ITS
Institut
Teknologi
Sepuluh Nopember

TUGAS AKHIR - CI1599

SIMULASI SISTEM NAVIGASI TRAFFIC LALU LINTAS BERBASIS TEKNOLOGI MOBILE AGENT

MAHAR FAIQU RAHMAN
NRP 5102 100 085

Dosen Pembimbing
Waskitho Wibisono S.Kom, M.Eng
Ary Mazharuddin S., S.Kom

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2007

ITS
Institut
Teknologi
Sepuluh Nopember

FINAL PROJECT - CI1599

SIMULATION OF TRAFFIC NAVIGATION SYSTEM BASED ON MOBILE AGENT TECHNOLOGY

MAHAR FAIQU RAHMAN
NRP 5102 100 085

Lecturer
Waskitho Wibisono S.Kom, M.Eng
Ary Mazharuddin S., S.Kom

DEPARTEMENT OF INFORMATICS ENGINEERING
Faculty of Information Technology
Institute Technology of Sepuluh Nopember
Surabaya 2007

SIMULASI SISTEM NAVIGASI TRAFFIC LALU LINTAS BERBASIS TEKNOLOGI MOBILE AGENT

Nama Mahasiswa : MAHAR FAIQU RAHMAN
NRP : 5102 100 085
Jurusan : Teknik Informatika FTIf-ITS
Dosen Pembimbing : WASKITHO WIBISONO S.Kom, M.Eng
ARY MAZHARUDDIN S. S.Kom

ABSTRAK

Sistem navigasi traffic lalu lintas, adalah merupakan suatu sistem yang memudahkan bagi sesorang untuk mendapatkan informasi mengenai rute atau jalur yang akan dilewati dari suatu lokasi asal, ke lokasi tujuan dalam suatu daerah tertentu. Sistem ini dapat membantu seseorang dalam mencari suatu lokasi tertentu yang berada pada suatu daerah.

Di dalam Tugas Akhir ini dikembangkan suatu sistem navigasi traffic lalu lintas dengan berbasis pada teknologi mobile agent. Teknologi mobile agent adalah suatu teknologi yang dikembangkan dari konsep dasar suatu agent, dimana agent tersebut dapat berpindah dari satu host ke host yang lain. Di dalam sistem ini diimplementasikan juga konsep multiagent, yaitu suatu sistem yang terdiri dari berbagai macam agent yang secara spesifik menangani proses-proses tertentu yang ada di dalam sistem.

Agent yang ada di dalam sistem ini dibagi menjadi empat, yaitu Client Agent yang berhubungan dengan interface user, Query Agent yang menerima request path dari user, Graph Agent yang melakukan komputasi penghitungan rute terpendek dari local area, dan Virtual Agent yang melakukan komputasi penghitungan rute terpendek dari global area.

Keunggulan dari dikembangkannya sistem ini adalah dapat mengurangi waktu yang dibutuhkan oleh client untuk mendapatkan data path. Dan juga dapat digunakan sebagai alternatif di dalam melakukan pemrosesan data dan komputasi secara terdistribusi.

Kata Kunci: multiagent system, mobile agent, graph, djikstra

[Halaman Ini Sengaja Dikosongkan]

SIMULATION OF TRAFFIC NAVIGATION SYSTEM BASED ON MOBILE AGENT TECHNOLOGY

Name	**: MAHAR FAIQU RAHMAN**
NRP	**: 5102 100 085**
Departement	**: Teknik Informatika FTIf-ITS**
Lecturer	**: WASKITHO WIBISONO S.Kom, M.Eng**
	ARY MAZHARUDDIN S. S.Kom

ABSTRACT

Traffic navigation system, is a kind of system that can make someone easier to get information about a path from source location to the destination location in a certain area..This system can help someone to find a certain location in a certain place or area.

This Final Project developed a kind of traffic navigation system, based on mobile agent technology, that developed from the agent concept. Mobile Agent technology is a technology that developed from the base concept of an agent that can move from one host to another. This system also implemented the concept of multiagent system. Multiagent system, is a kind of system that have some agent that hold some specific process in the system.

The Agents in this system devide into four chategory.They are Client Agent that connected to user interface, Query agent that receive path request from user, Graph Agent that compute the shortest path in local area graph, and Virtual Agent that compute shortest path in global area graph.

The advantage of this system is can reduce the time that needed by client to get path data. Also can be use as an alternative in distributed data processing and computing.

Keywords: multiagent system, mobile agent, graph, djikstra

[Halaman Ini Sengaja Dikosongkan]

SIMULASI SISTEM NAVIGASI TRAFFIC LALU LINTAS BERBASIS TEKNOLOGI MOBILE AGENT

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
Pada
Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
MAHAR FAIQU RAHMAN
Nrp: 5102 100 085

Disetujui oleh Pembimbing Tugas Akhir

Waskitho Wibisono S.Kom, M.Eng .....
NIP: 132 256 272 (pembimbing 1)

Ary Mazharuddin Shiddiqi S.Kom .....
NIP: 132 309 748 (pembimbing 2)

SURABAYA
JANUARI, 2007

[Halaman Ini Sengaja Dikosongkan]

KATA PENGANTAR

Segala puji dan syukur Alhamdulillah dipanjatkan kehadirat Allah SWT, atas segala nikmat, karunia dan hidayah yang tak ternilai, sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul "***SIMULASI SISTEM NAVIGASI TRAFFIC LALU LINTAS BERBASIS TEKNOLOGI MOBILE AGENT***".

Tugas akhir ini disusun sebagai dan diajukan oleh penulis sebagai salah satu syarat untuk menyelesaikan program Strata 1 (S1) pada Jurusan Teknik Informatika, Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember Surabaya.

Dalam penyusunan tugas akhir ini, penulis berusaha untuk menerapkan ilmu yang telah didapat selama menjalani perkuliahan dengan tidak terlepas dari petunjuk, bimbingan, bantuan, dan dukungan dari berbagai pihak. Tidak lupa penulis mengucapkan banyak terima kasih kepada:

1. Ibu, Ayah, Kakak, dan Adik tercinta, yang tiada pernah berhenti memberikan dukungan, dan motivasi.
2. Bapak Waskitho Wibisono S.Kom, M.Eng selaku pembimbing I. Bapak Ary Mazharuddin Shidiqi S.Kom selaku pembimbing II. Ir. Bapak Joko Lianto Buliali, S.Kom, M.Sc. selaku Dosen Wali
3. Semua teman-teman yang sudah banyak membantu.

Dengan tidak melupakan kodratnya sebagai manusia, penulis menyadari bahwa dalam dalam Tugas Akhir ini masih mengandung berbagai kekurangan. Sehingga dengan segala kerendahan hati, penulis sangat mengharapkan berbagai kritik dan saran yang membangun guna pengembangan berikutnya.

Surabaya, Januari 2007

Penulis

[Halaman Ini Sengaja Dikosongkan]

DAFTAR ISI

[Halaman Ini Sengaja Dikosongkan]

DAFTAR GAMBAR

[Halaman Ini Sengaja Dikosongkan]

DAFTAR TABEL

[Halaman Ini Sengaja Dikosongkan]

BAB I

PENDAHULUAN

Dalam bab ini dijelaskan beberapa hal dasar yang meliputi latar belakang, permasalahan, batasan permasalahan, tujuan dan manfaat, metodologi pelaksanaan serta sistematika penulisan buku Tugas Akhir ini. Dari uraian tersebut diharapkan, gambaran umum permasalahan dan pemecahan yang diambil dapat dipahami dengan baik

1.1 Latar Belakang

Sistem navigasi *traffic* lalu lintas merupakan sistem yang dikembangkan untuk melakukan pencarian rute dari suatu lokasi ke lokasi yang lain. Sistem ini memberikan informasi kepada *user* berupa jalur atau rute yang harus dilewati dari lokasi asal menuju ke lokasi tujuan. Dengan dikembangkannya sistem navigasi ini, maka dapat membantu bagi *user* dalam mencari suatu lokasi yang berada pada area tertentu.

Di dalam sistem yang ada saat ini seorang *user* akan melakukan proses *request* pencarian lokasi ke satu buah *server* yang terpusat, dimana segala database yang menyimpan data *graph* suatu area, berada jadi satu pada sebuah *server*. Selain itu, di dalam sistem yang ada saat ini, proses komputasi pencarian rute suatu lokasi ditangani oleh satu buah *server* yang bertugas untuk melakukan pemrosesan data *graph*. Hal ini menjadi tidak efisien ketika terdapat banyak *client* yang melakukan *request* ke *server*. Selain itu *traffic* data yang ada di dalam jaringan yang menghubungkan *client* dan *server* akan menjadi sangat padat.

Di dalam Tugas Akhir ini dikembangkan suatu sistem navigasi *traffic* lalu lintas dengan memanfaatkan teknologi *mobile agent*, yaitu suatu teknologi yang berbasis pada *agent*, dimana *agent* tersebut dapat berpindah dari satu *host* ke *host* yang lain untuk mendapatkan suatu informasi atau data pada *host* tersebut Selain *mobile agent*, sistem ini juga memanfaatkan

teknologi *multiagent*, dimana terdapat banyak *agent* yang masing-masing memiliki fungsi dan tugas yang berbeda-beda untuk menangani proses-proses yang ada di dalam sistem ini. Semua agent yang ada terjadi saling komunikasi dan kerjasama untuk memperoleh satu buah *goal* atau tujuan utama dari sistem navigasi *traffic* lalu lintas ini. Sistem ini akan merubah pendekatan sistem terpusat menjadi sistem terdistribusi, dimana seluruh data graph dan penanganan *request path* yang ada akan di pecah ke dalam beberapa buah *server*.

1.2 Permasalahan

Permasalahan yang diangkat dalam penyelesaian tugas akhir ini adalah sebagai berikut:

- Bagaimana membuat suatu aplikasi *mobile agent* dengan menggunakan teknologi yang ada pada saat ini, serta *framework* apa yang menjadi pendukung bagi *mobile agent.*
- Bagaimana caranya suatu *agent* berkomunikasi dengan *agent* yang lainnya. Serta protokol apa yang digunakan dalam komunikasi tersebut.
- Bagaimana arsitektur yang dimiliki oleh keseluruhan *agent* dalam menyelesaikan permasalahan pada sebuah sistem navigasi *traffic* lalu lintas.
- Pembentukan perilaku-perilaku dan sifat-sifat suatu agent yang berkaitan dengan respon dari luar agent, dalam menangani suatu sistem navigasi *traffic* lalu lintas.
- Algoritma apa yang akan digunakan dalam melakukan komputasi untuk menentukan rute tercepat dari suatu lokasi ke lokasi yang lain, dalam sebuah sistem navigasi.

1.3 Batasan Permasalahan

Dalam sub-bab ini akan dijelaskan batasan-batasan permasalahan yang ada di dalam sistem.

1. Sistem ini tidak menangani pengambilan data secara *real-time*.
2. Data yang dipakai di dalam Tugas Akhir ini adalah data simulasi.
3. Dalam Tugas Akhir ini dibatasi tidak sampai menangani masalah *generate* data.
4. Tugas akhir ini ditekankan pada pembuatan sistem *mobile agent* sebagai alternatif dalam transfer data.
5. Sistem ini hanya bisa berjalan pada host yang memiliki *Java Virtual Machine*.
6. Sistem baru diuji pada lingkungan sistem operasi Windows.

1.4 Tujuan dan Manfaat

Tujuan dari tugas akhir ini adalah untuk membuat suatu simulasi sistem navigasi *traffic* dengan memanfaatkan teknologi *mobile agent*. Sehingga nantinya dapat digunakan untuk memperoleh *data* path yang di *request* oleh user, dari beberapa server yang melakukan komputasi dan pemrosesan data secara terdistribusi.

1.5 Metodologi

Pembuatan Tugas Akhir ini terdiri dari beberapa tahapan sebagai berikut:

a. Studi Literatur

Pada tahap ini dilakukan pengumpulan informasi – informasi yang diperlukan dalam proses perancangan dan implementasi sistem yang akan dibangun. Adapun informasi – informasi yang diperlukan diantaranya adalah *framework* apa yang akan digunakan dalam membangun suatu agent, algortima yang digunakan untuk penentuan rute terpendek dari satu lokasi ke lokasi yang lainnya.

b. Perancangan dan Implementasi Sistem

Pada tahap ini dilakukan perancangan dan implementasi sistem yang meliputi:

- Perancangan Arsitektur Sistem
  Perancangan arsitektur sistem adalah tahapan untuk menentukan model arsitektur sistem untuk membangun suatu sistem navigasi lalu lintas dengan berbasis pada *mobile agent*.
- Perancangan Data
  Yaitu perancangan format baku dalam pertukaran data antar agent, serta perancangan data *traffic* yang merupakan komponen-komponen dari graph yang akan diolah dalam proses pencarian rute terpendek.
- Perancangan Proses
  Yaitu perancangan proses pencarian data lokasi suatu tempat, yang merupakan proses *request* dari suatu *client* ke suatu *agent* untuk melakukan pencarian lokasi yang diingikan oleh *client* tersebut. *Agent* tersebut akan melakukan proses *request* ke *agent* yang lain (*server*) yang menangani proses pengolahan data *traffic*.
- Perancangan Antarmuka
  Yaitu perancangan antarmuka untuk memudahkan pengoperasian perangkat – perangkat lunak di dalam sistem yang akan dibangun.
- Implementasi
  Yaitu pembuatan perangkat lunak sesuai dengan perancangan yang telah dilakukan dengan menggunakan bahasa pemrograman Java J2SDK 1.50_2, serta *framework* JADE 3.4 sebagai *framework* untuk membangun agent

c. Uji Coba dan Analisis

Pada tahap ini dilakukan uji coba dengan skenario tertentu untuk mendapatkan percepatan sistem dan efektifitas dalam melakukan proses pencarian dan penghitungan data traffic.

d. Penyusunan Laporan Tugas Akhir

Penyusunan laporan Tugas Akhir ini berisi informasi – informasi mengenai sistem yang telah dibangun dan sekaligus merupakan tahapan akhir dari pelaksanaan Tugas Akhir.

1.6 Sistematika Penulisan

Buku Tugas Akhir ini terdiri dari beberapa bab yang tersusun secara sistematis, yaitu :

BAB I PENDAHULUAN

Bab ini merupakan gambaran umum yang membahas latar belakang dan tujuan pembuatan perangkat lunak, serta permasalahan yang dihadapi dalam pengerjaan Tugas Akhir ini. Selain itu dijelaskan pula pembatasan terhadap masalah yang akan dihadapi serta metodologi yang dipakai untuk menyelesaikannya.

BAB II TEORI PENUNJANG

Bab ini membahas teori-teori dasar yang menunjang pengerjaan Tugas Akhir, meliputi arsitektur *multi agent*, penjelasan mengenai *framework JADE*, konsep *graph* serta algoritma *Djikstra* dalam penentuan lokasi tercepat..

BAB III PERANCANGAN SISTEM

Dalam bab ini diuraikan mengenai sistem yang akan dibangun yakni mengenai perancangan arsitektur sistem, perancangan data beserta proses – proses yang ada, serta perancangan antarmuka untuk menjalankan perangkat – perangkat lunak yang berada dalam sistem.

BAB IV IMPLEMENTASI

Dalam bab ini diuraikan implementasi dari perancangan yang telah dilakukan sebelumnya yakni implementasi arsitektur sistem dengan pengadaan komputer – komputer *client* dan *server* beserta perangkat – perangkat lunaknya.

BAB V UJI COBA DAN ANALISIS

Berisi tentang uji coba yang telah dilakukan terhadap sistem yang telah dibuat beserta analisis dari hasil uji coba tersebut.

BAB VI PENUTUP

Berisi kesimpulan yang didapat dari pembuatan Tugas Akhir ini dan dilengkapi dengan saran untuk kemungkinan pengembangan selanjutnya.

BAB II

DASAR TEORI PENUNJANG

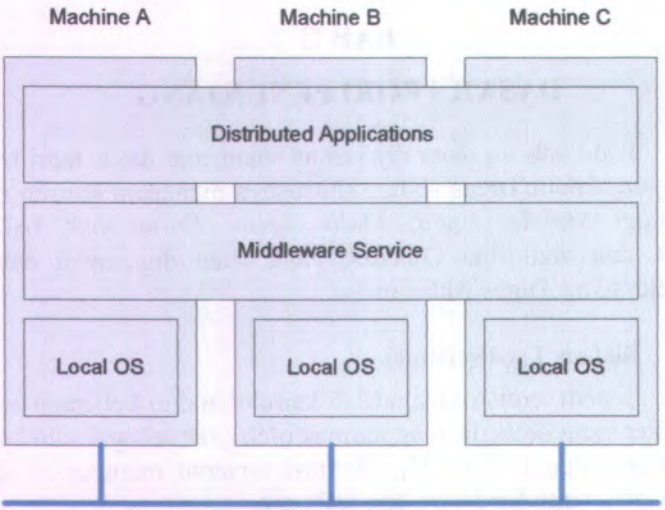
Pada bab ini akan dijelaskan mengenai dasar teori-teori penunjang dalam Tugas Akhir, khususnya mengenai konsep dari Teknologi *Mobile Agent*, *Multi Agent*, *Framework* JADE, *Graph*, dan algortima Djikstra, yang akan digunakan dalam menyelesaikan Tugas Akhir ini.

2.1 Sistem Terdistribusi

Sistem terdistribusi adalah kumpulan dari beberapa buah komputer yang berbeda yang tampak oleh *user* sebagai satu buah komputer tunggal [TAN02]. Definisi tersebut mempunyai dua buah aspek, yaitu hardware dan software.

Ada beberapa karakteristik penting dari sistem terdistribusi. Satu karaktersitik yang penting adalah *user* tidak dapat mengetahui cara berkomunikasi antara komputer yang satu dengan komputer lain yang berbeda. Selain itu karaktersitik yang lain adalah *user* dan aplikasi dapat berinteraksi dalam sistem terdistribusi dengan cara yang konsisten dan seragam, tergantung dari kapan dan dimana proses interaksi berlangsung.

Untuk mendukung penggunaan komputer yang berbeda-beda dalam suatu jaringan, maka sistem terdisribusi mempunyai beberapa fitur dimana terdapat *layer* software, yang secara logika berada diantara *layer* dengan level yang lebih tinggi yang terdiri dari aplikasi *user*, dan *layer* yang lebih rendah yang menghubungkan dengan sistem operasi. Seperti pada gambar berikut ini, konsep tersebut dinamakan *middleware*.



Gambar 2.1 Middleware pada Sistem Terdistribusi[TAN02]

Dalam membangun suatu sistem terdistribusi, maka kita harus memperhatikan beberapa konsep-konsep, yaitu:

- *Connecting User and Resource.*
  Tujuan utama dari sistem terdistribusi adalah untuk memudahkan *user* mengakses *remote resource* (sumber daya dari tempat lain), dan melakukan *sharing* terhadap sumber daya tersebut dengan *user* lainnya.
- *Transparency.*
  Tujuan utama dari system terdistribusi adalah untuk menyembunyikan kenyataan bahwa suatu proses dan *resource* secara fisik didistribusikan melawati berbagai komputer. Sistem terdistribusi yang mampu menunjukan dirinya pada *user* dan aplikasi sebagai sebuah sistem computer tunggal dikatakan memiliki *transparency*. Beberapa aspek *transparency* dalam sistem terdistribusi dapat ditunjukan pada tabel berikut ini (Tabel-2.1).

Tabel 2.1 Macam-macam Bentuk Transparancy dalam Sistem Terdistribusi[TAN02]

Transparency	Description
Access	Hide Differences in data representation an how a *resource* is accessed
Location	Hide where a *resource* is lecoted
Migration	Hide that a *resource* may move to another location
Relocation	Hide that a *resource* may be moved to another location while in use
Replication	Hide that a *resource* is replicated
Concurrency	Hide that a *resource* may be shared by severalcompetitive *users*
Failure	Hide the failure and recovery of a *resource*
Persistence	Hide wheter a (software) *resource* is ini memory or on disk

- *Openness.*
  Opennes merupakan salah satu tujuan terpenting dari sistem terdistribusi. Suatu *Open Distributed System* adalah sistem terdistribusi yang menawarkan beberapa *service* berdasarkan pada aturan standar yang menggambarkan sintaks dan semantik pada beberapa *service* tersebut. Peraturan-peraturan tersebut dibentuk dalam suatu protokol. Dalam sistem terdistribusi *service* tersebut secara umum dispesifikan melalui suatu *interface*, yang sering disebut *Interface Definition Languange* (IDL).
- *Scalability.*
  Scalability dari suatu sistem, dapat diukur dari tiga dimensi yang berbeda. Yang pertama adalah berhubungan dengan ukuran, yang berarti kita dapat dengan mudah menambah *user* dan *resource*. Yang kedua adalah berhubungan dengan lokasi, dimana *user* dan *resource* dapat terpisah jauh. Dan yang terakhir berkaitan dengan administrasi, dimana *user* dapat dengan mudah mengatur sistem tersebut, walaupun bebeda administrasi.

2.1.1 Konsep hardware dan software

2.1.1.1 Arsitektur hardware

Secara konsep, konfigurasi hardware pada sistem terdistribusi ini dibagi menjadi dua bagian, yaitu *multiprocessors* dan *multicomputers*. Perbedaan mendasar adalah pada *multiprocessor*, terdapat satu buah *space* untuk *physical address* (alamat fisik) yang di-*share* oleh semua CPU. Sedangkan pada *multicomputers* terdapat beberapa komputer yang terhubung dengan jaringan untuk melakukan proses.

Konsep *multicomputers* dibedakan menjadi dua, yaitu *homogenus multicomputers* dan *heterogenus multicomputers*. Pada *homogenus multicomputers* terdapat satu buah koneksi jaringan yang menggunakan spesifikasi yang sama. Komputer-komputer tersebut, memiliki prosesor yang sama dan secara umum mempunyai akses yang sama pada *private memory*. *Homogenus multicomputer* lebih cenderung digunakan sebagai sistem paralel, seperti pada *multiprocessor*.

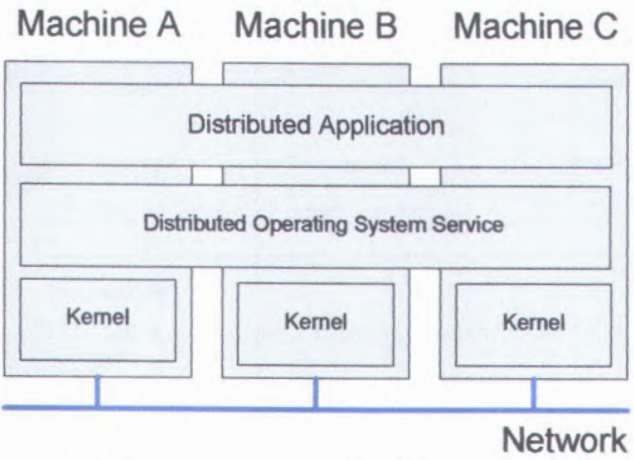
Sedangkan pada *heteregenous multicomputers* terdapat beberapa computer yang berbeda dan *independent* satu sama lainnya, yang terhubung dalam jaringan yang berbeda. Sebagai contoh adalah *distributed computer system* bisa dibentuk dari kumpulan *local-area computer network* yang berbeda., yang dihubungkan dengan FDDI atau ATM-*switched*.

2.1.1.2 Arsitektur software

Sistem operasi yang digunakan untuk system terdistribusi dibagi menjadi dua kategori, yaitu: *toughly-couple system* dan *loosely coupled system*. Dalam *toughly-couple system*, sistem operasi melakukan *maintain* satu *resource* global yang diatur. Sedangkan pada *loosely-couple system* dapat diumpamakan sebagai kumpulan beberapa komputer, dimana masing-masing komputer tersebut menjalanan sistem operasi sendiri. Akan tetapi sistem tersebut bekerja bersama memberikan servis dan *resource* agar bisa digunakan oleh komputer lain.

Sesuai dengan kategori yang ada, arsitektur software dalam system terdistribusi ini dibagi menjadi 3 macam, yaitu:

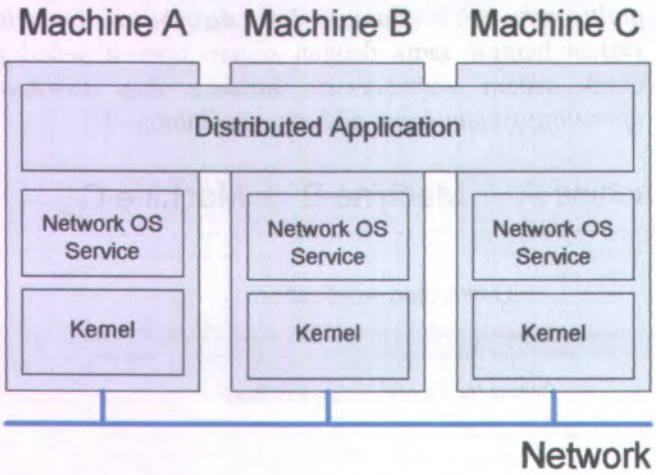
- *Distributed Operating System (DOS)*
  Ada dua macam tipe *distributed operating system*, yaitu *multiprocessor operating system* yang mengatur *resource multiprocessor* dan didesain dengan tujuan untuk mendukung performa yang tinggi melalui banyaknya CPU, dan *multicomputer operating system*, yaitu sistem operasi yang di disain untuk *homogenous multicomputers*. Fungsi dari *distributed operating system* hampir sama dengan sistem operasi sederhana untuk sistem *uniprocessor*. Struktur dari *distributed operating system* dapat dilihat pada Gambar-2.1.



Gambar 2.1 Struktur umum Distributed Operating System[TAN02]

- *Network Operating System*
  Secara umum *network operating system* dibentuk atas dasar konfigurasi dari kumpulan sistem *uniprocessor*,

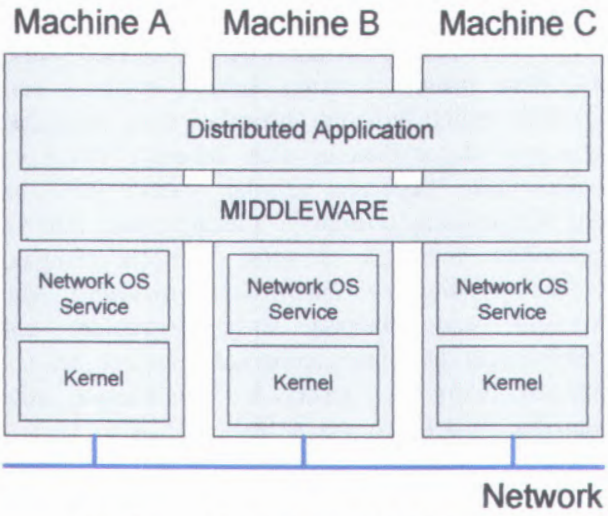
dimana masing-masing bagian tersebut mempunyai sistem operasi sendiri, seperti yang ditunjukan pada Gambar-2.3. Komputer dan sistem operasi yang digunakan bisa jadi berbeda-beda, tetapi kesemuanya terhubung dalam jaringan komputer. *Network operating system* menyediakan fasilitas yang memungkin *user* untuk menggunakan *service* yang tersedia pada masing-masing komputer yang ada.



Gambar 2.3 Struktur umum Network Operating System[TAN02]

- *Middleware*
  Middleware merupakan salah satu arsitektur pada sistem terdistribusi, yang merupakan penghubung antara aplikasi sistem terdistribusi dengan sistem operasi yang ada pada masing-masing komputer (*network operating system*) dalam sistem terdistribusi (Gambar-2.4). Setiap sistem lokal yang memiliki *network operating system* diasumsikan menyediakan manajemen *resource* secara lokal untuk proses komunikasi antar komputer. Dengan

kata lain *middleware* tidak akan melakukan manejemen pada komputer local.



Gambar 2.4 Struktur umum Middleware[TAN02]

2.1.2 Proses komunikasi dalam sistem terdistribusi

Proses komunikasi dalam sistem terditribusi merupakan hal sangat penting sekali. Komunikasi ini berdasarkan pada pengiriman pesan *low-level* yang dilakukan oleh jaringan pokok. Dalam sistem terdistribusi, proses komunikasi antar *host* dapat dilakukan dengan berbagai cara, yaitu:

- *Remote Procedure Call* (RPC)
  RPC adalah protokol yang memungkinkan suatu program yang berjalan di atas suatu komputer dapat melakukan eksekusi terhadap *subroutine* yang ada di komputer yang lain.
  RPC diinisialisasi oleh suatu *client* yang mengirimkan pesan pada *remote server* untuk mengeksekusi suatu prosedur yang ada pada *remote server* tersebut, dengan

menggunakan parameter tertentu. Respon akan dikembalikan ke *client* dimana aplikasi tersebut akan berjalan selama prose situ berjalan.
Ada beberapa jenis imlementasi dan hasil dari beberapa protokol RPC. Kunci utamanya adalah konsep *blocking client*, atau proses *waiting*, selama aplikasi *server* melakukan respon terhadap informasi yang dibutuhkan. Agar *server* dapat diakses oleh beberapa *client* yang berbeda, maka beberapa standar sistem RPC telah dibuat. Kebanyakan standar ini menggunakan IDL yang mengizinkan beberapa *platform* berbeda mengakses RPC. IDL (*Interface Definition Language*) adalah merupakan suatu bahasa yang digunakan untuk mendefinisikan dan menggambarkan *interface* dari suatu komponen software, termasuk didalamnya adalah komunikasi antar komponen software.

- *Remote Object Invocation*
  Gagasan utama dari *Remote Object Invocation* ini adalah konsep dari RPC. Dalam sistem objek terdistribusi ini, maka yang didistribusikan adalah objek.
  Dalam sistem ini, setiap *resource* didefinisikan oleh suatu objek, dimana objek tersebut berisi beberapa data dan operasi yang memanipulasi data tersebut. Setiap objek tersebut memiliki nama dan alamat yang unik.
  Di dalam system objek terdistribusi ini, suatu objek dapat menempati node tertentu dalam suatu jaringan. Objek tersebut dapat berkomunikasi antara satu dengan yang lain melalui proses *invoking*. Proses *invoking* adalam merupakan proses RPC, dimana dalam proses *invoking* ini menspesifikasikan objek target, operasi-operasi pada objek tersebut, dan beberapa parameter.
  Seperti halnya RPC, proses *invocation* ini berjalan secara *synchronous*. Proses *binding* pemanggilan objek dari objek yang dipanggil, seperti halnya pelewatan

parameter, ditangani oleh *run-time system* dan kernel secara otomatis.

- *Message Oriented Communication*
  Message-oriented communication adalah salah satu cara komunikasi antar proses. Pesan yang berhubungan dengan suatu *event*, adalah merupakan unit dasar dari data yang dikirim. *Message-Oriented Communication* dibagi menjadi dua, yaitu *synchronous communication* atau *asynchronous communication* dan *transient* atau *persistent communication.*
  Pada *synchrounus communication*, *sender* berada dalam kondisi *block* ketika menuggu *reply* dari *receiver.* Sedangkan pada *asynchronous communication* antara *sender* dan *receiver* tidak perlu melakukan eksekusi secara simultan. *Transient communication* menyimpan pesan hanya pada saat *sender* dan *receiver* sedang melakukan eksekusi. Sedangkan *persistent communication*, pesan disimpan sampai *receiver* menerima pesan tersebut.
- *Stream Oriented Communication.*
  Stream Oriented Communication, adalah suatu proses berkomunikasi antar proses dimana media yang dikomunikasikan mempunyai sifat kontinyu. Di dalam media yang sifatnya kontinyu data ditransmisikan melalui *stream.* Mode transmisi ini dibagi menjadi menjadi beberapa macam, yaitu: *asynchrounus transmission*, *synchronous transmission*, dan *isochronous transmission.*
  Asynchronous transmission yaitu transmisi dimana data yang dilewatkan dalam stream ditransmisikan secara berurutan, dan tidak ada batasan waktu untuk transmisi data. *Synchronous transmission* yaitu transmisi dimana tiap data yang ditransmisikan mempunyai batasan waktu *delay Isochronous transmission* yaitu transmisi dimana data yang dikomunikasikan mempunyai batas waktu

minimum dan maksimum *delay* dalam proses transmisinya.

Stream sering dihubungkan dengan koneksi virtual antara *source* dan *sink*. *Source* dan *sink* ini dapat berupa proses ataupun suatu *device*.

2.2 Multi Agent

Berikut ini akan dijelaskan beberapa konsep yang berkaitan dengan *Multi Agent* yang meliputi definisi dari *Agent*, arsitektur serta konsep-konsep lain yang berkaitan dengan *Multi Agent*.

2.2.1 Definisi agent

Terdapat beberapa definisi tentang *agent* yaitu: "Software *agent* adalah program yang digunakan untuk komunikasi dan dialog dalam pertukaran informasi" (Michael Coen 1996). "Intellegent *Agent* adalah suatu entitas software yang bertanggung jawab terhadap sejumlah operasi bagi *user* atau program lain dengan secara otonomi, dan juga dapat melakukan pekerjaan dengan pengetahuan yang dimilikinya" (definisi IBM *Agent* 1996).

Berdasarkan dua definisi *agent* diatas, maka suatu *agent* memiliki beberapa karakteristik yang dibagi ke dalam dua kelompok (Brenner et al.1998 p.23), yaitu:

- *Internal properties*

  Internal properties menggambarkan berbagai macam aksi dan tingkah laku *agent*, yaitu:

  - *learning* (kemampuan belajar)

  Suatu *rational agent* tidak hanya menggali informasi, tetapi juga mampu mempelajari segala sesuatu yang ditangkap dari lingkungan luarnya. Dengan kata lain, suatu *agent* dapat belajar dari pengalaman-pengalaman terhadap segala sesuatu yang berasal dari luar.

  - *autonomy* (bersifat otonomi)

Suatu *agent* bersifat otonomi apabila mampu mempelajari aksi yang berasal dari luar, dan mampu mempersepsikan aksi dari luar tersebut ke dalam reaksi, yang dilakukan secara mandiri (tanpa adanya pengaruh dari luar).

- *proactivity*
- *goal oriented* (berorientasi pada hasil),
- *omniscience*

Suatu *omniscience agent* dapat mengetahui suatu hasil dari setiap tindakan, dan dapat bertindak berdasarkan pengetahuan tersebut.

- *reactivity*.

Suatu *agent* mampu melakukan monitoring terhadap lingkungannya dan melakukan respon secara cepat dan efektif untuk melakukan perubahan pada lingkungannya.

- *External properties*

External properties merupakan karakteristik dari *agent* yang berhubungan dengan interaksi dengan *agent* yang lain atau *users*, yaitu : *communication*, *cooperation*, *character*, dan *coordination*.

Suatu *agent* membutuhkan suatu ukuran atau kriteria yang menentukan apakah tindakan dan sifat-sifat yang dimiliki oleh suatu *agent* dapat dikatakan sukses dan sesuai dengan tujuan yang hendak dicapai. Ukuran tersebut dinamakan *performance measure*. Pada saat suatu *agent* dilepaskan dalam suatu lingkungan, maka akan menghasilkan suatu tindakan-tindakan secara sekuensial beradasarkan persepsi yang dia terima. Dalam satu aturan umum dikatakan bahwa, akan lebih baik untuk menentukan *performance measure* berdasarkan pada apa yang dibutuhkan dalam lingkungan, dari pada bagaimana suatu *agent* harus bertindak.

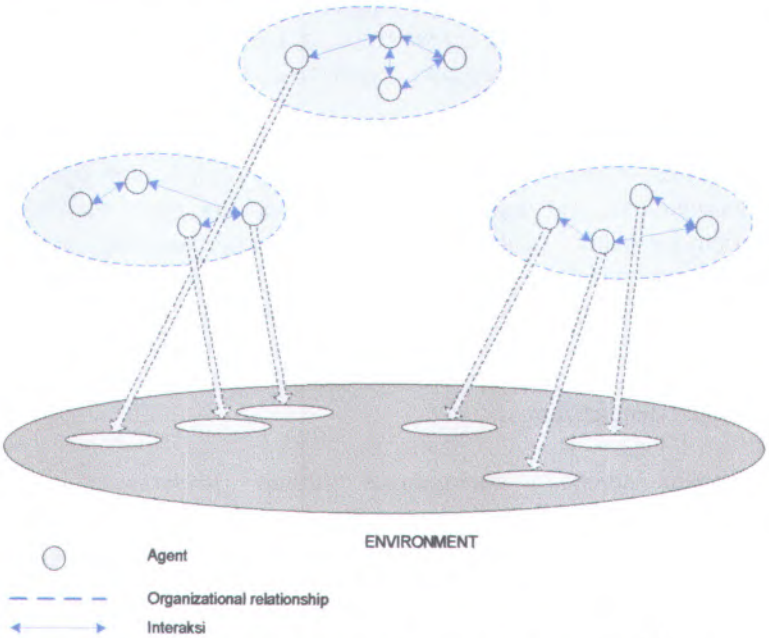
2.2.2 Environtment agent

Environment Agent yaitu lingkungan di sekitar *agent* yang dapat mempengaruhi *agent* tersebut. *Environment agent* ini dibagi ke dalam beberapa jenis [RUS03], yaitu;

- *Accessible versus Inaccessible.*
  Accessible environment adalah suatu *environment* dimana suatu *agent* dapat memperoleh suatu informasi terhadap *environment* secara lengkap, akurat, dan *up-to-date*.
- *Deterministic versus non deterministic.*
  Deterministic environment adalah suatu *environment* setiap aksi padanya akan menyebabkan satu buah afek. Ada ketidak pastian mengenai *state* yang akan dihasilkan dari melakukan sebuah aksi.
- *Static versus dynamic*
  Static environment adalah *environment* yang diasumsikan tidak dapat dirubah kecuali oleh sejumlah aksi yang dilakukan oleh suatu *agent*. Sedangkan *dynamic environment* adalah *environtment* yang memiliki sejumlah proses yang lain yang berjalan didalamnya, dan menangani perubahan diantara kontrol *agent*.
- *Episodic versus sequential.*
  Dalam *episodic environment* suatu pengalaman *agent* dibagi menjadi beberapa bagian kecil. Setiap bagian terdiri dari beberapa *agent perceiving* dan melakukan sejumlah aksi tunggal. Bagian berikutnya tidak tergantung pada aksi yang diakibatkan oleh bagian sebelumnya.
- *Discrete versus continue*
  Environment dikatakan bersifat diskrit, juka memiliki sejumlah aksi yang bersifat tetap dan terbatas.

2.2.3 Definisi multi agent

Multi Agent System (MAS) adalah suatu sistem yang dibangun dari beberapa *agent*, yang saling bekerjasama dalam menyelesaikan permasalahan yang tidak mungkin ditangani oleh sebuah *agent* dalam suatu sistem. Inti dari sistem multi *agent* ini adalah adanya interaksi antar *agent* dengan tujuan untuk menyelesaikan sebuah *task* [WOO02].



Gambar 2.5 Struktur dari multi agent system [WOO02].

Pada Gambar-2.5 dijelaskan struktur dari sistem multi *agent*. Sistem tersebut berisi sejumlah *agent* yang saling berinteraksi satu dengan lainnya melalui komunikasi. *Agent* tersebut dapat bertindak dalam *environtment*. Sebuah *agent* mempunyai pengaruh berbeda dari *agent* lainnya pada suatu

environment. Pengaruh pada lingkungan ini serupa dalam beberapa kasus. Secara nyata pengaruh pada lingkungan ini akan menimbulkan *dependency* (ketergantungan) diantara *agent* yang ada.

2.2.4 Utilities dan preferences dari multi agent

Pertama kali diasumsikan bahwa terdapat dua buah *agent*, yaitu *agent i* dan *agent j*. Setiap *agent* tersebut diasumsikan mempunyai kepentingan sendiri (*self-interested*). Dikatakan bahwa setiap *agent* tersebut mempunyai keinginan dan pilihan sendiri mengenai bagaimana seharusnya kondisi lingkungan yang ada.

Kemudian diasumsikan bahwa ada satu set *state* atau *outcome* $\Omega = \{\omega 1, \omega 2, \omega 3....\}$, yang harus dipilih oleh *agent*. Kemudian dari sini dapat diketahui bahwa dua *agent* tersebut mempunyai dua buah fungsi *utility* sendiri-sendiri yang ditetapkan pada setiap *outcome* sebuah bilangan real, yang mengindikasikan tingkat kebaikan dari outcome tersebut. Semakin besar nilainya, semakin baik fungsi *utility* dari *agent* tersebut. *Agent* i mempunyai pilihan (*preference*) yang di notasikan dengan fungsi:

$$Ui: \Omega \rightarrow R$$

Sedangkan *agent* j mempunyai pilihan (*preference*) yag dinotasikan dengan fungsi

$$Uj: \Omega \rightarrow R$$

2.2.5 Relasi dependent dalam Sistem multiagent.

Suatu relasi yang *dependent* terdapat pada dua buah *agent* jika salah satu *agent* membutuhkan *agent* yang lain untuk mencapai tujuan yang diinginkan. Ada beberapa relasi *dependent* pada *agent* yang mungkin [WOO02], yaitu:

- *Independence*, yaitu tidak adanya ketergantungan antara dua buah *agent*.

- *Unilateral,* yaitu suatu *agent* tergantung pada *agent* yang lain, tapi tidak sebaliknya.
- *Mutual,* yaitu kedua *agent* mempunyai ketergantungan antara satu dengan yang lain, untuk mencapai tujuan yang sama.
- *Reciprocal dependence.,* yaitu *agent* yang pertama tergantung pada *agent* yang lain untuk mencapai tujuan tertentu. Sedangkan *agent* yang kedua juga tergantung pada *agent* yang pertama untuk tuujuan tertentu (kedua tujuan *agent* tersebut berbeda).

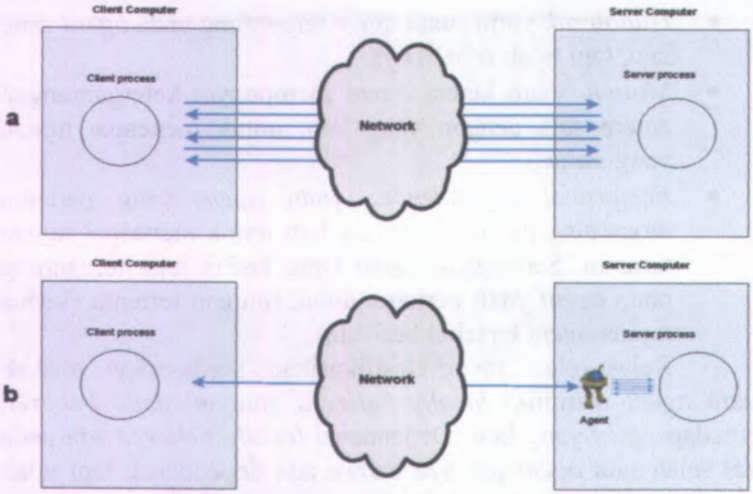
Relasi-relasi ini diklasifikasikan berdasarkan apakah suatu *agent* memiliki *locally believed* atau *mutually believed* terhadap *agent* yang lain. Dependensi *locally believed* ada pada saat salah satu *agent* percaya bahwa ada dependensi, tapi tidak dengan *agent* yang lain. Dependensi *mutually believed* ada pada saat *agent* percaya bahwa ada dependensi, dan juga percaya bahwa *agent* yang lain mempunyai dependesi.

2.3 Teknologi Mobile Agent

2.3.1 Definisi mobile agent

Teknologi *mobile agent* adalah suatu teknologi dimana suatu *agent* mempunyai kemampuan untuk memindahkan dirinya sendiri, program, dan *state*, melalui jaringan komputer, dan dapat melakukan eksekusi pada *remote site* [WOO02]..

Ide dari *mobile agent* ini adalah sangat sederhana. Ide ini bertujuan agar *mobile agent* mampu menggantikan konsep RPC (*Remote Procedure Call*) sebagai suatu proses untuk berkomunikasi antar jaringan. Lihat Gambar-2.6. Dengan RPC, suatu proses dapat mengirimkan prosedur (*method*) ke proses yang lain yang berada di lokasi *remote*. Umpakan proses *A* mengirimkan *method m* pada proses *B* dengan argumen *args*. Nilai yang dikembalikan oleh proses *B* akan diberikan kepada variabel *v*.



Gambar 2.6 Remote Procedure Calls (a) Mobile Agent (b) [WOO02].

V=B.m(args)

Satu hal yang penting dalam konsep RPC adalah, komunikasi dilakukan secara *asynchronous*. Oleh karena itu proses A akan melakukan blok mulai dari dimulainya eksekusi dari instruksi, sampai pada waktu B mengembalikan *return value*. Jika B tidak pernah mengembalikan *return value* karena ada kegagalan dalam jaringan, maka A akan berada dalam kondisi *suspend*, menunggu *reply* dari B yang tidak akan pernah datang. Oleh karena itu koneksi jaringan antara A dan B harus selalu terbuka, walaupun tidak sedang digunakan.

Gagasan dari *mobile agent* ini menggantikan RPC dengan cara mengirimkan semua objek *agent* ke *remote location* untuk melakukan komputasi. Metode ini merupakan pengganti dari proses pengiriman prosedur, dimana proses A akan mengirimkan *mobile agent* ke proses *B*. Pada saat *agent* selesai melakukan interaksi, maka *agent* tersebut akan kembali ke *A* dengan membawa hasil dari proses komputasi di *B*. Selama

proses interaksi berlangsung, maka koneksi dibutuhkan hanya pada saat proses pengiriman *agent* dari *A* ke *B*, dan sebaliknya. Hal ini meyebabkan penggunaan *resource* pada jaringan dapat efisien.

Ada beberapa pokok persoalan yang terakait dengan penggunaan mobile *agent*, yaitu:

- *Serialization*. Yaitu bagaimana *agent* melakukan proses serialisasi dan *encoding* program, atau data sebelum dikirmkan melalui jaringan.
- *Hosting and Remote Execution*. Yaitu pada waktu *agent* tiba di remote location bagaimana *agent* tersebut dieksekusi
- *Security*. Yaitu pada saat *agent* dari *A* dikirimkan ke komputer yang mengani proses *B*, apakah ada kemungkinan *agent* tersebut akan menimbulkan masalah di *host* yang menangani proses *B*.

Ada beberapa alternatif skenarioa untuk menyelesaikan beberapa permasalahan diatas, diantaranya adalah:

- Pada waktu proses pengiriman *agent*, *agent* yang dikirmkan beserta *state*-nya juga dikirimkan. *State* tersebut tmeliputi program counter. Sebagai contoh adalah *agent* akan mengingat *state* terakhir sebelum dia berpindah, dan pada saat sudah mencapai tujuannya, maka *state* tersebut kembali dieksekusi mulai dari saat terakhir tadi.
- *Agent* berisi program dan nilai dari variabel-variabel, tetapi tidak berisi program counter, sehingga suatu *agent* dapat mengingat semua nilai dari variabel-variabel tersebut, tetapi tidak pada saat *agent* tersebut dilewatkan melalui suatu jaringan.
- *Agent* yang ditransmisikan berupa script, tanpa adanya *state* (meskipun nantinya *state* tersebut akan diambil dari *host* asal pada saat *agent* tiba di *host* tujuan).

2.3.2 Telescript pada mobile agent

Telescript merupakan suatu *language-based environtment* (bahasa dasar) yang digunakan untuk membangun suatu *agent* [WOO02].. Ada dua konsep utama dari teknologi *telescript*, yaitu lokasi dan *agent*. Lokasi adalah tempat (lingkungan) dimana *agent* dapat hidup. Lokasi bisa berkaitan dengan suatu *host* dimana terdapat suatu *agent*. Sedangkan *agent* merupakan penyedia dan pemakai segala sesuatu yang ada di *host*.

Telescript agent dapat berpindah dari satu lokasi ke lokasi yang lain, dengan membawa *state* dan program yang sudah ter-*encode* melalui suatu jaringan. Untuk berpindah ke lokasi lain, suatu *agent* memerlukan suatu *ticket*, yang merupakan suatu parameter dalam proses perpidahan tersebut, yaitu:

- *Destination* (tujuan *agent*)
- Waktu yang dibutuhkan untuk menyelesaikan proses perpindahan

Telescript agent berkomunikasi antara satu dengan lainnya melalui berbagai cara, yaitu :

- Jika dua *agent* berada pada lokasi yang berbeda, maka membentuk koneksi melalui jaringan
- JIka dua *agent* berada pada lokasi yang sama, maka dua *agent* tersebut akan bertemu.

2.3.3 Agent TCL

Agent TCL (*Tool Control Language*) adalah bahasa standar yang digunakan oleh *agent* untuk melakukan komunikasi [WOO02].. TCL menyediakan suatu implementesi yang mudah untuk mendefiniskan bahasa yang digunakan oleh *agent* untuk berkomunikasi.

2.4 Framework JADE

JADE (Java Agent DEvelopment *Framework*) adalah suatu *framework* software yang secara sederhana mengimplementasikan sistem *multi gent* melalui *middleware*, dengan mengikuti spesifikasi dari FIPA (*Foundation for Intelligent Physical Agent*). *Platform agent* tersebut dapat didistribusikan antar mesin (tidak pada Sistem Operasi yang sama), dan konfigurasinya dapat di control melalui *remote* GUI.

JADE merupakan *middleware* yang memberikan fasilitas untuk mengembangkan dan membangun suatu sistem *multi agent*. Komponen-komponen yang dimiliki oleh JADE antara lain:

- *Runtime environtment,* dimana *agent* dapat hidup, dan harus aktif pada *host* sebelum *agent* dieksekusi.
- *Libarary,* dari kelas yang dapat digunakan untuk mengembangkan *agent*.
- *Graphical Tools,* untuk memudahkan proses administrasi dan monitoring dari *agent* yang sedang aktif.

Arsitektur komunikasi dari *framework* ini menawarkan pemrosesan pesan secara fleksibel dan efisien , dimana JADE membuat dan mengatur antrian dari pesan yang masuk, dan bersifat pribadi untuk setiap *agent*. Suatu *agent* dapat mengakses antrian tersebut dengan menggunakan kombinasi beberapa mode, yaitu: *block*ing, *polling*, *timeout*, dan pengenalan pola dari suatu pesan.

2.4.1 Fitur-fitur dalam JADE

Beberapa fitur yang ada dalam *framework* JADE ini antara lain sebagai berikut:

- *Distributed agent platform*. *Agent platform* dapat dipecah ke dalam beberapa *host* (yang saling terhubung melalui mekanisme RMI). Masing-masing *agent* tersebut merupakan implementasi dari *java thread* yang hidup

dalam suatu *Container Agent*, yang menyediakan runtime untuk proses eksekusi *agent*

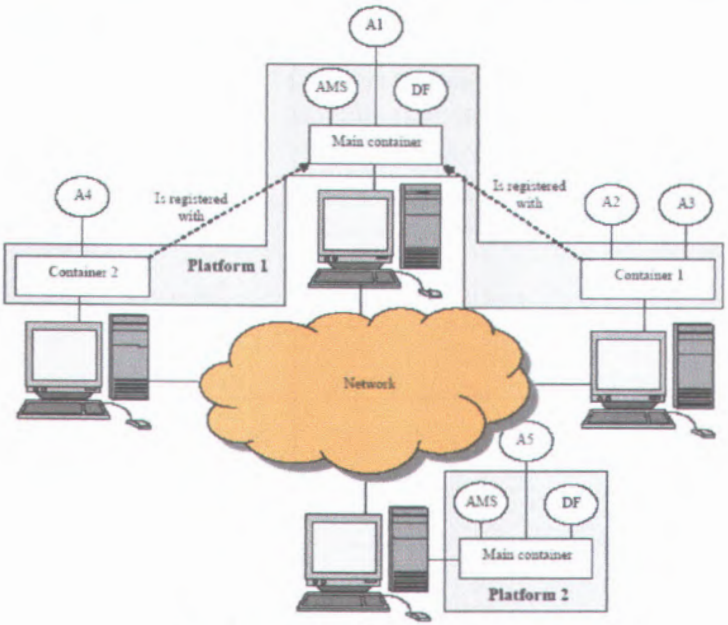
- Adanya *tool* untuk melakukan proses *debugging* dan *develop* aplikasi multi *agent* berdasarkan arsitektur JADE
- Mobilitas *agent* antar *platform*, yang mendukung pertukaran *state* dan kode antar *platform* tersebut.
- Mendukung proses ekseskusi secara multi, pararel dan konkuren dari aktifitas *agent* melalui pemodelan *behaviour*.
- FIPA-compliant *Agent Platform*, yang meliputi AMS (*Agent Management System*), DF (*Directory Facilitator*), dan ACC (*Agent Communication Channel*), yang akan aktif pada saat *agent startup*.
- Tersedianya *library* dari protokol interaksi FIPA yang siap pakai.
- Proses registrasi dan deregistrasi otomatis melalui AMS.
- Mendukung pemodelan bahasa dan ontologi yang sudah terdefinisi dalam aplikasi.
- *InProcess Interface* yang membolehkan aplikasi eksternal untuk mengaktifkan *autonomous agent*.

2.4.2 Container dan platform

Setiap *instance* yang aktif pada *runtime environment* JADE disebut *Container*, yang dapat berisi beberapa *agent*. Set *Container* yang aktif disebut *Platform*. Sebuah *Main-Container* tunggal harus aktif dalam suatu *platform*, dan setiap *container* yang ada harus teregistrasi pada *Main-Container* tersebut.

Jika *Main-Container* yang lain sedang aktif di suatu tempat lain dalam suatu jaringan, maka akan terdapat beberapa *platform* yang berbeda dimana suatu *container* dapat teregistrasi. Pada Gambar 2.7 merepresentasikan konsep-konsep diatas, melalui contoh scenario, dimana dua buah *platform* JADE yang terdiri dari 1-3 *container*. *Agent* JADE diidentifikasi dengan nama yang unik, dimana masing-masing *agent* tersebut dapat

saling mengenali dan dapat berkomunikasi antara satu dengan lainnya.



Gambar 2.7 Platform dan Container

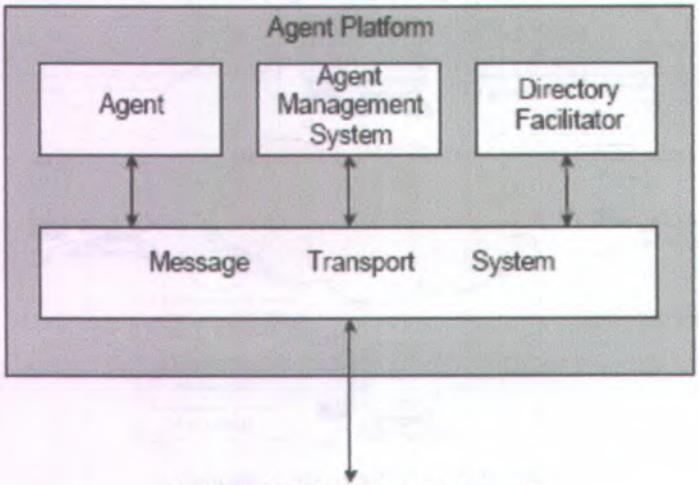
2.4.3 Pemodelan platform agent

Agent Management System (AMS) adalah *agent* yang mempunyai akses kontrol dan pengawasan terhadap *platform agent*. Dalam satu buah *platform* hanya terdapat satu buah AMS, yang menyediakan *white-page* dan *life-cycle service*, menjaga directory dari *agent identifier* (AID) dan *agent state*. Setiap *agent* harus teregistrasi dalam AMS untuk mendapatkan AID. Dikatakan bahwa suatu AMS menyediakan suatu *naming service* untuk setiap *agent* yang aktif. *Directory Facilitator* (DF) adalah

agent yang secara *default* menyediakan servis *yellow pages* dalam suatu *platform*.

Sistem Transportasi Pesan, atau yang biasa disebut *Agent Communication Channel* (ACC), adalah merupakan komponen yang mengontrol segala pertukaran pesan dalam suatu *platform*, termasuk pesan dari/ke *remote platform*.

Pemodelan standar dari *platform agent* sesuai dengan standar FIPA adalah sebagai berikut:



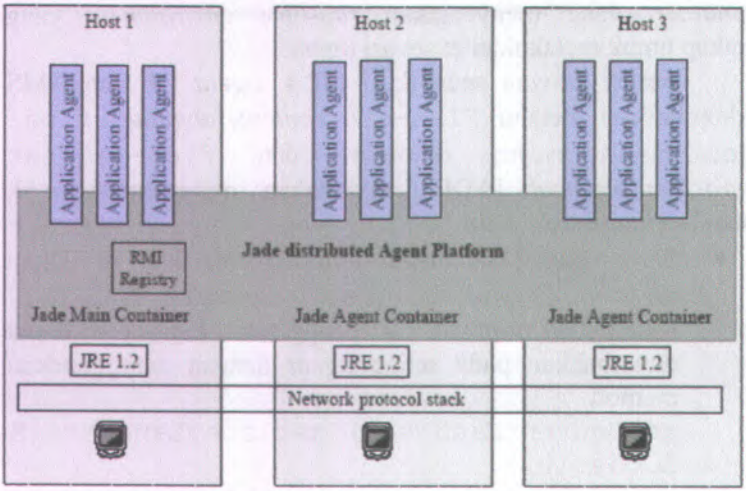
Gambar 2.8 Arsitektur Fipa Agent

Pada saat suatu *agent* aktif, maka secara otomatis akan memiliki arsitektur seperti diatas. AMS dan DF akan pertama kali diaktifkan dan selanjutnya apabila terdapat pesan maka ACC akan diaktifkan. Suatu *platform agent* dapat tersebar pada setiap *host* yang berbeda. Pada setiap *host* tersebut harus memiliki *Java Virtual Machine* (JVM) sebagai dasar dari *container* dimana suatu *agent* dapat aktif. *Main-Container*, adalah *container agent* dimana terdapat AMS dan DF dan registry RMI (*Remote Method Invocation*), yang terdapat dalam *framework* JADE. *Container agent* yang lain harus terhubung dan teregistrasi ke *Main-*

Container, yang menyediakan *run-time environment* yang lengkap untuk melakukan eksekusi *agent*.

Sesuai dengan spesifikasi FIPA, *agent* DF dan AMS berkomunikasi melalui `FIPA-SL0` *content language*, `fipa-agent-management` *ontology*, dan `fipa-request` *interaction protocol*. JADE menyediakan implementasi untuk setiap komponen ini, yaitu:

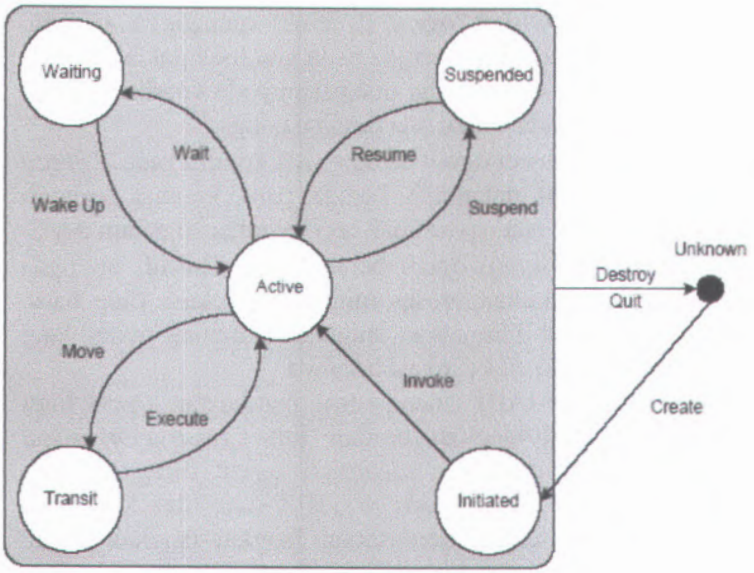
- SL-0 content language, diimplementasikan di dalam kelas `jade.content.lang.sl.SLCodec`. Kemampan otomatis dari penggunaan bahasa ini dapat ditambahkan pada setiap *agent* dengan menggunakan method `getContentManager().registerLanguage(SLCodec(0))`.
- Konsep dari *ontology* diimplementasikan ke dalam kelas di dalam *package* `jade.domain.FIPAAgentManagement`. Kelas `FIPAManagementOntology` mendefinisikan kosa kata dengan semua symbol *ontology*.
- `Fipa-request` *interaction protocol* diimplementasikan sebagai suatu *behaviour* dalam package `jade.proto`.



Gambar 2.9 pendistribusian platform *JADE* agent *pada beberapa* container

2.4.4 Kelas agent

Kelas *agent* merepresentasikan kelas dasar bagi pembuatan *agent*. Dalam setiap pembuatan kelas *agent* kita perlu meng-*inherit* dari kelas *Agent* agar dapat berinteraksi dengan *platform agent* (registrasi, konfigurasi, *remote management*, dan lain-lain), dan dapat memanggil *method* dasar untuk mengimplementasikan *Behaviour* suatu *agent*.



Gambar 2.10 Agent life cycle

Di dalam JADE, *agent* dapat berada pada salah satu dari beberapa *state*, berdasarkan pada siklus hidup *agent* yang didefinisikan oleh FIPA, yaitu:

- ***Initiated*** : Object *agent* dibuat, tetapi belum melakukan registrasi ke dalam AMS, belum mempunyai nama dan alamat, sehingga tidak dapat berkomunikasi dengan *agent* yang lain.
- ***Active*** : Object *agent* teregistrasi dalam AMS, mempunyai nama dan alamat dan dapat mengakses semua fitur yang ada dalam JADE.
- ***Suspended*** : Object *agent* baru saja di stop. *Thread* yang ada di dalamnya berada pada kondisi *suspend* dan tidak ada *behaviour agent* yang sedang dieksekusi.

- ***Waiting*** : Object *agent* di blok, menunggu sesuatu. Thread di dalamnya berada pada kondisi tidak aktif pada *Java Monitor*, dan akan diaktifkan pada kondisi tertentu (biasanya pada waktu ada pesan datang).
- ***Deleted*** : Object *agent* berada pada kondisi mati. *Thread* yang ada di dalamnya berada pada kondisi berhenti (*terminate*), dan *agent* tidak lagi terdaftar di dalam AMS
- ***Transit*** : *Mobile agent* berada pada konsidi ini pada waktu melakukan proses migrasi ke lokasi yang baru. System akan dilanjutkan untuk menampung pesan yang akan dikirimkan ke lokasi barunya.

Framework JADE mengontrol pembuatan *agent* baru berdasarkan langkah-langkah berikut ini: *Constructor agent* diekseskusi, *agent* diberikan *identifier*, *agent* diregistrasi ke dalam AMS, *agent* berada pada *ACTIVE state*, dan kemudian *method setup()* dieksekusi. Berdasarkan langkah-langkah diatas, suatu agent memiliki beberapa atribut, yaitu:

- *Globally unique name*: Suatu *agent* akan memiliki nama yang unik yang berbeda antara agent satu dengan *agent* yang lain.
- Sebuah *agent address*.
- Sebuah *resolver*.

Untuk menghentikan eksekusi *agent*, suatu *bahaviour agent* dapat memanggil *method Agent.doDelete()*. *Method Agent.takeDown()* dieksekusi pada saat *agent* akan berada pada kondisi *DELETED*. Pada saat *method takeDown()* dieksekusi, *agent* masih teregistrasi dalam AMS, sehingga masih bisa mengirimkan pesan ke *agent* yang lain, setelah method ini dieksekusi agent akan di-*deregistrasi* dari AMS, dan semua *thread* dimatikan.

2.4.5 Agent Communication Language (ACL) message

Kelas ACLMessage merepresentasikan suatu pesan ACL yang dapat dipertukarkan antara agent yang satu dengan *agent*

yang lain. *ACLMessage* berisi satu set atribut yang terdefinisi dalam FIPA.

Suatu *agent* yang akan mengirimkan pesan harus membuat objek baru *ACLMessage*, mengisikan atribut-atribut yang ada di dalamnya dengan nilai yang tepat, dan kemudian memanggil *method Agent.send()* untuk mengirimkan pesan ke *agent* yang lain. Sedangkan untuk menerima pesan dari *agent* lain, maka suatu *agent* harus memanggil *method receive()* dan *blockingReceive()*.

Semua atribut dari objek *ACLMessage* dapat diakses melalui *method set/get<attribute>()*. Semua atribut tersebut diberi nama setelah nama parameter, yang didefinisikan dalam FIPA.

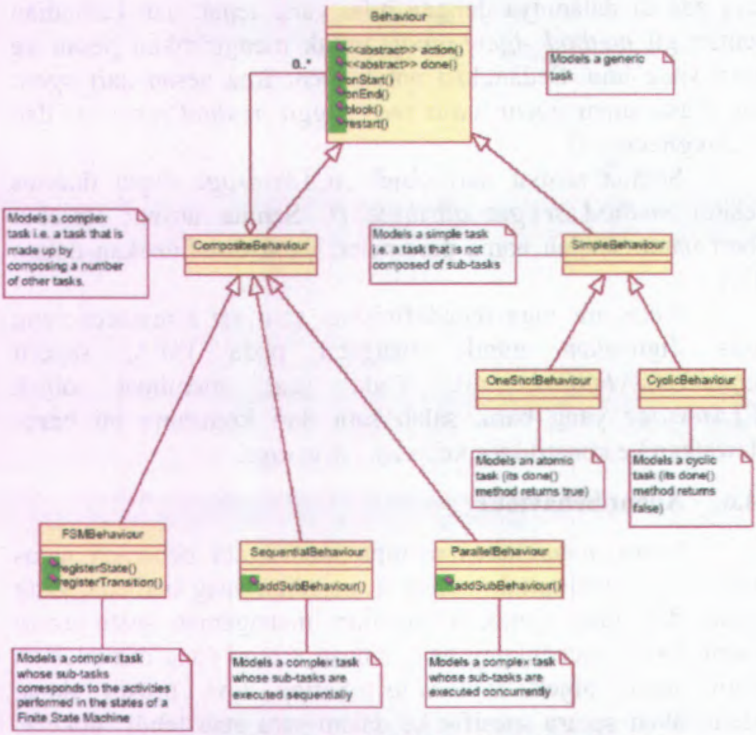
Kelas ini juga mendefiniskan satu set konstanta yang harus digunakan untuk mengacu pada FIPA, seperti *REQUEST,INFORM*, dll. Pada saat membuat objek *ACLMessage* yang baru, salah satu dari konstanta ini harus dilewatkan ke konstruktor kelas *ACLMessage*.

2.4.6 Agent behaviour

Suatu *agent* harus mampu menangani beberapa tugas secara *concurrent* untuk merespon kejadian yang berbeda, yang berasal dari luar. Untuk menjadikan managemen suatu *agent* efisien, JADE mendukung penggunaan *thread* yang dimodelkan dalam suatu objek *Behaviour*. Setiap *task* proses dapat didefinisikan secara spesifik ke dalam satu atau lebih subkelas *Behaviour*, meninstansiasi kelas tersebut, dan kemudian menambahkan *behaviour* tersebut dalam *task list*. Dua buah *method* yang sering digunakan yaitu *addBehaviour(behaviour)* untuk menmbahkan *behaviour* pada *task list*, dan *method removeBehaviour(behaviour)* untuk menghilangkan *behaviour* dari *task list*.

Kelas *Behaviour* mempunyai beberapa *method* utama, yaitu method *action()* yang berisi tindakan-tindakan yang dilakukan oleh *agent*, *method done()* yang akan mengembalikan

nilai *bolean* 1 pada saat suatu *behaviour* selesai dijalankan, dan method *block()* yang akan melakukan proses *blocking* terhadap *behaviour* apabila kondisi tertentu tidak dipenuhi.



Gambar 2.11 Pemodelan UML dari hierarki behaviour

Dalam *package jade.core.behaviours* terdapat bermacam-macam tipe kelas *behaviour*, diantaranya adalah:

- Kelas *Behaviour. Abstract* kelas *behaviour* ini merupakan pemodelan dasar dari *task agent*, dan penjadwalan *behaviour* yang membolehkan perubahan dan pergantian *state*.

- Kelas *Simple Behaviour. Abstract* kelas ini memodelkan *behaviour atomic* yang sederhana. Memiliki *method reset()* yang secara default tidak melakukan aksi apapun, tetapi dapat dilakukan *override* oleh *user* yang mendefiniskannya.
- Kelas *OneShotBehaviour. Abstract* kelas ini memodelkan *behaviour atomic* harus dieksekusi sekali, dan tidak dapat dilakukan proses *block*. Sehingga memiliki *method done()* yang selalu bernilai *true*.
- Kelas *CyclicBehaviour. Abstract* kelas ini memodelkan *behaviour atomic* yang harus dieksekusi selamanya. Memiliki *method done()* yang selalu bernilai *false*.
- Kelas *CompositeBehaviour. Abstract* kelas ini memodelkan *behaviour* yang dibuat dengan menggabungkan beberapa *behaviour* (*child behaviour*). Sehingga operasi yang dilakukan oleh *behaviour* ini, tidak didefinisikan oleh *behaviour* ini sendiri, akan tetapi juga didefinisikan oleh *child behaviour* yang ada di dalamnya.
- Kelas *SequentialBehaviour.* Kelas ini merupakan *CompositeBehaviour* yang melakukan eksekusi sub-*behaviour* yang dimiliki secara sekuensial, dan akan berhenti pada saat semua *behaviour* selesai dijalankan.
- Kelas *ParallelBehaviour.* Adalah merupkan *CompositeBehaviuor* yang melakukan ekseskusi pada *sub-behaviour* secara konkuren, dan akan berhenti pada saat kondisi tertentu pada sub-*behaviour* ditemui.
- Kelas *FSMBehaviour.* Kelas ini adalah merupakan *CompositeBehaviour* yang melakukan eksekusi pada *sub-behaviour* berdasarkan pada *Finite State Machine* yang didefinisikan oleh *user*
- Kelas *WakerBehaviour.* Kelas ini meng-*implements one-shot task* yang harus dieksekusi hanya sekali setelah waktu yang diberikan selesai.

- Kelas *TickerBehaviour*. Kelas ini meng-implements *cyclic task* yang harus dieksekusi secara periodik.

2.4.7 Agent mobility

JADE mendukung mekanisme mobilitas *agent*, dimana suatu *agent* dapat berpindah atau menggandakan dirinya sendiri melalui beberapa *host* dalam suatu jaringan. *Mobile agent* JADE dapat melakukan navigasi melewati beberapa *container agent* yang bebeda, tetapi tetap dibatasi pada satu *platform* JADE.

Mobile agent harus mengetahui suatu lokasi, untuk menentukan kapan dan dimana agent tersebut berpindah. JADE menyediakan *property ontology* yang dinamakan *jade-mobility-ontology*, yang menangani mekanisme perpindahan tersebut. Ontology ini berada pada *package jade.mobility.ontology*.

Dua buah *public method doMove()* dan *doClone()* memungkinkan *agent* JADE untuk melakukan migrasi ke suatu tempat dan meggandakan dirinya sendiri dengan nama yang berbeda. *Method doMove()* mempunyai parameter tunggal *jade.core.Location*, yang mereprenstasikan tujuan dari *agent*. *Method doClone()* juga mempunyai parameter *jade.core.Location*, dan *String* yang berisi nama dari *agent* baru yang akan dibuat sebagai duplikat dari *agent* tersebut.

Memindahkan *agent* melibatkan proses pengiriman kode dan *state* melalui jaringan , jadi kita harus melakukan pengaturan proses serialisasi dan sebaliknya. Beberapa *resource* yang digunakan oleh *mobile agent* akan ikut dibawa, dan sebagian lagi akan diputus sebelum *agent* berpindah . *Method beforeMove()* dipanggil pada lokasi awal pada saat proses perpindahan selesai dilakukan. *Method afterMove()* dipanggil pada lokasi tujuan langsung pada saat *agent* tiba di lokasi tersebut.

2.5 Graph dan Algoritma Djikstra

2.5.1 Graph

Graph merupakan struktur diskrit yang terdiri dari *vertex* dan *edge* yang menghubungkan *vertex* tersebut [DOU01]. *Graph* dibedakan menjadi beberapa tipe berdasarkan jumlah *edge* yang menghubungkan *dua* buah *vertex*. *Graph* dapat digunakan untuk memodelkan jaringan komputer.

Simple graph adalah graph yang tidak mempunyai *loop* (dimana *vertex* akhir sama dengan *vertex* awal) pada *multiple edge*-nya (*edge* yang mempunyai *vertex* akhir yang sama).

2.5.2 Algortima djikstra

Algoritma Djikstra dikembangkan oleh seorang ilmuwan bernama Edsger W. Djikstra. Algoritma ini memecahkan permasalahan untuk menentukan rute tercepat (terpendek) pada suatu *graph* berarah dengan suatu nilai *edge* yang positif, dari suatu titik asal ke titik tujuan [ROS03]. Sebagai contoh, jika suatu titik pada graph mewakili suatu kota, dan *edge* mewakili jalan yang menghubungkan dua kota tersebut, maka Algoritma Djikstra dapat digunakan untuk menentukan rute terpendek diantara dua kota tersebut.

Input dari algortima ini terdiri dari *weight* dari graph berarah (*direct graph*) *G*, dan *vertex* dari *graph G* tersebut. Untuk setiap set dari semua *vertex* pada *graph G* dinotasikan dengan *V*. Setiap *edge* dari *graph* adalah merupakan pasangan terurut dari *vertex (u,v)*, yang merepresentasikan koneksi antara *vertex u* dengan *vertex v*. Set dari semua *edge* dinotasikan dengan *E*. *Weight* dari setiap *edge* diberikan dengan fungsi weight w: $E \rightarrow [0, \infty)$; oleh karena itu nilai $w(u,v)$ akan selalu bernilai positif, dari setiap pergerakan dari *u* ke *v*. Nilai dari *edge* merupakan nilai jarak antara *vertex u* dengan *vertex v*, sedangkan nilai dari suatu *path* merupakan penjumlahan dari nilai setiap *edge* pada *path* tersebut.

2.5.3 Deskripsi algoritma

Algoritma Djikstra ini bekerja dengan cara menemukan nilai dari *d[v]* dari nilai rute terpendek yang ditemukan antara *s* dan *v*. Inisialisasi awal dari nilai ini adalah 0 untuk *vertex* awal (*source*) *s* (d[s]=0) dan tak terhingga untuk *vertex* yang lainnya, hal ini berdasarkan kenyataan bahwa pada awal kita tidak mengetahui nilai *vertex* tersebut (*d*[*v*]=∞ untuk setiap *vertex v* dalam set *vertex V*, kecuali *s*). Pada saat algortima selesai, d[v] akan mempunyai nilai rute terpendek dari *s* ke *v*, atau tak hingga apabila tidak ada path antara *s* ke *v*.

Operasi utama dari algortima Djikstra adalah *edge relaxation*. Yaitu suatu proses dimana jika ada *edge* dari *u* ke *v*, maka rute terpendek dari *s* ke *u* (*d[u]*) dapat diberikan pada *path* dari *s* ke *v* dengan cara menambah *edge* (*u,v*) di akhir. Path ini akan mempunyai panjang *d[u]+w(u,v)*. Jika nilai ini lebih sediki dari nilai *d[v]* yang ada, maka nilai *d[v]* akan digantikan dengan nilai baru yang lebih kecil tersebut. Proses *edge relaxation* tersebut akan dilakukan sampai semua nilai dari *d[v]* merepresentasikan nilai dari rute terpendek antara *s* dan *v*.

Algoritma ini menyimpan dua buah nilai dari set *vertex* S dan Q. Set S berisi semua *vertex* dimana nilai dari *d[v]* pertama kali didefinisikan, dan set Q berisi semua *vertex* yang lain. Sebuah *array pi* menyimpan nilai *predecesor* pada tiap *vertex*. Set dari S dimulai dengan nilai 0, dan setiap langkah, satu buah *vertex* dipindah dari Q ke S. *Vertex* yang dipindah ini merupakan *vertex* dengan nilai *d[u]* paling kecil. Pada saat *vertex u* dipindah ke *S*, algortima ini melakukan proses *relaxation* setiap meninggalkan *edge* (*u,v*).

Secara dasar, operasi-operasi pada algortima djikstra digambarkan sebagai berikut:

1. Inisialisasi nilai *d* dan *pi*
2. Set nilai S agar bernilai kosong
3. *Loop* jika masih ada *vertex* Q-S
    a. Urutkan *vertex* Q-S berdasarkan pada perkiraan nilai terbaik dari *vertex* asal

b. Tambahkan *veretex* u (*vertex* paling dekat) dalam Q-S, ke S
c. Lakukan proses *relaxation* pada semua *vertex* yang masih ada di Q-S yang terhubung dengan u.

Proses Inisilisasi

Proses ini menginisilisasi nilai dari *d* dan *pi*

```
initialise_single_source( Graph g, Node s )
   for each vertex v in Vertices( g )
       g.d[v] := infinity
       g.pi[v] := nil
   g.d[s] := 0;
```

Proses Rekaxation

Relaxation merupakan proses *update* pada nilai semua *vertex* v yang terhubung dengan *vertex* u, yang merupakan jarak terpendek.

```
relax( Node u, Node v, double w[][] )
    if d[v] > d[u] + w[u,v] then
       d[v] := d[u] + w[u,v]
       pi[v] := u
```

[Halaman Ini Sengaja Dikosongkan]

BAB III

PERANCANGAN SISTEM

Pada bab ini akan dijelaskan mengenai latar belakang permasalahan, arsitektur sistem, perancangan data, perancangan proses, serta perancangan antarmuka aplikasi-aplikasi yang terdapat dalam sistem.

3.1 Latar Belakang Permasalahan

Dalam tugas akhir ini, dikembangkan suatu aplikasi Simulasi Sistem Navigasi Traffic Lalu Lintas dengan Menggunakan Teknologi *Mobile agent*. Sistem ini mengadopsi dua macam teknologi yaitu teknologi AI (*Artificial Intelligence*) dengan memanfaatkan *agent*, dan teknologi Sistem Terdistribusi (*Distributed System*) yang memanfaatkan konsep *multiagent system*.

Adapun berbagai permasalahan yang diangkat dalam tugas akhir ini meliputi beberapa hal, yaitu sebagai berikut:

1. Apakah yang menjadi dasar dikembangkannya suatu sistem simulasi navigasi *traffic* lalu lintas
2. Bagaimana memanfaatkan suatu *agent* di dalam sistem navigasi *traffic* lalu lintas.
3. Bagaimana caranya melakukan pendistribusian data *traffic* yang ada dalam sistem.
4. Bagaimana mengimplementasikan konsep *multiagent* dalam membangun suatu sistem terdistribusi, dalam studi kasus sistem simulasi navigasi *traffic* lalu lintas.
5. Bagaimana caranya memanfaatkan teknologi *mobile agent* di dalam pencarian data pada masing-masing *server* terdistribusi.
6. Interaksi-interaksi apa saja yang terjadi diantara *agent-agent* yang ada, dan bagaimana mengimplementasikan interaksi tersebut.

7. Bagaimana proses pencarian rute terpendek, serta algoritma apa yang digunakan untuk mencari rute terpendek tersebut

3.2 Deskripsi Umum Sistem

Aplikasi ini dirancang untuk mempermudah orang dalam mendapatkan data mengenai suatu lokasi daerah tertentu. Dengan kata lain sistem ini membantu *user* dalam menemukan rute dari satu lokasi asal ke lokasi tujuan. Ketika seorang *user* melakukan *request* ke sistem untuk mencari lokasi tertentu, maka sistem akan merespon dengan melakukan komputasi pada data graph yang ada pada database untuk mencari rute terpendek. Kemudian akan mengirimkan hasilnya ke *user* tersebut. *Request user* ke sistem ditangani oleh suatu *agent*, begitu juga untuk proses komputasi pencarian rute terpendek.

Sistem ini terdiri dari beberapa *server* database yang menyimpan data *graph*. *Server* database tersebut ada dua macam, yaitu *server* database yang menyimpan data *graph* lokal tiap *local area* tertentu, dan *server* database yang menyimpan data *graph* global untuk keseluruhan area. Pada tiap *server* database tersebut terdapat sebuah *agent*, yang akan bertanggung jawab dalam menerima *request* dari agent lain untuk melakukan proses pencarian *path*, termasuk didalamnya adalah proses komputasi untuk menghitung rute terpendek dari *path* tersebut. *Agent* yang menangani database *local area* dinamakan *Graph Agent* (GA). Sedangkan agent yang menangani database *global area* dinamakan *Virtual Agent* (VA).

Pemodelan *graph* yang digunakan di dalam sistem ini, menggunakan pemodelan *two-level graph*. *Two-level graph* adalah pemodelan *graph* dimana suatu *graph* dibagi menjadi dua level yaitu *graph* global dan *graph* lokal. *Graph* global merupakan *graph* yang menggambarkan area secara keseluruhan, sedangkan *graph* lokal adalah *graph* yang menggambarkan beberapa sub-area yang terdapat pada *graph*

global. Secara singkat *graph* global akan dibagi-bagi ke dalam *graph* yang lebih kecil lagi yang disebut *graph* lokal.

Secara umum sistem yang dibangun nanti terdiri dari beberapa bagian subsistem yang menangani proses-proses tertentu, yaitu:

a. Proses *request user*.

Proses request *user* ini ditangani oleh suatu *agent*, yang dinamakan *Query Agent*. *Query Agent* ini di-*generate* dari *agent* yang berada pada *client* yang disebut *Client* Agent . *Client* Agent ini berhubungan langsung dengan *interface user*, dan menerima *request path* dari lokasi asal ke lokasi tujuan.

Setelah menerima *request* dari *user*, *Client* Agent akan meng-*generate Query Agent* sekaligus mengirimkan data lokasi asal dan tujuan. Selanjutnya *Query Agent* akan berpindah dari lokasi *client* ke lokasi *server* tertentu yang menyimpan data *graph* yang akan dicari. Setelah itu *Query Agent* akan meneruskan request ke *agent* yang terdapat pada *server* tersebut, untuk melakukan proses komputasi pencarian *path*.

b. Proses komputasi pencarian rute terpendek.

Proses pencarian rute terpendek ini ditangani oleh *agent* yang terdapat pada masing-masing *server*. *Agent* ini ada dua macam, yaitu *agent* yang terdapat pada *server* lokal yang menangani data pada *local area* graph, dan *agent* yang terdapat pada *server* global yang menangani data pada *global area* graph.

Request user yang dibawa oleh *Query Agent* pertama kali akan ditangani oleh *agent* yang terdapat pada *server* lokal. Sedangkan agent yang terdapat pada *server* global akan menangani *request path*, apabila tujuan *path* tersebut tidak berada pada area yang sama dengan *client*.

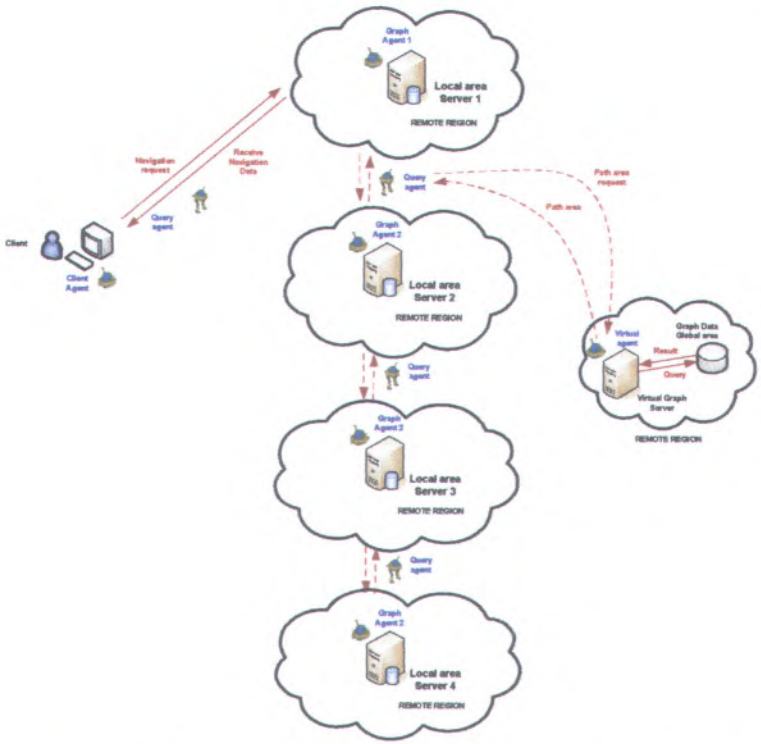
3.3 Perancangan Arsitektur Sistem

Sistem yang dibuat ini merupakan sistem yang berbasis pada konsep sistem terdisribusi. Di dalam sistem ini terdapat *client* yang akan melakukan *request* informasi mengenai rute tercepat antara suatu lokasi asal ke lokasi tujuan. Sedangakan *server*, akan menerima *request* dari *user* tersebut, dan kemudian melakukan proses komputasi dari data *graph* yang ada pada *server* itu untuk mencari rute tercepat.

Sistem ini dikembangkan diatas suatu *middleware*, yang merupkan suatu software sistem terdistribusi yang berada diantara *layer operating system* dan *layer distributed application*. *Middleware* yang digunakan dalam sistem ini berupa *framework* JADE, yaitu suatu *framework* yang berbasis java yang digunakan untuk mengembangkan dan membangun suatu *agent*. *Framework* ini juga menyediakan *resource* bagi *container*, yaitu suatu lingkungan aktif tempat tinggal suatu agent.

Proses komunikasi antara *client* dengan *server*, digunakan suatu *agent*, yang akan menjadi penghubung antara *client* dan *server*.

Sistem yang dikembangkan ini menggunakan arsitektur *multiagent*, dimana di dalamnya terdapat beberapa jenis *agent* yang dibedakan menurut fungsi dan tugasnya, yang masing-masing dapat berkomunikasi antara satu dengan lainnya. Masing-masing *agent* tersebut mempunyai tingkah laku yang berbeda-beda sesuai dengan fungsi dan tugasnya yang disebut *behaviour*. Arsitektur umum sistem ini dapat dilihat pada Gambar 3.1



Gambar 3.1 Arsitektur umum sistem navigasi traffic lalu-lintas

3.3.1 Arsitektur client

Pada sisi *client* ini terdapat suatu *interface* yang menghubungkan antara *user* dengan sistem. *Interface* ini terhubung dengan *framework* JADE yang ada pada komputer *client*. Di dalam *client* ini terdapat *Client Agent* yang akan berkomunikasi langsung dengan *interface*. Ketika *user* melakukan *request* untuk mencari *path* antara lokasi asal dengan lokasi tujuan, maka *Client Agent* akan meng-*generate* suatu *agent* yang dinamakan *Query Agent* pada komputer *client* tersebut.

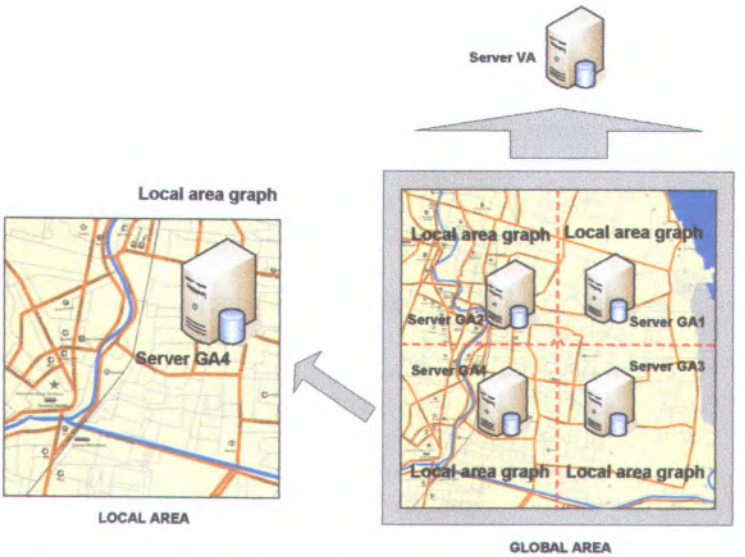
Query Agent ini nantinya yang akan membawa pesan dari *container* yang berada pada *host client* ke ke *container* yang berada pada *host server*. *Query Agent* merupakan *agent* yang memiliki sifat *mobile*, sehingga dapat berpindah dari *host client* ke *host server*, yang berada dalam satu buah *platform*. Proses pertukaran data antara *client* dengan *server* ditangani oleh *Query Agent*. Pada waktu proses perpindahan berlangsung, *Query Agent* akan membawa data-data dan *state-state* yang ada di dalamnya dari *client* ke *server*, begitu juga sebaliknya.

3.3.2 Arsitektur server

Di dalam sistem ini terdapat dua buah *server*, yaitu *server* yang menangani database yang menyimpan *log user* (*Query Agent*) yang dinamakan *log server*, dan *server* yang menangani database yang menyimpan data *graph*, yang dinamakan *graph server*. *Log server* ini nantinya akan melakukan pencatatan *log* terhadap semua aktifitas yang dilakukan oleh *Query Agent*. Sedangkan *graph server* akan melakukan manajemen terhadap database *graph*.

Seperti halnya pada pemodelan *graph*, di dalam *graph server* ini digunakan pemodelan *server* dua level. Dimana terdapat dua macam *server* yang akan berinteraksi dengan *user*, yaitu *server* yang yang menangani database *local area graph*, dan *server* yang menagani database *global area graph*.

Server yang menangani *local area graph* dinamakan *server GA*. *Server* ini terdiri dari beberapa buah sesuai dengan pembagian *local area graph* yang ada di dalam sistem. Sehingga untuk setiap *local area graph* akan ditangani oleh satu buah *server GA*. Di dalam masing-masing *server* ini terdapat suatu *agent* yang hidup, yang dinamakan *Graph Agent* (GA). *Graph Agent* akan menangani request data *path* yang diterima dari *Query Agent*. Selanjutnya *Graph Agent* akan melakukan proses komputasi untuk menentukan rute terpendek dari lokasi asal ke lokasi tujuan, sesuai dengan yang diminta oleh *client*.



Gambar 3.2 Arsitektur server

Server yang menangani *global area graph* dinamakan *server VA*. *Server* ini hanya terdapat satu buah, yang menangani database *global area graph*, dan menerima *request* untuk melakukan pencarian *path area* pada *graph*. *Request* pada *server* ini dilakukan apabila lokasi tujuan yang diminta oleh *user* tidak berada pada lokasi dimana *user* berada, sehingga *user* perlu melakukan *request* pada *server VA* untuk menanyakan *path* area menuju ke lokasi tujuan. Di dalam *server VA* ini terdapat satu buah *agent* yang dinamakan *Virtual Agent* (VA). *Agent* ini menerima *request* untuk melakukan pencarian dan penghitungan *path area* dari lokasi asal ke lokasi tujuan. Desain arsitektur server dapat dilihat pada Gambar 3.2

Data *path area* ini nantinya diperlukan oleh *Query Agent* untuk menentukan ke lokasi (area) mana saja dia harus berpindah, untuk mendapatkan *path* total dari tiap-tiap area.

3.4 Perancangan Proses

Pada sub-bab ini akan dijelaskan mengenai proses-proses apa saja yang terdapat di dalam sistem ini. Secara umum seperti pada Gambar-3.2 algoritma sistem ini dijelaskan sebagai berikut:

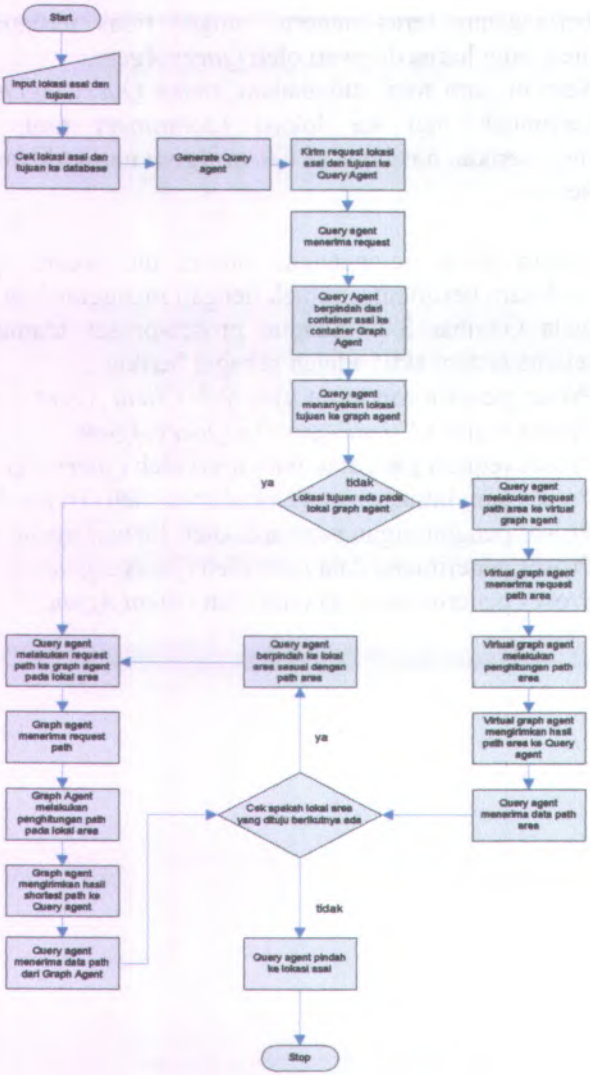
1. *User* akan melakukan *request* data *path* pada lokal area tertentu, dengan parameter yang dimasukkan berupa lokasi asal dan lokasi tujuan *user*.
2. Sistem terlebih dahulu akan melakukan pengecekan pada data input lokasi asal dan lokasi tujuan apakah ada dalam database atau tidak.
3. Jika data ada, *Client Agent* yang terdapat pada *client* akan meng-*generate Query Agent*. *Query Agent* akan meneruskan *request user* ke *Graph Agent* pada lokal area tersebut, dengan terlebih dahulu menanyakan apakah lokasi tujuan *user* berada pada lokal area tersebut.
4. Jika lokasi tujuan berada pada lokal area, maka *Query Agent* akan mengirimkan data lokasi asal dan tujuan ke *Graph Agent*. Kemudian *Graph Agent* akan melakukan penghitungan untuk mencari rute terpendek antara dua lokasi tersebut. Setelah itu hasilnya akan dikirim balik ke *Query Agent*
5. Jika lokasi tujuan tidak berada pada lokal area, maka *Query Agent* akan menanyakan ke *Virtual agent* yang melakukan *maintenance* pada database *global area graph*. *Virtual agent* akan melakukan proses penghitungan *path area*, yaitu lokal area mana saja yang harus dilewati dari lokasi awal menuju ke lokasi tujuan.
6. Setelah *path area* didapatkan, maka *Virtual agent* akan mengirimkan hasilnya ke *Query Agent*.
7. Setelah itu *Query Agent* akan berpindah ke lokal area sesuai dengan *path area* yang didapatkan dari *Virtual agent*, untuk menanyakan *path* pada *Graph Agent* di masing-masing area tersebut. Proses ini akan

berlangsung terus-menerus hingga tidak terdapat lagi area yang harus dilewati oleh *Query Agent*.

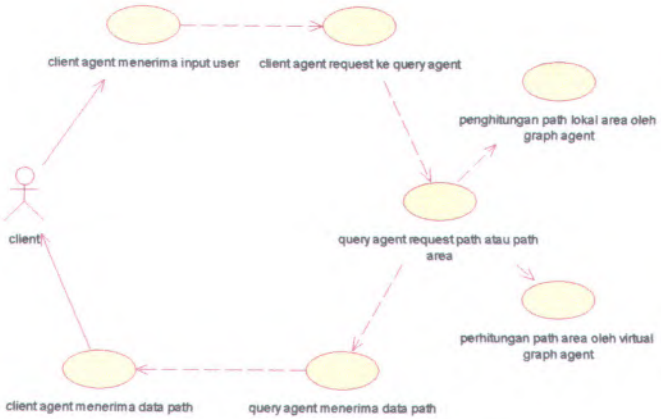
8. Setelah *path* total didapatkan, maka *Query Agent* akan berpindah lagi ke lokasi (*container*) asal, untuk memberikan hasilnya ke *Client Agent* untuk ditampilkan ke *user*.

Dalam tahap perancangan proses ini, secara spesifik digunakan desain berorientasi objek dengan menggunakan UML, seperti pada Gambar 3.4. Adapun proses-proses utama yang bekerja selama sistem aktif adalah sebagai berikut :

1. Proses penerimaan input *user* oleh *Client Agent*
2. Proses request *Client Agent* ke *Query Agent*.
3. Proses request path atau *path area* oleh *Query Agent*
4. Proses penghitungan *path local area* oleh *Graph Agent*
5. Proses penghitungan *path* area oleh *Virtual agent*
6. Proses penerimaan data *path* oleh *Query Agent*
7. Proses penerimaan data path oleh *Client Agent*



Gambar 3.3 Flowchart Proses di Dalam Sistem

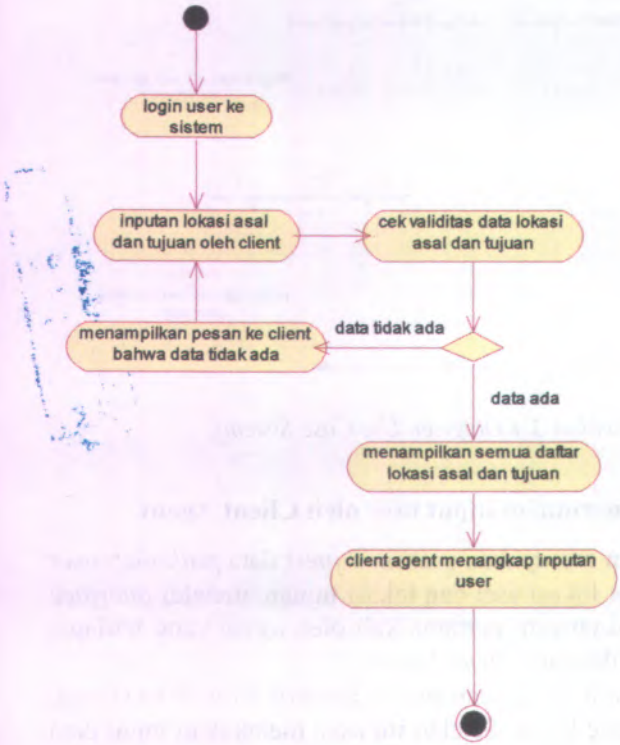


Gambar 3.4 Diagram Use Case Sistem

3.4.1 Proses penerimaan input user oleh Client Agent

Proses ini merupakan proses *request* data *path* oleh *user* dengan parameter lokasi asal dan lokasi tujuan, melalui *interface user*. Proses ini ditangani pertama kali oleh *agent* yang terdapat pada *client* yang disebut *Client Agent*.

Proses ini di mulai dari proses autentikasi *user* ke sistem, melalui mekanisme login. Setelah itu *user* melakukan input data lokasi asal dan lokasi tujuan. Setelah data lokasi asal dan lokasi tujuan divalidasi maka inputan dari *user* tadi akan diterima oleh *Client Agent* yang terdapat di dalam aplikasi *client*.



Gambar 3.5 Activity Diagram proses penerimaan input user oleh Client Agent

3.4.2 Proses request Client Agent ke Query Agent

Proses ini merupakan proses meneruskan *request* data *path* dari Client Agent ke *Query Agent*. Setalah *Client Agent* menangkap inputan *user*, maka Client Agent akan meng-*generate Query Agent* dengan nama seperti nama *user* login, pada *container* yang terdapat pada aplikasi *client*. Tiap *agent* akan mempunyai nama (*Agent ID*) yang berbeda untuk tiap

client. Ketika *Query Agent* dibuat, maka database *log* juga akan dibuat dengan nama seperti nama *Query Agent*.

Setelah *Query Agent* terbentuk maka nilai parameter *request path* yang berupa lokasi asal dan lokasi tujuan akan dilewatkan oleh *Client Agent* ke *Query Agent*.

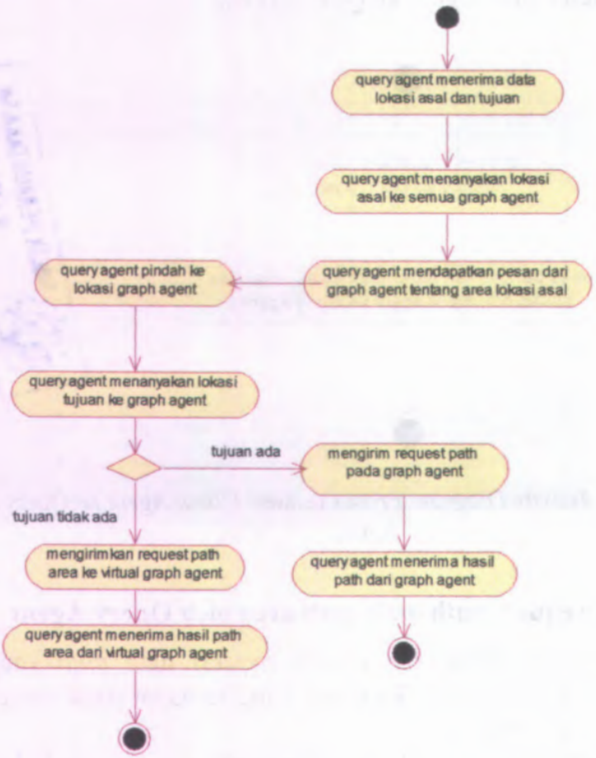
Gambar 3.5 Activity Diagram Proses request *Client Agent ke* Query Agent

3.4.3 Proses request path atau path area oleh Query Agent

Proses ini merupakan proses *request* data *path* yang dilakukan oleh *Query Agent* ke *agent* yang terdapat pada *server* lokal maupun *server* global.

Setelah *Query Agent* menerima data *request path* dari Client Agent, maka *Query Agent* akan menanyakan nama *container* ke semua *Graph Agent* yang teregistrasi dalam AMS. Selanjutnya *Query Agent* akan berpindah ke lokasi *container* (*host*) *Graph Agent*, dimana data lokasi asal terdapat di dalam databasenya. Setelah itu *Query Agent* akan menanyakan lokasi tujuan pada *Graph Agent* tersebut. Jika lokasi tujuan berada pada database *local area* tersebut, maka *Query Agent* akan mengirimkan *request path* ke *Graph Agent* tersebut. Sebaliknya

jika lokasi tujuan tidak berada pada database *local area* tersebut, maka *Query Agent* akan mengirimkan *request path* area ke *Virtual agent*.



Gambar 3.6 Activity Diagram Proses request path *atau* path area *oleh* Query Agent

3.4.4 Proses penghitungan path local area oleh Graph Agent

Proses penghitungan rute terpedek dilakukan oleh *Graph Agent* setelah menerima *request path* dari *Query Agent*. Setelah mendapatkan hasilnya, maka *Graph Agent* akan mengirim hasil penghitungan rute terpendek tersebut pada *Query Agent* yang terdapat pada *container* yang sama dengan *Graph Agent*.

Gambar 3.7 Activity Diagram Proses penghitungan path local area oleh Graph Agent

3.4.5 Proses penghitungan path area oleh Virtual Agent

Proses ini dilakukan oleh *Query Agent* ketika lokasi tujuan yang dicari, tidak terdapat pada *local area* yang sama

dengan lokasi awal. Setelah menerima *request path area* oleh *Query Agent* dengan parameter lokasi asal dan lokasi tujuan, maka *Virtual agent* akan melakukan penghitungan untuk mencari data *path area* antara tersebut.

Setelah path area didapatkan, maka *Virtual agent* akan mengirimkan hasil penghitungan tersebut ke *Query Agent*.

Gambar 3.8 Activity Diagram Proses penghitungan path area oleh Virtual agent

3.4.6 Proses penerimaan data path oleh Query Agent.

Query Agent menerima pesan *path* atau *path area* dari *Graph Agent* atau *Virtual agent*. Setelah itu *Query Agent* akan melakukan pengecekan terhadap asal pesan tersebut. Jika pesan berasal dari *Graph Agent*, maka *Query Agent* akan langsung mengirim data path tersebut ke *Client Agent*. Sebaliknya apabila pesan tersebut berasal dari *Virtual agent*, maka *Query Agent*

akan pindah ke *container* (*host*) *Graph Agent* yang terdapat pada daftar path area, untuk menanyakan *path* pada masing-masing *local area* tersebut.

Gambar 3.9 Activity Diagram Proses penerimaan data path oleh Query Agent

3.4.7 Proses penerimaan data path oleh Client Agent

Setelah *Client Agent* menerima data *path* dari lokasi asal ke lokasi tujuan yang berasal dari *Query Agent*, maka *Client Agent* akan menampilkan hasil path tersebut ke *interface user* yang terdapat pada aplikasi *client*.

Gambar 3.10 Proses penerimaan data path oleh Client *Agent*

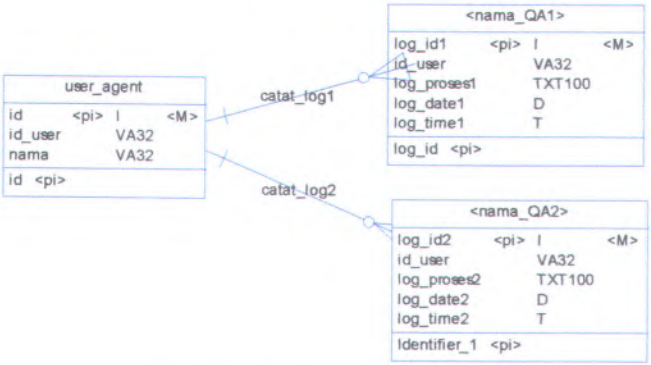
3.5 Perancangan Data

Pada sub-bab ini akan dijelaskan mengenai perancangan data yang ada pada sistem, yaitu perancangan konseptual data dan perancangan fisikal data.

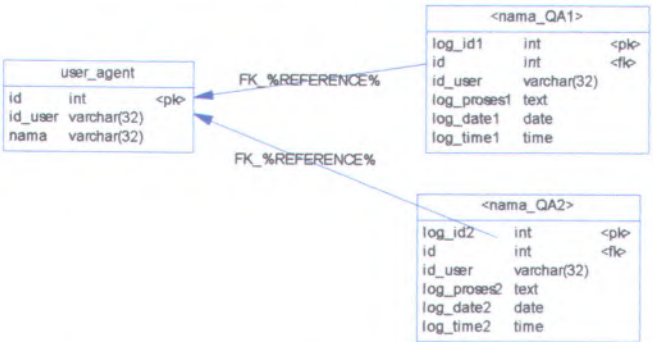
Dalam sistem ini terdapat tiga macam perancangan data, yaitu perancangan data pada database *client*, perancangan data pada *server local area graph*, dan perancangan data pada *server global area graph*.

3.5.1 Perancangan data pada client

Database yang ada pada *client* ini digunakan untuk menyimpan data *log* segala aktifitas dari *Query Agent* yang di-*generate* oleh *client*. Database ini terdiri dari dua buah tabel yaitu tabel *datalog* dan tabel nama *Query Agent*. Tabel nama *Query Agent* akan dibuat setiap kali *client* meng-*generate Query Agent*, dan memiliki nama yang sama dengan *Query Agent*.



Gambar 3.11 Desain CDM pada database client



Gambar 3.12 Desain PDM pada Database Client

Tabel 3.1 Tabel user_agent

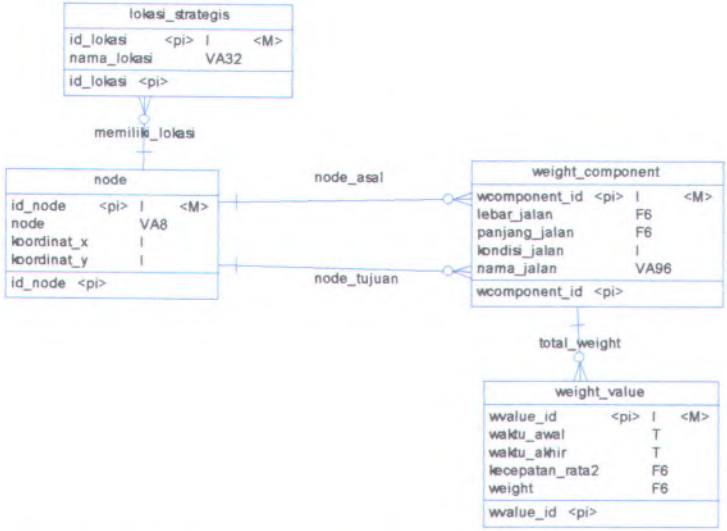
id_field	Deskripsi	tipe & length	keterangan
ID	id kolom	integer	primary_key
ID_*user*	id *client*	varchar (32)	
nama	nama *client*	varchar (32)	

Tabel 3.2 Tabel <nama Query Agent>

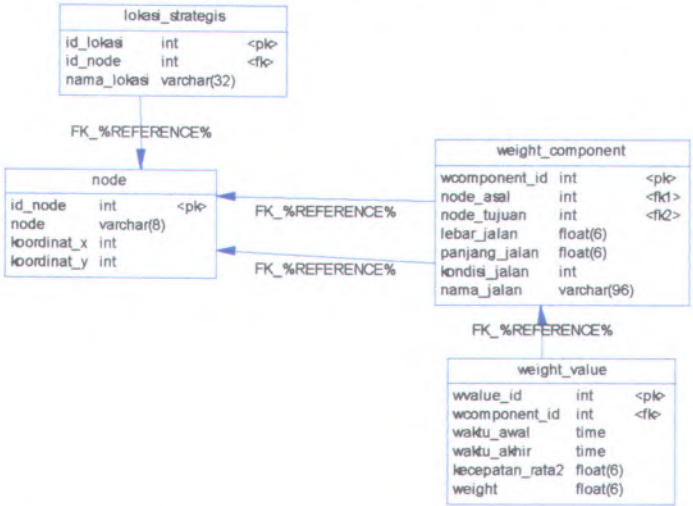
id_field	deskripsi	tipe & length	keterangan
log_id	id log proses	integer	primary_key
id	id datalog	integer	foreign key
id_user	id client	varchar (32)	
log_proses	Data log setiap aksi dari qery agent	text	
log_date	tanggal data masuk	date	
log_time	waktu data masuk	time	

3.5.2 Perancangan data pada server local area

Database ini terdapat pada masing-masing *server GA* pada *local area graph*, dan digunakan untuk menyimpan data graph pada lokal area tertentu. Database ini akan diakses oleh *Graph Agent* (GA) ketika ada *request* dari *Query Agent* untuk melakukan penghitungan pencarian rute tercepat. Terdiri dari beberapa tabel, yaitu tabel *node*, tabel *lokasi_strategis*, tabel *weight_component*, tabel *weight_value*.



Gambar 3.12 Desain CDM pada Database server area lokal



Gambar 3.13 Desain PDM pada Database local area server

Tabel 3.3 Tabel lokasi_strategis

id_field	deskripsi	tipe & length	keterangan
id_lokasi	id lokasi strategis	integer	primary_key
id_node	id node	integer	foreign key
nama_lokasi	nama lokasi strategis	varchar (32)	

Tabel 3.4 Tabel node

id_field	deskripsi	tipe & length	keterangan
id _node	id node	integer	primary_key
node	nama node	varchar(8)	
koordinat_x	posisi absis node	integer	
koordinat_y	posisi ordinat node	integer	

Tabel 3.5 Tabel weight_component

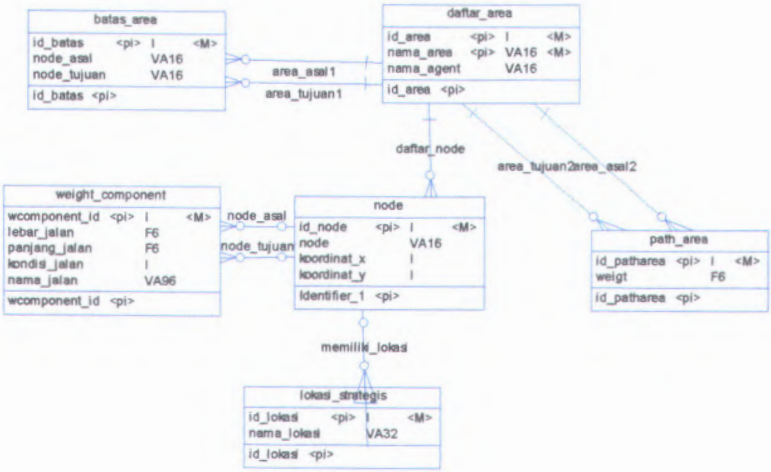
id_field	deskripsi	tipe & length	keterangan
wcomponent_id	id weight component	integer	primary_key
node_asal	node asal	varchar(8)	foreign_key
node_tujuan	node tujuan	varchar(8)	foreign_key
lebar_jalan	lebar jalan	float	
panjang_jalan	panjang jalan	float	
kondisi_jalan	kondisi_jalan	integer	
nama_jalan	nama jalan antara node asal dengan node tujuan	varchar(96)	

Tabel 3.6 Tabel weight_value

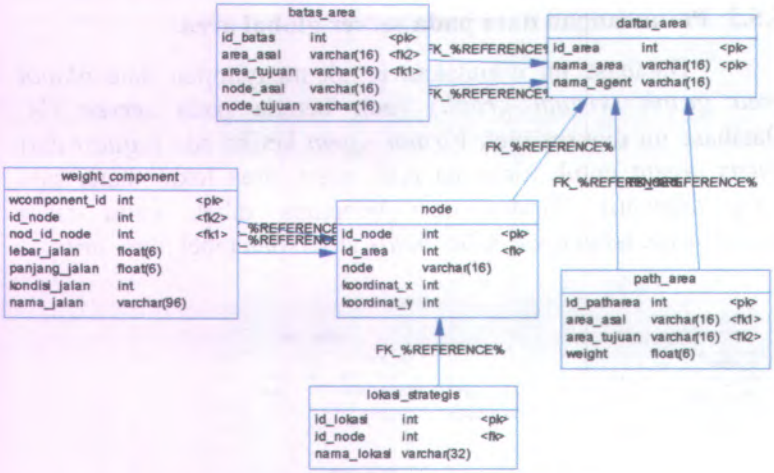
id_field	deskripsi	tipe & length	keterangan
wcomponent_id	id weight component	integer	foreign_key
wvalue_id	id weight value	integer	primary_key
waktu_awal	waktu awal	time	
waktu_akhir	waktu akhir	time	
kecepatan_rata2	kecepatan rata-rata kendaraan	double	
weight	weight	double	

3.5.3 Perancangan data pada server global area.

Database ini digunakan untuk menyimpan data *global area graph* (*virtual graph*) yang berada pada *server VA*. Database ini diakses oleh *Virtual Agent* ketika ada *request* dari *Query Agent* untuk meminta *path area* (area lokal mana saja yang dilewati). Terdiri dari beberapa tabel, yaitu tabel *daftar_area*, tabel *node*, tabel *batas_area*, dan tabel *path_area*.



Gambar 3.14 Desain CDM pada Database global area server



Gambar 3.15 Desain PDM pada Database global area server

Tabel 3.7 Tabel daftar_area

id_field	deskripsi	tipe & length	keterangan
id area	id area	integer	primary key
nama_area	nama area	varchar(16)	
nama_agent	nama agent yang menangani area	varchar(16)	

Tabel 3.8 Tabel batas_area

id_field	deskripsi	tipe & length	keterangan
id_batas	id batas	integer	primary key
area_asal	area asal batas	varchar(16)	foreign key
area_tujuan	area tujuan batas	varchar(16)	foreign key
node_asal	node batas asal	varchar(16)	
node_tujuan	node batas tujuan	varchar(16)	

Tabel 3.9 Tabel node

id_field	deskripsi	tipe & length	keterangan
id_node	id node	integer	primary key
id_area	id area	integer	foreign key

id_field	deskripsi	tipe & length	keterangan
node	nama node	varchar(16)	
koordinat_x	koordnat absis x	integer	
koordinat_y	koordinat absis y	integer	

Tabel 3.10 Tabel path_area

id_field	Deskripsi	tipe & length	keterangan
id_patharea	id path area	integer	primary key
area_asal	area batas asal	varchar(16)	foreign key
area_tujuan	area batas tujuan	varchar(16)	foreign key
Weight	weight area	float	

Tabel 3.11 Tabel lokasi_strategis

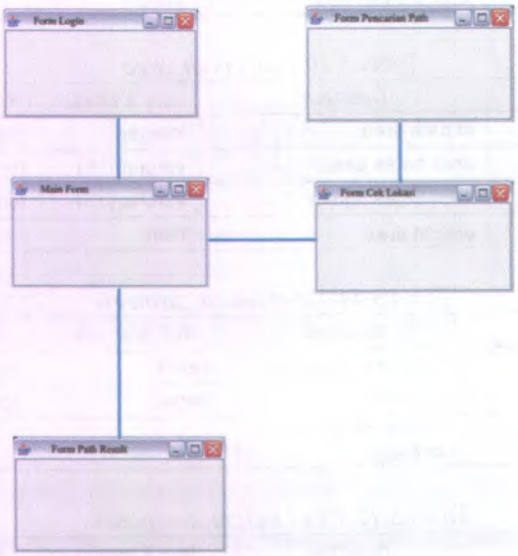
id_field	deskripsi	tipe & length	keterangan
id_lokasi	id lokasi strategis	integer	primary_key
id_node	id node	integer	foreign key
nama_lokasi	nama lokasi strategis	varchar (32)	

Tabel 3.12 Tabel weight_component

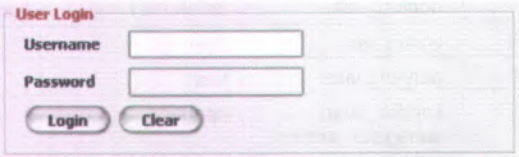
id_field	deskripsi	tipe & length	keterangan
wcomponent_id	id weight component	integer	primary_key
node_asal	node asal	varchar(8)	foreign_key
node_tujuan	node tujuan	varchar(8)	foreign_key
lebar_jalan	lebar jalan	float	
panjang_jalan	panjang jalan	float	
kondisi_jalan	kondisi_jalan	integer	
nama_jalan	nama jalan antara node asal dengan node tujuan	varchar(96)	

3.6 Perancangan Antarmuka

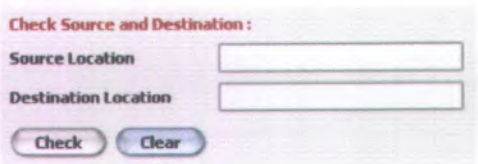
3.6.1 Perancangan antar muka client



Gambar 3.16 Antar muka interface client



Gambar 3.17 Antar muka login user



Gambar 3.17 Antar muka cek source *dan destination location*

Gambar 3.19 Antar muka pencarian path

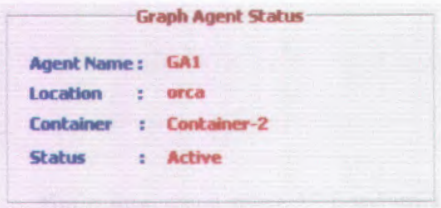
Gambar 3.20 Antar muka Query Agent Log Process

Gambar 3.21 Antar muka path result

3.6.2 Perancangan antar muka *server*

3.6.2.1 Perancangan antar muka server *GA*

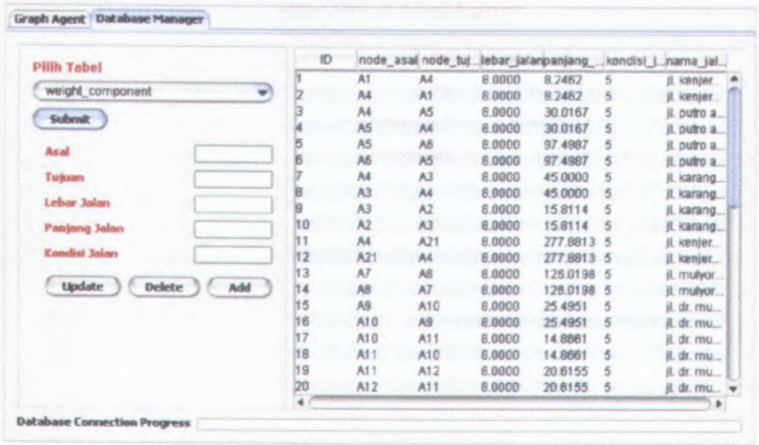
Gambar 3.22 Antar muka server GA



Gambar 3.23 Antar muka Graph Agent Status

Gambar 3.23 Antar muka Graph Agent Status

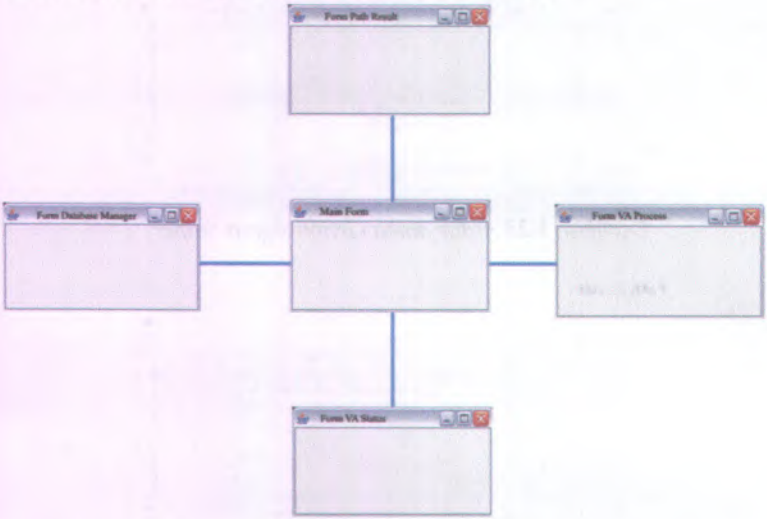
Gambar 3.25 Antar muka Path Result



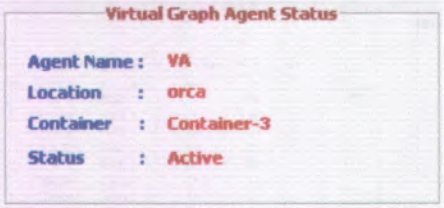
ID	node_asal	node_tuj	lebar_jalan	panjang_	kondisi_j	nama_jal
1	A1	A4	8.0000	8.2462	5	jl. kenjer
2	A4	A1	8.0000	8.2462	5	jl. kenjer
3	A4	A5	8.0000	30.0167	5	jl. putro a
4	A5	A4	8.0000	30.0167	5	jl. putro a
5	A5	A6	8.0000	97.4987	5	jl. putro a
6	A6	A5	8.0000	97.4987	5	jl. putro a
7	A4	A3	8.0000	45.0000	5	jl. karang
8	A3	A4	8.0000	45.0000	5	jl. karang
9	A3	A2	8.0000	15.8114	5	jl. karang
10	A2	A3	8.0000	15.8114	5	jl. karang
11	A4	A21	8.0000	277.8813	5	jl. kenjer
12	A21	A4	8.0000	277.8813	5	jl. kenjer
13	A7	A8	8.0000	126.0198	5	jl. mulyor
14	A8	A7	8.0000	126.0198	5	jl. mulyor
15	A9	A10	8.0000	25.4951	5	jl. dr. mu
16	A10	A9	8.0000	25.4951	5	jl. dr. mu
17	A10	A11	8.0000	14.8661	5	jl. dr. mu
18	A11	A10	8.0000	14.8661	5	jl. dr. mu
19	A11	A12	8.0000	20.6155	5	jl. dr. mu
20	A12	A11	8.0000	20.6155	5	jl. dr. mu

Gambar 3.26 Antar muka Database Manager

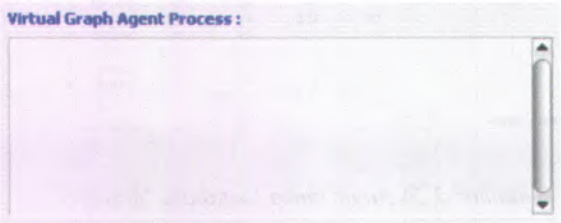
3.6.2.2 Perancangan antar muka server VA



Gambar 3.27 Antar muka server VA

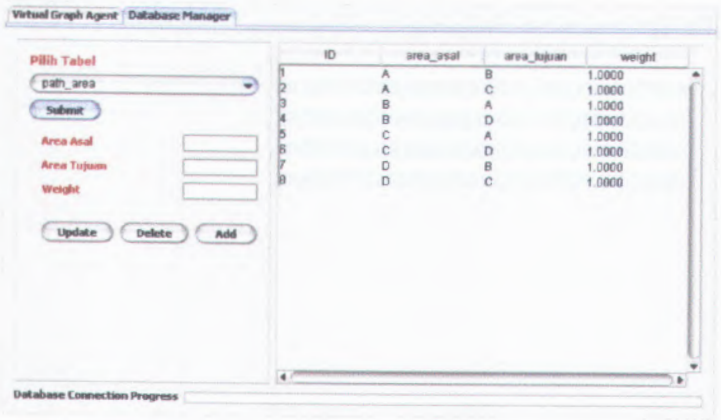


Gambar 3.28 Antar muka Virtual agent Status



Gambar 3.29 Antar muka Virtual agent Status

Gambar 3.30 Antar muka Path Result



Gambar 3.31 Antar muka Database Manager

[Halaman Ini Sengaja Dikosongkan]

BAB IV

IMPLEMENTASI SISTEM

4.1 Implementasi Database

Pada sub bab ini akan dijelaskan tentang implementasi database yang ada di dalam sistem. Sesuai dengan desain database yang ada tahap perancangan data, database ini dibagi menjadi tiga, yaitu: database *client*, database *server local area*, dan database *server global area*.

4.1.1 Implementasi database client

Database *client* ini menyimpan login *user* yang akan mengakses sistem. Selain itu database ini juga menyimpan *action log* dari *Query Agent*. Database ini terdiri dari tabel *user_agent* dan tabel *<Query Agent>* yang akan menyimpan *action log*.

- *Script* DDL untuk tabel *user_agent*

```
CREATE TABLE `user_agent` (
  `Id` int(6) unsigned NOT NULL auto_increment,
  `id_user` varchar(16) default NULL,
  `name` varchar(32) default NULL,
  `passwd` text,
  PRIMARY KEY  (`Id`)
)
```

- *Script* DDL untuk tabel *<nama agent>*

```
CREATE TABLE `<nama Query Agent>` (
  `log_id` int(11) NOT NULL auto_increment,
  `id_user` varchar(16) default NULL,
  `log_process` text,
  `log_date` date default NULL,
  `log_time` time default NULL,
  PRIMARY KEY  (`log_id`)
)
```

4.1.2 Implementasi database server local area

Database *server local area* menyimpan data *graph* pada *local area* tertentu, yang kemudian akan diakses dan diolah oleh *Graph Agent* yang terdapat pada *local area* terebut untuk mendapatkan rute terpendek dari lokasi asal ke lokasi tujuan. Database ini terdiri tabel *node*, *lokasi_strategis*, *weight_component*, dan *weight_value*.

- *Script* DDL untuk tabel *node*

```
CREATE TABLE `node` (
  `id_node` int(11) unsigned NOT NULL
   auto_increment,
  `node` varchar(8) default NULL,
  `koordinat_x` int(11) default NULL,
  `koordinat_y` int(11) default NULL,
  PRIMARY KEY  (`id_node`)
)
```

- *Script* DDL untuk tabel *lokasi_strategis*

```
CREATE TABLE `lokasi_strategis` (
  `Id_lokasi` int(6) unsigned NOT NULL
auto_increment,
  `nama_node` varchar(8) default NULL,
  `nama_lokasi` varchar(64) default NULL,
  PRIMARY KEY  (`Id_lokasi`)
)
```

- *Script* DDL untuk tabel *weight_component*

```
CREATE TABLE `weight_component` (
  `wcomponent_id` int(6) unsigned NOT NULL
auto_increment,
  `node_asal` varchar(8) default NULL,
  `node_tujuan` varchar(8) default NULL,
  `lebar_jalan` float(16,4) default '0.0000',
  `panjang_jalan` float(16,4) default '0.0000',
  `kondisi_jalan` int(11) default '0',
  `nama_jalan` varchar(96) default NULL,
  PRIMARY KEY  (`wcomponent_id`)
)
```

- *Script* DDL untuk tabel *weight_value*

```
CREATE TABLE `weight_value` (
  `wvalue_id` int(11) NOT NULL auto_increment,
  `wcomponent_id` int(11) default NULL,
  `waktu_awal` time default NULL,
  `waktu_akhir` time default NULL,
  `kecepatan_rata2` double(16,4) default '0.0000',
  `weight` double(16,4) default '0.0000',
  PRIMARY KEY  (`wvalue_id`)
)
```

4.1.3 Implementasi database server global area

Database pada *server global area* ini menyimpan data-data *global graph* beserta data-data *graph* lainnya seperti batas-batas antar area. Database ini akan diakses oleh *Virtual Agent* untuk mendapatkan *path area* yang akan dilewati oleh *Query Agent* dari lokasi asal ke lokasi tujuan. Database ini terdiri dari beberapa tabel, yaitu: *node*, *lokasi_strategis*, *weight_component*, *daftar_area*, *path_area*, dan *batas_area*.

- *Script* DDL untuk tabel *node*

```
CREATE TABLE `node` (
  `id_node` int(11) unsigned NOT NULL
auto_increment,
  `id_area` int(11) default NULL,
  `node` varchar(8) default NULL,
  `koordinat_x` int(11) default NULL,
  `koordinat_y` int(11) default NULL,
  `graph_flag` tinyint(6) default NULL,
  PRIMARY KEY  (`id_node`)
)
```

- *Script* DDL untuk tabel *lokasi_strategis*

```
CREATE TABLE `lokasi_strategis` (
  `Id_lokasi` int(6) unsigned NOT NULL
auto_increment,
  `nama_node` varchar(8) default NULL,
  `nama_lokasi` varchar(64) default NULL,
  PRIMARY KEY  (`Id_lokasi`)
)
```

- *Script* DDL untuk tabel *weight_component*

```
CREATE TABLE `weight_component` (
  `wcomponent_id` int(6) unsigned NOT NULL
auto_increment,
  `node_asal` varchar(8) default NULL,
  `node_tujuan` varchar(8) default NULL,
  `lebar_jalan` float(16,4) default '0.0000',
  `panjang_jalan` float(16,4) default '0.0000',
  `kondisi_jalan` int(11) default '0',
  `nama_jalan` varchar(96) default NULL,
  PRIMARY KEY  (`wcomponent_id`)
)
```

- *Script* DDL untuk *daftar_area*

```
CREATE TABLE `daftar_area` (
  `id_area` int(6) unsigned NOT NULL
auto_increment,
  `nama_area` varchar(16) default NULL,
  `nama_agent` varchar(4) default NULL,
  PRIMARY KEY  (`id_area`)
)
```

- *Script* DDL untuk *path_area*

```
CREATE TABLE `path_area` (
  `id_patharea` int(6) unsigned NOT NULL
auto_increment,
  `area_asal` varchar(16) default NULL,
  `area_tujuan` varchar(16) default NULL,
  `weight` double(16,4) default NULL,
  PRIMARY KEY  (`id_patharea`)
)
```

- *Script* DDL untuk *batas_area*

```
CREATE TABLE `batas_area` (
  `id_batas` int(6) unsigned NOT NULL
   auto_increment,
  `area_asal` varchar(16) default NULL,
  `area_tujuan` varchar(16) default NULL,
  `node_asal` varchar(16) default NULL,
  `node_tujuan` varchar(16) default NULL,
  PRIMARY KEY  (`id_batas`)
)
```

4.2 Implementasi Graph

Pada sub-bab ini akan dijelaskan mengenai implementasi graph yang ada di dalam sistem, yang meliputi implementasi

data graph yang dipakai dalam sistem, dan implementasi penghitungan nilai bobot antar node dari data graph yang ada.

4.2.1 Implementasi data graph

Di dalam Tugas Akhir ini data graph yang ada pada sistem merupakan data simulasi yang diambil dari studi kasus sebagian area yang ada pada peta surabaya pada skala 1 : 161

Gambar 4.1 Global Area Graph

Data *graph* global dari keseluruhan area yang ada, akan dipecah menjadi beberapa data *graph* lokal yang disimpan ke dalam database yang berbeda-beda. Dimana di dalam masing-masing database tersebut akan disimpan informasi graph berupa nama *lokasi strategis* (nama lokasi tertentu), nama node,

koordinat node, nama jalan, panjang jalan, lebar jalan, kondisi jalan, dan kecepatan rata-rata antara dua node. Posisi *lokasi strategis* yang ada dalam graph, ditentukan berdasarkan jarak terdekat lokasi strategis tersebut dengan suatu node.

Gambar 4.2 Local Area Graph

Proses mendapatkan data dilakukan dengan cara menentukan node-node yang berada di sepanjang jalan yang ada

di dalam peta. Node-node ini terhubung antara satu dengan lainnya melalui suatu edge yang berada pada jalur jalan yang ada pada peta. Selanjutnya data node yang berupa nama node dan lokasi koordinat node pada peta akan disimpan di dalam tabel *node*.

Object Browser | Data Browser | SQL Editor

id_node	node	koordinat_x	koordinat_y
1	A1	4	122
2	A2	3	182
3	A3	12	169
4	A4	12	124
5	A5	13	94
6	A6	48	3
7	A7	3	306
8	A8	128	322
9	A9	3	347
10	A10	28	352
11	A11	39	362
12	A12	59	367

Gambar 4.3 Tabel node

Pengukuran jarak antar dua node dilakukan dengan menggunakan metode *manhatan distance* (|x1-x2| + |y1-y2|). Setelah data jarak antara dua buah node didapatkan, maka data tersebut akan disimpan di dalam tabel *weight_component*. Sedangkan data lebar jalan, dan kondisi jalan yang ada pada tabel *weight_component*, merupakan data simulasi.

wcomponent	node_asal	node_tujuan	lebar_jalan	panjang_jalan	kondisi_jalan	nama_jalan
1	A1	A4	8.0000	8.2462	5	jl. kenjeran
2	A4	A1	8.0000	8.2462	5	jl. kenjeran
3	A4	A5	8.0000	30.0167	5	jl. putro agung wetar
4	A5	A4	8.0000	30.0167	5	jl. putro agung wetar
5	A5	A6	8.0000	97.4987	5	jl. putro agung wetar
6	A6	A5	8.0000	97.4987	5	jl. putro agung wetar
7	A4	A3	8.0000	45.0000	5	jl. karang asem
8	A3	A4	8.0000	45.0000	5	jl. karang asem
9	A3	A2	8.0000	15.8114	5	jl. karang asem
10	A2	A3	8.0000	15.8114	5	jl. karang asem
11	A4	A21	8.0000	277.8813	5	jl. kenjeran
12	A21	A4	8.0000	277.8813	5	jl. kenjeran
13	A7	A8	8.0000	126.0198	5	jl. mulyorejo
14	A8	A7	8.0000	126.0198	5	jl. mulyorejo

Gambar 4.3 Tabel weight_component

Data-data yang ada pada tabel *weight_component* tersebut menentukan data *weight* (nilai bobot antara dua node) yang ada pada tabel *weight_value*. Di dalam table *weight_value* tersebut terdapat data mnegenai keceptan rata-rata antara dua node dalam periode waktu tertentu, dan nilai bobot antara dua node.

wvalue_id	wcomponent_id	waktu_awal	waktu_akhir	kecepatan_rata2	weight
1	1	06:00:00	06:59:59	80.5000	0.0706
2	1	07:00:00	08:59:59	40.5000	0.1404
3	1	09:00:00	14:59:59	70.6000	0.0806
4	1	15:00:00	17:59:59	40.5000	0.1404
5	1	18:00:00	21:59:59	70.7000	0.0804
6	1	22:00:00	23:59:59	100.8000	0.0564
7	1	00:00:00	05:59:59	100.8000	0.0564
8	2	06:00:00	06:59:59	80.5000	0.0706
9	2	07:00:00	08:59:59	40.5000	0.1404
10	2	09:00:00	14:59:59	70.6000	0.0806
11	2	15:00:00	17:59:59	40.5000	0.1404
12	2	18:00:00	21:59:59	70.7000	0.0804
13	2	22:00:00	23:59:59	100.8000	0.0564
14	2	00:00:00	05:59:59	100.8000	0.0564

Gambar 4.3 Tabel weight_value

4.2.2 Implementasi penghitungan nilai bobot

Di dalam Tugas Akhir ini nilai bobot (*weight*) yang ada pada database, digunakan di dalam penghitungan algoritma Djikstra, untuk menentukan rute terpendek antara node yang satu dengan node yang lain.

Di dalam Tugas Akhir ini diasumsikan bahwa, nilai bobot antara dua node yang berada pada tabel *weight value* ditentukan oleh beberapa komponen, yaitu komponen lebar jalan, panjang jalan, kondisi jalan, kecepatan rata-rata (yang berubah tiap periode waktu tertentu).

Untuk penghitungan nilai bobot antara dua node digunakan rumus berikut.

$$Weight = \frac{\text{lebar\_jalan x panjang\_jalan x faktor\_kondisi\_jalan x skala}}{\text{Kecepatan\_rata2}}$$

Dimana lebar jalan berada dalam satuan *m*, panjang jalan antara dua buah node berada pada satuan *mm*, faktor kondisi jalan merupakan faktor skala kondisi jalan, nilai skala adalah faktor skala jarak antara dua node pada peta, sedangkan kecepatan rata-rata berada pada satuan m/s.

Karena di dalam perhitungan ini, nilai *weight* berbanding lurus dengan waktu tempuh antara dua buah node, maka penghitungan rute terpendek di dalam algortima djikstra ditentukan berdasarkan nilai *weight* yang paling rendah atau paling kecil.

Penghitungan nilai *weight* yang dipakai diatas masih merupakan asumsi berdasarkan pengamatan pada kondisi nyata. Dan untuk simulasi yang dilakukan pada sistem ini, data-data pendukung untuk percobaan, merupakan data yang di-*generate* terlebih dahulu (bukan data *realtime*).

4.3 Implementasi Aplikasi Client

Aplikasi yang ada pada *client* tersebut menangani *request path* pada sistem, dengan parameter berupa lokasi asal dan lokasi tujuan. Pada aplikasi *client* tersebut terdapat suatu *interface* yang menghubungkan antara *user* dengan sistem.

Agent yang berperan pada aplikasi *client* ini ada dua yaitu *agent* yang berinteraksi dengan *interface user* yang disebut *Client Agent*, dan *agent* yang menerima *request path* dari *user* yang disebut *Query Agent*

4.3.1 Implementasi Client Agent dalam sistem

Client *Agent* merupakan *agent* yang terdapat dalam aplikasi *client* yang mempunyai fungsi utama mengkomunikasikan antara *interface user* dengan sistem terutama *Query Agent*. *Agent* ini berhubungan langsung dengan

interface user, sehingga dapat menangkap inputan yang berasal dari *user* dan *event-event* yang terjadi pada *interface user*.

Ketika *interface user* pertama kali diaktifkan, maka aplikasi *client* ini juga mengaktifkan *Client Agent* dan *container* agent dimana *Client Agent* dapat hidup.

Kelas *Client Agent* ini diturunkan dari kelas *Agent*. Untuk dapat terintegrasi dengan *interface user* yang merupakan turunan dari kelas *JFrame*, maka agent ini harus membuat suatu objek yang merupakan instansiasi dari kelas *QAInterface* (kelas yang mendefinisikan *interface user* dan merupakan turunan dari *JFrame*). Sebuah *method setupUI()* digunakan untuk mendefinisikan properti objek hasil instansiasi dari kelas *QAInterface*, dan dipanggil di dalam *method setup()*.

```
private void setupUI() {
  try {
      javax.swing.UIManager.setLookAndFeel
                  (new LiquidLookAndFeel());
      this.QAFrame =  new QAInterface(this);
      this.QAFrame.setSize(865,550);
      this.QAFrame.setLocation(100,100);
      this.QAFrame.setVisible(true);
      this.QAFrame.validate();
  }
  catch(Exception ex) {
      ex.printStackTrace();
  }
}
```

Method yang lain yang terdapat di dalam kelas ini yaitu *sendMessageToQA()* dengan parameter berupa *String asal*, *String tujuan*, dan *String nama_agent*. *Method* ini dipanggil pada saat terdapat *action event* pada tombol *search* di dalam *interface*. *Method* ini digunakan untuk menangkap *request path* dari *user* berupa lokasi asal dan lokasi tujuan, kemudian dikirimkan ke *Query Agent*.

```
protected void sendMessageToQA(String asal, String
tujuan,String agentname) {
  try {
     ACLMessage msg = new ACLMessage
                      (ACLMessage.INFORM);
     msg.addReceiver(new AID(agentname,
                     AID.ISLOCALNAME));
     msg.setOntology("agent-communication-
                     ontology");
     msg.setContent("getpath_"+asal+"_"+tujuan);
     send(msg);
     msg.clearAllReceiver();
  }
  catch(Exception ex){
     ex.printStackTrace();
  }
}
```

Method *createQueryAgent()* dipanggil pada saat terdapat *action event* pada tombol login. Method ini dignakan untuk membuat *Query Agent* pada saat *user* melakukan login. Pembuatan *Query Agent* ini dilakukan melalui kelas *AgentController* dengan memanggil method *createNewAgent()* pada objek *container* yang memiliki parameter berupa nama *agent*, kelas *agent*, dan *argument.*

```
protected void createQueryAgent() {
   try {
     Object[] args=new Object[4];
     args[0] =this.getLocalName();
     PlatformController container =
                        getContainerController();
     AgentController QA = container.createNewAgent
       (this.QAName,"queryagent.QueryAgent",args);
     QA.start();
  }
   catch(Exception ex) {
      ex.printStackTrace();
   }
}
```

Di dalam method *setup()* yang terdapat pada kelas *Client Agent* ini dipanggil kelas *CyclicBehaviour*, yang di dalamnya terdapat method *receive()* yang akan terus mendengarkan pesan dari *Query Agent*. Apabila tidak ada pesan yang datang maka *behaviour* ini akan berada pada kondisi blok dengan pemangilan method *block()*. Ketika ada pesan yang datang dari *Query Agent* dengan parameter "path_" maka akan dilakukan proses penulisan pesan tersebut kedalam *interface user*.

```
for(int i=0;i<temp2.length;i++) {
  temp3 = temp2[i].split("-");
  ((QAInterface) QAFrame).setTextJTArea2(temp3[1]+
     " -> "+temp3[0]+" KM");
}
```

4.3.2 Implementasi Query Agent dalam sistem

Query Agent merupakan salah satu dari beberapa *agent* yang ada dalam sistem. *Query Agent* ini merupakan *agent* yang berinteraksi dengan *Client Agent*. Fungsi utama dari *Query Agent* ini adalah sebagai *agent* penghubung antara *Client Agent*

(aplikasi *client*) dengan *Graph Agent* (aplikasi *server*) yang ada pada *server*.

Query Agent di definisikan ke dalam kelas *public QueryAgent* yang diturunkan dari kelas *Agent*. Di dalam kelas *Query Agent* ini terdapat beberapa kelas *inner* atau sub-kelas yang akan diinstansi di dalam objek *Query Agent* tersebut. Sub-kelas tersebut mendefinisikan struktur data maupun *behaviour* yang dimiliki oleh *Query Agent* dalam sistem. Karena *Query Agent* ini mempunyai sifat *mobile* (dapat bergerak dari *host* yang satu ke *host* yang lain), maka diperlukan pendefinisian semua kelas yang dipakai oleh *Query Agent* menjadi kelas *inner* dari kelas *Query Agent*. Hal ini dilakukan agar pada waktu *Query Agent* berpindah ke *host* yang lain, *behaviour* (*state* dan data) yang ada di dalamnya juga ikut berpindah.

Kelas *Query Agent* terdiri dari beberapa *method* utama. *Method setup()* pertama kali dijalankan pada saat agent melakukan *startup* dan baru saja tersegistrasi dalam AMS, *method beforeMove()* yang akan melakukan aksi sebelum *agent* berpindah lokasi, dan *method afterMove()* yang akan melakukan aksi setelah *agent* berpindah lokasi. Pada *method setup()* dipanggil *behaviour CheckLocationSource* dan *QueryAgentBehaviour*. Sedangkan pada *method beforeMove()* dan *afterMove()* dipanggil *behaviour WriteLogBehaviour*.

```
public Class QueryAgent extends Agent{
        ..........
        private Location destination;
        ..........
        protected void setup() {
        ..........
        addBehaviour(new WriteLogBehaviour
        ("create Query Agent named:
        "+this.getAID().toString()));
        addBehaviour(new CheckLocationSource(argCmd));
        addBehaviour(new QueryAgentBehaviour(argCmd));
        }

        protected void beforeMove() {
        addBehaviour(new WriteLogBehaviour("agent in
         "+this.getContainerController().getName()
        +" and will move to
        "+this.destination.getName()));
        }

        protected void afterMove() {
        addBehaviour(new WriteLogBehaviour("agent has
         moved to "+this.destination));
        }
        ..........
}
```


4.3.2.1 Kelas QueryAgentBehaviour

Kelas *QueryAgentBahaviour* merupakan salah satu *behaviour* yang diturunkan dari kelas *CyclicBehaviour*. Kelas ini memiliki *method action()* yang mendefisikan aksi-aksi yang terdapat di dalam *behaviour*. Di dalam method ini terdapat variabel *msg* dengan tipe data *ACLMessage* yang akan menerima semua pesan yang dikirm dari *agent* yang lain melalui *method receive()*. *Behaviour* ini akan diaktifkan secara terus menerus untuk melakukan *listening* pesan dari agent yang lain.

Ketika tidak ada pesan yang datang maka dipanggil *method block()* yang menjadikan *state behaviour* ini berada dalam kondisi *block* (*idle*). Sebaliknya ketika ada pesan yang

datang dan ditangkap oleh *method receive()*, maka *behaviour* ini melakukan sejumlah aksi sesuai dengan protokol yang didefinisikan.

```
Class QueryAgentBehaviour extends CyclicBehaviour {
        private String argCmd;

        public QueryAgentBehaviour(String cmd) {
        this.argCmd=cmd;
        }

        public void action() {
        try {
                ACLMessage msg = receive();
                if (msg!= null) {
                ..............
                //melakukan aksi sesuai dengan protocol
                //dari pesan yang diterima
                ..............
                }
                else {
                block();
                }
        }
        catch(Exception e) {
                e.printStackTrace();
        }
        }
}
```

4.3.2.2 Kelas MobilityBehaviour

Kelas *MobilityBehaviour* diturunkan dari kelas *Behaviour*. *Behaviour* ini dipanggil di dalam kelas QueryAgentBehaviour ketika ada pesan dengan *content* berupa *recheck_true, path_* , dan *patharea_* yang berasal dari *Graph Agent*. Kelas ini memiliki parameter *constructor* berupa *String com* (berupa perintah "move") dan *String dst* (berupa nama *container* tujuan perpindahan). Di dalam *method action()* terdapat pemanggilan fungsi *doMove()* dengan parameter berupa nama *container* yang berada dalam tipe data *Location*.

```
Class MobilityBehaviour extends Behaviour {
        private String cmd;
        private String destContainer;
        private AID controller;

        public MobilityBehaviour(String com,String dst) {
                this.cmd=com;
                this.destContainer=dst;
        }

        public void action() {
         try {
           if(this.cmd.equalsIgnoreCase("move")) {
             Location dest = new jade.core.ContainerID
                             (destContainer,null);
             destination = dest;
             myAgent.doMove(dest);
           }
         }
        }
        catch(Exception ex) {
                ex.printStackTrace();
        }
        public boolean done() {
                return true;
        }
}
```

Method *doMove()* ini digunakan oleh agent untuk melakukan perpindahan dari satu *container* ke *container* yang lain.

4.3.2.3 Kelas WriteLogBahaviour

Kelas *WriteLogBehaviour* ini diturunkan dari kelas *Behaviour*. *Behaviour* ini dipanggil oleh *behaviour* yang lain yang ada di dalam *Query Agent*. *WriteLogBehaviour* ini akan melakukan pencatatan segala aktifitas dari *Query Agent* pada database *log*.

Di dalam method *action()* yang ada di dalam kelas ini, terdapat objek *db* yang merupakan instansiasi dari kelas *dbConnection()*. Objek ini akan memanggil method *queryExec()*

dengan parameter berupa isi *log*, untuk melakukan pencatatan *log* pada database.

```
Class WriteLogBehaviour extends Behaviour {
        private String eventLog;
        public WriteLogBehaviour(String log) {
                this.eventLog=log;
        }
        public void action() {
          try {
           dbConnection db = new dbConnection();
           db.Connect();
           db.queryExec("insert into "
            +myAgent.getLocalName()+" "+
           "(id_user,log_process,log_date, log_time)
            values('"+myAgent.getLocalName()+
           "','"+this.eventLog+"',CURDATE(),CURTIME())");
           db.Disconnect();
          }
         catch(Exception e) {
            e.printStackTrace();
         }
        }
        public boolean done() {
          return true;
        }
}
```

4.3.2.4 Kelas DeleteLogTable

Kelas ini diturunkan dari kelas *Behaviour*. *Behaviour* ini dipanggil oleh *Query Agent* ketika semua proses (*task*) yang ada di dalam *QueryAgent* selesai dilakukan. *Behaviour* ini akan melakukan penghapusan pada tabel *log Query Agent*.

```
Class DeleteLogTable extends Behaviour {
        public DeleteLogTable() {
        }
        public void action() {
          try {
           dbConnection db = new dbConnection();
           db.Connect();
           db.queryExec("drop table "+
                        myAgent.getLocalName());
           db.Disconnect();
          }
          catch(Exception ex) {
            ex.printStackTrace();
          }
        }
        public boolean done() {
          return true;
        }
}
```

4.3.2.5 Kelas CreateLogTable

Kelas *CreateLogTable* diturunkan dari kelas *Behaviour*. *Behaviour* ini dijalankan segera setelah *Query Agent* selesai dibuat. *Behaviour* ini melakukan proses pembuatan tabel *log* pada database *log*.

```
Class DeleteLogTable extends Behaviour {
        public DeleteLogTable() {
        }
        public void action() {
          try {
           dbConnection db = new dbConnection();
           db.Connect();
           db.queryExec("create table "
            +myAgent.getLocalName()+" (log_id integer
            AUTO_INCREMENT,id_user varchar(16),
           log_process text, log_date date,log_time time,
            primary key (log_id))");
           db.Disconnect();
          }
          catch(Exception ex) {
            ex.printStackTrace();
          }
        }
        public boolean done() {
          return true;
        }
}
```

4.3.2.6 Kelas CheckLocationSource

Kelas ini diturunkan dari kelas *Behaviour*. *Behaviour* ini akan dijalankan oleh *Query Agent* untuk melakukan pengecekan data lokasi asal yang dicari. Di dalam Kelas *CheckLocationSource* ini terdapat satu *method* utama, yaitu *method askSourceToGA()*, yang dipanggil di dalam *method action()* dan memiliki parameter berupa *source node*.

Method *askSourceToGA()* akan mengirimkan pesan untuk menanyakan lokasi asal ke semua *Graph Agent* yang terdaftar dalam AMS. Untuk itu terdapat mekanisme untuk mendapatkan daftar semua nama *Graph Agent* yang teregistrasi di dalam AMS, yaitu melalui pemanggilan *method search()* yang ada di dalam kelas *AMSService*. Hasil balik dari method ini adalah berupa *array* dari *AMSAgentDescription*.

```
AMSAgentDescription [] agents = null;
.....
SearchConstraints c = new SearchConstraints();
c.setMaxResults (new Long(-1));
agents = AMSService.search( myAgent, new
                            AMSAgentDescription(), c );
```

Setelah daftar semua *agent* tersebut didapatkan dari AMS, maka akan dilakukan proses *looping* untuk mengirimkan pesan ke *Graph Agent* (mempunyai awalan GA). Di bawah ini terdapat *code* untuk mengirimkan pesan ke *Graph Agent*.

```
ACLMessage mess = new ACLMessage(ACLMessage.CFP);
.....
for (int i=0; i<agents.length;i++)
{
 if (agents[i].getName().toString().contains("GA")) {
   agentID = new AID(agents[i].getName().getLocalName()
             ,AID.ISLOCALNAME);
   mess.addReceiver(agentID);
   mess.setOntology("agent-communication-ontology");
   mess.setContent("checksrc_"+msg+"_"+i);
   myAgent.send(mess);
   mess.clearAllReceiver();
 }
}
```

Method addReceiver() memiliki parameter berupa *agentID* (nama *agent* yang dikirimi pesan). Isi dari pesan didefinisikan dalam *method setContent()* yang memiliki parameter berupa pesan ("checkscr_<lokasi asal>"). Sedangkan untuk mengirimkan pesan digunakan *method send()* dengan parameter berupa objek pesan. Sedangkan *method clearAllReceiver()* digunakan untuk membersihkan semua daftar *receiver* yang ditambahkan.

4.3.2.7 Kelas GetPathBehaviour

Kelas *GetPathBehaviour* diturunkan dari kelas *Behaviour*. *Behaviour* ini akan dipanggil di dalam kelas QueryAgentBehaviour. Yaitu ketika ada pesan *recheck_true* dari *Graph Agent*.

Behaviour ini melakukan aksi dengan mengirim pesan ke *Graph Agent* untuk menanyakan *path* dari lokasi asal ke lokasi tujuan. Konstruktor dari kelas *GetPathBehaviour* ini mempunyai parameter berupa nama *agent* yang dikirimi pesan dan isi dari pesan yang dikirim. Dibawah ini adalah *source* dari *method action()* yang terdapat pada kelas *GetPathBehaviour*.

```
public GetPathBehaviour(String name, String content) {
       this.agentName=name;
       this.msgContent=content
}
public void action() {
       try {
         message.addReceiver(new AID(agentName,
         AID.ISLOCALNAME));
         message.setOntology("agent-communication-
         ontology");
         message.setContent(msgContent);
         myAgent.send(message);
         message.clearAllReceiver();
       }
       catch(Exception e) {
         e.printStackTrace();
       }
}
```

4.3.2.8 Kelas PathVariabel

Kelas ini digunakan untuk mendeklarasikan struktur data yang akan digunakan untuk menampung data *path* yang berupa daftar *Graph Agent* dari lokasi yang akan dilewati, dan node batas antara dua buah *graph* pada *local area* yang berbeda. Kelas ini dipanggil di dalam kelas *QueryAgentBehaviour* ketika ada pesan *listarea_* yang berasal dari *Virtual agent*. Setelah pesan tersebut diterima maka parameter *listarea_* akan diparsing dan

kemudian ditampung dalam struktur data tersebut. Di dalam kelas ini terdapat *method setter* dan *getter* yang akan melakukan set nilai ke variabel di dalamnya dan mendapatkan nilai dari variabel itu.

```
Class PathVariabel implements Serializable {
        private String areaName;
        private LinkedList areaAsal;
        private LinkedList areaTujuan;
        private LinkedList nodeAsal;
        private LinkedList nodeTujuan;

        public PathVariabel() {
          this.areaName=null;
          this.areaAsal= new LinkedList();
          this.areaTujuan= new LinkedList();
          this.nodeAsal= new LinkedList();
          this.nodeTujuan= new LinkedList();
        }
        ..........
        //method setter dan getter variabel
        ..........
}
```

4.3.2.9 Kelas dbConnection

Kelas *dbConnection* merupakan kelas *public* yang akan diinstansiasi menjadi objek koneksi ke database. Kelas ini memiliki lima *method* utama yang digunakan untuk melakukan koneksi ke database, dan operasi *sql* lainnya.

Method Connect() digunakan untuk melakukan koneksi ke database *log Query Agent*. Didalam method ini didefiniskan pemanggilan *driver* koneksi ke database, nama *server*, nama database, *url* koneksi, nama *user*, dan *password*.

```
public boolean Connect() {
  try {
    // mysql driver
    String name = "org.gjt.mm.mysql.Driver";
    Kelas.forName(name);

    //create connection to database.
    String server = "10.126.11.151"; //server database
    String dbase = " datalog"; //database name
    String url = "jdbc:mysql://"+server+"/"+dbase;
    String user = "queryagent"; //user
    String pass = "riset"; //password
    //create connection
    con = DriverManager.getConnection(url,user,pass);
    if (con!= null) {
      //System.out.println("koneksi berhasil");
    }
    return true;
  }
  catch(Exception ex) {
    System.out.println(ex);
    return false;
  }
}
```

Method Disconect() digunakan untuk melakukan pemutusan koneksi ke database. Di dalam *method* ini dilakukan pemanggilan *method close()* pada objek koneksi.

```
//disconnect database
public boolean Disconnect() {
   try {
      if (!con.isClosed()) {
         con.close();
         return true;
      }
      else {
         return false;
      }
   }
   catch(Exception e) {
      System.out.println(e);
      System.out.println("tidak ada koneksi");
      return false;
   }
}
```

Method queryExec() digunakan untuk melakukan eksekusi perintah *sql* selain *query* ke objek database. *Method* ini tidak mengembalkan nilai (mempunya *return value void()*).

```
//method for execute sql
public void queryExec(String query) {
    try {
        stat = con.createStatement();
        stat.execute(query);
        stat.close();
    }
    catch(SQLException sqlx) {
        System.out.println(sqlx);
    }
}
```

Method dbRecordset() digunakan untuk melakukan *query* ke objek database (tabel atau *view*). *Method* ini mempunyai *return value* berupa *ResultSet* hasil dari query ke database tersebut.

```
public ResultSet dbRecordset(String query){
  ResultSet record=null;
  rs=null;
  try {
     stat = con.createStatement(ResultSet.
      TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
     rs = stat.executeQuery(query);
     record=rs;
  }
  catch(SQLException sqlx){
     System.out.println(sqlx);
  }
  return record;
}
```

Method rsClose() digunakan untuk melakukan penutupan *recordset. Method* ini mengambalikan nilai *boolean true* apabila *resulset* berhasil ditutup, dan sebaliknya akan mengembalikan nilai *boolean false.*

```
//method to close recordset
public boolean rsClose(){
 try {
   rs.close();
   return true;
 }
 catch(Exception e){
   System.out.println(e);
```

4.4 Implementasi Aplikasi Server

Aplikasi *server* ini akan menangani setiap *request* dan pesan yang berasal dari *Query Agent.* Setiap pesan yang masuk, akan diterima oleh *agent* yang terdapat pada *server* ini. Kemudian agent tersebut akan melakukan operasi dan tindakan sesuai dengan pesan yang diterimanya.

Agent yang terdapat pada *server* ini dibagi menjadi dua jenis, yaitu *agent* yang terdapat pada *server local area* yang disebut *Graph Agent*, dan *agent* yang terdapat pada *server* global area yang disebut *Virtual agent*. Kedua *agent* tersebut akan menerima pesan dari *Query Agent*, dan melakukan interaksi secara langsung dengan database yang ada pada *server*. Implementasi masing-masing *agent* ini akan dijelaskan pada sub bab berikut ini.

4.4.1 Implementasi Graph Agent dalam sistem

Kelas *Graph Agent* diturunkan dari kelas *Agent*. *Graph Agent* ini memiliki tugas utama melakukan penghitungan rute terpendek dari lokasi asal ke lokasi tujuan, setelah menerima *request* (berupa pesan *getpath_* dan *getpatharea_*) dari *Query Agent*.

Di dalam *method setup()* yang terdapat pada *Graph Agent* ini, dipanggil sebuah *CyclicBehaviour* yang akan terus menunggu pesan yang datang melaui *method receive()*, dan akan berada dalam kondisi *block* jika tidak ada pesan yang datang.

```
addBehaviour(new CyclicBehaviour(this){
        private int i=1;
        public void action() {
          try {
            ACLMessage msg = receive();
            if(msg!=null) {
                ..............
                // melakukan beberapa aksi sesuai dengan
                // pesan yang diterima oleh Graph Agent
                ..............
            }
          }
          catch(Exception ex) {
            ex.printstacktrace();
          }
        }
```

Di dalam kelas *GraphAgent* ini terdapat *method CheckDataMethod()* yang mempunyai parameter berupa *String arg* yang menujukan nama *node* lokasi asal atau lokasi tujuan

dan *int argv* menunjukan jenis lokasi yang dicari. *Method* ini akan melakukan *query* ke database untuk mencari lokasi yang di maksud. Jika lokasi yang dimaksud ada, maka akan menghasilkan *return value true*, dan sebaliknya akan menghasilkan *return value false* jika lokasi yang dimaksud tidak ada.

```
//method that check source is available
public boolean CheckDataMethod(String arg,int argv) {
   boolean checkData=false;
   ResultSet rs=null;
   try {
        dbConnection db = new dbConnection();
        db.Connect();
        if(argv==2) {
        rs= db.dbRecordset("select * from
             weight_component where node_tujuan like '"+
             arg+"'");
        }
        else if(argv==1) {
        rs= db.dbRecordset("select * from
             weight_component where node_asal like '"+
             arg+"'");
        }
        if (rs.next()) {
             checkData=true;
        }
        else {
             checkData=false;
        }
   }
   catch(Exception ex) {
        ex.printStackTrace();
   }
   return checkData;
}
```

4.4.1.1 Kelas GABehaviour

Kelas *GABehaviour* diturunkan dari kelas *Behaviour*, dan merupakan salah satu *behaviour* yang ada di dalam *Graph*

Agent. *GABehaviour* mempunyai parameter konstruktor berupa *ACLMessage*, dan *AID sender*. Ketika aktif, *behaviour* ini akan melakukan pengecekan pada lokasi tujuan yang diambil dari parameter *ACLMessage* (pesan yang dikirim oleh *Query Agent*). Pengecekan dilakukan dengan pemanggilan *method CheckDataMethod()*. Jika lokasi tujuan ada maka akan dilakukan pemangilan *behaviour FindPathBehaviour*, sebaliknya jika tidak ada maka akan mengirimkan pesan (*destination_<lokasi tujuan>_false*) kepada *Query Agent*. Inti dari *method action()* yang terdapat pada *behaviour* ini dapat dilihat sebagai berikut.

```
if(this.msgTemp[0].equalsIgnoreCase("getpath")) {
   if(this.msgTemp[1]!=null && this.msgTemp[2]!=null) {
     //check if the destination are in the database
     if(CheckDataMethod(this.msgTemp[2],2)==true) {
        //behaviour that request graph from Query Agent
        addBehaviour(new FindPathBehaviour(
                this.msgTemp[1],this.msgTemp[2],
                message.getSender(),this.msgTemp[0]));
     }
     else {
        ACLMessage reply = new
        ACLMessage(ACLMessage.INFORM);
        reply.setContent("destinaton_"+this.msgTemp[2]
        +"_false");
        reply.addReceiver(this.agentSender);
        send(reply);
        reply.clearAllReceiver();
     }
   }
  else {
     reply = new ACLMessage(ACLMessage.INFORM);
     reply.setContent("Warning!! message parameter is
     getpath-<source>-<destination>");
     reply.addReceiver(message.getSender());
     send(reply);
     reply.clearAllReceiver();
  }
}
```

4.4.1.2 Kelas FindPathBehaviour

Kelas ini diturunkan dari kelas *behaviour*, dan mendefinisikan *behaviour agent* yang melakukan pencarian *path* dari lokasi asal ke lokasi tujuan, sekaligus mengirimkan hasil rute terpendek tersebut ke *Query Agent*. Di dalam *method action()*, kelas ini menginstansiasi kelas *shortestPath()*. Untuk mendapatkan jarak terdekat antara lokasi asal dengan lokasi tujuan. Objek dari kelas tersebut memanggil *method Djikstra()*.

```
for (int k=0;k<destination.length;k++) {
   shortestPath sp = new shortestPath(source,
                     destination[k]);
   path = sp.Djikstra();
   if (Double.parseDouble(path.get(path.toArray().
   length-1).toString())<tmpWeight) {
     tmpWeight=Double.parseDouble(path.get(
     path.toArray().length-1).toString());
     pathMsg="";
     if (path!=null) {
        for (int i=0;i<path.toArray().length-1;i++) {
          if (pathMsg.equalsIgnoreCase("")) {
             pathMsg=path.get(i).toString();
          }
          else {
             pathMsg=pathMsg+"-"+path.get(i).toString();
          }
        }
        pathMsg=pathMsg+"-"+destination[k];
     }
   }
   path.remove();
}
```

Untuk pengiriman hasil penghitungan rute terpendek ke *Query Agent* dapat dilihat pada kode berikut ini.

```
if(this.command.equalsIgnoreCase("getpath")) {
   reply.setContent("path_"+pathMsg);
}
else if(this.command.equalsIgnoreCase("getpatharea")) {
   reply.setContent("patharea_"+pathMsg);
}
reply.addReceiver(sender);
send(reply);
reply.clearAllReceiver();
```

4.4.1.3 Kelas CheckSourceLocation

Kelas *CheckSourceLocation* merupakan salah satu *behaviour* yang terdapat di dalam *Query Agent*. Kelas ini akan diinstansiasi menjadi sebuah *behaviour* di dalam *GABehaviour*, pada waktu ada pesan untuk melakukan pengecekan lokasi asal (*checkscr_*) oleh *Query Agent*. Kelas ini memiliki parameter konstruktor berupa *String scr* (nama lokasi asal yang dicari), dan *AID sender* (AID *Query Agent* yang mengirimkan *request*).

Method *setup()* dalam kelas ini melakukan aksi berupa pengecekan lokasi asal dengan memanggil *method CheckDataMethod()*. Kemudian akan mengirimkan pesan *recheck_true_* kepada *Query Agent*, apabila data lokasi asal berada pada database *local area graph* tersebut. Sebaliknya akan mengirimkan pesan *recheck_false_*, apabila data lokasi asal tidak berada pada database tersebut.

```
if(CheckDataMethod(this.source,1)==true) {

   ACLMessage recheck = new ACLMessage(ACLMessage.
                        CONFIRM);
   recheck.setContent("recheck_true_"+myAgent.
        getContainerController().getContainerName());
   recheck.addReceiver(this.agentSender);
   send(recheck);
   recheck.clearAllReceiver();
   this.flagCheck=true;
}
else {
  ACLMessage recheck = new ACLMessage(ACLMessage.
                        CONFIRM);
   recheck.setContent("recheck_false_"+myAgent.
        getContainerController().getContainerName());
   recheck.addReceiver(this.agentSender);
   send(recheck);
   recheck.clearAllReceiver();
   this.flagCheck=false;
}
```

4.4.1.4 Kelas ShortestPath

Kelas *ShortestPath* ini akan diinstansiasi menjadi sebuah objek di dalam *behaviour FindPathBehaviour*. Konstruktor yang ada pada Kelas ini memiliki parameter berupa *String nstart* yang merupakan node asal, dan *String nstop* yang merupakan node tujuan. Kelas ini memiliki *method* utama *Djisktra()* yang memberikan *return value* berupa *LinkedList* dari *path* antara node asal dengan node tujuan. *Method Djikstra()* ini yang nantinya akan dipanggil oleh objek dari kelas *ShortestPath* yang terdapat pada *behaviour FindPathBehaviour*, untuk mendapatkan nilai *path* antara node asal dengan node tujuan.

Langkah pertama kali di dalam implementasi algortima Djikstra ini adalah kita memasukkan semua data graph yang ada pada database ke dalam objek variabel yang diinstansiasi dari kelas *graphComponent* (kelas yang berisi variabel-variabel yang menyimpan data graph). Data yang disimpan ke variabel berupa

nama node (*node[i].name*), node *adjacent* (*node[i].adjVertex*), dan *weight* antara node dengan node *adjacent* (*node[i].weight[jj]*).

```
dbConnection db = new dbConnection();
db.Connect();
rs = db.dbRecordset("SELECT DISTINCT node_asal FROM
weight_component");
rs.last();
size=rs.getRow(); //get the number of vertex
graphComponenent[] node =new graphComponenent[size];
rs.beforeFirst();
i=0;
while(rs.next()) {
   node[i]=new graphComponenent(size);
   node[i].name = rs.getString(1);
   query="SELECT node_asal,node_tujuan,waktu_awal,
          waktu_akhir,weight FROM weight_component,
          weight_value WHERE weight_component.
          wcomponent_id =weight_value.wcomponent_id "+
           "AND weight_component.node_asal like
          '"+rs.getString(1)+"' AND waktu_awal <=
           CURTIME() AND "+ "waktu_akhir > CURTIME()";
   rs2=db.dbRecordset(query);
   j=0;
   while (rs2.next()) {
     node[i].adjVertex.add(rs2.getString(2));
     node[i].weight[j]=Double.parseDouble(
                        rs2.getString(5));
     j=j+1;
  } i=i+1;
}
```

Selanjutnya akan dilakukan proses inisialisasi dari variabel node asal dan node tujuan.

```
//looping for initialization node label
for(i=0;i<node.length;i++) {
//Initialization for source node
   if(node[i].name.equals(nodeStart)) {
        node[i].vStart=nodeStart;
        node[i].vPath.add(nodeStart);
        node[i].totWeight=0.0;
   }
   else if (node[i].name.equals(nodeStop)) {
        node[i].vEnd=nodeStop;
        node[i].vPath.clear();
        node[i].totWeight=null;
   }
   //Initialization for destination node
   else {
        node[i].vPath.clear();
        node[i].totWeight=null;
   }
}
```

Setelah itu dilakukan pengecekan apakah node yang dilewati dalam iterasi sama dengan node tujuan (node akhir). Jika sama maka proses iterasi berhenti. Jika tidak sama maka akan dijalankan langkah-langkah dibawah ini.

- Melakukan pencarian node yang mempunyai nilai *totWeight* (*total weight*) paling kecil dan tidak berada pada node yang dilewati. Nilai dari node yang paling kecil tersebut dimasukkan ke variabel *passNode* (variabel yang menyimpan node yang sudah dilewati)

```
minTotalWeight=1000.0;
for (k=0;k<node.length;k++) {
   flag1=false;
   for(i=0;i<passNode.toArray().length;i++) {
      //check if node that has weight label is
      //not exist in passNode
      if (node[k].name.equalsIgnoreCase(passNode.get(i).
      toString())) {
         flag1=true;
      }
   }
   if(flag1==false) {
      if (minTotalWeight!=0.0 && node[k].totWeight
      !=null) {
         if (minTotalWeight>node[k].totWeight ) {
           minTotalWeight=node[k].totWeight;
           idx=k;
         }
       }
   }
}
passNode.add(node[idx].name);
```

- Memasukkan nilai *totWeight* pada setiap node yang *adjacent* dengan node terakhir yang berada pada variabel *passNode*.

```
//check if totWeight of adjacent node is null
if (node[k].totWeight==null) {
   //save totWeight
   node[k].totWeight=node[idx].totWeight+
                  node[idx].weight[j];
}
node[k].vPath.clear();
for(c=0;c<node[idx].vPath.toArray().length;c++) {
    node[k].vPath.add(node[idx].vPath.get(c).
                      toString());
}
node[k].vPath.add(node[idx].name);
```

- Melakukan *update* nilai *totWeight*, pada tiap node yang *adjacent* dengan node terakhir yang berada pada *passNode*, apabila nilai *totWeight* pada node tersebut lebih besar dari nilai *totWeight* node sebelumnya ditambah dengan nilai *weight* antara node tersebut dengan node sebelumnya.

```
for (k=0;k<node.length;k++) {
  for(j=0;j<node[idx].adjVertex.toArray().length;j++) {
     if(node[k].name.equalsIgnoreCase(node[idx].
     adjVertex.get(j).toString())) {
        if(node[idx].totWeight+node[idx].
         weight[j]<node[k].totWeight) {
            node[k].totWeight = node[idx].
                    totWeight+node[idx].weight[j];

        }
        pathWeight=node[k].totWeight;
     }
  }
}
```

4.4.1.5 Kelas dbConnection

Kelas *dbConnection* merupakan kelas *public* yang akan diinstansiasi menjadi objek koneksi ke database. Kelas ini memiliki lima method utama yang digunakan untuk melakukan koneksi ke database, dan operasi *sql* lainnya.

Method Connect() digunakan untuk melakukan koneksi ke database *local area graph*. Didalam *method* ini didefiniskan pemanggilan *driver* koneksi ke database, nama *server*, nama database, *url* koneksi, nama *user*, dan *password*.

```
public boolean Connect() {
  try {
    // mysql driver
    String name = "org.gjt.mm.mysql.Driver";
    Kelas.forName(name);

    //create connection to database.
    String server = "10.126.11.151"; //server database
    String dbase = "server1"; //database name
    String url = "jdbc:mysql://"+server+"/"+dbase;
    String user = "agent1"; //user
    String pass = "riset"; //password
    //create connection
    con = DriverManager.getConnection(url,user,pass);
    if (con!= null) {
      //System.out.println("koneksi berhasil");
    }
    return true;
  }
  catch(Exception ex) {
    System.out.println(ex);
    return false;
  }
}
```

Method Disconect() digunakan untuk melakukan pemutusan koneksi ke database. Di dalam method ini dilakukan pemanggilan *method close()* pada objek koneksi.

```
//disconnect database
public boolean Disconnect() {
   try {
      if (!con.isClosed()) {
         con.close();
         return true;
      }
      else {
         return false;
      }
   }
   catch(Exception e) {
      System.out.println(e);
      System.out.println("tidak ada koneksi");
      return false;
   }
}
```

Method queryExec() digunakan untuk melakukan eksekusi perintah *sql* selain *query* ke objek database. Method ini tidak mengembalikan nilai (mempunyai *return value void()*).

```
//method for execute sql
public void queryExec(String query) {
    try {
        stat = con.createStatement();
        stat.execute(query);
        stat.close();
    }
    catch(SQLException sqlx) {
        System.out.println(sqlx);
    }
}
```

Method dbRecordset() digunakan untuk melakukan *query* ke objek database (tabel atau view). *Method* ini mempunyai *return value* berupa *ResultSet* hasil dari *query* ke database tersebut.

```
public ResultSet dbRecordset(String query){
  ResultSet record=null;
  rs=null;
  try {
     stat = con.createStatement(ResultSet.
      TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
     rs = stat.executeQuery(query);
     record=rs;
  }
  catch(SQLException sqlx){
     System.out.println(sqlx);
  }
  return record;
}
```

Method rsClose() digunakan untuk melakukan penutupan *recordset*. *Method* ini mengambalikan nilai *boolean true* apabila *resulset* berhasil ditutup, dan sebaliknya akan mengembalikan nilai *boolean false*.

```
//method to close recordset
public boolean rsClose(){
  try {
     rs.close();
     return true;
  }
  catch(Exception e){
     System.out.println(e);
     return false;
  }
}
```

4.4.2 Implementasi Virtual Agent dalam sistem

Virtual agent merupakan *agent* yang menangani *server global area graph*. *Agent* ini akan menerima *request path* area dari *Query Agent* dengan parameter pesan berupa lokasi asal dan lokasi tujuan, kemudian melakukan penghitungan untuk mencari *path* area dari lokasi asal ke lokasi tujuan tersebut.

Virtual Agent ini diinstansiasi dari kelas *VirtualGraphAgent* yang diinherit dari kelas *Agent*. Di dalamnya terdapat method *setup()* yang akan dipanggil pada waktu agent melakukan *startup*. Di dalam method `setup()` tersebut terdapat *CyclicBehaviour* yang akan terus dijalankan oleh *Virtual agent* untuk menuggu pesan atau *request* yang masuk. Ketika ada pesan yang masuk melalui *method receive()*, maka pesan tersebut akan ditangkap oleh variabel *msg* yang memiliki tipe data *ACLMessage*. Ketika tidak ada pesan masuk maka *method block()* akan dijalankan di dalam *CyclicBehaviour* ini, yang menjadikan *behaviour* berada dalam kondisi *block*.

```
protected void setup() {
  try {
     addBehaviour(new CyclicBehaviour(this){
        public void action() {
        //waiting for the message from Query Agent
          try {
           System.out.println("receive message from...");
           ACLMessage msg = receive();
              if(msg!=null) {
                ..........
                // menjalankan aksi behaviour
                ..........
              }
              else {
                //block agent if there is no message
                block();
              }
          }
          catch(Exception ex) {
             ex.printStackTrace();
          }
        }
      });
  }
   catch(Exception e) {
       e.printStackTrace();
   }
}
```

Ketika *CyclicBehaviour* tersebut menangkap pesan melaui method *receive()*, maka pesan tersebut akan di cek apakah pesan

tersebut mempunyai content *findpatharea_<lokasi asal>_<lokasi tujuan>*. Jika benar, maka *agent* tersebut akan menjalankan *behaviour CheckPathArea*.

```
if(msg!=null) {
  contentMsg = msg.getContent().split("_");
  if(contentMsg[0].equalsIgnoreCase("findpatharea")) {
       //check for path area of destination
       addBehaviour(new CheckPathArea(contentMsg[1],
                    contentMsg[2],msg.getSender()));
  }
}
```

4.4.2.1 Kelas CheckPathArea

Kelas ini diturunkan dari kelas *Behaviour*. Merupakan salah satu *behaviour* yang terdapat pada *Virtual agent*. *Behaviour* ini dipanggil ketika ada pesan *findpatharea_<lokasi asal>_<lokasi tujuan>* yang berasal dari *Query Agent* untuk meminta *path* area. Konstruktor dari kelas ini mempunyai tiga parameter yaitu *String src* (node asal), *String dst* (node tujuan), dan *AID rcv* (nama agent *receiver*).

Di dalam *method setup()*, dipanggil *method Djikstra()* dari objek yang diinstansiasi dari kelas *shortestPath*, dan setelah itu dilakukan proses pengiriman hasil komputasi pencarian rute terpendek ke *Query Agent*.

Ketika pertama kali aktif, *behaviour* ini membuat objek *sp* yang merupakan *instance* dari kelas *shortestPath*. Kemudian objek ini akan memanggil method *Djikstra()* untuk mendapatkan *path* area terpendek yang akan dilewati oleh *Query Agent*.

```
LinkedList path;
.....
shortestPath sp = new
shortestPath(this.tempSrc,this.tempDst);
path = sp.Djikstra();
.....
```

Kemudian dilakukan *query* untuk mendapatkan nama *Graph Agent* yang ada pada area-area tersebut, dan menyimpan hasilnya di variabel bertipe data *LinkedList*.

```
pathAgent = new LinkedList();
  for(int a=0;a<path.toArray().length;a++) {
     rs3 = dbcon.dbRecordset("select nama_agent from
           daftar_area where nama_area like
           '"+path.get(a).toString()+"'");
     while(rs3.next()) {
        pathAgent.add(rs3.getString(1));
     }
  }
```

Setelah daftar nama *Graph Agent* pada area lokal, dan node batas antara dua area didapatkan, maka disimpan dalam sebuah variabel bertipe data *String*. Setelah itu hasilnya dikirim ke *Query Agent*.

```
try {
  ACLMessage reply = new ACLMessage(ACLMessage.INFORM);
  reply.addReceiver(this.receiver);
  reply.setContent(this.temporaryMsg);
  reply.setOntology("agent-communication-object
                    -ontology");
  myAgent.send(reply);
  reply.clearAllReceiver();
}
catch (Exception ex) {
  ex.printStackTrace();
}
```

4.4.2.2 Kelas shortestPath

Kelas *ShortestPath* ini akan diinstansiasi menjadi sebuah objek di dalam *behaviour CheckPathArea*. Konstruktor yang ada pada kelas ini memiliki parameter berupa *String nstart* yang merupakan node asal, dan *String nstop* yang merupakan node tujuan. Kelas ini memiliki *method* utama *Djisktra()* yang memberikan *return value* berupa *LinkedList* dari *path* antara node asal dengan node tujuan. *Method Djikstra()* ini yang

nantinya akan dipanggil oleh objek dari kelas *ShortestPath* yang terdapat pada *behaviour CheckPathArea*, untuk mendapatkan nilai *path* antara node asal dengan node tujuan.

Langkah pertama kali di dalam implementasi algortima Djikstra ini adalah kita memasukkan semua data graph yang ada pada database ke dalam objek variabel yang diinstansiasi dari kelas *graphComponent()* (kelas yang berisi variabel-variabel yang menyimpan data graph). Data yang disimpan ke variabel berupa nama node (*node[i].name*), node *adjacent* (*node[i].adjVertex*), dan *weight* antara suatu node dengan node *adjacent* (`node[i].weight[j]`).

```
dbConnection db = new dbConnection();
db.Connect();
rs = db.dbRecordset("SELECT DISTINCT node_asal FROM
weight_component");
rs.last();
size=rs.getRow(); //get the number of vertex
graphComponenent[] node =new graphComponenent[size];
rs.beforeFirst();
i=0;
while(rs.next()) {
   node[i]=new graphComponenent(size);
   node[i].name = rs.getString(1);
   query="SELECT node_asal,node_tujuan,waktu_awal,
          waktu_akhir,weight FROM weight_component,
          weight_value WHERE weight_component.
          wcomponent_id =weight_value.wcomponent_id "+
           "AND weight_component.node_asal like
          '"+rs.getString(1)+"' AND waktu_awal <=
           CURTIME() AND "+ "waktu_akhir > CURTIME()";
   rs2=db.dbRecordset(query);
   j=0;
   while (rs2.next()) {
     node[i].adjVertex.add(rs2.getString(2));
     node[i].weight[j]=Double.parseDouble(
                        rs2.getString(5));
     j=j+1;
  } i=i+1;
}
```

Selanjutnya akan dilakukan proses inisialisasi dari variabel node asal dan node tujuan.

```
//looping for initialization node label
for(i=0;i<node.length;i++) {
//Initialization for source node
   if(node[i].name.equals(nodeStart)) {
        node[i].vStart=nodeStart;
        node[i].vPath.add(nodeStart);
        node[i].totWeight=0.0;
   }
   else if (node[i].name.equals(nodeStop)) {
        node[i].vEnd=nodeStop;
        node[i].vPath.clear();
        node[i].totWeight=null;
   }
   //Initialization for destination node
   else {
        node[i].vPath.clear();
        node[i].totWeight=null;
   }
}
```

Setelah itu dilakukan pengecekan apakah node yang dilewati dalam iterasi sama dengan node tujuan (node akhir). Jika sama maka proses iterasi berhenti. Jika tidak sama maka akan dijalankan langkah-langkah dibawah ini.

- Melakukan pencarian node yang mempunyai nilai *totWeight* (*total weight*) paling kecil dan tidak berada pada node yang dilewati. Nilai dari node yang paling kecil tersebut dimasukkan ke variabel *passNode* (variabel yang menyimpan node yang sudah dilewati)

```
minTotalWeight=1000.0;
for (k=0;k<node.length;k++) {
   flag1=false;
   for(i=0;i<passNode.toArray().length;i++) {
      //check if node that has weight label is
      //not exist in passNode
      if (node[k].name.equalsIgnoreCase(passNode.get(i).
      toString())) {
         flag1=true;
      }
   }
   if(flag1==false) {
      if (minTotalWeight!=0.0 && node[k].totWeight
      !=null) {
        if (minTotalWeight>node[k].totWeight ) {
          minTotalWeight=node[k].totWeight;
          idx=k;
        }
      }
   }
}
passNode.add(node[idx].name);
```

- Memasukkan nilai *totWeight* pada setiap node yang *adjacent* dengan node terakhir yang berada pada variabel *passNode*.

```
//check if totWeight of adjacent node is null
if (node[k].totWeight==null) {
   //save totWeight
   node[k].totWeight=node[idx].totWeight+
                  node[idx].weight[j];
}
node[k].vPath.clear();
for(c=0;c<node[idx].vPath.toArray().length;c++) {
    node[k].vPath.add(node[idx].vPath.get(c).
                      toString());
}
node[k].vPath.add(node[idx].name);
```

- Melakukan *update* nilai *totWeight*, pada tiap node yang *adjacent* dengan node terakhir yang berada pada *passNode*, apabila nilai *totWeight* pada node tersebut lebih besar dari nilai *totWeight* node sebelumnya, ditambah dengan nilai *weight* antara node tersebut dengan node sebelumnya.

```
for (k=0;k<node.length;k++) {
  for(j=0;j<node[idx].adjVertex.toArray().length;j++) {
     if(node[k].name.equalsIgnoreCase(node[idx].
     adjVertex.get(j).toString())) {
        if(node[idx].totWeight+node[idx].
         weight[j]<node[k].totWeight) {
            node[k].totWeight = node[idx].
                    totWeight+node[idx].weight[j];

        }
        pathWeight=node[k].totWeight;
     }
  }
}
```

4.4.2.3 Kelas dbConnection

Kelas *dbConnection* merupakan kelas *public* yang akan diinstansiasi menjadi objek koneksi ke database. Kelas ini memiliki lima *method* utama yang digunakan untuk melakukan koneksi ke database, dan operasi *sql* lainnya.

Method Connect() digunakan untuk melakukan koneksi ke database *global area graph*. Didalam method ini didefiniskan pemanggilan *driver* koneksi ke database, nama *server*, nama database, *url* koneksi, nama *user*, dan *password*.

```
public boolean Connect() {
  try {
    // mysql driver
    String name = "org.gjt.mm.mysql.Driver";
    Kelas.forName(name);

    //create connection to database.
    String server = "10.126.11.151"; //server database
    String dbase = "virtual_graph"; //database name
    String url = "jdbc:mysql://"+server+"/"+dbase;
    String user = "virtualagent"; //user
    String pass = "riset"; //password
    //create connection
    con = DriverManager.getConnection(url,user,pass);
    if (con!= null) {
      //System.out.println("koneksi berhasil");
    }
    return true;
  }
  catch(Exception ex) {
    System.out.println(ex);
    return false;
  }
}
```

Method Disconect() digunakan untuk melakukan pemutusan koneksi ke database. Di dalam *method* ini dilakukan pemanggilan *method close()* pada objek koneksi.

```
//disconnect database
public boolean Disconnect() {
   try {
      if (!con.isClosed()) {
         con.close();
         return true;
      }
      else {
         return false;
      }
   }
   catch(Exception e) {
      System.out.println(e);
      System.out.println("tidak ada koneksi");
      return false;
   }
}
```

Method queryExec() digunakan untuk melakukan eksekusi perintah *sql* selain *query* ke objek database. *Method* ini tidak mengembalkan nilai (mempunya *return value void()*).

```
//method for execute sql
public void queryExec(String query) {
    try {
        stat = con.createStatement();
        stat.execute(query);
        stat.close();
    }
    catch(SQLException sqlx) {
        System.out.println(sqlx);
    }
}
```

Method dbRecordset() digunakan untuk melakukan *query* ke objek database (tabel atau view). *Method* ini mempunyai *return value* berupa *ResultSet* hasil dari *query* ke database tersebut.

```
public ResultSet dbRecordset(String query){
  ResultSet record=null;
  rs=null;
  try {
     stat = con.createStatement(ResultSet.
      TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
     rs = stat.executeQuery(query);
     record=rs;
  }
  catch(SQLException sqlx){
     System.out.println(sqlx);
  }
  return record;
}
```

Method rsClose() digunakan untuk melakukan penutupan *recordset*. *Method* ini mengambalikan nilai *boolean true* apabila *resulset* berhasil ditutup, dan sebaliknya akan mengembalikan nilai *boolean false*.

```
//method to close recordset
public boolean rsClose(){
  try {
     rs.close();
     return true;
  }
  catch(Exception e){
     System.out.println(e);
   return false;
  }
}
```

4.5 Protokol yang Digunakan dalam Sistem

Untuk komunikasi antara *user* dengan *agent*, atau antara *agent* dengan *agent* dibuat suatu protokol yang didefinisikan tersendiri. Suatu *agent* dalam melakukan komunikasi dengan *agent* yang lain memerlukan suatu protokol. Berikut ini adalah beberapa protokol yang digunakan untuk berkomunikasi antar *agent*.

- *checksrc_<sourcenode>*
  Yaitu pesan yang dikirimkan oleh *Query Agent* ke *Graph Agent* untuk melakukan pengecekan apakah node asal berada pada database *Graph Agent* yang dimaksudkan. Parameter *<sourcenode>* merupakan node asal yang dicari.

- *getpath_<sourcenode>_<destinationnode>*
  Yaitu pesan yang dikirimkan oleh *Query Agent* ke *Graph Agent* untuk mendapatkan rute terpendek pada *local area* tertentu. Parameter *<sourcenode>* merupakan node asal dan parameter <destinationnode> merupakan node tujuan yang dicari *path*-nya.
  Selain itu pesan ini juga dikirmkan dari *Client* Agent ke *Query Agent* yang akan meminta *request path area* ke *Graph Agent*.

- *getpatharea_<sourcenode>_<destinationnode>*
  Yaitu pesan yang dikirimkan oleh *Query Agent* ke *Graph Agent* untuk mendapatkan rute terpendek pada *local area* tertentu. Parameter *<sourcenode>* merupakan node asal dan parameter *<destinationnode>* merupakan node tujuan yang dicari *path*-nya. Bedanya dengan *getpath_* adalah, *getpatharea_* dikirim ke *Graph Agent* jika *Graph Agent* yang menjadi tujuan lebih dari satu (lokasi asal dan lokasi tujuan berada pada lokal area yang berbeda).

- *findpatharea_<sourcenode>_<destinationnode>*
  Yaitu pesan yang dikirim oleh *Query Agent* ke *Virtual agent* untuk mendapatkan *path area* (daftar area yang akan dilewati) dari lokasi asal ke lokasi tujuan. Parameter *<sourcenode>* merupakan node asal dan parameter *<destinationnode>* merupakan node tujuan yang dicari *path*-nya.

- *destinaton_<destinationnode>_false*

Yaitu pesan yang dikirimkan oleh *Graph Agent* ke *Query Agent*, yang memberitahukan bahwa lokasi tujuan yang ditanyakan oleh *Query Agent* tidak berada pada database *Graph Agent* tersebut. Parameter <*destinationnode*> merupakan lokasi tujuan yang ditanyakan oleh *Query Agent*.

- *recheck_true_*<*containername*>
  Pesan ini dikirimkan oleh *Graph Agent* ke *Query Agent*, yang memberitahukan bahwa lokasi asal yang dicari oleh *Query Agent* berada pada database *Graph Agent* tersebut. Parameter <*containername*> merupakan nama *container Graph Agent* yang mengirimkan pesan. Paramater ini akan digunakan oleh *Query Agent* untuk melakukan perpindahan dari *container* asal ke *container Graph Agent* tersebut.

- *recheck_false_*<*agentname*>
  Pesan ini dikirimkan oleh *Graph Agent* ke *Query Agent*, yang memberitahukan bahwa lokasi asal yang dicari oleh *Query Agent* tidak berada pada database *Graph Agent* tersebut. Parameter <*containername*> merupakan nama *container Graph Agent* yang mengirimkan pesan. Paramater ini akan digunakan oleh *Query Agent* untuk melakukan perpindahan dari *container* asal ke *container Graph Agent* tersebut.

- *path_*<*listofpath*>
  Pesan ini merupakan pesan balik yang dikirim oleh *Graph Agent* ke *Query Agent*, setelah mendapatkan rute terpendek yang di-*request* oleh *Query Agent* pada *local area* tertentu. Parameter <*listofpath*> adalah daftar *path* dari lokasi asal ke lokasi tujuan.

- *patharea_*<*listofpatheacharea*>
  Sama dengan *path_* pesan ini dikirimkan oleh *Graph Agent* ke *Query Agent*, setelah mendapatkan rute terpendek yang

di-*request* oleh *Query Agent* pada *local area* tertentu. Yang membedakan adalah *path_* me-*reply* pesan *getpath_*, sedangkan *patharea_* me-*reply* pesan *getpatharea_*.

- *listarea_<listofGA>_<listofadjacentGA>/<nodelimit>* Pesan ini dikirim oleh *Virtual agent* untuk me-*reply* pesan *findpatharea_* dari *Query Agent*. Parameter pesan *<listofGA>* adalah daftar dari *Graph Agent* yang akan dilewati areanya oleh *Query Agent*, dari lokasi asal ke lokasi tujuan. Parameter pesan *<listofadjacentGA>* adalah daftar dari *Graph Agent* yang berada pada dua lokasi yang bersebelahan. Dan parameter *<nodelimit>* adalah daftar node batas antara dua area yang bersebelahan.

BAB V

UJI COBA DAN EVALUASI

5.1 Lingkungan Uji Coba

Uji coba di lakukan dengan spesifikasi komputer sebgai berikut:

- Komputer *server VA*, dan server database agent
  Pentium 4 CPU 2.40 GHz, memori 512 MB, Sistem Operasi Windows XP SP2, server database MySQL 5.0, JDK 1.5.0
- Komputer *server GA*
  Intel Pentium II MMX, 400 MHz, memori 256 MB, Sistem Operasi Windows Server 2000, JDK 1.5.0
- Komputer client
  Intel Pentium II MMX, 400 MHz, memori 256 MB, Sistem Operasi Windows 2000 Professional, JDK 1.5.0

5.2 skenario Uji Coba Fungsionalitas

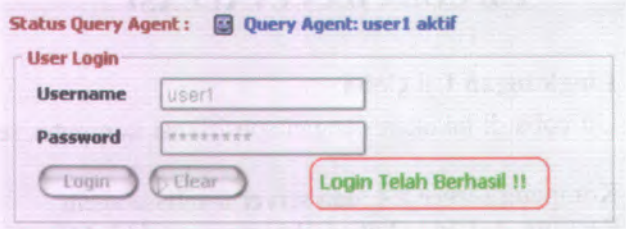
Pada sub bab ini akan dijelaskan mengenai skenario-skenario uji coba untuk mengetahui fungsionalitas dari program yang dibuat. Uji coba ini dilakukan mulai dari proses masuknya input *user* dalam sistem, proses pengolahan data, hingga proses menampilkan output ke *user*.

5.2.1 Skenario uji coba client

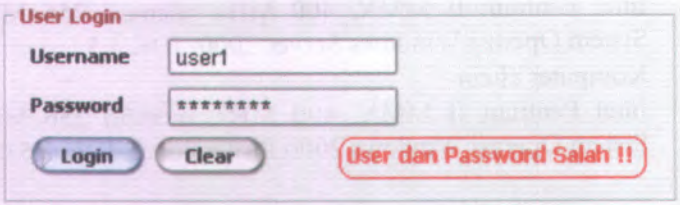
5.2.1.1 Skenario login dan generate Query Agent

Login pada sistem dilakukan dengan cara mengisikan nama *user* pada *form username* dan mengisikan *password* pada *form password*. Apabila login berhasil maka akan muncul pesan

bahwa login telah berhasil. Sedangkan apabila login gagal maka pada aplikasi ditampilkan pesan bahwa login tidak berhasil

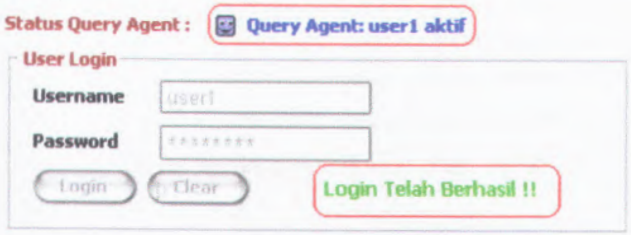


Gambar 5.1 Skenario login berhasil

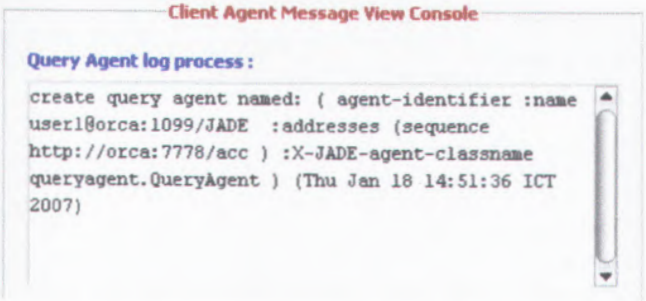


Gambar 5.2 Skenario login gagal

Jika proses login telah berhasil, maka aplikasi *client* ini secara otomatis akan meng-*generate Query Agent*. *Query Agent* ini memiliki nama yang sama dengan nama *username* pada waktu login. Pada *Query Agent Log Process* bisa dilihat pesan bahwa *Query Agent* telah berhasil di-*generate*.



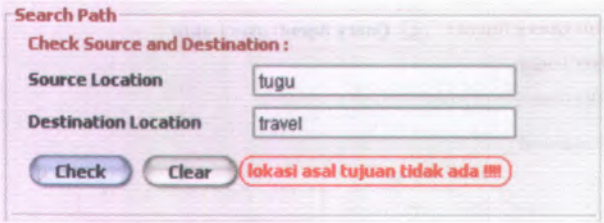
Gambar 5.3 Status Query Agent *setelah di*-generate



Gambar 5.4 Status Query Agent *pada* View Console *setelah di*-generate

5.2.1.2 Skenario pengecekan lokasi yang dicari

Proses pengecekan lokasi yang dicari dilakukan dengan cara memasukan lokasi asal pada form *source location,* dan lokasi tujuan pada form *destination location.* Apabila proses pencarian lokasi asal dan lokasi tujuan pada database tidak ditemukan, maka akan menampilan pesan bahwa proses pengecekan tidak berhasil. Apabila proses pengecekan berhasil, maka *combo box* pada *form search path* akan berisi data berupa daftar lokasi asal dan lokasi tujuan dari path yang dicari.

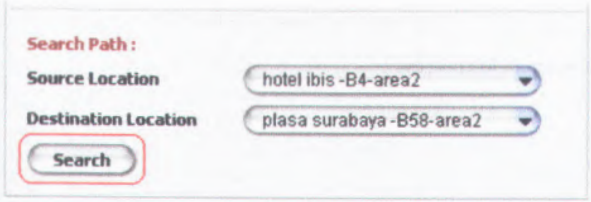


Gambar 5.5 Skenario pengecakan lokasi gagal

Gambar 5.6 Skenario pengecakan lokasi gagal

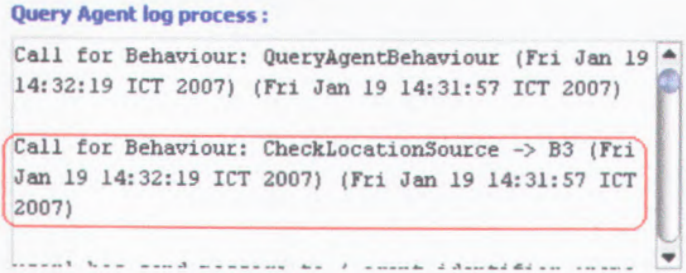
5.2.1.3 Skenario request path

Setelah proses pengecekan lokasi asal dan lokasi tujuan berhasil dilakukan, maka dilanjutkan dengan proses *request* path. Proses *request* path dimulai ketika *user* memilih daftar lokasi asal dan lokasi tujuan pada *combo box* yang terdapat pada *form search path*. Setelah menekan tombol *search*, maka *request user* dengan parameter berupa lokasi asal dan lokasi tujuan akan dikirim ke *Query Agent*.



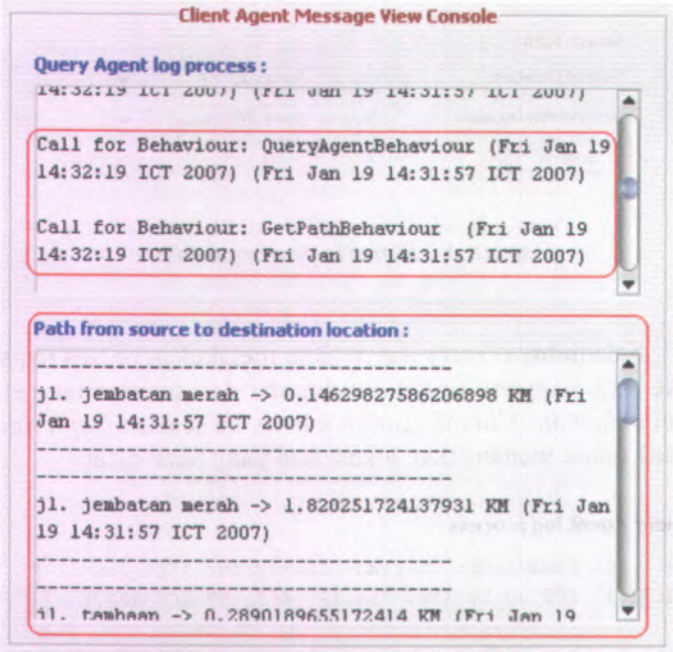
Gambar 5.7 Skenario pencarian path

Selanjutnya *Query Agent* akan melakukan proses request path ke *Graph Agent*. Proses ini dimulai dari pengiriman pesan melalui *behaviour CheckLocationSource*, ke semua *Graph Agent* yang ada untuk menanyakan lokasi asal yang akan dicari.



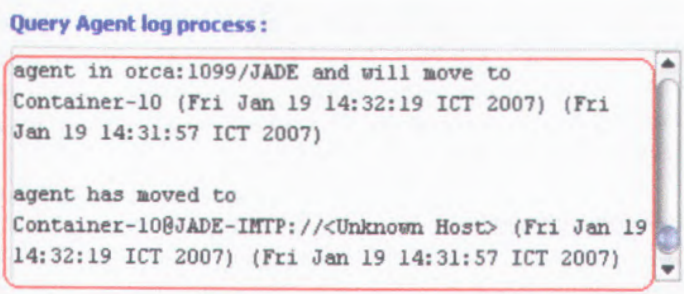
Gambar 5.8 Pemanggilan behaviour CheckLocationSource

Setelah itu akan dilakukan proses *request path* antara lokasi asal dengan lokasi tujuan, melalui pemanggilan *behaviour GetPathBehaviour*. Setelah *path* di dapatkan, maka hasilnya yang berupa nama jalan dan panjang jalan yang dilewati, akan ditampilkan dalam *interface client*.



Gambar 5.9 Path hasil pencarian

Setelah semua proses mendapatkan path selesai dilakukan, maka *Query Agent* melakukan proses perpindahan dari *container* tempat *Graph Agent* berada, ke *container* asal (*container Client* Agent).

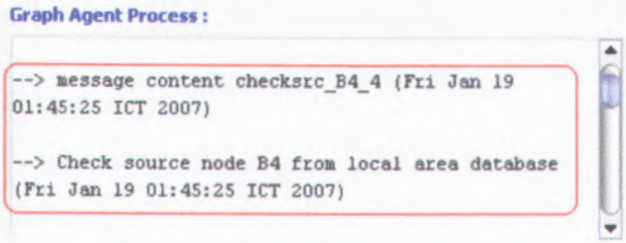


Gambar 5.10 Perpindahan Query Agent *ke lokasi asal*

5.2.2 Skenario uji coba server

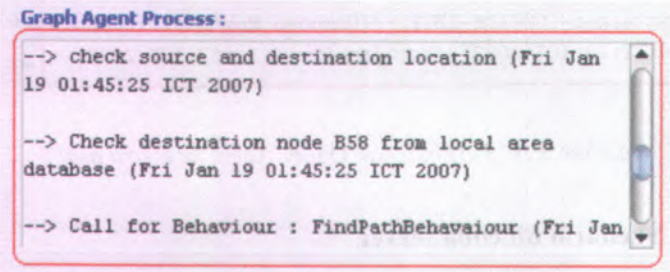
5.2.2.1 Skenario uji coba server Graph Agent

Skenario ini dimulai dari proses penerimaan pesan *checkscr_* dari *Query Agent*. Proses penerimaan pesan *checksrc_* dilakukan setelah *Query Agent* mengirimkan pesan tersebut ke *Graph Agent*, untuk melakukan cek lokasi asal pada database. Setelah pesan tersebut diterima maka *Graph Agent* melakukan cek lokasi asal tersebut pada database.



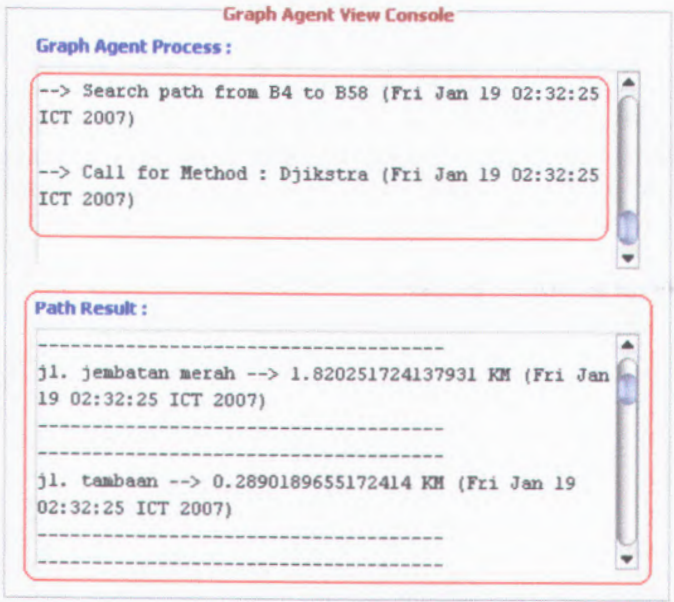
Gambar 5.11 Penerimaan pesan cek lokasi dari Query Agent

Setelah itu *Graph Agent* akan melakukan proses pengecekan lokasi tujuan. Jika lokasi tujuan berada pada database, maka dilakukan proses pemangilan *behaviour* FindPathBehaviour untuk mencari rute terpendek antara lokasi asal dengan lokasi tujuan.



Gambar 5.12 Pemanggilan bahaviour FindPathBehaviour

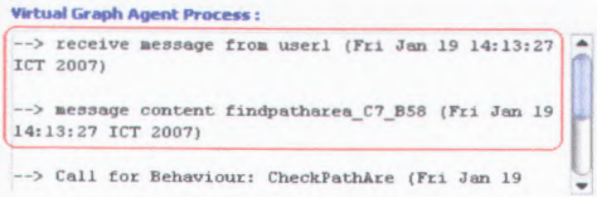
Di dalam *behaviour* FindPath*Behaviour* tersebut dipanggil method *Djikstra()*, yang akan melakukan komputasi untuk mencari rute terpendek antara lokasi asal dengan lokasi tujuan. Setelah rute terpendek yang berupa nama jalan dan pandajang jalan didapatkan, maka akan ditampilkan pada *interface user*.



Gambar 5.13 Pemanggilan method *oleh* Graph Agent

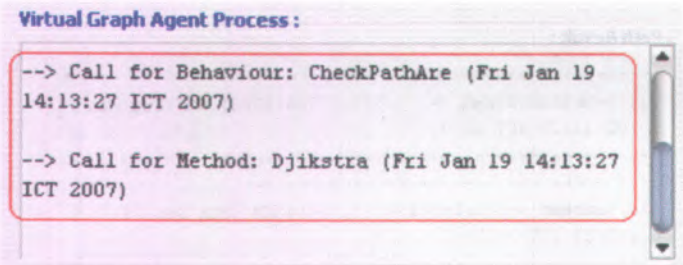
5.2.2.2 Skenario uji coba server Virtual Agent

Skenario ini dimulai dari proses penerimaan pesan *findpatharea_*dari *Query Agent*, yang meminta *Virtual Agent* untuk mendapatkan *path* area (area yang harus dilewati) dari lokasi asal ke lokasi tujuan.



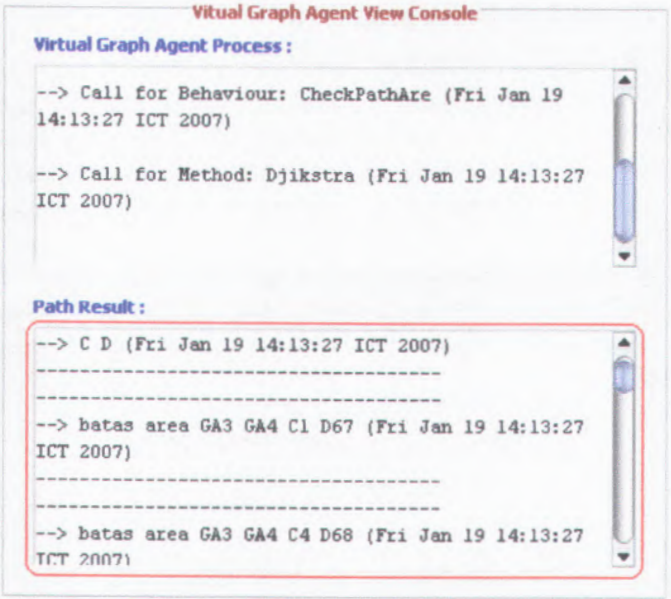
Gambar 5.14 Pemanggilan method *Dijikstra oleh* Graph Agent

Setelah itu akan dilakukan pemanggilan *behaviour* FindPathBehaviour yang akan melakukan pencarian *path area* dari lokasi asal ke lokasi tujuan. Di dalam *behaviour* ini terdapat methode *Djikstra()*, yang akan melakukan komputasi untuk mencari rute terpendek dari data *graph* area yang ada.



Gambar 5.15 Pemanggilan method *Dijikstra oleh* Graph Agent

Setelah proses pemanggilan method *Djikstra()* dilakukan, maka hasilnya ditampilkan pada *interface user*. Hasil penghitungan *method* ini berupa daftar dari area yang harus dilewati, dan node-node batas antara area yang satu dengan area yang lain.



Gambar 5.15 Path Result *hasil pencarian* path area

5.3 Evaluasi Uji Coba Performa

Di dalam uji coba performa ini dievaluasi mengenai kecepatan *request path client* terhadap *server*, kecepatan komputasi algoritma Djikstra pada *server* Graph Agent, kecepatan komputasi algortima Djikstra pada *server* Virtual Agent, serta pengaruh penambahan jumlah *client* terhadap masing-masing kecepatan tersebut.

Untuk uji coba pada sisi *client* dibuat skenario *request path* dari beberapa *client* yang memiliki lokasi asal pada *host* yang berbeda (*multi host*), ke lokasi tujuan yang sama (*single host*). Sedangkan untuk uji coba pada *server* GA maupun server VA, dibuat skenario *request path* dari *client* yang memiliki lokasi asal dan lokasi tujuan pada *host* yang sama dengan *server-server* tersebut.

5.3.1 Evaluasi performa pada client

Evaluasi ini dilakukan untuk mengetahui seberapa cepat *client* (*Query Agent*) dalam melakukan *request path* ke *server* (*Graph Agent*). *Request path* dilakukan oleh client yang berasal dari host yang berbeda-beda, ke *server* yang sama (*host* sama).

Untuk menghitung seberapa besar pengaruh penambahan jumlah *client* yang melakukan *request* pada *server*, maka uji coba ini dilakukan sebanyak tiga kali, yaitu *request* dari tiga *client*, *request* dari lima *client*, dan *request* dari delapan *client*. Berikut ini adalah data uji coba pada masing-masing skenario.

Tabel 5.1 Request dari 3 client

client	waktu (milisecond)
1	18457
2	26869
3	39597
waktu rata-rata	**28307.66667**

Tabel 5.2 Request dari 5 client

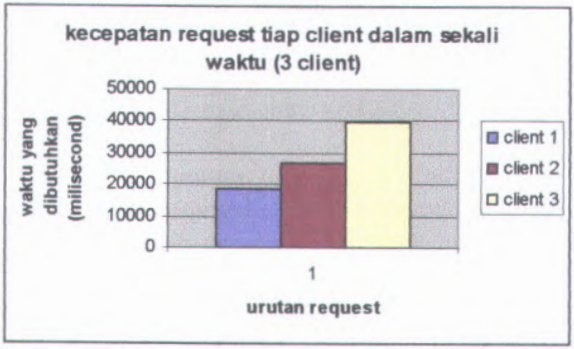
client	waktu (milisecond)
1	4156
2	17685
3	24205
4	57082
5	69580
waktu rata-rata	**34541.6**

Tabel 5.3 Request dari 8 client

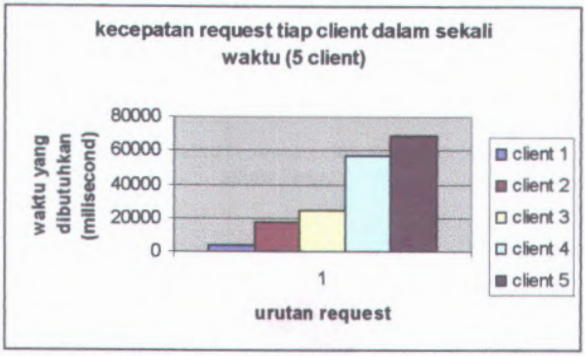
client	waktu (milisecond)
1	18590
2	24636
3	33598
4	49451
5	95707
6	107645

client	waktu (milisecond)
7	119322
8	129046
waktu rata-rata	**57799.5**

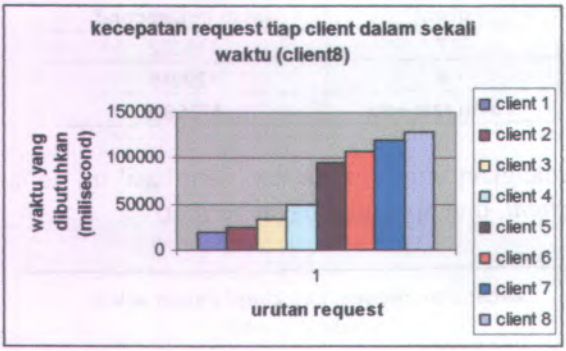
Sedangkan untuk grafik dari data hasil uji coba dengan beberapa *client*, ditampilkan sebagai berikut.



Gambar 5.1 Grafik kecepatan request oleh 3 client



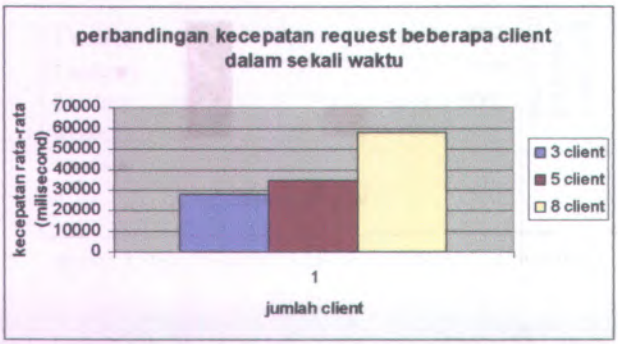
Gambar 5.2 Grafik kecepatan request oleh 5 client



Gambar 5.3 Grafik kecepatan request oleh 8 client

Grafik tersebut menjelaskan tentang pengaruh kecepatan *request path* oleh beberapa *client*, terhadap urutan proses *request* pada masing-masing *client*. Dari masing-masing grafik tersebut dapat dianalisis bahwa, ketika ada *request* dari beberapa *client* secara hampir bersamaan terhadap suatu *server*, maka *client* yang lebih awal melakukan *request* memiliki waktu *request* yang lebih sedikit dari pada *client* yang terakhir melakukan *request* dalam sekali waktu.

Grafik berikut ini akan menjelaskan tentang pengaruh penambahan *client* terhadap kecepatan *request path* dari *client* ke *server*.



Gambar 5.4 perbandingan kecepatan request oleh beberapa client

Dari data dan grafik yang diperoleh dari hasil uji coba, dapat dihitung dan dianalisis pengaruh penambahan jumlah *client* dalam sekali waktu *request* ke *server*. Jika *Δjumlah_client* merupakan penambahan jumlah *client*, dan *Δwaktu_request* merupakan penambahan waktu rata-rata *request client* ke *server* (dalam *milisecond*), maka dapat dihitung sebagai berikut

- *Δjumlah_client1* = 5 client – 3 client = **2** client
  Δwaktu_request1 = 34541.6 - 28307.7 = **6233.9** milisecond
- *Δjumlah_client2* = 8 client – 5 client = **3** client
  Δwaktu_request2 = 34541.6 - 28307.7 = **23257.9** milisecond

Dari penghitungan tersebut dapat dianalisis bahwa peningkatan jumlah client berbanding lurus secara linier dengan peningkatan waktu *request* yang dilakukan oleh *client* tersebut.

5.3.2 Evaluasi performa pada server

Evaluasi performa pada yang dilakukan pada *server*, terdiri dari dua macam, yaitu evaluasi performa pada *server GA* dan evaluasi performa pada *server VA*.

5.3.2.1 Evaluasi performa pada server GA

Pada *server GA*, evaluasi dilakukan untuk mencari kecepatan penghitungan algoritma Djikstra, untuk setiap *request path* yang berasal dari *client*. Dalam evaluasi ini dibuat skenario *request path* yang dilakukan oleh beberapa *client* ke salah satu *server GA*, yaitu *request* oleh tiga *client*, delapan *client*, dan tiga belas *client*. Data hasil uji coba dapat dilihat pada tabel berikut ini.

Tabel 5.4 Request pada server GA oleh 3 client

client	waktu (milisecond)
1	1042
2	1503
3	961
waktu rata-rata	**1168.666667**

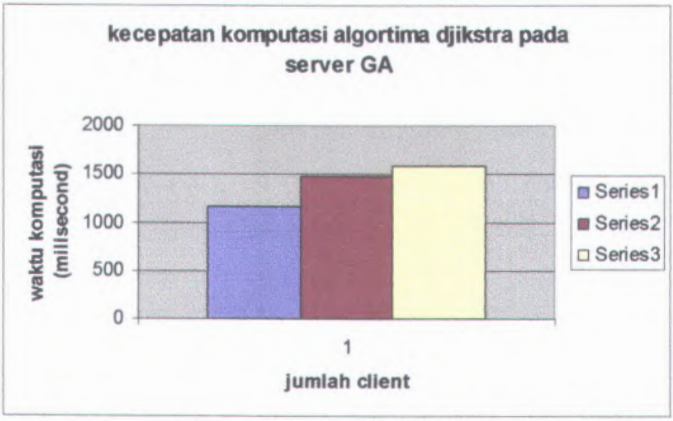
Tabel 5.5 Request pada server GA oleh 8 client

client	waktu (milisecond)
1	1072
2	1812
3	1122
4	1152
5	1712
6	1613
7	1852
8	1462
waktu rata-rata	**1474.625**

Tabel 5.6 Request pada server GA oleh 13 client

client	waktu (milisecond)
1	1181
2	1382
3	1352
4	1613
5	1392
6	1382
7	1392
8	1782
9	2003
10	1152
11	1643
12	2253
13	2093
waktu rata-rata	**1586.153846**

Grafik berikut ini menjelaskan tentang pengaruh kecepatan komputasi algortima Djikstra terhadap penambahan jumlah *request client* ke *server GA.*



Gambar 5.4 kecepatan komputasi algortima Djikstra pada server GA

Dari data dan grafik yang diperoleh dari hasil uji coba, dapat dihitung dan dianalisis pengaruh penambahan jumlah *client* dalam sekali waktu *request* ke *server,* terhadap kinerja *server GA*. Jika *Δjumlah_client* merupakan penambahan jumlah *client,* dan *Δwaktu_komputasi* merupakan penambahan waktu rata-rata komputasi algortima djikstra pada *server GA* (dalam *milisecond*), maka dapat dihitung sebagai berikut

- *Δjumlah_client1* = 8 client – 3 client = **5** client
  Δwaktu_komputasi1 = 1474.6 - 1168.7= **305.96** milisecond
- *Δjumlah_client2* = 13 client – 8 client = **5** client
  Δwaktu_komputasi2 = 1586.1 - 1474.6 = **111.52** milisecond

Dari penghitungan tersebut dapat dianalisis bahwa peningkatan jumlah client berbanding lurus secara linier dengan peningkatan waktu komputasi algortima djikstra oleh *server GA*.

5.3.2.2 Evaluasi performa pada server VA

Seperti halnya *server GA*, pada *server VA* evaluasi dilakukan untuk mencari kecepatan penghitungan algoritma Djikstra, untuk setiap *request path* yang berasal dari *client*.

Dalam evaluasi ini dibuat skenario *request path area* yang dilakukan oleh beberapa *client* ke satu *server VA*, yaitu *request* oleh tiga *client*, delapan *client*, dan tiga belas *client*. Hasil uji coba dapat dilihat pada tabel berikut ini.

Tabel 5.7 Request pada server VA oleh 3 client

client	waktu (milisecond)
1	94
2	47
3	31
waktu rata-rata	**57.33333333**

Tabel 5.8 Request pada server VA oleh 8 client

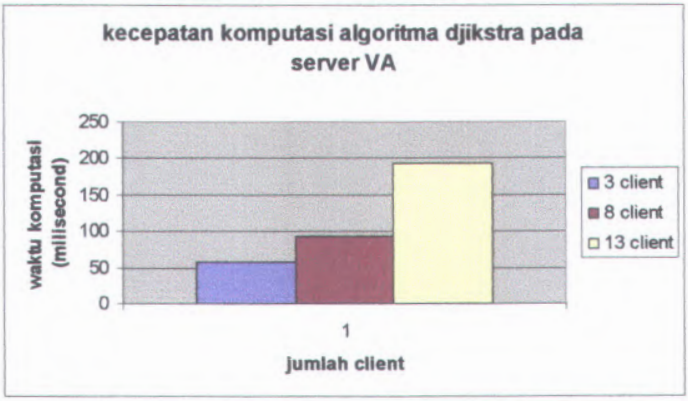
client	waktu (milisecond)
1	79
2	47
3	391
4	31
5	47
6	62
7	47
8	31
waktu rata-rata	**91.875**

Tabel 5.9 Request pada server VA oleh 13 client

client	waktu (milisecond)
1	281
2	765
3	78
4	47
5	63
6	594
7	63
8	62

client	waktu (milisecond)
9	110
10	250
11	109
12	31
13	47
waktu rata-rata	**192.3076923**

Grafik dibawah ini menjelaskan tentang pengaruh komputasi algortima Djikstra terhadap penambahan jumlah *client* yang melakukan *request* ke *server VA.*



Gambar 5.5 perbandingan kecepatan komputasi algortima Djikstra pada server VA

Dari data dan grafik yang diperoleh dari hasil uji coba, dapat dihitung dan dianalisis pengaruh penambahan jumlah *client* dalam sekali waktu *request* ke *server,* terhadap kinerja *server VA.* Jika *Δjumlah_client* merupakan penambahan jumlah *client,* dan *Δwaktu_komputasi* merupakan penambahan waktu rata-rata komputasi algortima djikstra pada *server VA* (dalam *milisecond*), maka dapat dihitung sebagai berikut

- *Δjumlah_client1* = 8 client – 3 client = **5** client

Δwaktu_komputasi1 = 91.9 - 57.3 = **34.5** milisecond

- *Δjumlah_client2* = 13 client – 8 client = **5** client

  Δwaktu_komputasi2 = 192.3 - 91.9 = **100.4** milisecond

Dari penghitungan tersebut dapat dianalisis bahwa peningkatan jumlah client berbanding lurus secara linier dengan peningkatan waktu komputasi algortima djikstra oleh *server VA*.

BAB VI

PENUTUP

Pada bab ini akan diberikan kesimpulan dari hasil uji coba yang dilakukan beserta saran-saran yang dijadikan acuan untuk pengembangan selanjutnya.

6.1 Kesimpulan

Di dalam sistem ini digunakan konsep *multiagent* yang masing-masing mempunyai karakteristik dan *behaviour* yang berbeda antara satu dengan yang lain. Masing-masing behaviour yang ada di dalam agent tersebut menjalankan satu buah *thread*.

- Dari hasil uji coba dapat dilihat pengaruh dari banyaknya proses yang ada di dalam sistem. Peningkatan waktu proses yang ada di dalam sistem berbanding lurus secara linier dengan semakin banyak jumlah *client* yang melakukan *request*.
- Proses migrasi *agent* yang dilakukan oleh *Query Agent*, juga mempunyai pengaruh terhadap kinerja sistem. Karena yang berpindah adalah objek *agent* dan *state agent*, maka untuk proses perpindahan ini membutuhkan waktu yang lebih lama daripada proses perpindahan data secara langsung.
- Setelah beberapa kali dilakukan uji coba, sistem ini dapat menangani *request* dari user untuk mendapatkan data path, dari beberapa server yang terdistribusi.

6.2 Saran

1. Perlu adanya penelitian lebih lanjut mengenai keamanan pertukaran data antar *agent*, sehingga dapat diterapkan secara nyata dalam suatu jaringan.
2. Perlu adanya perbaikan-perbaikan algoritma di dalam mengimplementasikan *behaviour* yang ada dalam *agent*.

[Halaman Ini Sengaja Dikosongkan]

DAFTAR PUSTAKA

1 [TAN02] Tanenbaum, A.S, V.S., Marteen. 2002. **Distributed Systems Principle and Paradigms, International edition**. Prentice Hall.

2 [COU05] Couloris, G., Dollimore, J., Kindberg, Tim. 2005. **Distributed Systems Concept and Design, fourth edition**. Addison Wesley.

3 [DOU01] B. West, Douglas. 2001. **Introduction to graph theory 2nd editon**. University of Illonois Urbana, Prentice Hall.

4 [ROS03] Rossen, K.H. 2003. **Discrete Mathematics and Its Applications, fifth edition**. Mc Graw Hill

5 [RUS03] Russel S.J., Norvig Peter. 2003. **Artificial Intelligence A Modern Approach 2nd Edition**. Prentice Hall.

6 [TLB06] Tilab, Nov. 2006. **Java Agent Development *Framework***, <URL: http://jade.tilab.com/ >

7 [TLB06] Tilab, Nov. 2006. **Java Agent Development Framewrok, Community Add-Ons**. <URL: http://jade.tilab.com/community-addons.php >

8 [WIK06] Wikipedia, Apr. 2006. **Middleware Concept**. <URL: http://en.wikipedia.org/wiki/Middleware >

9 [WK206] Wikipedia, Apr. 2006. **Multiagent System**. <URL: http://en.wikipedia.org/wiki/Multiagent>

10 [WOO02] Wooldridge, Michael, 2002. **An Introduction to Multiagent System,** Departement of Computer Science, University of Liverpool, UK, John Wiley and Sons.

11 [BOO01] Booger, Marko, 2001. **Java in Distributed System**. University of Hamburg, John Wiley and Sons.

12 [UMS06] UMASS, DES 2006, **Multiagent Ssystem Lab**.<URL: http://dis.cs.umass.edu/research/cdps/>

LAMPIRAN

[Halaman Ini Sengaja Dikosongkan]

BIODATA PENULIS

Penulis dilahirkan di Banyuwangi, 19 November 2007. Merupakan anak kedua dari 3 bersaudara. Penulis telah menempuh pendidikan formal di TK ABA 06 Sempu Banyuwangi, MI Muhammadiyah Sempu Banyuwangi, SLTPN 1 Genteng Banyuwangi dan SMUN 1 Genteng Banyuwangi. Setelah lulus SMU pada tahun 2002, Penulis mengikuti SPMB dan diterima di Jurusan Teknik Informatika FTIf-ITS pada tahun 2002 dan terdaftar dengan NRP 5102100085.

Di Jurusan Teknik Informatika ini penulis mengambil Bidang Studi NCC (*Net Centric Computing*) di dalam Tugas Akhirnya. Selama menjadi mahasiswa, penulis sempat aktif menjadi Administrator di Laboratorium Pemrograman Jurusan Teknik Informatika. Selain itu penulis juga pernah aktif sebagai Asisten Praktikum Basis Data, dan sebagai Asisten Dosen Program D1 PIKTI pada Jurusan Teknik Informatika ITS.