memberi akses ke sumber daya yang ada di bawah penguasaannya.
- (release, T_k, k) : P_k melepas sumber daya yang sebelumnya dialokasikan untuk P_k .

Algoritmanya adalah sebagai berikut.

1. Saat P_i memerlukan akses ke sebuah sumber daya, maka P_i mengirimkan (request, T_i, i), distempel waktu dengan clock lokal saat itu. P_i meletakkan *message* ini pada *array*-nya sendiri di $q[i]$ dan mengirim *message* ini ke semua proses yang lain.
2. Ketika P_j menerima (request, T_i, i), P_j meletakkan *message* ini di *array*-nya (yaitu $q[i]$) dan mengirimkan (reply, T_j, j) ke seluruh proses.

3. P_i bisa memasuki seksi kritis sumber daya jika kedua kondisi berikut dipenuhi :
 - *Message* permintaan P_i dalam array q adalah *message* permintaan paling dahulu dalam *array*. Karena *message* telah diurutkan secara konsisten di seluruh situs, maka aturan ini hanya memperbolehkan satu proses mengakses sumber daya pada setiap waktu.
 - Semua *message* di array lokal lebih belakangan dibandingkan *message* pada $q[i]$. Aturan ini menjamin bahwa P_i telah mendeteksi seluruh permintaan yang mendahului permintaan terbarunya.
4. P_i melepas sebuah sumber daya dengan mengirimkan (*release*, T_i, i) yang diletakkan dalam *array*-nya sendiri dan mengirim *release* ini ke seluruh sistem.
5. Jika P_i menerima (*release*, T_j, j), P_i mengganti isi $q[j]$ dengan *message* (*release*, T_j, j).
6. Jika P_i menerima (*reply*, T_j, j), P_i mengganti isi $q[j]$ dengan (*reply*, T_j, j).

4.2. Algoritma Dua Pesan

Algoritma Dua Pesan berupaya mengoptimasikan algoritma Tiga Pesan dengan jalan menghilangkan *release message*. Asumsi yang digunakan sama dengan asumsi tiga pesan kecuali bahwa *message* yang dikirim dari satu proses ke proses yang lain tidak harus diterima dengan urutan yang sama dengan urutan pada saat dikirim.

Algoritmanya adalah sebagai berikut.

1. Jika P_i memerlukan akses ke sumber daya, P_i mengeluarkan permintaan (*request*, T_i, i), distempel waktu dengan clock lokalnya. P_i meletakkan *message* ini dalam *array*-nya di $q[i]$ dan mengirim *message* ke proses yang lain.
2. Saat P_j menerima (*request*, T_i, i), P_j mengikuti langkah-langkah berikut ini.
 - Jika P_j sedang berada di seksi kritisnya, P_j menunda pengiriman *message* balasan.
 - Jika P_j tidak sedang menunggu *request* untuk masuk ke seksi kritis, P_j mengirim (*reply*, T_j, j) ke seluruh proses.
 - Jika P_j sedang menunggu untuk masuk ke seksi kritis dan jika *message* yang diterima lebih lambat dibanding dengan *request* P_j , P_j memasukkan *message* tersebut dalam *array* $q[i]$ dan menunda pengiriman *message* jawaban.
 - Jika P_j menunggu untuk memasuki seksi kritis, dan jika *message* yang masuk lebih dahulu daripada permintaan P_j , maka P_j meletakkan *message* tersebut dalam *array* $q[i]$ dan mengirim (*reply*, T_j, j) ke P_i .
3. P_i dapat mengakses sumber daya (berada pada seksi kritis) jika P_i telah menerima *reply message* dari seluruh proses yang lain.
4. Jika P_i telah meninggalkan seksi kritis, P_i melepas sumber daya dengan mengirim *reply message* ke setiap request yang ditunda.

4.3. Pendekatan dengan Penyerahan Token (A Token-Passing Approach)

Pendekatan ini mempunyai prinsip penyerahan token di antara proses-proses yang berperan. Token adalah *entity* yang pada suatu saat dipegang oleh suatu proses. Proses yang memegang token dapat memasuki seksi kritis tanpa meminta ijin. Bila proses meninggalkan seksi kritis, maka ia akan menyerahkan token tersebut ke proses yang lain.

Token yang diserahkan oleh satu proses kepada proses lain sebenarnya adalah *array* yang mana elemen ke- k mencatat stempel waktu terakhir saat token masuk proses P_k . Tiap proses mempunyai *array*, *request*, yang mana elemen ke- k mencatat stempel waktu *request* terakhir yang diterima dari P_j .

Langkah-langkahnya adalah sebagai berikut.

1. Awalnya, sebuah token diberikan sembarang pada salah satu proses dengan membuat *token-present* berharga *true* untuk proses yang bersangkutan.
2. Ketika sebuah proses hendak menggunakan seksi kritisnya, proses yang bersangkutan dapat melakukannya jika proses tersebut memiliki token. Jika tidak, proses tersebut akan mengirim

request dengan stempel waktu ke seluruh proses yang lain dan menunggu sampai proses tersebut memperoleh token.

3. Ketika proses P_i meninggalkan seksi kritis, maka P_i mengirimkan token yang dimilikinya ke proses lain. P_i memilih proses berikutnya yang akan menerima token berdasarkan urutan dalam *array* dengan urutan :

$$i+1, i+2, \dots, 1, 2, \dots, j-1$$

untuk permintaan masuk pertama [j] sedemikian sehingga stempel waktu permintaan P_j yang terbaru untuk token lebih besar daripada nilai yang tercatat pada token di P_j yang berisi nilai pemegang token yang terakhir atau $\text{request}[j] > \text{token}[j]$

Algoritma tersebut memerlukan satu dari dua hal berikut.

- N buah message (N-1 dikirimkan untuk request dan 1 untuk dikirim ke token) saat proses yang meminta ijin tidak memiliki token.
- 0 buah message saat proses memiliki token.

5. PENUTUP

Secara umum, pengaturan proses terdistribusi jauh lebih rumit daripada pengaturan proses pada satu mesin atau pengaturan proses secara terpusat. Namun pengaturan proses dalam sistem terdistribusi tetap dipelajari dan dikembangkan karena secara teoritis pendekatan terdistribusi jauh lebih menguntungkan daripada pendekatan yang lain.

Konsep-konsep yang ada pada sistem terdistribusi tidak semuanya telah diimplementasikan dan dibakukan untuk digunakan user pada level aplikasi. Sebagai contoh konsep migrasi, yang sebenarnya telah lama diselidiki oleh para ahli, pada kenyataannya konsep tersebut sangat sulit diimplementasikan. Faktor penyebabnya antara lain adalah harga yang sangat tinggi dibanding dengan manfaat yang dirasakan pada level aplikasi. Oleh karena itu konsep migrasi sampai saat ini baru sampai pada tahap penelitian. Walaupun demikian, ada baiknya semua teori sistem terdistribusi yang ada saat ini dipelajari agar gagasan awal yang pernah muncul tidak hilang tetapi justru dapat dikembangkan sehingga suatu saat teori-teori tersebut dapat diimplementasikan dan dibakukan untuk kepentingan bersama.

DAFTAR PUSTAKA

- Tanenbaum, A.S., 1995, *Distributed Operating Systems*, Prentice-Hall, Inc.
 Nutt, G.J., 1992, *Centralized and Distributed Operating Systems*, Prentice-Hall, Inc.
 Beveridge, J.E., Robert Wiener, 1997, *Multithreading Applications in Win32 : The Complete Guide to Threads*, Addison Wesley Developers Press.
 Stallings, W., 1992, *Operating Systems*, Macmillan, Inc.
 Stallings, W., 1995, *Operating Systems*, Prentice-Hall, Inc.