PEMBUATAN *MIDDLEWARE* SEBAGAI SARANA PEMBANTU KOMUNIKASI ANTAR PENGGUNA DALAM APLIKASI *SOCKET* TCP MENGGUNAKAN XMPP

Seminar Pra Skripsi

Disusun untuk memenuhi salah satu syarat
tugas mata kuliah Seminar Pra Skripsi

Resa Fajar Sukma

1313618029

Program Studi Ilmu Komputer

Fakultas Matematika dan Ilmu Pengetahuan Alam

Universitas Negeri Jakarta

2024

LEMBAR PENGESAHAN

Dengan ini saya mahasiswa Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta

Nama	:	RESA FAJAR SUKMA
No. Registrasi	:	1313618029
Jurusan	:	Ilmu Komputer
Judul	:	Pembuatan *Middleware* Sebagai Sarana Pembantu Komunikasi Antar Pengguna Dalam Aplikasi Socket TCP Menggunakan XMPP

Menyatakan bahwa proposal skripsi ini telah siap diajukan untuk sidang proposal.

Menyetujui,

Dosen Pembimbing I	Dosen Pembimbing II
Muhammad Eka Suryana, M.Kom.	**Med Irzal, M.Kom.**
NIP. 19851223 201212 1 002	NIP. 19770615 200312 1 001

Mengetahui

Koordinator Program Studi Ilmu Komputer

Dr. Ria Arafiyah, M. Si.

NIP. 19851121 200501 2 004

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I PENDAHULUAN

1.1 Latar Belakang

Komunikasi adalah proses sosial yang melibatkan dua orang atau lebih, di mana terjadi pertukaran informasi, gagasan, dan perasaan, yang bertujuan untuk mencapai saling pengertian.

Tujuan dari komunikasi adalah untuk mendapatkan suatu informasi yang dibutuhkan atau yang sedang terjadi pada lingkungan. Informasi yang didapat dari hasil komunikasi dapat membantu seseorang atau sekelompok orang untuk mengambil suatu tindakan yang perlu dilakukan.

Komunikasi memerlukan pengiriman ide, pikiran atau perasaan dari pengirim ke penerima melalui cara verbal ataupun nonverbal. Pengiriman ini mempunyai arti khusus dalam bisnis. Komunikasi memainkan peran yang penting dalam hal perencanaan atau strategi berkelanjutan apapun (Genç, 2017).

Dalam era teknologi seperti saat ini banyak penyedia layanan komunikasi yang memberikan akses kepada setiap orang untuk berkomunikasi secara *real-time*. Penyedia layanan komunikasi ada yang memberikan layanannya secara gratis dengan persyaratan tertentu dan juga ada yang berbayar. Penyedia layanan komunikasi gratis dengan adanya persyaratan contohnya adalah Facebook, Whatsapp, Instagram, dan lain-lain. Penyedia layanan komunikasi yang berbayar seperti Amazon Connect, dan API Komunikasi 8x8.

Adapun harga dari layanan pengiriman pesan pada Amazon Connect sebesar 0.004 USD/pesan atau setara dengan Rp64/pesan dengan catatan nilai tukar rupiah terhadap dollar adalah Rp16.0000. Harga bisa dilihat pada link berikut https://aws.amazon.com/id/connect/pricing/, sedangkan untuk API Komunikasi 8x8 harga yang ditawarkan adalah Rp65/pesan yang terkirim. Harga tertera dilink https://www.8x8.com/id/cpaas/products/communication-apis/pricing. Hal ini memiliki arti semakin banyak pesan yang dikirimkan akan semakin besar juga uang yang harus dikeluarkan untuk membayar biaya layanan.

Tentu ada perbedaan yang mencolok dari dua tipe layanan komunikasi tersebut selain dari gratis dan berbayar. Pada layanan komunikasi yang gratis akan terdapat sebuah hambatan yaitu fitur yang dapat digunakan terbatas kepada fitur yang disediakan oleh penyedia layanan, sedangkan pada layanan komunikasi berbayar hambatannya adalah semakin banyak menggunakan fitur maka akan semakin besar biaya yang harus dibayarkan ke penyedia layanan.

Ada sebuah permasalahan yang diangkat dalam penelitian ini yaitu menghubungkan pengguna dalam aplikasi sehingga mereka dapat melakukan komunikasi. Permasalahan ini juga menginginkan pengeluaran dana yang seminimal mungkin dan mengharapkan dapat dilakukan perluasan fitur yang ingin digunakan. Yang mana kebutuhan dari permasalahan ini tidak cocok dengan layanan komunikasi gratis atau berbayar yang sudah disebutkan sebelumnya.

Ada sebuah protokol yaitu *The Extensible Messaging and Presence Protocol* (XMPP). XMPP adalah *open technology* untuk komunikasi secara *real-time* (Saint-Andre et al., 2009). Penjelasan dari XMPP menjadi dasar terpilihnya XMPP sebagai protokol yang akan digunakan dalam pembuatan *middleware* pada penelitian ini. Selain dari penjelasan, terpilihnya XMPP juga didukung oleh sebuah tulisan yaitu "*Instant messaging using xmpp*" yang menyatakan kegunaan XMPP sebagai sarana komunikasi berbasis web dan keunggulannya untuk perusahaan kecil dari segi keamanan maupun finansial (Gupta et al., 2021).

Sebuah protokol yang bersifat *open technology* hanyalah akan menjadi sebuah format data. Arsitektur komunikasi agar penelitian ini dapat berjalan terinspirasi dari penelitian Muhammad Ridho Rizqillah dengan judul Improvisasi Crawling Pada Peta Web Menggunakan Algoritma Terdistribusi Dengan Model Koordinasi Berbasis Socket Programming, yang mana pada penelitian telah dilakukan pengumpulan data dengan *crawler* terdistribusi. Arsitektur yang digunakan masih *single server* dengan agen/*client* terdistribusi (Muhammad Ridho Rizqillah, 2024). Sedikit perubahan dilakukan pada penelitian ini, yang mana akan mengubah sifat *single server* tersebut menjadi *multiple server* dengan banyak agen/*client*.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dipaparkan, dapat dirumuskan beberapa permasalahan yaitu:

1. Bagaimana cara membuat *middleware* komunikasi menggunakan protokol XMPP dan *socket programming*?
2. Berapa biaya yang dibutuhkan untuk membuat sebuah *middleware* komunikasi secara mandiri?

1.3 Batasan Masalah

Penelitian ini akan memiliki beberapa batasan masalah diantaranya adalah:

1. Penelitian hanya akan berfokus kepada pembuatan *middleware* komunikasi.
2. Penelitian tidak akan membahas topik keamanan data.
3. Pengujian pada penelitian akan berfokus pada pengujian fungsionalitas dari *middleware* komunikasi.

1.4 Tujuan Penelitian

Penelitian memiliki tujuan yaitu membuat *middleware* komunikasi dengan protokol XMPP dan *socket programming*.

1.5 Manfaat Penelitian

Penelitian ini diharapkan dapat memberikan beberapa manfaat yang mana diantaranya adalah

1. Mengetahui cara pembuatan *middleware* komunikasi dengan protokol XMPP dan *socket programming*.
2. Mengetahui cara pembuatan *middleware* komunikasi yang dapat dilakukan perluasan fitur karena memiliki standar protokol.
3. Mengetahui biaya yang dibutuhkan untuk membuat sistem penghubung komunikasi.
4. Sistem yang telah dibuat dapat digunakan selama sistem masih aktif.

BAB II KAJIAN PUSTAKA

2.1 Jaringan Komputer

Menurut Melwin Syafrizal jaringan komputer adalah hubungan interkoneksi yang menggunakan media kabel atau tanpa kabel antara dua komputer atau lebih yang saling terhubung. Jaringan komputer terbentuk karena perangkat komputasi saling terhubung satu sama lain melalui suatu koneksi tertentu. Jaringan komputer ini berfungsi untuk jalur pertukaran data yang akan dilakukan oleh sekumpulan perangkat komputasi yang saling terhubung tersebut. Hubungan koneksi antar perangkat ini dapat terkoneksi secara fisik melalui kabel atau lainnya, maupun terhubung secara nirkabel seperti gelombang radio atau internet.

Peran jaringan komputer sangat besar hal ini dikarenakan tanpa adanya jaringan komputer maka perangkat komputasi tidak dapat saling bertukar data yang dimiliki atau yang diinginkan. Contoh pertukaran informasi seperti ketika seseorang ingin mencari sebuah bahan bacaan maka ia dapat mencarinya melalui internet menggunakan mesin pencari seperti Google Chrome ataupun seseorang ingin mencari temannya melalui media sosial seperti Instagram atau Facebook. Arsitektur yang digunakan biasanya memiliki model *client-server*, dimana setiap keinginan *client* akan dipenuhi oleh *server* selama permintaan tersebut dapat ditangani oleh *server* (Amira, 2021).

2.2 *Socket*

Socket adalah titik komunikasi dari lalu lintas komunikasi antar proses di dalam sebuah jaringan komputer. Hampir semua komunikasi antar komputer sekarang berdasarkan protokol internet, oleh karena itu hampir semua *socket* di jaringan komputer adalah *socket internet*. Sebuah *socket* memiliki alamat dan alamat tersebut terdiri dari kombinasi alamat IP dan juga nomor *port*. Setiap paket yang dikirimkan atau diterima pasti akan melalui *socket* yang terdiri dari dua kombinasi tersebut. *Socket* memiliki beberapa protokol diantaranya adalah protokol TCP dan protokol UDP (Shafiei et al., 2012).

2.2.1 *Socket* Protokol TCP

Protokol TCP (*Transmission Control Protocol*) adalah protokol yang berorientasi koneksi (*connection-oriented*) sehingga jika ingin mengirimkan data melalui *socket* dengan protokol TCP perangkat yang akan terlibat didalamnya harus saling memiliki koneksi. Sebuah hal yang wajar dikatakan bahwa sebelum mengirimkan data melalui protokol TCP ini wajib hukumnya koneksi antar perangkat dibangun terlebih dahulu. Pada protokol TCP ini pengiriman data bisa dipastikan memiliki kemungkinan risiko data menjadi rusak atau hilang adalah kecil (Shafiei et al., 2012).

2.2.2 *Socket* Protokol UDP

Protokol UDP (*User Datagram Protocol*) adalah sebuah protokol sederhana yang mana pengiriman data melalui protokol UDP ini tidak mewajibkan untuk membangun koneksi terlebih dahulu (*connection-less*) (Shafiei et al., 2012).

Protokol UDP hanya memerlukan alamat IP dan nomor *port* dari perangkat yang dituju. Protokol ini memiliki kecepatan pengiriman data yang lebih cepat dibanding pengiriman data melalui protokol TCP. Namun protokol UDP memiliki kelemahan yaitu protokol ini memiliki sifat "*fire and forget*" yang mana data akan dikirimkan tanpa memperdulikan bahwa data tersebut telah tersampaikan dengan sempurna kepada penerima atau terdapat kecacatan dalam proses pengiriman data (Ben Gorman, 2023).

2.3 IP *Address*

IP (*Internet Protocol*) *address* atau alamat IP adalah alamat yang memiliki peran sebagai perwakilan dari sebuah koneksi individual baik itu dalam lingkup luas di internet atau dalam lingkup kecil atau lokal (Roberts & Challinor, 2000). Alamat IP pada saat ini memiliki dua tipe yaitu IPv4 dan IPv6. Adapun jenis dari alamat IP adalah IP publik dan IP *private* (Faradilla A., 2023). Bagaimanakah cara komunikasi antara kedua jenis alamat IP ini.

2.3.1 IPv4

IPv4 adalah tipe alamat IP yang umum digunakan dengan panjang 32-bit dan terdiri dari empat oktet yang terpisah oleh titik dua. Setiap oktet memiliki cakupan panjang mulai dari 0 sampai 255. IPv4 memiliki kemungkinan untuk mewakili sekitar 4,3 miliar perangkat berbeda diseluruh dunia. Contoh dari IPv4 adalah 123.142.132.200 (Faradilla A., 2023).

2.3.2 IPv6

IPv6 adalah tipe alamat IP terbaru yang mana alamat IP ini muncul untuk menggantikan IPv4 dikarenakan alamat IPv4 yang awalnya diharapkan dapat mewakili seluruh perangkat yang ada didunia ini tiba-tiba menjadi sebuah cakupan yang sangat kecil. Hal ini dikarenakan meledaknya teknologi gawai. Permasalahan tersebutlah yang memunculkan sebuah inovasi baru yaitu IPv6.

Tidak seperti IPv4 yang memiliki panjang 32-bit, IPv6 memiliki panjang 128-bit. IPv6 juga dituliskan dalam rangkaian angka heksadesimal dan huruf serta dipisahkan pula oleh titik dua sama seperti IPV4. Contoh dari IPv6 2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF (Faradilla A., 2023).

2.3.3 Alamat IP Publik

Alamat IP publik adalah sebuah alamat yang mewakili perangkat dalam lingkup lingkungan yang luas yaitu internet. Ketika berkomunikasi melalui internet perangkat wajib hukumnya memakai alamat IP publik karena alamat IP publik adalah alamat unik yang hanya dimiliki oleh satu perangkat diseluruh dunia. Alamat IP publik juga dapat dijadikan sebagai sebuah alamat tampungan seperti yang digunakan oleh ISP (*Internet Service Provider*) agar satu alamat IP publik dapat dipakai oleh beberapa perangkat sekaligus.

Pemakaian satu alamat IP publik oleh beberapa perangkat sekaligus membuat penghematan ketika krisis alamat IP publik pada IPv4. Ketika satu alamat IP publik dipakai secara bersama-sama oleh beberapa perangkat, cara alamat IP publik menentukan perangkat mana yang memberikannya permintaan itulah yang harus menerima balasan atas permintaan tersebut adalah dengan cara pemberian sebuah alamat khusus yaitu alamat IP *private*. Contoh dari alamat IP publik 103.8.90.201 (Biznet News, 2023b).

2.3.4 Alamat IP *Private*

Alamat IP *private* adalah sebuah alamat yang mewakili perangkat dalam lingkup lingkungan yang kecil atau lokal yaitu sebuah alamat yang hanya dapat diakses dalam satu area yang terhubung oleh sebuah alamat IP publik dan tidak dapat langsung diakses melalui internet. Alamat IP *private* membantu IP publik dalam membedakan setiap perangkat yang terhubung ke IP publik. Adapun tipe untuk IP *private* adalah sebagai berikut:

1. 10.0.0.0 hingga 10.255.255.255.
2. 172.16.0.0 hingga 172.16.255.255.
3. 192.168.0.0 hingga 192.168.255.255 (Faiz, 2023).

2.3.5 Komunikasi Antar Alamat IP

Saat melakukan proses pertukaran data pastinya perlu komunikasi antar alamat IP sehingga data dapat dikirimkan oleh pengirim dan diterima oleh penerima. Cara berkomunikasi antar alamat IP publik dengan publik, alamat IP publik dengan *private*, alamat IP *private* dengan publik, alamat *private* dengan *private*.

1. Publik dengan publik

   Komunikasi antara IP publik dengan publik sungguh sangat dapat dilakukan. Hal ini dikarenakan alamat IP publik mewakili perangkat dalam lingkup luas sehingga sudah dapat dipastikan mana perangkat yang mengirim dan mana perangkat yang menerima.

2. Publik dengan *private*

   Komunikasi antara IP publik dengan *private* dapat dilakukan ketika IP *private* diwakili oleh sebuah IP publik dalam jaringannya yang mana ketika data dikirimkan ke IP publik perwakilannya maka data tersebut dapat diteruskan kepada IP *private*. Hal ini disebut dengan NAT (*Network Address Translator*).

3. *Private* dengan publik

   Komunikasi antara IP *private* dengan publik dapat dilakukan ketika IP *private* ingin melakukan pengiriman data, maka IP *private* wajib memberikan data kepada IP publik yang mewakilkannya didalam lingkup internet luas.

4. *Private* dengan *private*

   Komunikasi langsung antara IP *private* dengan *private* dapat dilakukan jika kedua perangkat tersebut berada dalam jaringan IP publik yang sama. Akan tetapi jika kedua perangkat tersebut berada dalam jaringan IP publik yang berbeda maka harus menggunakan beberapa trik. IP *private* pengirim harus diubah menjadi IP publik dan mengirimkannya kepada IP publik penerima yang mewakili IP *private* perangkat penerima dan seterusnya (Rekhter et al., 1996). Adapun trik tersebut mengalami banyak kendala yang memerlukan banyak konfigurasi dimana belum tentu seseorang memiliki hak penuh untuk melakukan konfigurasi tersebut. Sehingga dapat dikatakan bahwa komunikasi dari alamat IP *private* dengan *private* agak sulit cenderung mustahil jika tidak memiliki kuasa penuh untuk konfigurasi. Salah satu cara agar alamat IP *private* dapat berkomunikasi dengan *private* lainnya adalah dengan menggunakan *relay* (Oikarinen & Reed, 1993).

2.4 *Relay*

Relay adalah jalur yang dibuat khusus untuk melewati semua konfigurasi-konfigurasi yang sangat rumit yang harus dilakukan ketika alamat IP *private* ingin berkomunikasi dengan *private* lainnya. Jalur ini dibuat dengan cara menjalankan sebuah sistem pada alamat IP publik yang dapat diakses oleh semua. Kemudian alamat IP *private* dapat terhubung *relay* dengan diwakili oleh alamat IP publik yang mewakili dirinya.

Bisa diperhatikan bahwa *relay* tidaklah mencoba untuk membuat koneksi terhadap alamat IP *private* melainkan IP *private* lah yang membuat koneksi terhadap *relay*. Hal ini dapat berhasil karena *relay* adalah sebuah alamat IP publik

yang dapat dikoneksikan oleh siapa saja tanpa perlu melakukan konfigurasi rumit seperti pada router atau sejenisnya.

Saat koneksi sudah terjadi dua kali antara *relay* dengan IP *private* maka kedua IP *private* dapat dikatakan terhubung melalui jalur *relay*. Sehingga kedua IP *private* sudah dapat melakukan proses pertukaran data sesuka hati mereka selama jalur *relay* masih tersedia. Jadi dapat dikatakan bahwa *relay* berfungsi sebagai jalur jalannya data antara IP *private* (Oikarinen & Reed, 1993).

2.5 Middleware

Komponen perangkat lunak yang menjembatani aplikasi, data, dan pengguna. Membantu menyederhanakan pengembangan aplikasi dan membuat pengembang bekerja secara lebih efisien karena mereka dapat berfokus pada pengembangan fitur perangkat lunak. (Revou, 2024)

2.6 VPS (*Virtual Private Server*)

VPS adalah mesin yang mengurus semua perangkat lunak dan data yang diperlukan untuk menjalankan sistem. *Server* ini disebut virtual karena hanya mengonsumsi sebagian dari sumber daya fisik yang mendasari *server* yang dikelola oleh penyedia pihak ketiga. Namun, pengguna mendapatkan akses ke sumber daya khusus pada perangkat keras tersebut. Adapun keunggulan atau kelebihan dari VPS adalah sebagai berikut (Biznet News, 2023a):

1. Mampu menangani trafik yang tinggi.
2. Sumber daya komputasi tidak terbagi-bagi.
3. Meningkatkan kinerja dari sistem.
4. Cukup memikirkan pembuatan aplikasi tanpa kesulitan untuk konfigurasi *server*.
5. Dapat diakses melalui jarak jauh atau *remote*.
6. Harga relatif terjangkau atau murah.
7. Penyedia layanan VPS sudah sangat banyak baik di Indonesia maupun luar negeri, sehingga memiliki banyak pilihan untuk menentukan VPS terbaik sesuai kebutuhan yang dimiliki (Biznet News, 2023a).

2.7 Python 3

Python 3 adalah salah satu bahasa pemrograman *open source* yang dapat digunakan untuk membantu dalam penangan program dengan menggunakan *socket*. Python 3 memiliki kemampuan untuk mengimplementasikan *socket* bertipe TCP dan juga UDP. Penerapan tipe TCP tentu berbeda dengan UDP. TCP pada *socket* Python 3 diwakili dengan SOCK_STREAM sedangkan UDP pada *socket* Python 3 diwakili dengan SOCK_DGRAM. Contoh dari pembuatan *socket* tipe TCP dan UDP pada Python 3 dengan model *client-server* adalah sebagai berikut (Python, 2024).

Implementasi *Socket Server* Python 3 Tipe TCP

```
import socket                                   ▷Import library socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
▷Inisialisasi socket
server.bind(('0.0.0.0', 55551))                 ▷Binding resource device
server.listen()                                 ▷Listening for client connection
client_socket, client_address = server.accept()  ▷Accept client
data = client_socket.recv(1024).decode("utf-8")  ▷Get data from client
print(data)
client_socket.send("Connection success!".encode("utf-8"))  ▷Send data to
client
```

Implementasi *Socket Client* Python 3 Tipe TCP

```
import socket                                   ▷Import library socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  ▷Inisialisasi
socket
```

```
client.connect(IP_ADDRESS_SERVER, PORT)          ▷ Connect to server
client.send("Hello Server!".encode("utf-8"))     ▷ Send data to server
data = client.recv(1024).decode("utf-8")         ▷ Get data from server
print(data)
```

Kode TCP diatas adalah cara arsitektur *client-server* TCP melakukan pengiriman data (Verma, 2023).

Implementasi *Socket Server* Python 3 Tipe UDP

```
import socket                                    ▷ Import library socket
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  ▷Inisialisasi socket
server.bind(('0.0.0.0', 55551))                  ▷ Binding resource device
data, address_initiate = server.recvfrom(1024)   ▷ Accept client
data = data.decode("utf-8")                      ▷ Get data from client
print(data)
server.sendto("Connection success!".encode("utf-8"), address_initiate)▷ Send data to client
```

Implementasi *Socket Client* Python 3 Tipe UDP

```
import socket                                    ▷ Import library socket
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  ▷Inisialisasi socket
server_address = (IP_ADDRESS_SERVER, PORT)
client.sendto("Hello Server!". encode("utf-8"), server_address)  ▷ Connect to
```

```
server
data, adddress_initiate = client.recvfrom(1024)          ▷ Send data to server
data = data.decode("utf-8")          ▷ Get data from server
print(data)
```

Kode UDP diatas adalah cara arsitektur *client-server* UDP melakukan pengiriman data (Virga, 2020). Hal yang dapat diperhatikan dari contoh pembuatan *socket* tipe TCP dan UDP pada Python 3 adalah sebagai berikut (Python, 2024):

1. Koneksi:
    - TCP mengharuskan *client* untuk melakukan koneksi kepada *server* terlebih dahulu sebelum melakukan pengiriman data dan saat melakukan pengiriman data tidak perlu memberikan alamat IP secara berulang-ulang.
    - UDP tidak mengharuskan *client* untuk melakukan koneksi kepada *server* terlebih dahulu sebelum melakukan pengiriman data akan tetapi saat ingin melakukan pengiriman data perlu memberikan alamat IP secara berulang-ulang.
2. Daya tahan:
    - TCP memiliki daya tahan waktu untuk mempertahankan koneksinya dalam jangka waktu yang cukup lama jika tidak terdapat pertukaran data melalui socket tersebut.
    - UDP memiliki daya tahan waktu yang relatif lebih cepat daripada TCP hanya sekitar beberapa menit jika tidak terdapat pertukaran data yang melalui socket tersebut.
3. Kecepatan:
    - TCP memiliki kecepatan yang lebih lambat dibandingkan UDP karena TCP akan memeriksa apakah data yang diterima mengalami kerusakan atau tidak.

- UDP memiliki kecepatan yang cepat karena sistem dari *socket* tipe UDP ini adalah "*fire-and-forget*". Artinya adalah data yang diterima tidak dicek apakah mengalami kerusakan atau tidak.

2.8 XMPP

Extensible Messaging Presence And Protocol (XMPP) adalah teknologi terbuka untuk komunikasi secara *real-time*, menggunakan *Extensible Markup Language* (XML) sebagai format dasar untuk pertukaran informasi. XMPP menyediakan cara untuk melakukan pengiriman potongan-potongan kecil XML dari satu entitas ke entitas lain dalam waktu yang hampir mendekati instan.

XMPP digunakan untuk jangkauan variasi aplikasi yang beragam. Pembahasan lebih dalam mengenai variasi penggunaan XMPP dapat dibagi menjadi dua hal yaitu dari sisi *services* dan dari sisi aplikasi. *Services* dalam XMPP terbagi menjadi dua bagian yaitu yang utama dan *extension*. Pembahasan mengenai *services* yang utama dipublikasikan oleh *Internet Engineering Task Force* (IETF) pada situs http://ietf.org disebut sebagai seri RFC (*Request For Comments*), dan mengenai *service* berupa *extension* dipublikasikan oleh XMPP *Standards Foundation* (XSF) pada situs http://xmpp.org disebut sebagai seri XEP (XMPP *Extensible Protocol*). Aplikasi adalah program perangkat lunak yang disebarkan melalui skenario ketertarikan umum secara individu ataupun organisasi (Saint-Andre et al., 2009).

2.8.1 *Services*

Service adalah fitur atau fungsi yang dapat digunakan oleh berbagai aplikasi apapun. Implementasi XMPP biasanya menyediakan layanan inti berupa:

1. *Channel Encryption*

   Didefinisikan pada [RFC 3920] yang mana adalah kemampuan enkripsi koneksi untuk klien dan *server*, atau antara dua *server* yang berhubungan.
2. *Authentication*

   Didefinisikan pada [RFC 3920] yang mana adalah kemampuan untuk mendeteksi atau memastikan entitas yang mencoba berkomunikasi

melalui jaringan sudah diautentikasi oleh *server* (bertindak seperti penjaga pada akses jaringan).

3. *Presence*

   Didefinisikan pada [RFC 3921] yang mana adalah kemampuan untuk mencari tahu tentang kondisi entitas pada suatu jaringan. Fitur *presence* didasarkan pada ikatan antara entitas yang sudah saling terkoneksi untuk menjaga privasi pengguna.

4. *Contact Lists*

   Didefinisikan pada [RFC 3921] yang mana adalah kemampuan untuk menyimpan daftar kontak atau *roster* pada XMPP *server*.

5. *One-to-one Messaging*

   Didefinisikan pada [RFC 3920] yang mana adalah sebuah kemampuan pada entitas untuk mengirimkan pesan kepada entitas lain (Saint-Andre et al., 2009).

2.8.2 Aplikasi

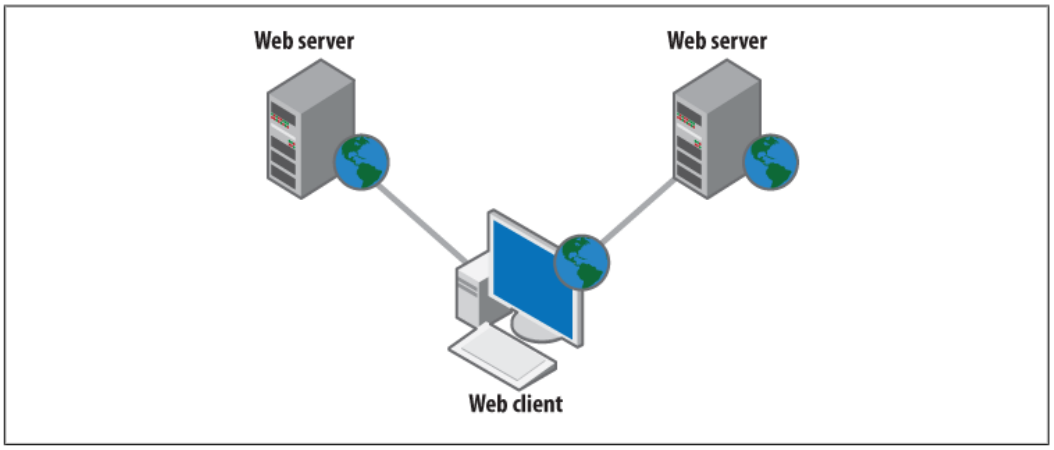
Banyak macam aplikasi yang dapat dibuat menggunakan XMPP sebagai dasar pembuatannya. Contoh dari aplikasi yang dapat dibuat adalah:

1. *Instant Messaging*.
2. *Middleware and Cloud Computing* (Saint-Andre et al., 2009).

2.8.3 Arsitektur XMPP

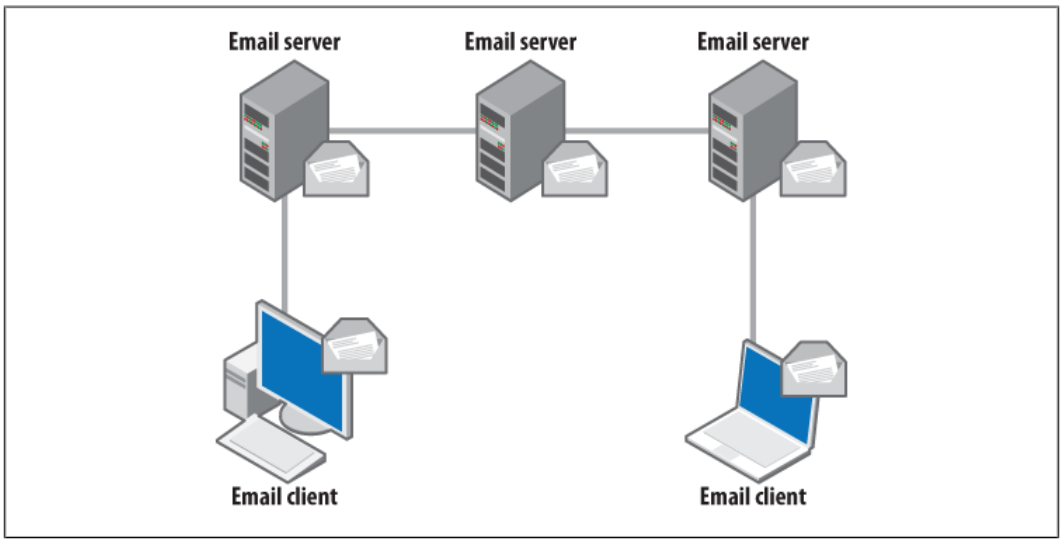
Semua teknologi yang ada dalam internet memiliki cara untuk saling terhubung dan berkomunikasi. Teknologi XMPP menggunakan arsitektur *client-server* yang terdesentralisasi serupa dengan arsitektur yang digunakan dalam *World Wide Web* dan jaringan *email*. Adapun perbedaan penting dalam arsitektur *World Wide Web*, *email*, dan XMPP akan dijelaskan dibawah.

Ketika menggunakan website, mesin pencari akan terkoneksi ke web *server*, tetapi antara web *server* biasanya tidak saling terkoneksi untuk menyelesaikan transaksi. Sebaliknya, HTML halaman *website* mungkin merujuk ke web *server* yang lainnya dan mesin pencari akan membuka session dengan web *server* tersebut untuk memuat halaman penuhnya. Dengan demikian *website* biasanya tidak memiliki koneksi antar-domain (*inter-domain*).



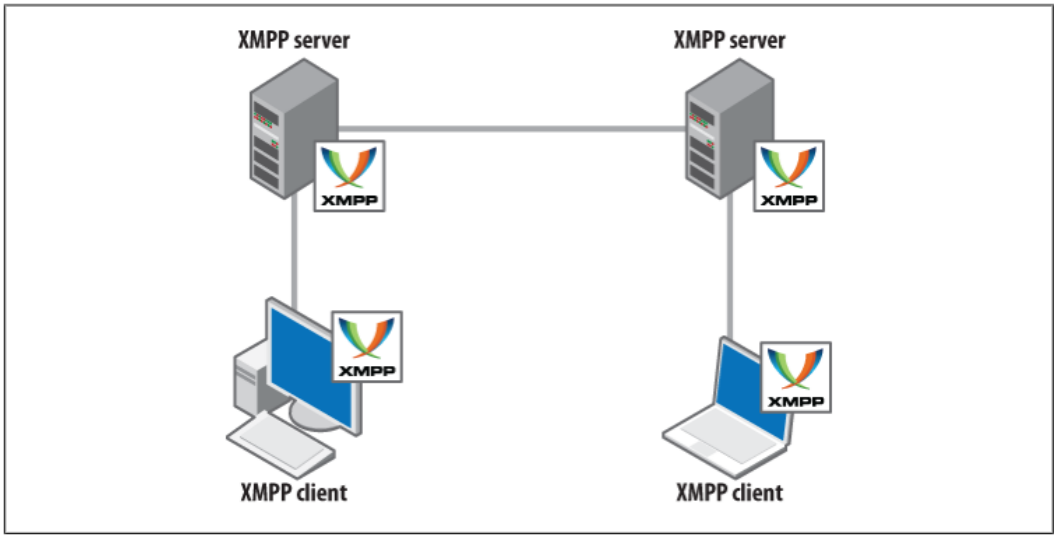
Gambar 2.1 Arsitektur Jaringan *World Wide Web*

Ketika mengirimkan *email* atau pesan ke salah satu kontak yang memiliki domain yang berbeda (xx@gmail.com ke xx@yahoo.com), pertama kali *email client* akan terkoneksi kepada domain utama *email server* pengirim, kemudian akan mencari rute menuju ke domain kontak yang dituju. Dengan demikian, tidak seperti *website*, sistem pada arsitektur *email* terdiri dari jaringan *server* gabungan. Bagaimanapun *email* yang dikirimkan memiliki kemungkinan akan melewati beberapa email *server* perantara sebelum mencapai tujuan akhir yaitu kontak yang ingin dituju. Dengan demikian jaringan *email* menggunakan beberapa lompatan antar *server* untuk mengirimkan pesan.



Gambar 2.2 Arsitektur Jaringan *Email*

Arsitektur XMPP lebih mirip dengan arsitektur *email* karena mampu melakukan koneksi antar domain (*inter-domain connections*). Ketika akan mengirimkan pesan kepada salah satu kontak dalam jaringan XMPP, maka pesan akan terkoneksi dengan domain miliknya kemudian *server* miliknya akan langsung terkoneksi pada *server* kontak yang dituju tanpa harus melalui *server* perantara. Koneksi langsung antar *server* ini memiliki kegunaan penting seperti mencegah *address spoofing* dan bentuk spam tertentu (Saint-Andre et al., 2009).



Gambar 2.3 Arsitektur Jaringan XMPP

Table 2.1 Perbedaan Arsitektur Jaringan

Arsitektur	*Interdomain Connections*	*Multiple Hops*
World Wide Web	*No*	-
Email	*Yes*	*Yes*
XMPP	*Yes*	*No*

2.8.4 Pengalamatan XMPP

Pada jaringan XMPP tentu saja akan mempunyai banyak entitas. Cara membedakan satu entitas dengan entitas lainnya adalah menggunakan alamat unik yang dimiliki tiap entitas. Adapun alamat unik dalam XMPP disebut sebagai JabberID (JID). JabberID memiliki format yang mirip seperti format alamat pada

email yaitu *user*@dns.com. Pengalamatan pada JabberID ada dua tipe yaitu *bare* JID dan *full* JID. Perbedaan diantara keduanya adalah *bare* JID ditulis seperti *user*@dns.com sedangkan *full* JID ditulis seperti *user*@dns.com/*resource* (Saint-Andre et al., 2009).

2.8.5 DNS (*Domain Name System*)

DNS adalah sebuah bentuk sederhana dari alamat IP. Penggunaan DNS dimaksudkan agar memudahkan mengingat alamat IP dengan menggunakan sebuah kata dibandingkan dengan nomor dari alamat IP itu sendiri. Contoh alamat IP untuk sebuah *website* pemberitahuan berita adalah 102.56.72.109 akan lebih sulit dihafal dibandingkan dengan https://beritakumilikku.com. Ketika DNS ini digunakan untuk membuat sebuah JabberID pada pengalamatan XMPP akan menjadi seperti *user*@beritakumilikku.com (Saint-Andre et al., 2009).

2.8.6 *Resource*

Resource yang tertera ketika JabberID adalah *full* JabberID adalah sebuah indikator untuk membedakan perangkat dari pengguna. Contoh pengguna atas nama Fajar dari *website* beritakumilikku.com memiliki dua perangkat yang terkoneksi ke dalam sistem XMPP, maka *full* JabberID untuk masing-masing perangkat adalah Fajar@beritakumilikku.com/perangkatKerja1 adalah JabberID pada perangkat pertama dan Fajar@beritakumilikku.com/perangkatKerja2 adalah JabberID pada perangkat kedua. Pada penamaan *resource* atau perangkat dalam sistem XMPP memiliki ketentuan yang berbeda-beda bergantung pada layanan sistem XMPP tersebut. Penamaan *resource* ada yang bersifat otomatis tidak dapat ditentukan oleh pengguna atau membebaskan pengguna untuk menentukan nama *resource* tersebut, selama *full* JabberID tidak menyamai dengan *full* JabberID yang sudah ada (Saint-Andre et al., 2009).

2.8.7 Komunikasi XMPP

Dalam XMPP terdapat sesuatu yang disebut *stanza*. *Stanza* adalah unit standar komunikasi dalam XMPP mirip seperti *packet* atau pesan didalam protokol jaringan. Beberapa faktor yang menentukan makna *stanza*:

1. Nama dari elemen *stanza* yang mana ada beberapa yaitu *message*, *presence*, iq (*info*/*query*). Masing-masing *stanza* ditangani berbeda-beda oleh *server* dan *client*.
2. Nilai dari atribut *type* yang bervariasi ditentukan berdasarkan sedang membicarakan *stanza* yang mana. Hal ini semakin membedakan setiap *stanza* berdasarkan cara pemrosesan oleh penerimanya.
3. *Child element* yang mana mendefinisikan sebuah payload untuk *stanza*. *Payload* mungkin diperlihatkan kepada pengguna atau diproses secara otomatis sebagaimana ditentukan oleh spesifikasi yang mendefinisikan *namespace payload* (Saint-Andre et al., 2009).

2.8.8 *Stanza Message*

Stanza message adalah metode dasar "*push*" untuk mengirimkan informasi dari satu tempat ke tempat lainnya. *Message* digunakan dalam sistem *instant messaging*, *groupchat*, *alerts and notifications*, dan sistem lainnya. *Message stanza* memiliki lima jenis yang dibedakan berdasarakan atribut *type* yaitu:

1. Normal

   Tipe ini mirip seperti pesan *email*. Tipe pesan yang mungkin akan ditanggapi atau tidak setelah dikirim.

2. *Chat*

   Tipe ini adalah bertukar pesan dalam waktu *real-time session* antara dua entitas. Mirip seperti teman yang saling bertukar pesan dalam aplikasi media sosial.

3. *Groupchat*

   Tipe ini adalah bertukar pesan oleh banyak pengguna dalam ruang obrolan, mirip seperti *Internet Relay Chat* (IRC).

4. *Headline*

   Tipe ini digunakan untuk mengirimkan *alerts and notifications* dan tidak mengharapkan tanggapan apapun dari penerima.

5. *Error*

   Tipe ini adalah untuk mengetahui apakah pesan yang dikirimkan mengalami kendala atau tidak, jika mengalami kendala maka tipe ini akan muncul sebagai tanggapan.

Stanza *message* selain memiliki atribut *type*, juga memiliki atribut *from* dan *to*, dimana atribut *from* menandakan siapakah yang mengirimkan pesan dan atribut *to* menandakan siapakah yang akan menerima pesan yang dikirimkan. Bagian *from* dan *to* akan berisikan JabberID masing-masing pengguna. Perbedaannya adalah bagian *from* didefinisikan oleh *server* pengirim untuk menghindari *address spoofing*. *Stanza message* juga memuat elemen *payload*. Spesifikasi inti XMPP mendefinisikan *payload* yang sangat dasar, seperti <*body*/> dan <*subject*/>. Contoh sederhana dari *stanza message* sebagai berikut (Saint-Andre et al., 2009).

***Stanza Message* XMPP**

<*message from*=”*person1@server.org*” *to*=”*friendPerson1@server2.org*” *type*=”*chat*”>

<*body*>Apa kabar bro?</*body*>

<*subject*>Pertanyaan</*subject*>

</*message*>

2.8.9 *Stanza Presence*

Salah satu ciri khas dari sistem komunikasi *real-time* adalah *presence* (kehadiran). Kegunaan dari *presence* adalah memperlihatkan ketersediaan jaringan dari suatu entitas dan dengan demikian memungkinkan pengguna melihat entitas tersebut sedang dalam jaringan dan dapat melakukan komunikasi. Untuk melihat kehadiran entitas lain pengguna harus melakukan sebuah otorisasi yaitu *presence subscription* dan harus diterima oleh entitas yang ingin dituju. Pengguna yang telah melakukan hal tersebut maka akan secara berkala menerima pembaruan informasi *presence* dari entitas. Prinsip paling dasarnya dari *stanza* ini adalah sebuah indikator untuk mengetahui apakah entitas sedang dalam jaringan

atau diluar jaringan (contoh keadaannya adalah “diluar” dan “tidak bisa diganggung”). Pada *stanza* ini juga dapat dilakukan perluasan fitur seperti memberikan pesan status seperti “sedang berada di kampus” atau “sedang mengemudi”. Contoh dari *stanza presence* adalah:

***Stanza Presence* XMPP**

<*presence from*=”*Fajar@beritakumilikku.com*”>

<*show*>*Away*</*show*>

<*status*>Chat saja di Whatsapp</*status*>

</*presence*>

Dalam aplikasi *instant messaging*, bagian *presence* biasanya dimunculkan pada *roster* (istilah untuk daftar kontak). *Roster* akan termuat daftar JabberID dan status dari para pengguna yang tersimpan didalamnya akan diberitahukan secara berkala (Saint-Andre et al., 2009).

2.8.10 *Stanza* IQ (*Info*/*Query*)

Stanza IQ menyediakan struktur untuk melakukan interaksi *request-response* dan alur kerja sederhana, mirip seperti metode *GET*, *POST*, dan *PUT* yang familiar pada HTTP. *Stanza* IQ hanya bisa memiliki satu *payload* tidak seperti *stanza message* yang dapat memiliki beberapa *payload* sekaligus. Dimana *payload* pada IQ mendefinisikan permintaan proses atau aksi yang harus dilakukan oleh penerima. Entiti yang mengirimkan *stanza* IQ harus selalu menerima tanggapan.

Permintaan dan tanggapan dapat dilacak menggunakan atribut id, yang dihasilkan melalui permintaan entitas dan akan dimasukan pada tanggapan entitas. Atribut *type* pada *stanza* IQ ada beberapa macam:

1. *Get*

   Atribut tipe permintaan untuk meminta suatu informasi.

2. *Set*

   Atribut tipe permintaan untuk memberikan suatu informasi atau membuat permintaan.

3. *Result*

   Atribut tipe tanggapan yang mengembalikan suatu permintaan, sesuai dengan yang meminta *GET* atau *SET*.

4. *Error*

   Atribut tipe tanggapan atau entitas perantara untuk memberitahukan bahwa permintaan *GET* atau *SET* mengalami suatu kesalahan atau permintaan tidak dapat diproses karena suatu hal.

Contoh pengguna *stanza* IQ dalam kasus meminta *roster* (daftar kontak) dan menambahkan kontak baru ke dalam *roster*.

Stanza IQ* XMPP *Get Roster

Client request:

<*iq from*=’manusiaGaul@*server*.*org*/*pda’*

id=’abcdefgh’

to=’manusiaGaul@*server*.org’

type=’get’>

<*query xmlns=’jabber*:*iq*:*roster’*/>

</*iq*>

Server response:

<*iq from*=’manusiaGaul@*server*.org’

id=’abcdefgh’

to=’manusiaGaul@*server*.*org*/*pda’*

```
    type='result'>
  <query xmlns='jabber:iq:roster'>
    <item jid='teman1@server1.org'/>
  </query>
</iq>
```

Pada contoh diatas pengguna manusiaGaul@*server.org* meminta daftar kontak yang dimiliki dengan cara memberitahukan melalui permintaan *stanza* IQ bertipe *GET* dan *query* berupa *jabber:iq:roster*.

Stanza IQ XMPP *Set Roster*

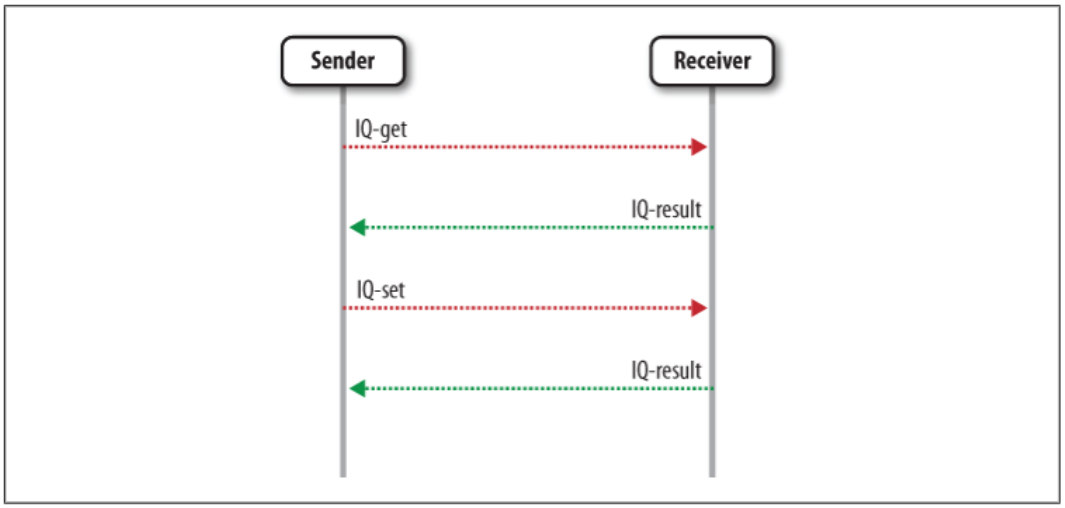
Client request:

```
<iq from='manusiaGaul@server.org/pda'
    id='abcdefgH'
    to='manusiaGaul@server.org'
    type='set'>
  <query xmlns='jabber:iq:roster'>
    <item jid='temanBaru@server.org'/>
  </query>
</iq>
```

Server response:

```
<iq from='manusiaGaul@server.org'
    id='abcdefgH'
    to='manusiaGaul@server.org/pda'
    type='result'/>
```

Pada contoh diatas pengguna manusiaGaul@*server.org* meminta untuk menambahkan kontak atas nama temamBaru@*server.org* dengan cara memberitahukan melalui permintaan *stanza* IQ bertipe *SET* dan *query* berupa *jabber*:*iq*:*roster* serta kontak baru tersebut dimasukkan kedalam *query* sebagai *payload* dengan tag berupa <*item*>. Penggunaan *stanza* IQ jika dilihat dalam gambar adalah sebagai berikut (Saint-Andre et al., 2009).



GAMBAR 2.4 Mekanisme *Request Response Stanza* IQ

2.9 Perbandingan Mengirim Data dengan XML dan JSON

XML adalah *Extensible Markup Language*, sedangkan JSON adalah *JavaScript Object Notation*. Bentuk data dari XML adalah memiliki sebuah tag pembuka, isi, dan tag penutup, sedangkan JSON berbentuk *key* dan *value*. Telah dilakukan uji perbandingan dari dua bentuk data tersebut untuk melakukan *GET*, *POST*, *PUT*, dan *DELETE*. Dari hasil pengujian yang telah dilakukan dapat disimpulkan bahwa, format data JSON memiliki keunggulan dibandingkan dengan format data XML diberbagai bentuk pengujian yang telah dilakukan (Breje et al., 2018). Hasil dari pengujiannya adalah sebagai berikut:

1. *GET Without GZIP Compression*

Table 2.2 Perbandingan JSON dan XML GET Without GZIP

Records No.	JSON Size	XML Size	JSON Second	XML Second
1000	281 KB	337 KB	0.49 s	1.03 s
5000	1402 KB	1689 KB	0.82 s	2.95 s
10000	2805 KB	3205 KB	1.63 s	5.1 s

2. *GET With GZIP Compression*

Table 2.3 Perbandingan JSON dan XML GET With GZIP

Records No.	JSON Size	XML Size	JSON Second	XML Second
1000	52.9 KB	73.4 KB	0.32 s	0.6 s
5000	263.1 KB	356.8 KB	0.61 s	2.21 s
10000	529.3 KB	731.3 KB	1.22 s	4.14 s

3. *POST*

Table 2.4 Perbandingan JSON dan XML POST

Records No.	JSON Second	XML Second	% XML over JSON
1000	1.25 s	2.21 s	176
5000	5.09 s	8.78 s	173
10000	11.11 s	17.02 s	153

4. PUT

Table 2.5 Perbandingan JSON dan XML PUT

Records No.	JSON Second	XML Second	% XML over JSON
1000	1.9 s	2.73 s	143
5000	9.73 s	17.43 s	179
10000	18.9 s	25.44 s	134

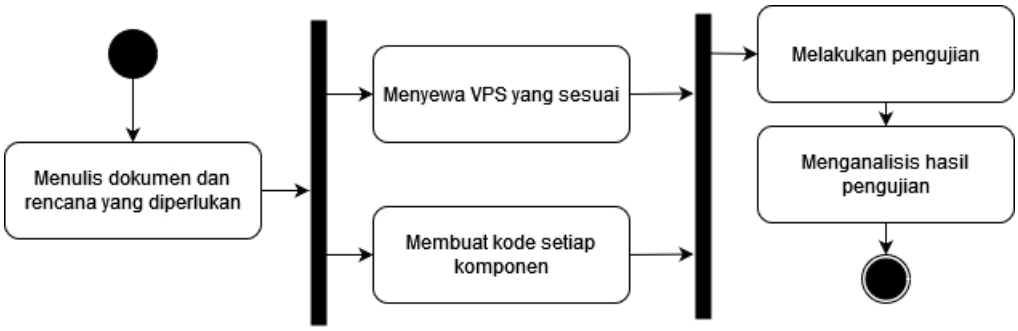
5. DELETE

Table 2.6 Perbandingan JSON dan XML DELETE

Records No.	JSON Second	XML Second	% XML over JSON
1000	0.4 s	0.55 s	137
5000	0.49 s	0.58 s	118
10000	0.58 s	0.7 s	120

BAB III METODOLOGI PENELITIAN

3.1 Tahapan Penelitian



Gambar 3.1 *Flowchart* Tahapan Penelitian Sistem Penghubung Komunikasi dengan Protokol XMPP

Permasalahan yang dihadapi pada penelitian ini adalah dua perangkat tidak dapat berkomunikasi jika tidak berada dalam jaringan yang sama. Sebagai gambaran seseorang memiliki satu komputer dan satu laptop. Komputer terhubung dengan jaringan Wifi dan laptop terhubung dengan jaringan *hotspot* gawainya. Maka komputer dan laptop tidak dapat berkomunikasi satu sama lain walaupun memiliki jarak yang dekat, hal ini dikarenakan mereka dipisahkan oleh jaringan. Bisa juga untuk aplikasi yang belum memiliki fitur berkomunikasi.

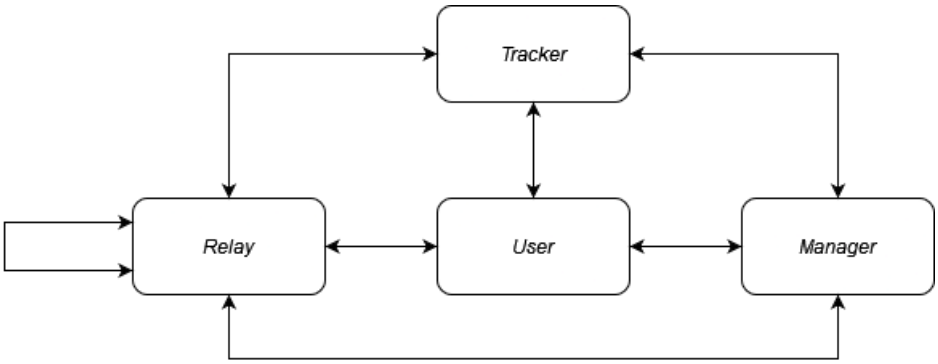
Penelitian akan dimulai dengan penulisan dokumen yang dibutuhkan, kemudian akan mencari penyedia IP publik yang berupa VPS (*Virtual Private Server*). VPS akan menjadi tempat menjalankan program *tracker*, *manager*, dan *relay*. Program akan dibangun dalam bentuk *socket programming*. Setelah semuanya dijalankan dan *user* melakukan koneksi *socket* ke *tracker*, *user* dapat melakukan pengiriman pesan ke *user* lainnya, dengan syarat *user* sudah melakukan otentikasi ke dalam sistem.

3.2 Arsitektur Sistem Penghubung Komunikasi dengan XMPP

Sistem dibuat berdasarkan permasalahan dua perangkat dengan IP *private* tidak dapat berkomunikasi secara langsung jika berada dalam jaringan yang berbeda atau sebuah aplikasi belum memiliki fitur untuk berkomunikasi. Arsitektur memiliki empat komponen utama yaitu *tracker*, *manager*, *relay*, dan *user*. Adapun peran dari masing-masing komponen adalah sebagai berikut:

1. ***Tracker*:** Memiliki peran sebagai tempat pertukaran informasi IP yang dimiliki oleh masing-masing komponen.
2. ***Manager*:** Memiliki peran sebagai pengatur keseimbangan jaringan pada *relay*. Ketika *user* akan bergabung ke jaringan percakapan, maka *manager* akan mengatur *user* tersebut akan masuk ke *relay* yang memiliki beban paling sedikit atau performa yang paling cepat.
3. ***Relay*:** Memiliki peran untuk menjembatani data yang berasal dari satu *user* ke *user* yang lain. Komponen *user* akan memiliki IP *Private*.
4. ***User*:** Memiliki peran utama seperti mengirimkan pesan ke sesama *user*.

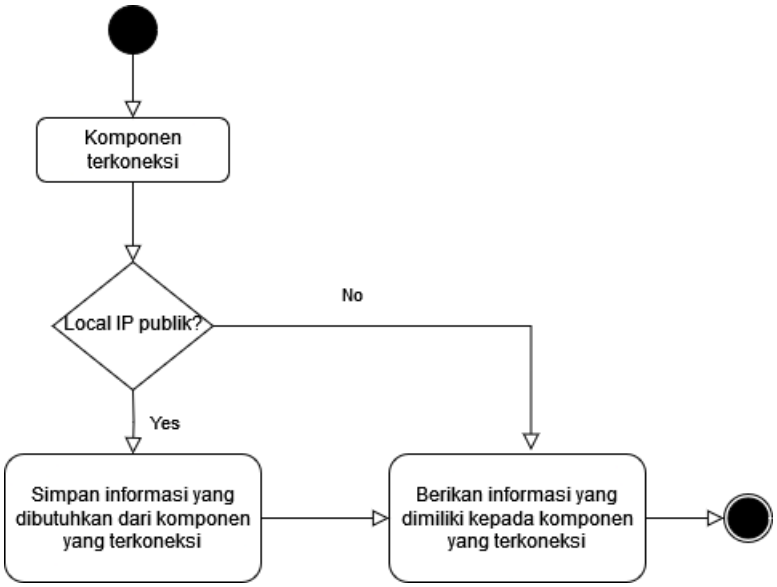
Arsitektur akan dibangun dengan jaringan terkoneksi menggunakan *socket* bertipe TCP (*Transmission Control Protocol*). Sistem *socket* tipe TCP menggunakan teknik koneksi (*connection*) antara dua *socket client*. Format protokol untuk *socket* TCP adalah (*socket_protocol*, *client_1_address*, *client_1_port*, *client_2_address*, *client_2_port*). Fungsi dari *socket* adalah sebagai jembatan pertukaran data antara dua perangkat berbeda yang saling terhubung melalui format protokol TCP.



Gambar 3.2 Arsitektur Sistem Penghubung Komunikasi dengan XMPP

3.2.1 Komponen *Tracker*

Komponen *tracker* akan memiliki peran sebagai pencatat alamat IP publik dari komponen. Komponen yang akan dicatat alamat IP publiknya adalah *manager*, dan *relay*.



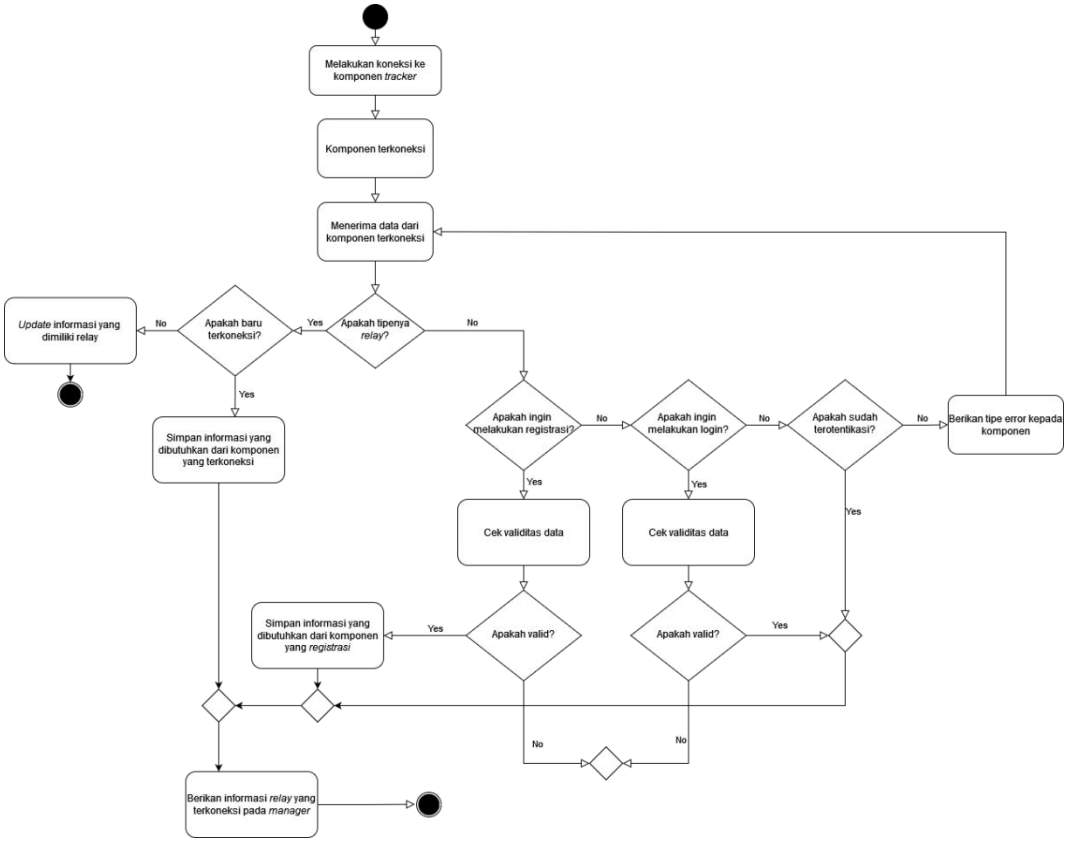
Gambar 3.3 *Activity Diagram* Komponen *Tracker*

3.2.2 Komponen *Manager*

Komponen *manager* adalah komponen pengatur administrasi, mulai dari keunikan tiap komponen hingga distribusi komponen. Komponen *manager* akan memiliki peran sebagai berikut.

1. Jika ada komponen *relay* yang menghubungkan diri kepada komponen *manager* maka tugas komponen *manager* adalah mencatat alamat IP dari *relay* tersebut dan memberikan seluruh alamat IP *relay* yang dimilikinya kepada *relay* yang baru saja melakukan penghubungan diri kepada *manager*.
2. Jika ada komponen *user* yang menghubungkan diri kepada komponen *manager* maka tugas komponen *manager* adalah mencari *relay* dengan jumlah *user* yang terhubung paling sedikit dan memberikan alamat IP dari *relay* tersebut kepada komponen *user* yang baru saja terhubung ke komponen *manager*. Selanjutnya komponen *manager* mencatat *username* dari *user* yang melakukan koneksi.

3. Peran komponen *manager* terhadap komponen *relay* adalah mencatat sudah berapa banyak *user* yang terhubung ke suatu *relay* dan memberikan *username* untuk dicatat dalam *database*.
4. Peran komponen *manager* terhadap komponen *user* adalah memberikan alamat IP *relay* yang memiliki beban paling ringan.

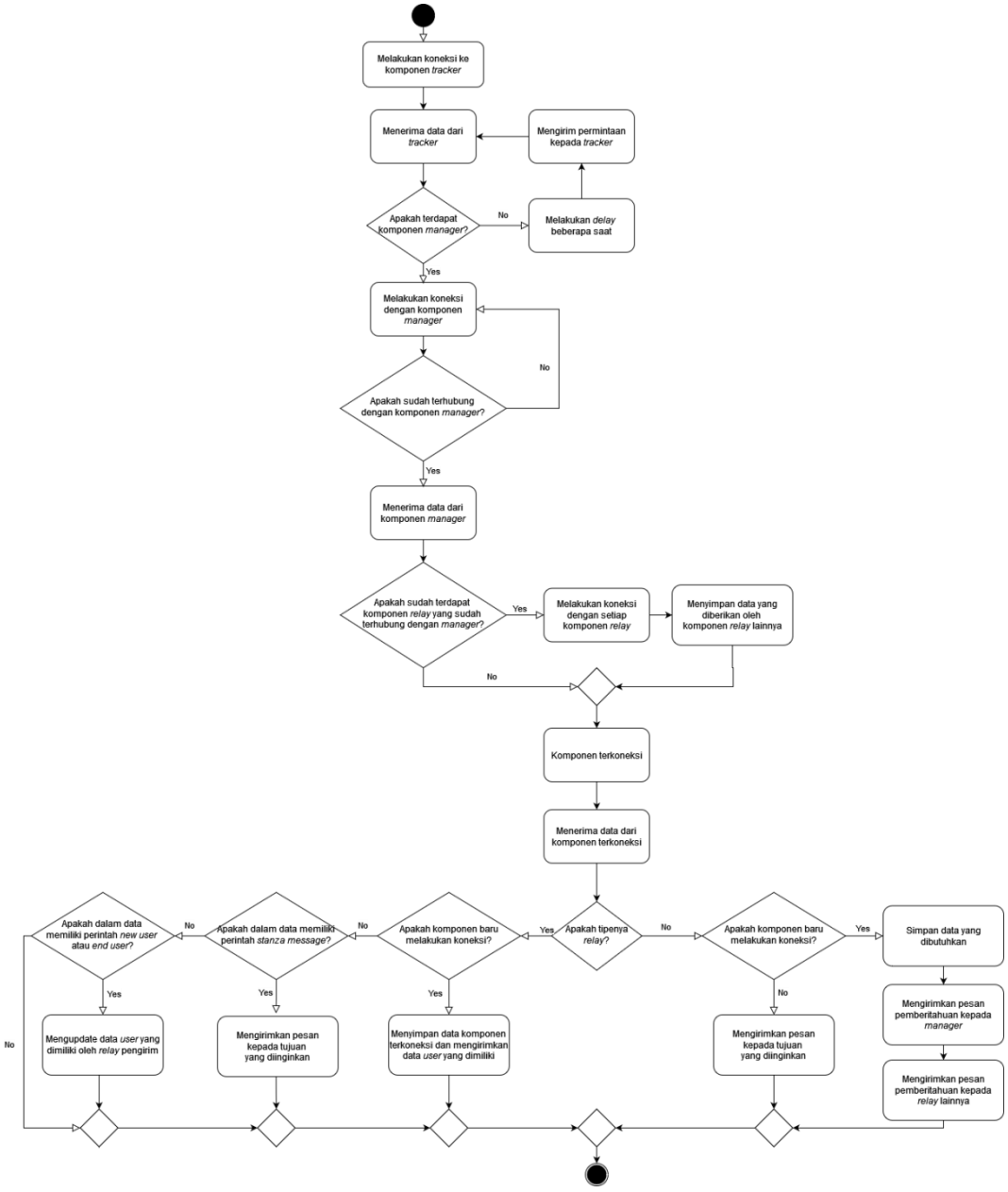


Gambar 3.4 *Activity Diagram* Komponen *Manager*

3.2.3 Komponen *Relay*

Komponen *relay* adalah komponen inti sebagai komponen penghubung dari sistem penghubung komunikasi, yang memiliki peran sebagai berikut.

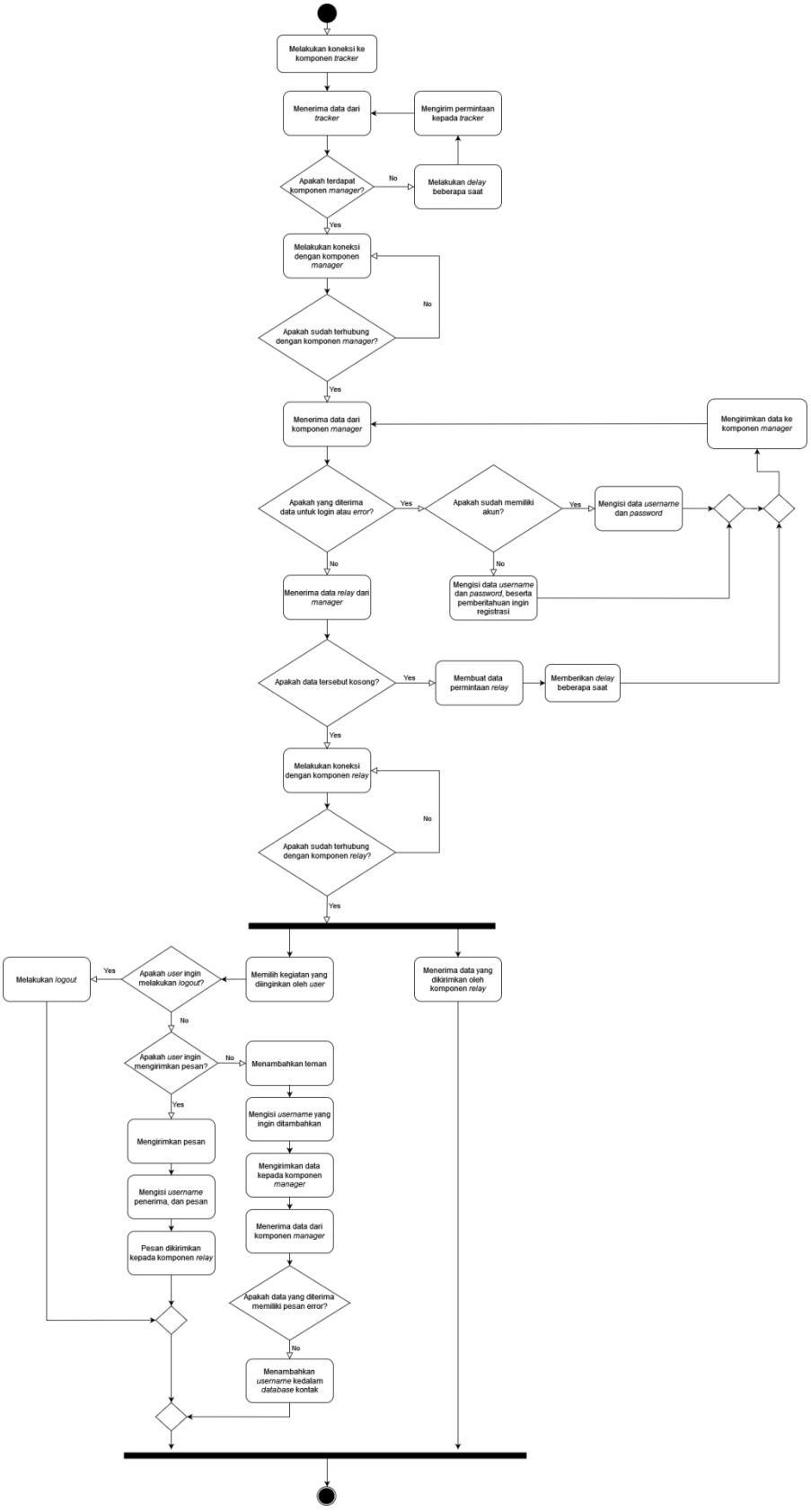
1. Mencatat seluruh *username* yang terhubung dengannya dan didapat dari komponen *manager*.
2. Sebagai jalur pertukaran data dari satu *user* ke *user* lainnya.
3. Saling berkaitan antar komponen *relay* agar *user* yang terhubung pada *relay* berbeda dapat saling berkomunikasi.



Gambar 3.5 *Activity Diagram* Komponen *Relay*

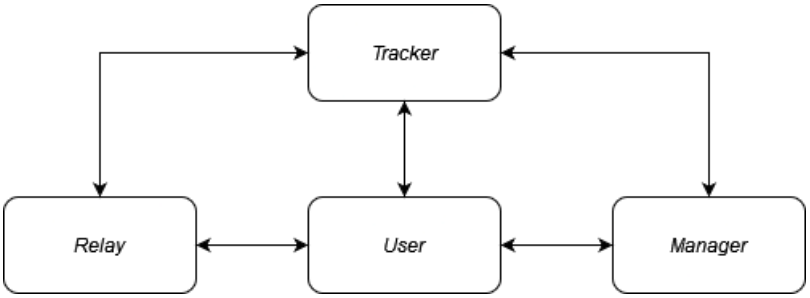
3.2.4 Komponen *User*

Komponen *user* akan memiliki peran untuk menguji sistem yang telah dibuat dapat bekerja dengan baik dalam hal mengirimkan data dari *user* yang tidak terhubung dalam jaringan yang sama atau berada dalam sebuah aplikasi. Komponen *User* manual akan menggunakan program sederhana untuk mengirimkan pesan.



Gambar 3.6 *Activity Diagram* Komponen *User*

3.2.5 Relasi *Tracker* dengan Komponen Lainnya



Gambar 3.7 Relasi Antara *Tracker* dengan Komponen Lainnya

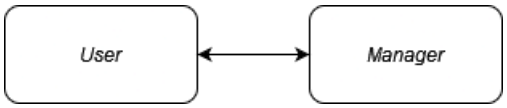
Peran dari *tracker* adalah untuk mengumpulkan alamat IP setiap koneksi yang terkonek ke dirinya dan memberikan setiap alamat IP yang sudah dicatat oleh dirinya kepada yang terkoneksi ke dirinya.

3.2.6 Relasi *Relay* dengan *Manager*

Gambar 3.8 Relasi Antara *Relay* dan *Manager*

Peran *manager* terhadap *relay* adalah sebagai penjaga kestabilan *relay* agar pekerjaan *relay* tidak berat. Komponen *manager* akan mencatat IP dari *relay* dan akan mengatur apabila ada *user* yang ingin bergabung ke dalam sistem *chat*. Komponen *manager* mengatur dengan cara mencari *relay* yang memiliki beban paling sedikit apabila *relay* terdapat lebih dari satu.

3.2.7 Relasi *User* dengan *Manager*

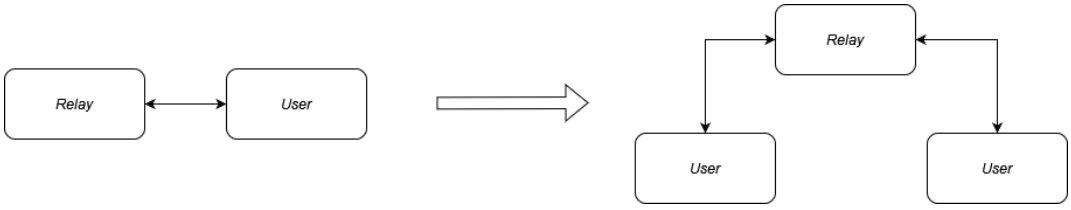


Gambar 3.9 Relasi Antara *User* dengan *Manager*

Peran *manager* terhadap *user* adalah sebagai pengarah dan administrator. Komponen *manager* akan mengarahkan komponen *user* agar komponen *user* melakukan koneksi dengan *relay* yang telah diberikan oleh *manager*. Cara

kerjanya adalah *user* melakukan koneksi dengan *manager*, *manager* akan mencari *relay* mana yang memiliki beban paling sedikit atau *relay* yang memiliki performa yang cepat. Kemudian komponen *relay* yang didapat, IP dari *relay* tersebut akan diberikan kepada *user* yang meminta untuk terhubung ke jaringan. Dengan syarat *user* sudah melakukan pendaftaran.

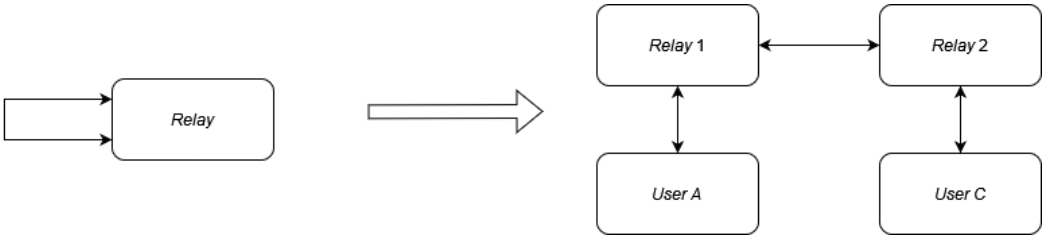
3.2.8 Relasi *User* dengan *Relay*



Gambar 3.10 Relasi Antara *Relay* dan *User*

Peran *relay* terhadap *user* adalah sebagai jembatan pertukaran data antar *user*. Hal ini dikarenakan *user* tidak dapat secara langsung mengirimkan data kepada *user* yang lainnya karena ketidakmampuan untuk melakukan koneksi TCP secara langsung antar *user*.

3.2.9 Relasi Antar *Relay*



Gambar 3.11 Relasi Antar *Relay*

Komponen *relay* berfungsi sebagai jembatan data untuk *user* mengirimkan data kepada *user* yang lainnya. Komponen *user* yang ingin mengirim dan yang menerima tentu saja haruslah memiliki koneksi dengan komponen *relay*. Lalu bagaimana jika *user* yang menerima tidak terhubung dengan *relay* yang sama. Tentu saja *user* tidak akan dapat menerima data tersebut karena tidak terdapat jalur yang menghubungkan keduanya. Disinilah fungsi dari relasi antar *relay*, kedua *relay* membangun koneksi untuk membentuk suatu jaringan sehingga *user*

pada *relay* A dapat mengirimkan pesan kepada *user* pada *relay* B yang sudah saling membangun koneksi.

3.3 Format XMPP

Format dari protokol XMPP adalah XML. Pada penelitian ini format XML akan diganti dengan format JSON. Contoh pengubahan format XML menjadi JSON akan terlihat seperti berikut (contoh menggunakan XMPP *stanza message*):

XMPP XML *Stanza Message*

```
<message from='initiate_entity' to='receiver_entity' lang='id'>
    <body> Hello World! </body>
</message>
```

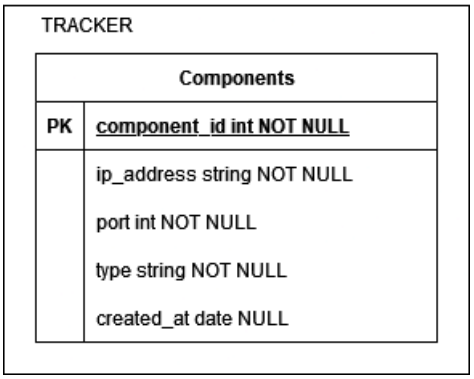
XMPP JSON *Stanza Message*

```
{
    "stanza" : "message",
    "from" : "initiate_entity",
    "to" : "receiver_entity",
    "lang" : "id",
    "body" : "Hello World!"
}
```

3.4 Penyimpan Data

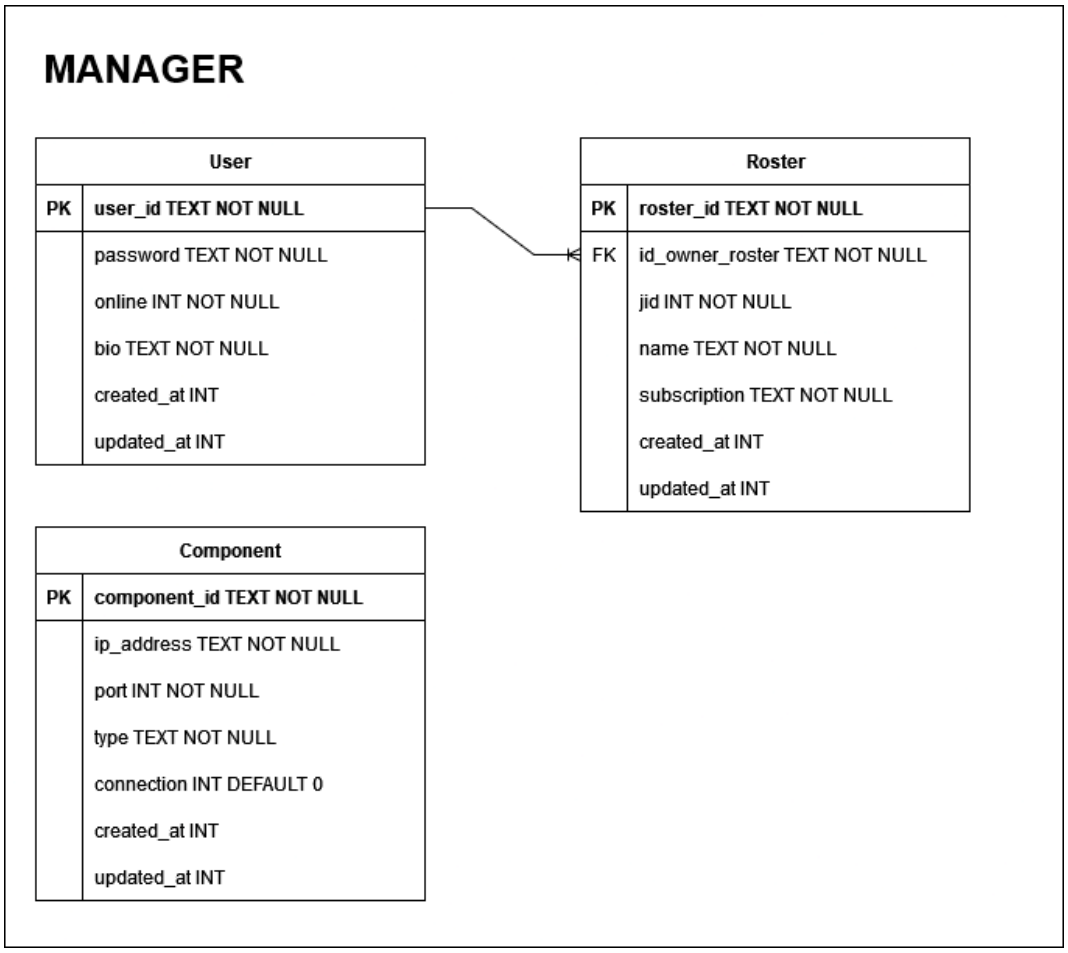
Setiap komponen memiliki data yang akan disimpan, oleh sebab itu komponen memerlukan struktur penyimpanan data untuk data yang ingin disimpan olehnya. Penyimpanan data akan dibentuk menggunakan teknologi SQL (*Structured Query Language*).

3.4.1 Struktur Penyimpan Data *Tracker*



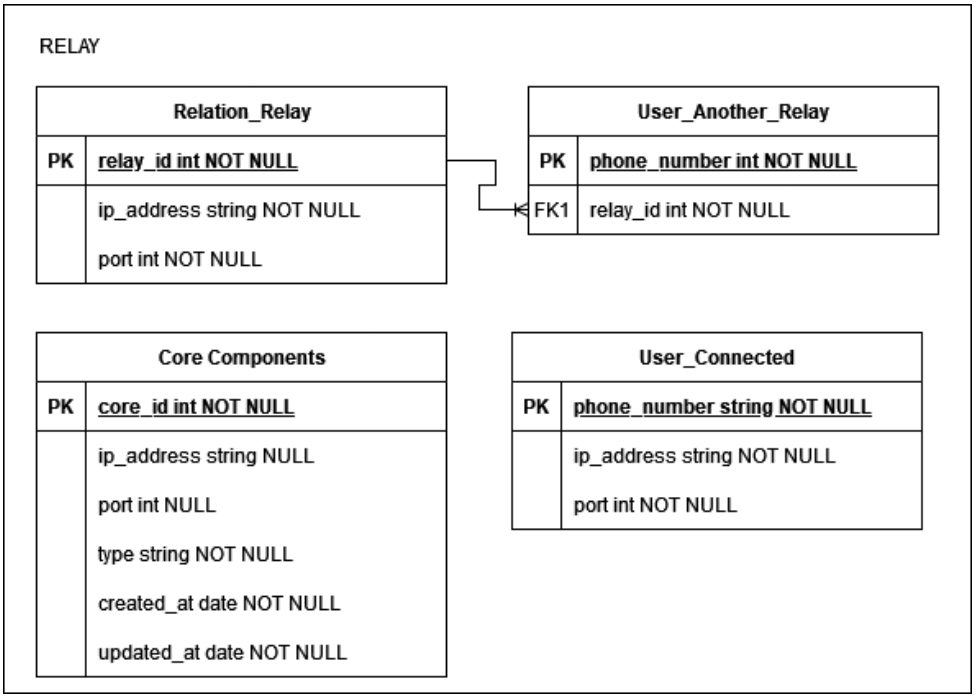
Gambar 3.12 *Database* Pada Komponen *Tracker*

3.4.2 Struktur Penyimpanan Data *Manager*



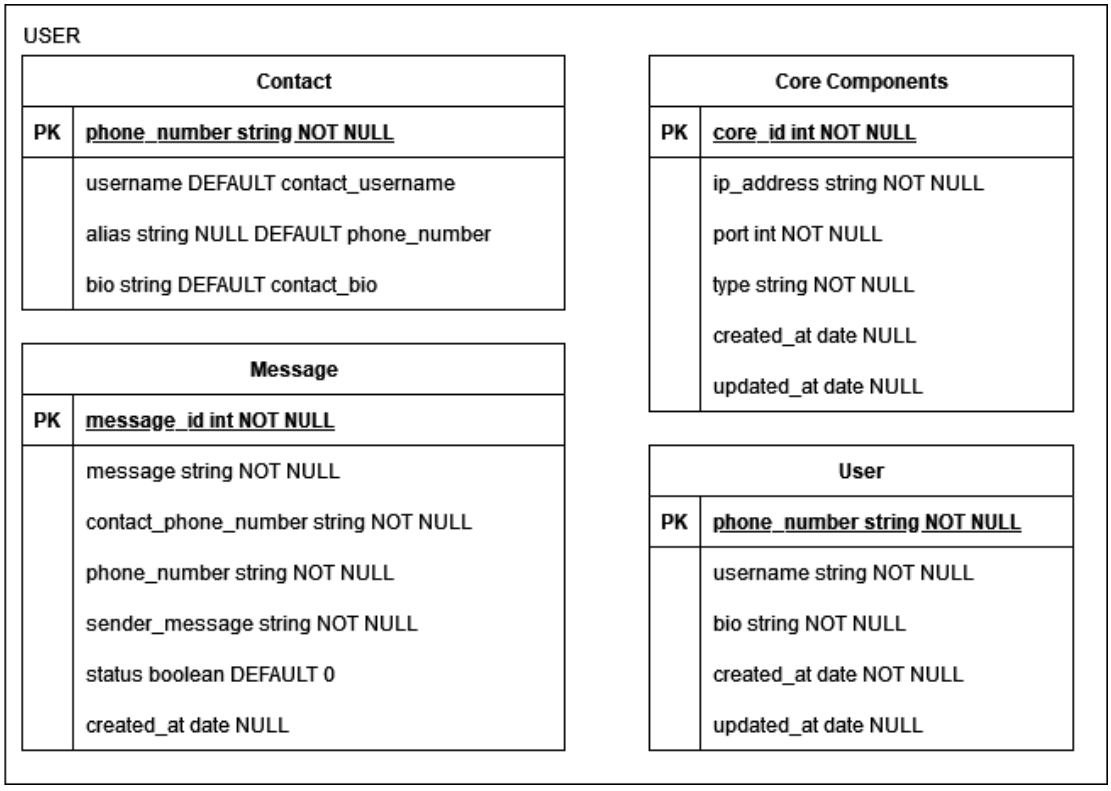
Gambar 3.13 *Database* Pada Komponen *Manager*

3.4.3 Struktur Penyimpanan Data *Relay*



Gambar 3.14 *Database* Pada Komponen *Relay*

3.4.4 Struktur Penyimpanan Data *User*



Gambar 3.15 *Database* Pada Komponen *User*

3.5 Alat dan Bahan Penelitian

Penelitian ini menggunakan beberapa alat untuk membuat sistem, menyimpan data, dan menjalankan sistem yaitu:

1. VSCode.
2. SQLite3.
3. Python 3.
4. VPS (*Virtual Private Server*) sebanyak dua buah.
5. Perangkat komputer/laptop.

3.6 Tahapan Pengembangan

3.6.1 Pemilihan VPS

VPS dipilih berdasarkan sumber daya yang ditawarkan beserta harga yang paling sesuai. Pada penelitian ini VPS yang dipilih adalah VPS dengan sumber daya yang tidak terlalu besar dan juga harga yang murah karena pada penelitian ini hanya akan dijalankan dalam ruang lingkup yang kecil dan sederhana.

3.6.2 System Development Life Cycle (SDLC)

Penelitian ini akan mengalami beberapa siklus dalam pembuatan sistem hingga melakukan pengujian sistem, adapun siklus yang direncanakan sebagai berikut:

- Siklus 1
    1. Siklus satu diestimasikan satu minggu.
    2. Mensurvei VPS termurah.
    3. Membuat komponen *tracker*.
    4. Membuat awalan komponen *manager*, dan *relay*. Memastikan agar dapat terhubung dan melakukan pertukaran data dengan *tracker*.
    5. Memastikan komponen *relay* dapat terhubung dan melakukan pertukaran data dengan *manager*.
- Siklus 2
    1. Siklus dua diestimasikan dua minggu.
    2. Membuat fungsionalitas *manager* untuk melakukan pemetaan *relay*, serta distribusi *relay* yang paling ringan koneksinya.

3. Membuat fungsionalitas pada *relay* untuk melakukan pemetaan *user*/*client* yang terkoneksi dengan sistem, dan dapat melakukan pertukaran data dengannya.
4. Membuat fungsionalitas *manager* untuk keunikan *relay* dan *user*/*client* yang terkoneksi dengan sistem.
5. Membuat fungsionalitas pada *relay* untuk mengirimkan data berapa *user*/*client* yang terkoneksi dengannya.
6. Membuat fungsionalitas agar komponen *relay* dapat saling terhubung.

- Siklus 3
    1. Siklus tiga diestimasikan satu minggu.
    2. Melakukan validasi apakah semuanya sudah baik dan benar.
    3. Membuat program pengujian otomatis untuk user/client pengirim pesan.
    4. Membuat program pengujian otomatis untuk user/client penerima pesan.
    5. Melakukan pengukuran dengan hasil yang telah diperoleh dari pengujian sistem otomatis.
    6. Melakukan pengujian manual dengan aplikasi pengguna sederhana yang dilakukan oleh beberapa orang yang bisa menggunakan aplikasinya.
    7. Melakukan pengukuran kecocokan dengan hasil yang telah diperoleh dari pengujian sistem otomatis.

3.7 Skema Uji

Skema untuk menguji sistem penghubung komunikasi dengan XMPP dapat berjalan dengan baik dan benar adalah sebagai berikut:

1. Menjalankan program *tracker* pada VPS pertama.
2. Menjalankan program *manager* pada VPS kedua.
3. Menguji fungsionalitas menerima koneksi dan penyimpanan komponen pada *tracker*.
4. Menjalankan program *relay* senior pada VPS kedua.

5. Menguji fungsionalitas memberikan komponen yang disimpan oleh *tracker* kepada komponen yang baru terkoneksi.
6. Menguji fungsionalitas menerima koneksi dan penyimpanan komponen pada *manager*.
7. Menjalankan program *relay* junior pada VPS pertama.
8. Menguji fungsionalitas menerima koneksi dan memberikan data konfigurasi awal pada *relay* senior.
9. Menjalankan program *user* pada beberapa perangkat komputer/laptop yang memiliki LAN berbeda.
10. Menguji setiap fungsionalitas pada *manager*, diantaranya adalah:
    a. Registrasi
    b. *Login*
    c. Mendapatkan komponen *relay* dengan koneksi paling sedikit.
    d. Pengelolaan *roster*.
    e. Pengelolaan *presence*.
11. Menguji fungsionalitas *relay* sebagai pengirim data antar *user*.

3.7.1 Metrik Pengujian

Performa sistem penghubung komunikasi dengan XMPP akan dinilai dengan menggunakan metrik sebagai berikut:

1. **Proses Komunikasi**: komunikasi antar komponen dapat berjalan dengan baik dan benar.
2. **Konsistensi Data**: kemampuan sistem untuk mengirimkan data sehingga tidak akan ada data yang hilang selama proses pengiriman data berlangsung dan konsistensi waktu pengiriman data.
3. **Fungsionalitas**: kemampuan setiap komponen dalam menjalan tugas dan perannya masing-masing.

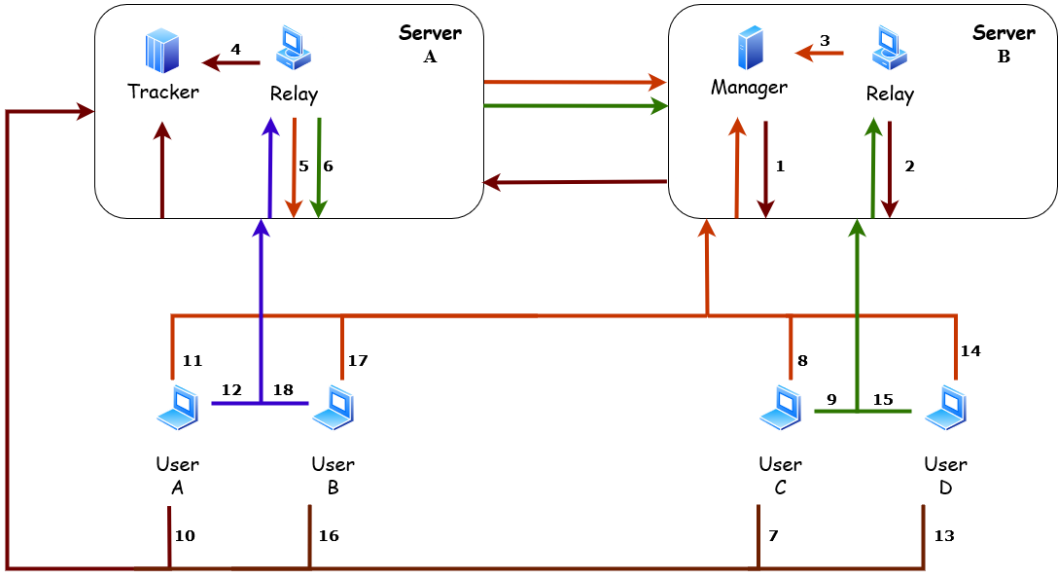
3.7.2 Rancangan Pengujian

Pengujian akan dirancang menjadi dua buah VPS/server, VPS pertama akan menjalankan program *tracker* dan *relay* junior, dan VPS kedua akan menjalankan program *manager* dan *relay* senior. Komponen penguji akan dijalankan pada komputer/laptop yang memiliki LAN berbeda.

BAB IV
HASIL DAN PEMBAHASAN

4.1 Implementasi Lingkungan Program dan Sistem

Lingkungan program dibuat menjadi dua yaitu, lingkungan program komponen *tracker* dan *relay*, dan komponen *manager* dan *relay*. Satu lingkungan program dibuat pada satu VPS atau *server*. IP yang dimiliki VPS atau *server* adalah IP publik dan dapat dilakukan koneksi *socket* TCP kepada dirinya.



Gambar 4. 1 Implementasi Relasi Antar Komponen

Sistem memiliki empat jenis folder utama, yaitu:

1. *Tracker*, berisi kebutuhan komponen agar menjadi komponen *tracker*.
2. *Manager*, berisi kebutuhan komponen agar menjadi komponen *manager*.
3. *Relay*, berisi kebutuhan komponen agar menjadi komponen *relay*.
4. *Utils*, berisi kebutuhan utilitas yang secara umum dibutuhkan oleh masing-masing komponen.

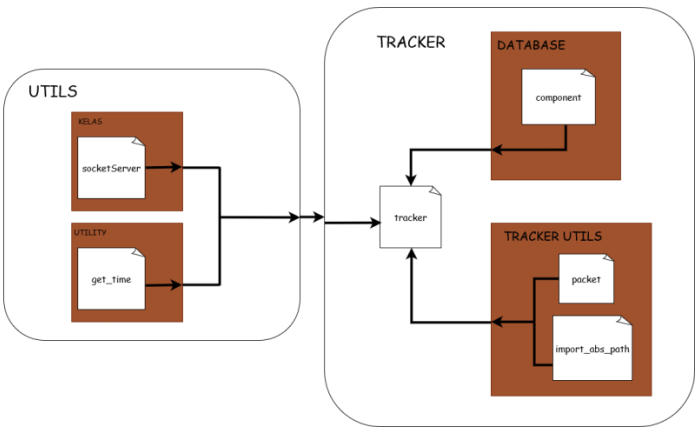
Sistem memiliki dua jenis file yaitu, file berisi tentang kelas dan file berisi fungsionalitas. Adapun file berisi tentang kelas yaitu init_db.py terdapat dalam database, socketServer.py terdapat dalam folder utils\kelas, dan socketClient.py.

4.2 Implementasi Komponen *Tracker*

Komponen *tracker* memiliki peran untuk memberikan informasi komponen *manager* atau *relay* yang terkoneksi di dalam sistem yang aktif. Maka adapun metode atau fungsionalitas yang diimplementasikan pada *tracker* adalah sebagai berikut:

1. Menerima setiap koneksi *socket* TCP yang menuju kepada dirinya.
2. Membedakan antara komponen, berdasarkan *packet* yang diterima. Bentuk *packet* adalah JSON dengan daftar kunci adalah *ip_local*, *port*, *type*, dan *is_private*.
3. Menyimpan data yang memiliki nilai dari kunci *type* adalah *manager* atau *relay*.
4. Memberikan komponen aktif yang disimpannya kepada komponen yang baru saja melakukan koneksi kepada dirinya.
5. Memberikan komponen aktif yang disimpannya kepada komponen yang mengirim *packet* dengan nilai *message* adalah "*get components*".
6. Melakukan pembaruan data untuk komponen yang *error* atau tidak lagi aktif.
7. Menerima segala *packet* pesan yang masuk dari komponen *manager*, *relay*, dan *user*.

Komponen *tracker* dapat dijalankan dengan memanggil file utamanya yaitu tracker.py yang terdapat dalam folder *tracker*. Adapun diagram dari komponen *tracker* secara lengkap bisa dilihat dibawah.



Gambar 4. 2 Diagram Lengkap Komponen *Tracker*

4.3 Implementasi Komponen *Manager*

Komponen *manager* memiliki tugas mengatur administrasi *user* dan *relay*, layanan pengelolaan *roster* (kontak *user*) untuk *user*, dan layanan pengelolaan *presence* (status *user*). Maka adapun metode atau fungsionalitas yang diimplementasikan pada *manager* adalah sebagai berikut:

1. Melakukan koneksi *socket* TCP kepada komponen *tracker*. Mengirimkan *packet* kepada TCP dengan nilai kunci *type* adalah *manager*.
2. Menerima setiap koneksi *socket* TCP yang menuju kepada dirinya.
3. Menyediakan *username* untuk *relay*.
4. Menyimpan data *relay* yang terkoneksi dengan dirinya.
5. Memberikan data *relay* yang terkoneksi dengan dirinya kepada *relay* yang baru terkoneksi.
6. Memperbarui data *connection* dari suatu *relay*, jika mendapatkan pesan "ncir" atau "ecir" dari *relay*-nya.
7. Menghapus *relay* yang tiba-tiba *error* atau terputus.
8. Melayani *user* yang ingin melakukan pendaftaran.
9. Menangani data yang tidak sesuai ketika dikirimkan oleh *user* saat mendaftar, diantaranya adalah:
    - Data *username* atau *password* kosong.
    - Data *username* atau *password* mengandung karakter *whitespace*.
    - Data *username* yang dimasukkan telah terdaftar dalam sistem.
10. Menyimpan data diri *user* ketika pendaftaran telah berhasil.
11. Melayani *user* yang ingin melakukan *login*.
12. Menangani data yang tidak sesuai ketika dikirimkan oleh *user* saat melakukan *login*, diantaranya adalah:
    - Data *username* atau *password* kosong.
    - Data *username* atau *password* mengandung karakter *whitespace*.
    - Data *username* tidak terdapat di dalam sistem.
    - Data *password* yang dimasukkan tidak sesuai dengan *password* akun.
13. Memberikan *relay* dengan koneksi paling sedikit kepada *user* yang baru saja melakukan *login*.

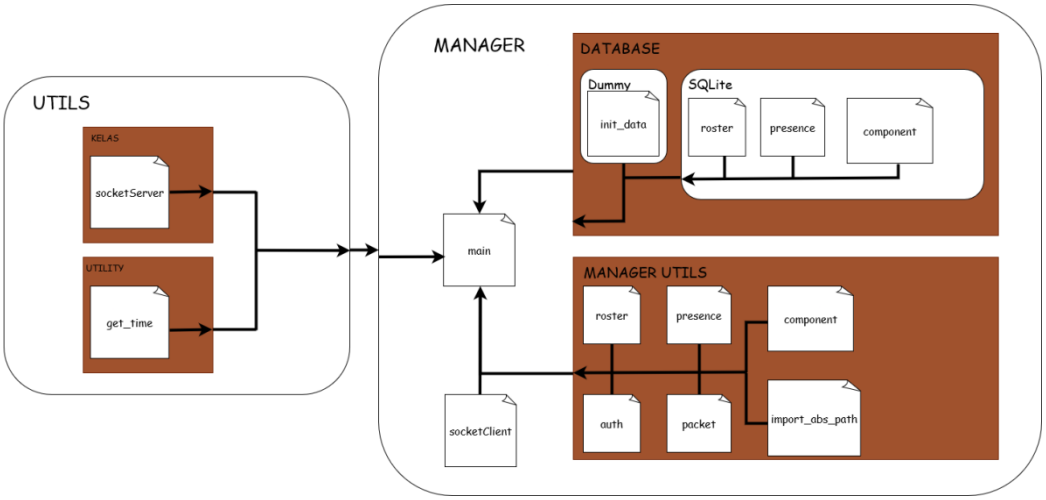
14. Memberikan *relay* dengan koneksi paling sedikit kepada *user* yang sedang *online* ketika menerima pesan bertipe "*get_relay_less_connection*".
15. Melayani *user* untuk menambahkan *roster item*.
16. Menangani data yang tidak sesuai ketika menambahkan *roster item*, diantaranya adalah:
    - Data yang dikirimkan formatnya bukan JSON.
    - Data yang dikirimkan tidak lengkap.
    - Data yang dikirimkan bertujuan untuk menambahkan *roster* orang lain.
    - Data *username roster item* tidak terdapat di dalam sistem.
17. Mengolah data yang valid dengan cara mengecek keterhubungan *user* dengan *roster item* yang ditambahkan ketika menambahkan *roster item* untuk mengatur nilai *subscription*-nya, diantaranya adalah:
    - Jika sudah memiliki keterhubungan maka nilai *subscription*-nya adalah "*both*" untuk kedua belah pihak.
    - Jika belum memiliki keterhubungan maka nilai *subscription* untuk yang menambahkan *roster item* adalah "*to*", dan sistem akan secara otomatis menambahkan *user* ke dalam *roster item's roster* dengan nilai subscription adalah "*from*".
18. Mengirimkan *presence* bertipe "*subscribed*" kepada *user* ketika berhasil menambahkan *roster item* ke dalam server dan mengirimkan *presence* dari *roster item* yang baru saja ditambahkan kepada *user*.
19. *User* dapat mengubah atribut *name* pada *roster item* yang diinginkan.
20. Melayani *user* untuk menghapus *roster item*.
21. Menangani data yang tidak sesuai ketika menghapus *roster item*, diantaranya adalah:
    - Data yang dikirimkan formatnya bukan JSON.
    - Data yang dikirimkan tidak lengkap.
    - Data yang dikirimkan bertujuan untuk menghapus *roster* orang lain.
    - Data *username roster item* tidak memiliki keterhubungan dengan *user*.

- Data *username roster item* pada *user's roster* memiliki nilai *subscription* "*from*".

22. Mengolah data yang valid dengan cara mengecek keterhubungan *user* dengan *roster item* yang dihapus ketika menghapus *roster item* untuk mengatur nilai *subscription*-nya, diantaranya adalah:
    - Jika keterhubungannya adalah "*both*" maka nilai *subscription* untuk yang menghapus *roster item* adalah "*from*", dan sistem akan secara otomatis mengubah tipe *subscription* pada *roster item's roster* menjadi "*to*".
    - Jika keterhubungan adalah "*to*" maka *roster item* akan dihapus dari *roster user*, dan *user* akan dihapus dari *roster item's roster*.
23. Mengirimkan *presence* bertipe "*unsubscribed*" kepada *user* ketika berhasil menghapus *roster item* dari dalam server.
24. Melayani *user* untuk melakukan inisialisasi *presence* saat pertama kali *online*, hal yang akan dilakukan untuk pelayanannya adalah:
    - Memberikan *presence user* kepada semua *roster item* yang memiliki nilai *subscription* adalah "*from*" atau "*both*".
    - Memberikan *presence roster item* kepada *user* yang memiliki nilai *subscription* pada *user's roster* adalah "*to*" atau "*both*".
25. Melayani *user* untuk melakukan perubahan bio.
26. Mengirimkan perubahan bio yang berhasil dilakukan oleh *user* kepada semua *roster item* yang memiliki nilai *subscription* adalah "*from*" atau "*both*".
27. Melayani *user* yang ingin meminta *presence* dari *username target*. Hal ini disebut juga sebagai permintaan *directed presence*. *User* akan mendapatkan *directed presence* dari *username target* dan *username target* akan mendapatkan *directed presence* dari *user*.
28. Menangani data yang tidak valid saat terjadi permintaan *directed presence* yaitu memasukkan *username target* yang tidak terdapat di dalam sistem.
29. Melayani *user* yang mengirimkan *presence* dengan tipe "*unavailable*" sebagai *user* yang akan melakukan *logout* dan mengubah status *online*-nya menjadi *offline*.

30. Memberikan pembaruan status *online* kepada semua *roster item user* yang memiliki nilai *subscription* adalah "*from*" atau "*both*".
31. Mengubah status *online* menjadi *offline* ketika *user* tiba-tiba *error* atau terputus dan akan melakukan fungsionalitas ke 29.
32. Menerima segala *packet* pesan yang masuk dari komponen *tracker*, *manager*, dan *relay*.

Komponen *manager* dapat dijalankan dengan memanggil file utamanya yaitu main.py yang terdapat dalam folder *manager*. Adapun diagram dari komponen *manager* secara lengkap bisa dilihat dibawah.



Gambar 4. 3 Diagram Lengkap Komponen Manager

4.4 Implementasi Komponen *Relay*

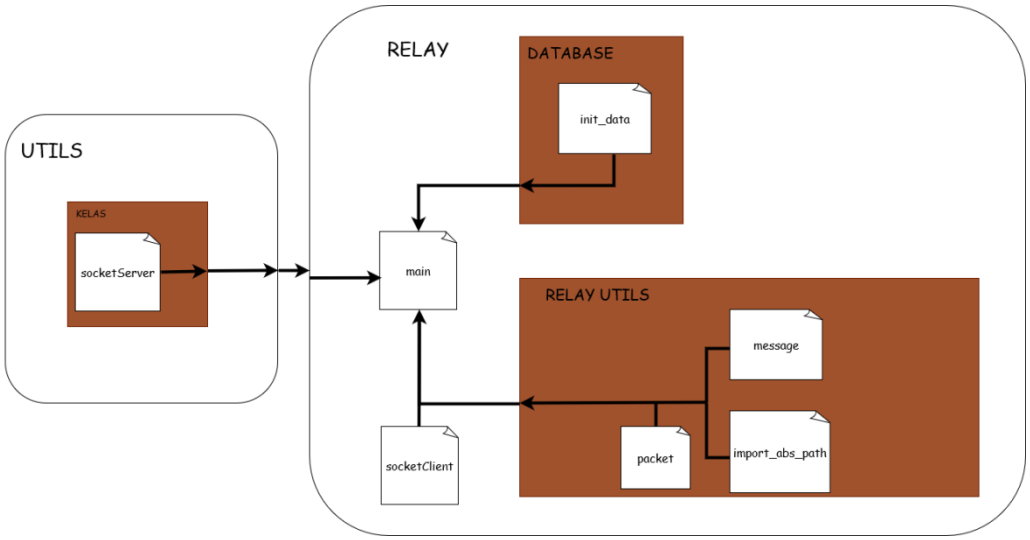
Komponen *relay* berfungsi sebagai jembatan pengiriman data pesan dari satu *user* ke *user* lainnya. Hal ini akan memiliki dua kondisi, yaitu *user* mengirimkan pesan ke *user* lain yang berada pada *relay* yang sama, atau *user* mengirimkan pesan ke *user* lain yang berada pada *relay* yang berbeda. Atas kondisi ini maka *relay* juga akan mengurus koneksi antar sesama *relay*. Adapun metode atau fungsionalitas yang diimplementasikan pada *relay* adalah sebagai berikut:

1. Melakukan koneksi *socket* TCP kepada komponen *tracker*. Mengirimkan *packet* kepada TCP dengan nilai kunci *type* adalah *relay*.

2. Melakukan koneksi *socket* TCP kepada komponen *manager*. Mengirimkan *packet* kepada TCP dengan nilai kunci *type* adalah *relay*.
3. Melakukan permintaan secara berkala mengenai data komponen *manager*, ketika belum terdapat komponen *manager* di dalam sistem kepada *tracker*.
4. Konfigurasi awal, ketika *relay* baru tergabung ke dalam sistem. Hal yang dilakukan untuk konfigurasi awal ini adalah:
    - Melakukan koneksi *socket* TCP kepada komponen *relay* yang sudah lebih dahulu bergabung dalam sistem. Mengirimkan *packet* kepada TCP dengan nilai kunci *type* adalah *relay*.
    - Menyimpan koneksi *socket* TCP dan menyimpan daftar *username* yang terhubung dengan *relay-relay* tersebut.
5. Menerima setiap koneksi *socket* TCP yang menuju kepada dirinya.
6. Mencatat username untuk *user* yang terkoneksi kepada dirinya.
7. Ketika *user* berhasil terkoneksi dengan dirinya maka, *relay* mengirimkan pesan “ncir” kepada komponen manager.
8. Ketika *user* berhasil terkoneksi dengan dirinya maka, *relay* mengirimkan informasi tentang *user* tersebut, yang berhasil terkoneksi dengan dirinya, kepada komponen *relay* lain yang terhubung dengan dirinya. Dikirimkan dengan menyertakan nilai *message* “*new user*”.
9. Menghapus *username* untuk *user* yang telah melakukan *logout* dari sistem.
10. Ketika *user* berhasil terputus koneksinya dengan *relay* maka, *relay* mengirimkan pesan “ecir” kepada komponen *manager*.
11. Ketika *user* berhasil terputus koneksinya dengan *relay* maka, *relay* mengirimkan informasi tentang *user* tersebut, yang berhasil terputus koneksinya dengan dirinya, kepada komponen *relay* lain yang terhubung dengan dirinya. Dikirimkan dengan menyertakan nilai *message* “*end user*”.
12. Menangani *user* yang tiba-tiba *error* atau terputus. Kemudian *relay* melakukan fungsionalitas 10 dan 11.
13. Mampu menerima *packet stanza message* dari *user*.
14. Memeriksa data *stanza message* untuk memastikan bahwa *user* yang dituju berada pada *relay* yang sama atau berbeda.

15. Mengatasi data *stanza message* jika *user* yang dituju tidak berada pada *relay* manapun maka, *relay* akan memberikan *packet stanza message error* kepada *user* pengirim dengan isi pesan adalah "*User not found or User not online*".
16. Mengirimkan *stanza message* kepada *user* yang dituju.
17. Mencatat *username relay* untuk *relay* yang terkoneksi dengan dirinya.
18. Memberikan *username* diri sendiri dan memberikan *username user* yang terkoneksi pada dirinya kepada *relay* lain yang baru melakukan koneksi kepadanya.
19. Mampu menerima *stanza message* dari *relay* lainnya.
20. Mengatasi data *stanza message* jika *user* yang dituju sudah tidak terkoneksi dengannnya, dengan cara mengirimkan *packet* dengan nilai dari *error* adalah 1 dan nilai *activity* adalah "*message from another relay*".
21. Mengelola aktivitas untuk menyimpan suatu *username* baru dalam data *user* yang terkoneksi pada *relay* lain, jika menerima *message* dengan nilai "*new user*".
22. Mengelola aktivitas untuk menghapus suatu *username* dalam data *user* yang terkoneksi pada *relay* lain, jika menerima *message* dengan nilai "*end user*".
23. Mampu mengirimkan *packet* dengan nilai dari *error* adalah 1 dan nilai *activity* adalah "*message from another relay*" dari *relay* lain kepada *user* pengirim. *Packet* yang akan dikirimkan kepada *user* pengirim adalah *packet stanza message error* dengan isi pesan adalah "*User not found or User not online*".
24. Mengatasi *relay* yang tiba-tiba terputus atau *error*.
25. Menghapus data koneksi dengan *relay* yang tiba-tiba terputus atau *error* dan menghapus koneksi seluruh *user* yang terkoneksi kepada *relay* tersebut.
26. Menerima segala *packet* pesan yang masuk dari komponen *tracker*, *manager*, dan *user*.

Komponen *relay* dapat dijalankan dengan memanggil file utamanya yaitu main.py yang terdapat dalam folder *relay*. Adapun diagram dari komponen *relay* secara lengkap bisa dilihat dibawah.



Gambar 4. 4 Diagram Lengkap Komponen Relay

4.5 Implementasi Komponen Penguji atau *User*

Implementasi komponen *user* ditujukan untuk menguji setiap fungsionalitas yang terdapat dalam daftar setiap komponen yang berhubungan dengan *user*. Metode atau fungsionalitas yang terdapat di dalam *user* untuk memenuhi perannya adalah sebagai berikut:

1. Melakukan koneksi *socket* TCP kepada komponen *tracker*. Mengirimkan *packet* kepada TCP dengan nilai kunci *type* adalah *client*.
2. Melakukan koneksi *socket* TCP kepada komponen *manager*. Mengirimkan *packet* kepada TCP dengan nilai kunci *type* adalah *client*.
3. Mampu mengirimkan *packet* untuk registrasi kepada komponen *manager*.
4. Mampu mengirimkan *packet* untuk *login* kepada komponen *manager*.
5. Melakukan koneksi *socket* TCP kepada komponen *relay*. Mengirimkan *packet* kepada TCP dengan nilai kunci *username* adalah *username* dari *user*.
6. Melakukan konfigurasi awal dengan mengirimkan permintaan *roster*-nya dan inisialisasi *presence* kepada komponen *manager*.
7. Melakukan permintaan *roster* kepada komponen *manager*.

8. Melakukan permintaan menambahkan *roster item* ke dalam *roster*-nya kepada komponen *manager*.
9. Melakukan permintaan mengubah *name* dari *roster item* kepada komponen *manager*.
10. Melakukan permintaan menghapus *roster item* dari *roster*-nya kepada komponen *manager*.
11. Melakukan permintaan *directed presence* terhadap suatu *username* tertentu kepada komponen *manager*.
12. Melakukan permintaan memperbarui bio dirinya kepada komponen *manager*.
13. Melakukan permintaan *logout* atau *presence* bertipe "*unavailable*" kepada komponen *manager*.
14. Melakukan pengiriman pesan kepada *username* tertentu kepada komponen *relay*.
15. Menerima segala *packet* pesan yang masuk dari komponen *tracker*, *manager*, dan *relay*.

4.6 Pengujian Proses Komunikasi

Proses komunikasi antar komponen dimulai dari komponen saling terkoneksi, pertukaran data yang terjadi dalam koneksi mereka, dan cara menangani apabila koneksi tersebut terputus. Berdasarkan lampiran hasil pengujian fungsionalitas dapat dilihat bahwa proses komunikasi telah berjalan dengan baik dan benar.

4.7 Pengujian Konsistensi Data

Saat proses komunikasi dilakukan akan ada pertukaran data yang terjadi, untuk setiap data yang dikirimkan akan dilihat apakah ada data yang tidak terkirim. Hal ini dapat dinilai dari lampiran hasil pengujian fungsionalitas yang menunjukan aktivitas pengujian fungsional. Hasil dari pengujian konsistensi data ini menunjukkan bahwa tidak ada data yang tidak terkirim selama terjadi proses komunikasi.

4.8 Pengujian Fungsionalitas

Fungsionalitas di dalam sistem dibagi menjadi beberapa kategori besar, diantaranya adalah:

1. *Tracker*
    - Pengelolaan detail komponen.
2. *Manager*
    - Melakukan koneksi kepada komponen *tracker*.
    - Pengelolaan detail informasi *relay* dan *user*.
    - Autentikasi.
    - Pengelolaan *roster*.
    - Pengelolaan *presence*.
3. *Relay*
    - Melakukan koneksi kepada komponen lainnya.
    - Pengelolaan informasi *user* pada dirinya dan *relay* lain.
    - Menerima pesan dari *user* dan *relay*.
    - Mengirimkan pesan kepada *user* dan *relay*.

Berdasarkan lampiran hasil pengujian fungsionalitas dapat dilihat bahwa semua fungsionalitas dapat digunakan semua dan sesuai dengan aktivitas fungsionalitas tersebut.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan implementasi dan pengujian untuk middleware komunikasi dengan protokol XMPP dan *socket programming* ini, dapat disimpulkan bahwa:

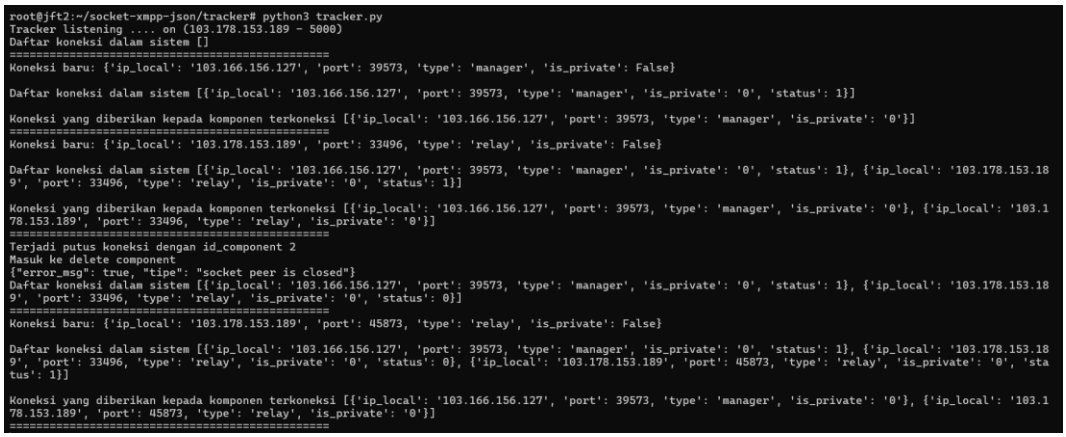
1. *Socket programming* hanyalah saluran untuk pertukaran data, pertukaran data dapat dilakukan seperti apa saja. Jika itu terjadi maka, sistem tidak akan bisa digunakan karena tidak memiliki format data yang jelas. Dibutuhkan sebuah protokol pertukaran data.
2. Pemahaman tentang protokol yang digunakan sebagai landasan format pertukaran data harus dikuasai dengan benar. Pemahaman ini akan membantu jika pengembang ingin melakukan inovasi pada protokol tersebut.
3. Pembuatan middleware komunikasi dengan XMPP dan *socket programming* bertujuan untuk menghubungkan user menggunakan *socket* TCP, dan melayani user untuk mengelola *roster* dan juga *presence user*.

5.2 Saran

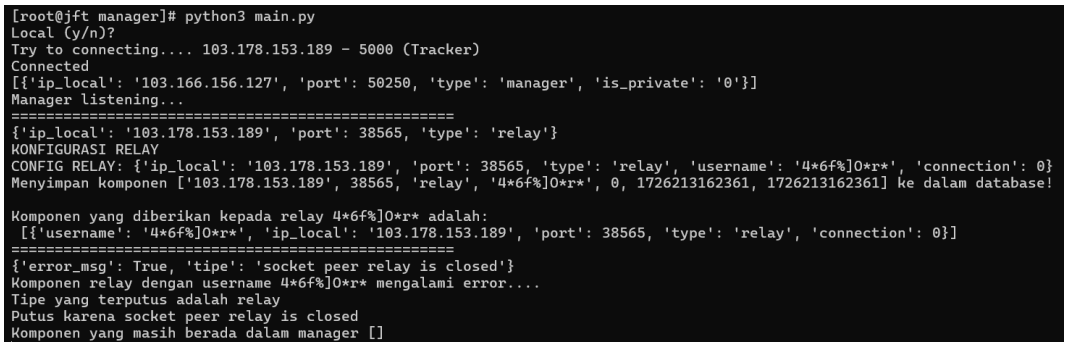
Adapun saran untuk penelitian selanjutnya adalah:

1. Keamanan jaringan pada penelitian ini belum diimplementasikan sama sekali, maka, ada baiknya jika memperbaiki permasalahan keamanan jaringan yang ada. Mengamankan jaringan dengan algoritma enkripsi ECC dan mengamankan pengiriman data yang dilakukan menggunakan enkripsi AES.
2. Pada penelitian ini, fitur konfirmasi penambahan kontak masih otomatis dilakukan oleh server. Bisa menambahkan spesifikasi fitur ini.
3. Fitur untuk mengirimkan gambar atau pesan suara juga bisa ditambahkan.
4. XMPP memiliki banyak sekali catatan fitur, gunakan yang ingin diimplementasikan.

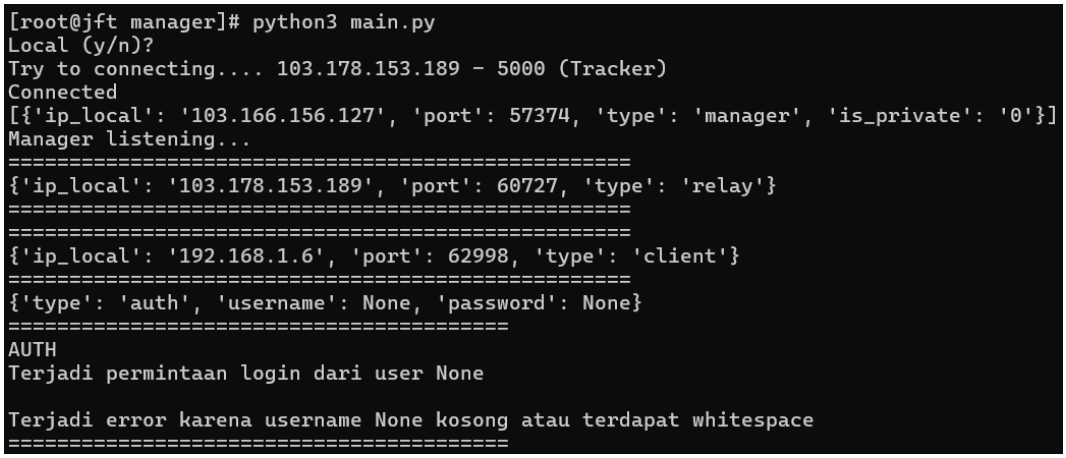
LAMPIRAN HASIL PENGUJIAN FUNGSIONALITAS



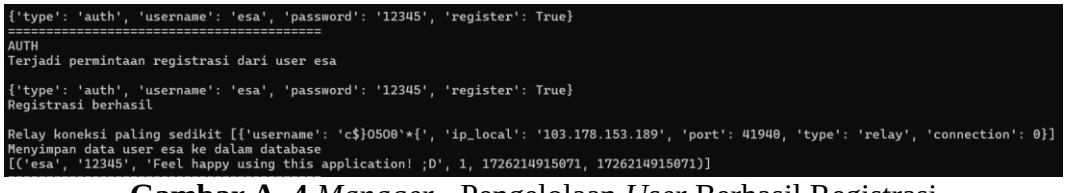
Gambar A. 1 *Tracker* - Pengeloaan Detail Komponen



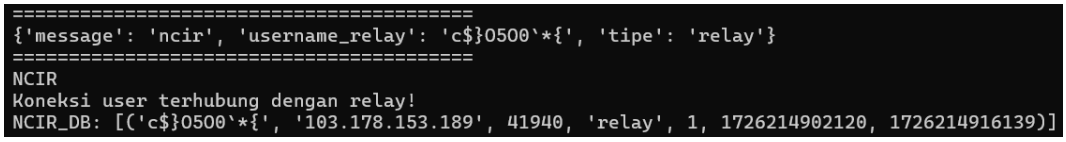
Gambar A. 2 *Manager* - Pengelolaan Detail Informasi *Relay*



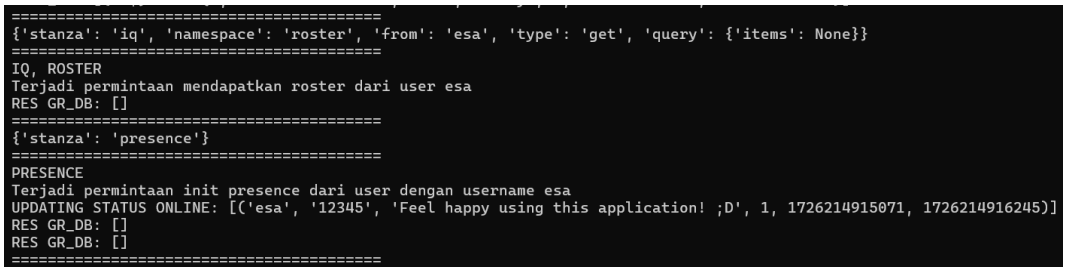
Gambar A. 3 *Manager* - Pengelolaan *User* Data Kosong atau *Whitespace*



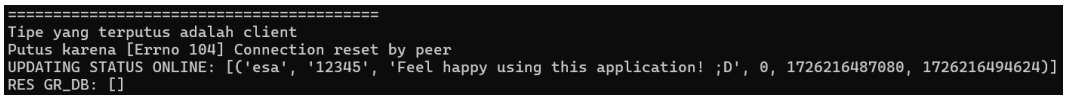
Gambar A. 4 *Manager* - Pengelolaan *User* Berhasil Registrasi



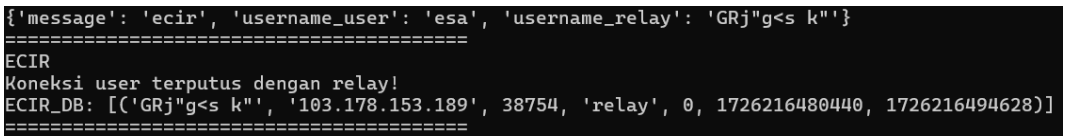
Gambar A. 5 *Manager* - Pengelolaan *Relay Message* NCIR



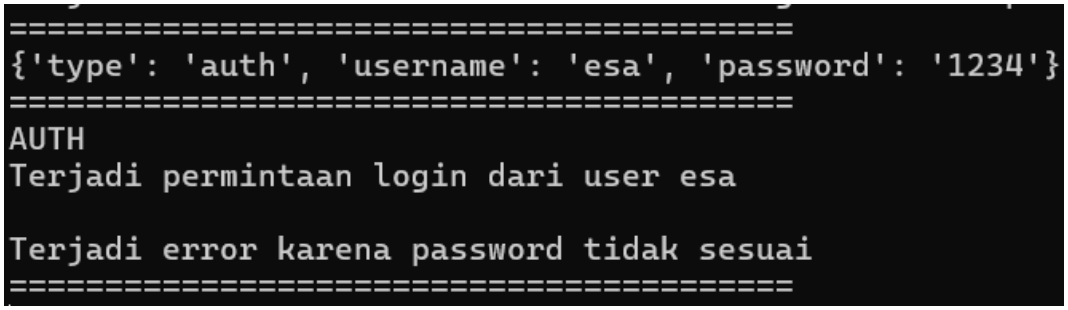
Gambar A. 6 *Manager* - Pengelolaan *User Get Roster* dan Inisialisasi *Presence*



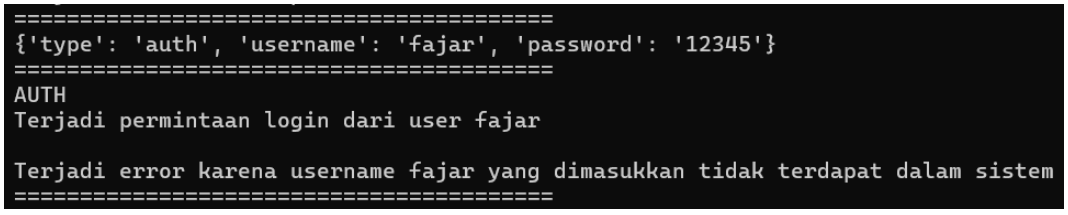
Gambar A. 7 *Manager* - Penanganan *User* Koneksi Terputus



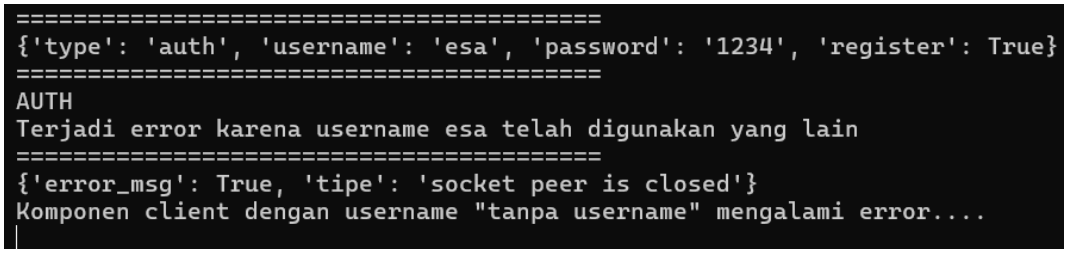
Gambar A. 8 *Manager* - Pengelolaan *Relay Message* ECIR



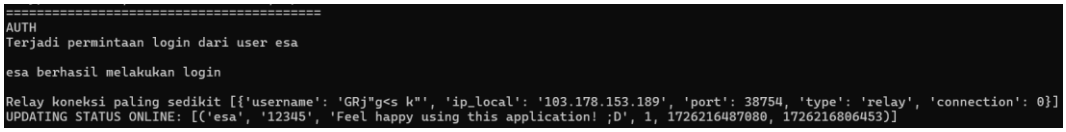
Gambar A. 9 *Manager* - Pengelolaan *User Password* Salah



Gambar A. 10 *Manager* - Pengelolaan *User* Memasukkan *Username* Salah

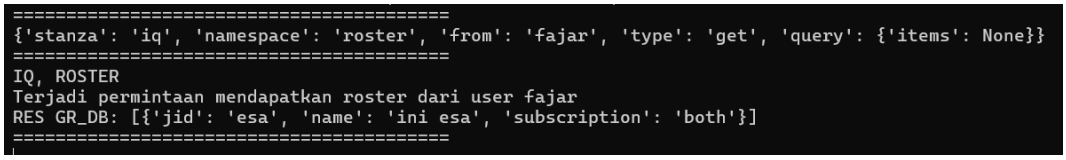


Gambar A. 11 *Manager* - Pengelolaan *User* Registrasi *Username Used*



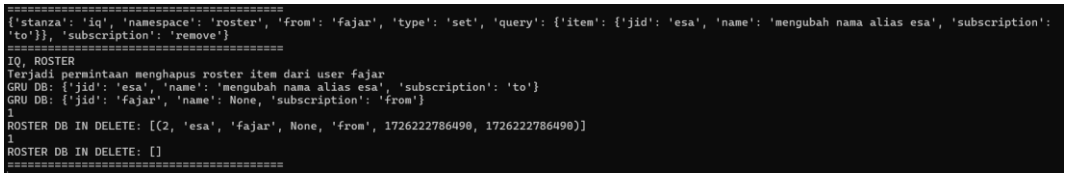
Gambar A. 12 *Manager* - Pengelolaan *User* Berhasil Login

Gambar A. 13 *Manager* - Pengelolaan *User* Menambahkan *Roster Item*

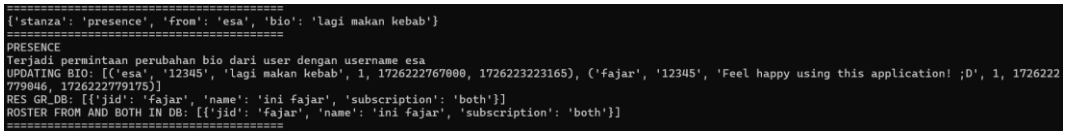


Gambar A. 14 *Manager* - Pengelolaan *User* Meminta *User's Item*

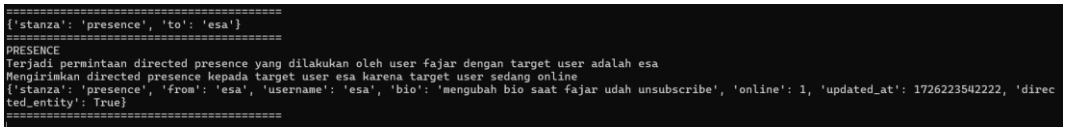
Gambar A. 15 *Manager* - Pengelolaan *User* Mengubah *Name Roster Item*



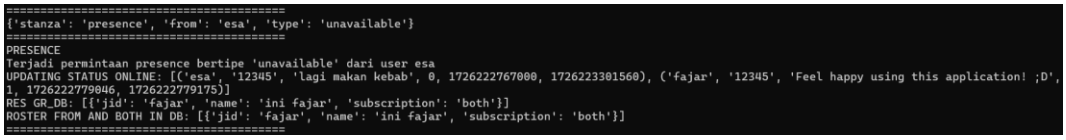
Gambar A. 16 *Manager* - Pengelolaan *User* Menghapus *Roster Item*



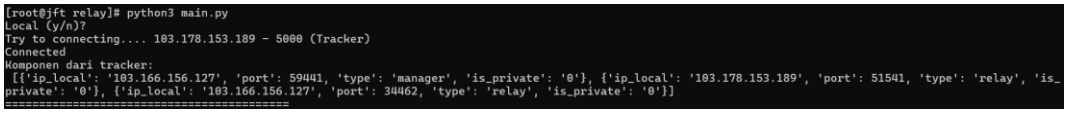
Gambar A. 17 *Manager* - Pengelolaan Mengirim Pembaruan Bio *User* Kepada *Roster Item*



Gambar A. 18 *Manager* - Pengelolaan *User* Meminta *Directed Presence*

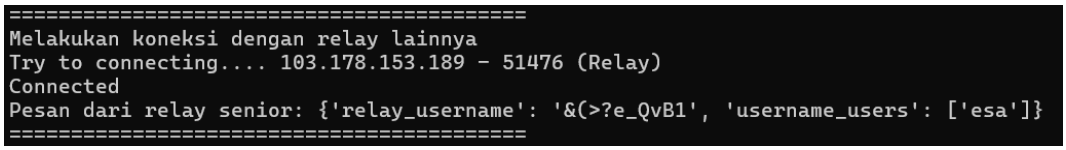


Gambar A. 19 *Manager* - Pengelolaan *User* Mengirim *Presence Unavailable*



Gambar A. 20 *Relay* - Melakukan Koneksi dengan *Tracker*

Gambar A. 21 *Relay* - Melakukan Koneksi dengan *Manager*

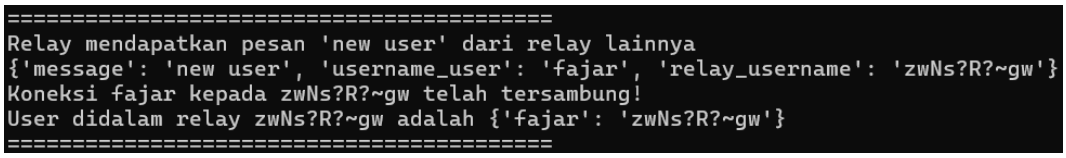


Gambar A. 22 *Relay* - Melakukan Koneksi dengan *Relay* Senior

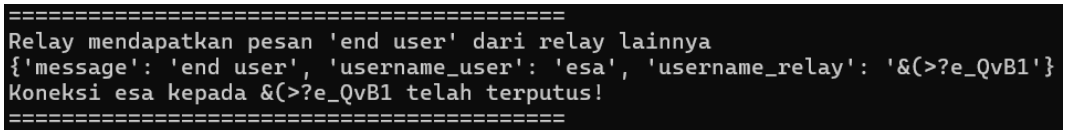
Gambar A. 23 *Relay* - Konfigurasi Saat Terkoneksi dengan *Relay* Senior yang Memiliki *User* Terkoneksi

Gambar A. 24 *Relay* - Mendapatkan Koneksi Baru dari *Relay* Junior

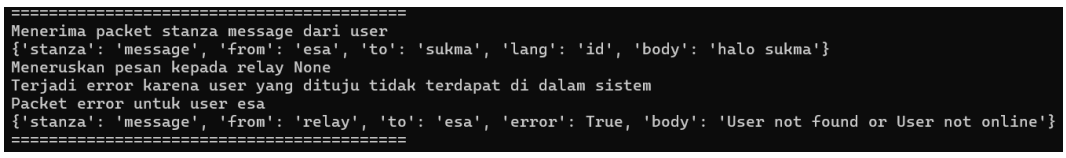
Gambar A. 25 *Relay* - Mendapatkan Koneksi Baru dari *User*



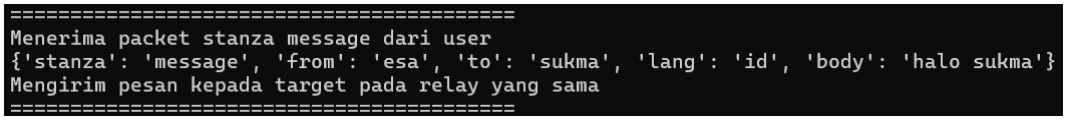
Gambar A. 26 *Relay* - Mengelola *New User* dari *Relay* Lainnya



Gambar A. 27 *Relay* - Mengelola *End User* dari *Relay* Lainnya

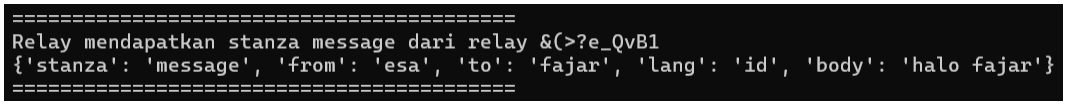


Gambar A. 28 *Relay* - Menangani *Stanza Mesage* Untuk *Target* Tidak Terdaftar

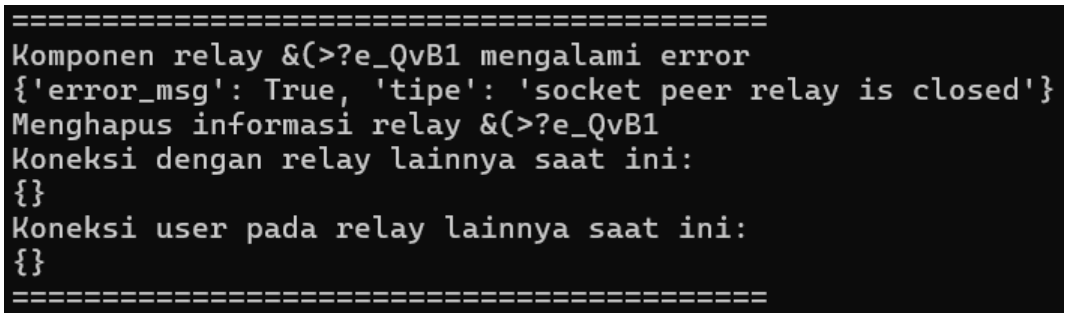


Gambar A. 29 *Relay* - Menangani *Stanza Message* Untuk *Target User* Pada *Relay* yang Sama

Gambar A. 30 *Relay* - Mengelola *Stanza Message* Untuk *Relay* Lainnya



Gambar A. 31 *Relay* - Meneruskan *Stanza Message Relay* Lain Kepada *User Target*



Gambar A. 32 *Relay* - Menangani Jika Koneksi dengan *Relay* Lain *Error*

LAMPIRAN KODE PROGRAM

A. Tracker

A.1 Menerima Koneksi Socket TCP dan Mengelola Komponen

Input: {ip_local: ip_address, port: port_num, type: tipe, is_private: Boolean}.

Output: selalu menerima koneksi baru.

```
while True:
    connection, address = server.accept()
    messages = get_message(connection)
    id_component = None
    for message in messages:
        message = json.loads(message)
        print(f"Koneksi baru: {message}\r\n")
        ip_is_private = message.get("is_private")
        tipe = message.get('type').lower()
        if(not ip_is_private or tipe == 'manager' or tipe == 'relay'):
            data = list(message.values())
            id_component = save_component_to_db(data, get_timestamp)
            # components.append(message)
        print(f"Daftar koneksi dalam sistem {get_all()}\r\n")
        feedback = get_components()
        print(f"Koneksi yang diberikan kepada komponen terkoneksi {feedback}")
        feedback = json.dumps(feedback)
        connection.send(feedback.encode())
        threading.Thread(target=handleComponent, args=(connection, message, id_component), daemon=True).start()
        print("================================================")
```

Gambar B. 1 Tracker - Menerima Koneksi Masuk dan Membedakan Koneksi Komponen

A.2 Mengelola Packet “Get Components”

Input: {message: get components}.

Output: daftar komponen yang terkoneksi.

```
def handleComponent(communicate:socket.socket, msg, id_component):
    error = False
    while not error:
        try:
            messages = get_message(communicate)
            for msg in messages:
                message = json.loads(msg)
                if(message.get("message") and message.get("message").lower() == "get components"):
                    send_message(communicate, get_components())
```

Gambar B. 2 Tracker - Mengelola Packet Get Components

A.3 Mengelola Komponen Yang Terputus

Input: error dari socket yang menghubungkan.

Output: pembaruan data untuk komponen yang terputus.

```
def delete_component(message, id_component):
    # print("Masuk ke delete component")
    print(message)
    delete_component_by_id(id_component, get_timestamp)
    print(f"Daftar koneksi dalam sistem {get_all()}")
    print("================================================")
```

Gambar B. 3 Tracker - Mengelola Komponen Koneksi Terputus

A.4 Menerima Packet Yang Masuk

Input: packet dari komponen yang mengirim.

Output: akan dikelola sesuai packet yang diterima.

```
def get_message(communicate):
    global msg
    message = ""
    message = communicate.recv(65536)
    message = message.decode()
    items = message.split('\x80\x81\x82')
    if(len(items) == 1 and not items[0]):
        yield '{"error_msg": true, "tipe": "socket peer is closed"}'
    else:
        for i in items:
            if(msg):
                i = msg + i
                msg = ""
                yield i
            elif(verify(i)):
                    yield i
            else:
                msg = i
```

Gambar B. 4 Tracker - Menerima Packet Yang Masuk

B. Manager

B.1 Melakukan Koneksi Kepada Komponen Tracker

Input: {ip_local: ip, port: port_num, type: tipe, is_private: Boolean}.

Output: akan terkoneksi dengan tracker.

```
def connect_to_tracker():
    ask = input("Local (y/n)?")
    if(ask.lower() == 'y'):
        client_tracker = SocketClient(None, 5000, tipe="Tracker")
    else:
        client_tracker = SocketClient(IP_TRACKER, 5000, tipe="Tracker")
    # ct = Client Tracker
    ct = client_tracker.socket
    my_ip = client_tracker.localAddress
    ip_is_private = ipaddress.ip_address(my_ip[0]).is_private

    objek = {
        "ip_local": my_ip[0],
        "port": my_ip[1],
        "type": "manager",
        "is_private": ip_is_private
    }

    send_message(ct, objek)
    message = get_message_tracker(ct)
    print(message)
    return ct, my_ip
```

Gambar B. 5 Manager - Melakukan Koneksi ke Tracker

B.2 Menerima Setiap Koneksi dan Mengelola Komponen Yang Terkoneksi

Input: {ip_local: ip, port: port_num, type: tipe}.

Output: koneksi yang baru terhubung dan data yang dikirimkannya akan dikelola sesuai tipe yang diberikan.

```
# Fungsionalitas untuk menerima setiap koneksi yang baru terhubung dengan manager
while True:
    try:
        connection, address = manager.accept()
        # Fungsionalitas penerimaan pesan
        messages = get_message(connection)
        for msg in messages:
            message = json.loads(msg)
            print("================================================")
            print(message)
            username_relay = None
            # Metode yang dilakukan jika koneksi yang baru bertipe relay
            if(message.get("type").lower() == "relay"):
                print("KONFIGURASI RELAY")
                # print(f'RELAY MESSAGE: {msg}')
                username_relay = random_username_for_relay(c_db, message)
                # username_relay = random_username_for_relay(c_db)
                ### jika ip address dan port yang sama sudah terdaftar maka tidak boleh menjadi relay
                if(not username_relay):
                    obj_error = {
                        "error_msg": True,
                        "msg": "IP address dan port sudah terdaftar",
                        "error-type": "bad request"
                    }
                    send_message(connection, obj_error)
                    continue
                # Konfigurasi awal relay
                config_new_relay(message, username_relay)
                # Mengambil semua data komponen yang terkoneksi dengan manager
                c = get_all_components(c_db.get_all())
                print(f"Komponen yang diberikan kepada relay {username_relay} adalah:\r\n {c}")
                # Membuat packet yang akan dikirimkan kepada relay yang baru terkoneksi
                objek = {
                    "components": c,
                    "username": username_relay
                }
                # Mengirimkan semua data komponen kepada relay yang baru terkoneksi
                # print(f"KONFIGURASI RELAY {objek}")
                send_message(connection, objek)
            # Mendapatkan tipe dari komponen yang terhubung
            tipe = message.get('type')
            # Membuat layanan eksklusif untuk menerima packet pesan dari komponen terkoneksi
            # lockThread = threading.Lock()
            threading.Thread(target=handle_component, args=(connection, tipe, username_relay), daemon=True)
            print("================================================")
```

Gambar B. 6 Manager – Menerima Setiap Koneksi dan Mengelola Komponen Yang Melakukan Koneksi

B.3 Mengelola Packet NCIR dan ECIR

Input: {message: NCIR/ECIR, username_relay: username_relay, type: relay}.

Output: memperbarui total koneksi dari suatu relay.

```
# new connection in relay
elif(message.get("message") and message.get("message").lower() == "ncir" and tipe == "relay"):
    username_relay = message.get("username_relay")
    update_total_connection(username_relay, get_timestamp, 'NCIR')

# end connection in relay
elif(message.get("message") and message.get("message").lower() == "ecir" and tipe == "relay"):
    username_relay = message.get("username_relay")
    update_total_connection(username_relay, get_timestamp, 'ECIR')
```

Gambar B. 7 Manager - Mengelola Packet NCIR dan ECIR

B.4 Mengelola Relay Yang Terputus Koneksi

Input: error dari socket yang menghubungkan.

Output: menghapus dari daftar koneksi.

```
def lost_connection(reason, tipe, username_relay, username):
    print(f"Tipe yang terputus adalah {tipe}")
    print(f"Putus karena {reason}")
    if(tipe and tipe.lower() == "relay"):
        # Menghapus komponen relay dari database
        delete_components_db_by_id(c_db, tuple_condition=username_relay)
    if(username):
        # Menghapus koneksi dari user
        del socket_user[username]
        # Konfigurasi user yang logout atau mengalami error
        timestamp = get_timestamp()
        logout(socket_user, username, timestamp)
```

Gambar B. 8 Manager - Mengelola Relay Yang Terputus Koneksi

B.5 Mengelola User Yang Ingin Registrasi atau Login

Input: {type: auth, username: username, password: password, register: Boolean}.

```
# Ketika mau login dan belum online, kalo udah online minta get_relay_less_connection aja
if(tipe == "client" and message.get("type") == "auth" and not get_user_online(username)):
    result = handle_auth(message, communicate, u_db, get_relay_with_less_connection_db)
    if(not result):
        username = None
    # continue
```

Gambar B. 9 Manager - Mengelola User Yang Ingin Registrasi atau Login

B.6 Mengelola User Yang Memasukkan Data Tidak Benar (Registrasi)

Input: seperti B.5.

Output: packet error.

```
def handle_auth(message, communicate, user_db=None, f=None):
    username = message.get("username")
    register = message.get("register")
    user = get_user_by_username(username)
    if(register):
        # print(f'Terjadi permintaan registrasi dari user {username}\r\n')
        if(user):
            print(f"Terjadi error karena username {username} telah digunakan yang lain")
            # Pembuatan packet untuk pemberitahuan kepada user bahwa registasi gagal
            objek = {"error": True, "msg": "Username used", "code": 409}
            print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
            send_message(communicate, objek)
            return False
        elif(not username or verify_whitespace(username)):
            print(f"Terjadi error karena username {username} kosong atau terdapat whitespace")
            # Pembuatan packet untuk pemberitahuan kepada user bahwa registasi gagal
            objek = {"error": True, "msg": "Bad request", "code": 400}
            print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
            send_message(communicate, objek)
            return False
        password = message.get("password")
        if(not password or verify_whitespace(password)):
            print(f"Terjadi error karena password kosong atau terdapat whitespace")
            # Pembuatan packet untuk pemberitahuan kepada user bahwa registasi gagal
            objek = {"error": True, "msg": "Bad request", "code": 400}
            print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
            send_message(communicate, objek)
            return False
```

Gambar B. 10 Manager - Mengelola User Yang Memasukkan Data Tidak Benar

B.7 Berhasil Registrasi

Input: seperti B.5.

Output: packet sukses, memberi relay koneksi paling sedikit dan menyimpan data user ke dalam database.

```
else:
    print(f'Terjadi permintaan registrasi dari user {username}\r\n')
    print(message)
    print("Registrasi berhasil\r\n")
    # Mendapatkan relay paling sedikit yang akan diberikan kepada user
    relay_for_user = get_relay_with_less_connection_db()
    # Pembuatan packet untuk pemberitahuan kepada user bahwa registasi berhasil
    objek = {"error": False, "msg": "Account created", "code": 201, "component": relay_for_user}
    print(f"Packet success yang dikirimkan kepada user {username}\r\n{objek}")
    password = message.get("password")
    timestamp = get_timestamp()
    # Menyimpan user ke dalam database
    helper_registering_user(username, password, timestamp)
    send_message(communicate, objek)
    return True
```

Gambar B. 11 Manager - Berhasil Registrasi

B.8 Menyimpan Data User Berhasil Registrasi

Input: seperti B.5.

Output: data user disimpan dalam database.

```
def helper_registering_user(username, password, timestamp):
    print(f"Menyimpan data user {username} ke dalam database")
    objek = {
        "username": username,
        "password": password,
        "bio": "Feel happy using this application! ;D",
        "online": 1,
        "created_at": timestamp,
        "updated_at": timestamp
    }
    save_user_to_db(list(objek.values()))
```

Gambar B. 12 Manager - Menyimpan Data User Berhasil Registrasi

B.9 Mengelola User Yang Memasukkan Data Tidak Benar (Login)

Input: seperti B.5

Output: packet error.

```
print(f"Terjadi permintaan login dari user {username}\r\n")
if(not username or verify_whitespace(username)):
    print(f"Terjadi error karena username {username} kosong atau terdapat whitespace")
    # Pembuatan packet untuk pemberitahuan kepada user bahwa login gagal
    objek = {"error": True, "msg": "Bad request", "code": 400}
    print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
    send_message(communicate, objek)
    return False
password = message.get("password")
if(not password or verify_whitespace(password)):
    print(f"Terjadi error karena password kosong atau terdapat whitespace")
    # Pembuatan packet untuk pemberitahuan kepada user bahwa login gagal
    objek = {"error": True, "msg": "Bad request", "code": 400}
    print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
    send_message(communicate, objek)
    return False
if(not user):
    print(f"Terjadi error karena username {username} yang dimasukkan tidak terdapat dalam sistem")
    # Pembuatan packet untuk pemberitahuan kepada user bahwa login gagal
    objek = {"error": True, "msg": "User not found", "code": 404}
    print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
    send_message(communicate, objek)
    return False
```

Gambar B. 13 Manager - Mengelola User Memasukkan Data Tidak Benar (Login)

B.10 Mengelola User Login Menggunakan Akun Status Online

Input: seperti B.5.

Output: packet error.

```
if(user[3]):
    if(not password == password_db):
        print(f"Terjadi error karena password tidak sesuai")
        # Pembuatan packet untuk pemberitahuan kepada user bahwa login gagal
        objek = {"error": True, "msg": "Bad request", "code": 400}
        print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
        send_message(communicate, objek)
        return False
    print(f"Terjadi error karena user mencoba login ke {username} yang sedang online")
    # Pembuatan packet untuk pemberitahuan kepada user bahwa login gagal
    objek = {"error": True, "msg": "Already logged in", "code": 409}
    print(f"Packet error yang dikirimkan kepada user {username}\r\n{objek}")
    send_message(communicate, objek)
    return False
```

Gambar B. 14 Manager - Mengelola User Login Menggunakan Akun Status Online

B.11 User Berhasil Login

Input: seperti B.5.

Output: packet sukses, memberikan relay dengan koneksi paling sedikit, dan memperbarui status user dari offline menjadi online.

```
if(password == password_db):
    print(f"{username} berhasil melakukan login\r\n")
    # Mendapatkan relay paling sedikit yang akan diberikan kepada user
    relay_for_user = get_relay_with_less_connection_db()
    # Pembuatan packet untuk pemberitahuan kepada user bahwa login berhasil
    objek = {"error": False, "msg": "Login successfull", "code": 200, "component": relay_for_user}
    # Memperbarui status dari user dari offline ke online
    print(f"Packet success yang dikirimkan kepada user {username}\r\n{objek}")
    update_status_online(["online", "updated_at"], (1, get_timestamp(),), ['user_id'], (username,))
    send_message(communicate, objek)
    return True
```

Gambar B. 15 Manager - User Berhasil Login

B.12 Mengelola Packet Type “get_relay_less_connection”

Input: {type: get_relay_less_connection}.

Output: memberikan relay dengan koneksi paling sedikit.

```
# Ketika ada relay yang tiba-tiba error dan user mau relay dengan koneksi paling rendah terbaru
elif(tipe == "client" and message.get("type") == "get_relay_less_connection" and get_user_online(username)):
    relay_for_user = get_relay_with_less_connection_db()
    objek = {"error": False, "msg": "Relay Less Connection", "code": 200, "component": relay_for_user}
    send_message(communicate, objek)
```

Gambar B. 16 Manager - Mengelola Packet Type "get_relay_less_connection"

B.13 Melayani User Yang Ingin Mendapatkan Roster

Input: {stanza: iq, namespace: roster, from: user_init, type: get, query: {items: null}}.

Output: {stanza: iq, namespace: roster, type: result, query: {items: roster's item}, to: user_init}.

```
# Mengelola yang berkaitan dengan roster
elif(message.get("stanza") and message.get("stanza").lower() == "iq" and message.get("namespace") and message.get("namespace").lower() == "roster"):
    print("========================================")
    print("IQ, ROSTER")
    # Mengelola user yang meminta rosternya
    if(message.get("type") and message.get("type").lower()  == "get"):
        print(f"Terjadi permintaan mendapatkan roster dari user {username}")
        objek = get_rosters(username, message)
        send_message(communicate, objek)
```

Gambar B. 17 Manager - Melayani User Yang Ingin Mendapatkan Roster

B.14 Melayani User Yang Ingin Menambahkan Roster Item

Input: {stanza: iq, namespace: roster, from: user_init, type: set, query: {item: {jid: jid_target, name: target_alias, subscription: to}}}.

Output:

1. {stanza: presence, from: user_target, to: user_init, type: subscribed, item: {jid: jid_target, name: target_alias, subscription: to/both}},
2. {stanza: presence, from: user_target, username: user_target, bio: target_bio, online: target_status, updated_at: last_updated_target, to: user_init},
3. {stanza: iq, namespace: roster, type: result, to: user_init}.

```
# Mengelola roster yang ingin menambahkan atau memperbarui roster item
elif(message.get("type") and message.get("type").lower()  == "set" and not message.get("subscription")):
    print(f"Terjadi permintaan untuk menambahkan atau memperbarui roster item dari user {username}")
    jid_target = message.get('query').get('item').get('jid')
    objek = set_roster(username, message)
    if(objek['type'] != "error"):
        # Membuat presence bertipe "subscribed" untuk user pengirim
        packet_presence_subscribed = get_packet_subscribed_for_init_entity(username, jid_target)
        # Mengambil data roster item di dalam roster terbaru untuk user pengirim dari database
        item = get_roster_user(username, jid_target)
        # Melengkapi packet presence bertipe "subscribed" dengan roster item terbaru
        packet_presence_subscribed["item"] = item
        # Mengirimkan packet lengkap presence bertipe "subscribed" kepada user pengirim
        send_message(communicate, packet_presence_subscribed)
        if(item.get("subscription") == "both" and get_user_online(jid_target)):
            packet_presence_subscribed_for_roster_item = get_packet_subscribed_for_init_entity(username, jid_target)
            item_for_roster_item = get_roster_user(jid_target, username)
            packet_presence_subscribed_for_roster_item["item"] = item_for_roster_item
            send_message(socket_user[jid_target], packet_presence_subscribed_for_roster_item)
        # Mengambil data presence terbaru dari roster item yang dikirimkan oleh user pengirim
        subscribed_entity_presence = get_presence_by_jid(jid_target)
        subscribed_entity_presence['to'] = username
        # Mengirimkan packet presence terbaru roster item kepada user pengirim
        send_message(communicate, subscribed_entity_presence)
    else:
        print(f"Terjadi error saat user {username} melakukan permintaan menambahkan atau mengubah roster item")
    # Mengirimkan hasil dari set roster item atau update roster item kepada user pengirim
    send_message(communicate, objek)
```

Gambar B. 18 Manager - Melayani User Ingin Menambahkan Roster Item

B.15 Melayani User Yang Ingin Mengubah Alias Roster Item

Input: seperti B.14.

Output: seperti B.14.

```
# Fungsionalitas untuk membantu menambahkan atau mengubah roster item
def set_to_rosters(username, item):
    jid_target = item["jid"]
    updating = False
    new = False
    # Mengecek apakah init_user sudah masuk ke roster_target
    tmp_item_jid = helper_check_and_get_roster(jid_target, username)
    # Logic komponen manager untuk menentukan apakah roster item akan ditambahkan atau diperbarui saja
    if(not tmp_item_jid):
        tmp_item_jid = {"jid": username, "subscription": None}
        new = True
    else:
        print("We will updating")
        updating = 1
    # Menentuka tipe subscription yang akan ditetapkan berdasarkan logic kepintaran pembeda
    if(not tmp_item_jid.get("subscription")):
        tmp_item_jid["subscription"] = "from"
    elif(tmp_item_jid.get("subscription") == "to"):
        updating = 2
        item["subscription"] = "both"
        tmp_item_jid["subscription"] = "both"
    if(updating == 2 or new):
        # Memanggil fungsionalitas untuk menyimpan atau mengubah roster item dari user's roster
        helper_set_rosters(username, item, updating)
        # Memanggil fungsionalitas untuk menyimpan atau mengubah roster item dari roster item's roster dengan roster itemnya adalah user pengirim
        helper_set_rosters(jid_target, tmp_item_jid, updating)
    else:
        helper_set_rosters(username, item, updating)
```

Gambar B. 19 Manager - Melayani User Ingin Mengubah Alias Roster Item

B.16 Melayani User Yang Ingin Menghapus Roster Item

Input: {stanza: iq, namespace: roster, from: user_init, type: set, query: {item: {jid: jid_target, name: target_alias, subscription: to}}, subscription: remove }.

Output:

1. {stanza: presence, from: user_target, to: user_init, type: unsubscribed},
2. {stanza: iq, namespace: roster, type: result, subscription: remove, to: user_init}.

```
# Menghapus roster item dari roster user pengirim
elif(message.get("type") and message.get("type").lower() == "set" and message.get("subscription") and message.get("subscription").lower() == "remove"):
    print(f"Terjadi permin                    {username}")
    jid_target = message.g  (variable) username: Any | None
    objek = delete_roster(username, message) # delete_roster_user
    if(objek['type'] != 'error'):
        # Membuat presence bertipe "unsubscribed" untuk user pengirim
        packet_presence_unsubscribed = get_packet_unsubscribed_for_init_entity(username, jid_target)
        # Mengirim packet presence kepada user pengirim
        send_message(communicate, packet_presence_unsubscribed)
    else:
        print(f"Terjadi error saat user {username} melakukan permintaan menghapus roster item")
    # Mengirimkan hasil dari menghapus roster item kepada user pengirim
    send_message(communicate, objek)
# print("Lewatin semua kok")
print("========================================")
```

Gambar B. 20 Manager - Melayani User Ingin Menghapus Roster Item

B.17 Melayani User Inisialisasi Presence

Input: {stanza: presence}.

Output: memberikan presence kepada roster item subscription “from”/”both” dan mendapatkan presence dari roster item subscription “to”/”both”.

```
# Mengelola init presence
if(not message.get('type') and not message.get("bio") and not message.get("to")):
    print(f"Terjadi permintaan init presence dari user dengan username {username}")
    init_presence(socket_user, username)
# Fungsionalitas untuk mengelola initial presence dari user
def init_presence(socket_user, username):
    update_status_online(['online', 'updated_at'], (1, get_timestamp(),), ['user_id'], (username,))
    roster_db_from = get_rosters_user(username, ("from", "both",))
    roster_db_to = get_rosters_user(username, ("to", "both",))
    objek_presence = get_presence_by_jid(username)
    if(roster_db_from):
        send_to_roster_subscription_from(roster_db_from, socket_user, objek_presence)
    if(roster_db_from):
        get_presence_entity_subscription_to(roster_db_to, socket_user, username)
```

Gambar B. 21 Manager - Melayani User Inisialisasi Presence

B.18 Melayani User Memperbarui Bio

Input: {stanza: presence, from: user_init, bio: bionya}.

Output: memperbarui data bio user dan mengirimkannya ke roster item “from”/”both”.

```
# Mengelola ketika user memperbarui bio-nya
elif(message.get('bio')):
    print(f"Terjadi permintaan perubahan bio dari user dengan username {username}")
    set_my_bio(socket_user, username, message)
# Fungsionalitas untuk mengelola permintaan user mengubah bio
def set_my_bio(socket_user, username, msg):
    if (type(msg) != dict):
        return helper_error(msg, "modify", "bad-request")
    elif(username != msg.get("from")):
        return helper_error(msg, "auth", "forbidden")
    updated_at = get_timestamp()
    update_bio(['bio', 'updated_at'], (msg.get('bio'), updated_at,), ['user_id'], (username,))
    my_presence = get_presence_by_jid(username)
    roster_db_from = get_rosters_user(username, ("from", "both",))
    if(not roster_db_from):
        return
    send_to_roster_subscription_from(roster_db_from, socket_user, my_presence, False)
```

Gambar B. 22 Manager - Melayani User Memperbarui Bio

B.19 Melayani User Meminta Directed Presence

Input: {stanza: presence, to: user_target}.

Output: {stanza: presence, from: user_target, username: user_target, bio: bio_target, online: Boolean, updated_at: last_updated_target, directed_entity: Boolean, to: user_init}.

```
# Mengelola ketika user membuat permintaan directed presence
elif(message.get('to')):
    jid_target = message.get('to')
    print(f"Terjadi permintaan directed presence yang dilakukan oleh user {username} dengan target user adalah {jid_target}")
    if(not get_user_by_username(jid_target)):
        print(f"Terjadi error saat permintaan directed presence karena target user {jid_target} tidak terdaftar di dalam sistem")
        objek_presence_error = {'stanza': 'presence', 'type': 'error', 'to': username, 'target': jid_target}
        send_message(communicate, objek_presence_error)
        continue
    # Kirim status presence init_entity ke target_entity jika target_entity online
    if(get_user_online(jid_target)):
        print(f"Mengirimkan directed presence kepada target user {jid_target} karena target user sedang online")
        init_entity_presence = get_presence_by_jid(username)
        init_entity_presence['directed_entity'] = True
        send_presence_to_someone(jid_target, socket_user, init_entity_presence)
    # Ambil status dari target_entity dan kirim ke init_entity
    target_entity_presence = get_presence_by_jid(jid_target)
    target_entity_presence['directed_entity'] = True
    print(target_entity_presence)
    send_presence_to_someone(username, socket_user, target_entity_presence)
```

Gambar B. 23 Manager - Melayani User Meminta Directed Presence

B.20 Melayani User Presence Unavailable

Input: {stanza: presence, from: user_init, type: unavailable}.

Output: memperbarui status user menjadi offline.

```
# Mengelola ketika user membuat permintaan presence tipe "unavailable" atau logout
elif(message.get('type') == 'unavailable'):
    print(f"Terjadi permintaan presence bertipe 'unavailable' dari user {username}")
    logout(socket_user, username)
    error = True
print("========================================")
# Fungsionalitas untuk mengelola permintaan user presence bertipe 'unavailable' atau logout
def logout(socket_user, username):
    update_status_online(['online', 'updated_at'], (0, get_timestamp(),), ['user_id'], (username,))
    roster_db_from = get_rosters_user(username, ("from", "both",))
    if(not roster_db_from):
        return
    objek_presence = {
        "stanza": "presence",
        "from": username,
        "type": "unavailable"
    }
    send_to_roster_subscription_from(roster_db_from, socket_user, objek_presence)
```

Gambar B. 24 Manager - Melayani User Presence Unavailable

C. Relay

C.1 Melakukan Koneksi dengan Tracker

Input: {ip_local: ip, port: port_num, type: tipe, is_private: Boolean}.

Output: akan terkoneksi dengan tracker.

```python
# Fungsionalitas terhubung ke tracker
def connect_to_tracker():
    ask = input("Local (y/n)?")
    if(ask.lower() == 'y'):
        client_tracker = SocketClient(None, 5000, tipe="Tracker")
    else:
        client_tracker = SocketClient("103.178.153.189", 5000, tipe="Tracker")
    # ct = Client Tracker
    ct = client_tracker.socket
    my_ip = client_tracker.localAddress
    ip_is_private = ipaddress.ip_address(my_ip[0]).is_private

    objek = {
        "ip_local": my_ip[0],
        "port": my_ip[1],
        "type": "relay",
        "is_private": ip_is_private
    }

    send_message(ct, objek)
    message = get_message_tracker(ct)
    print(f"Komponen dari tracker:\r\n {message}")
    print("========================================")
```

Gambar B. 25 Relay - Melakukan Koneksi dengan Tracker

C.2 Melakukan Koneksi dengan Manager

Input: {ip_local: ip, port: port_num, type: tipe}.

Output: akan terkoneksi dengan manager.

```python
# Connect ke manager
while not s2:
    target = None
    for m in message:
        if(m.get("type").lower() == "manager"):
            target = m
            break
    if(target):
        ip = target.get("ip_local")
        port = target.get("port")
        s2 = SocketClient(ip, port, tipe="Manager")
        relay_to_manager = s2.socket
        my_ip = s2.localAddress
        ip_is_private = ipaddress.ip_address(my_ip[0]).is_private
        objek = {
            "ip_local": my_ip[0],
            "port": my_ip[1],
            "type": "relay"
        }
        send_message(relay_to_manager, objek)
        messages = get_message_manager(relay_to_manager)
        for msg in messages:
            msg_from_manager = json.loads(msg)
            print(f"Pesan dari manager: {msg_from_manager}")
            print("----------------------------------------")
            # Melakukan konfigurasi awal saat relay pertama kali online, dengan daftar komponen yang diberikan oleh manager
            config_starter_relay(msg_from_manager)
            my_port = my_ip[1]
            r = SocketServer(my_port)
            relay = r.socket
    if(not s2 or not len(message)):
        print("Belum ada manager!")
        time.sleep(10)
        # Meminta manager kepada tracker
        message = get_manager(relay_to_tracker)
```

Gambar B. 26 Relay - Melakukan Koneksi dengan Manager

C.3 Melakukan Konfigurasi Relay Awal/Koneksi dengan Relay Senior

Input: {ip_local: ip, port: port_num, type: relay, is_private: Boolean, relay_username: username_relay}.

Output: terkoneksi dengan relay senior dan mendapatkan user yang terkoneksi dengan mereka.

```
# Fungsionalitas untuk konfigurasi awal saat relay online
def config_starter_relay(m): # m = message_from_manager
    global my_username, connection_relay, user_in_another_relay, my_ip
    mip = my_ip[0]
    mport = my_ip[1]
    my_username = m.get("username")
    connections = m.get("components") # Components Relay yang diberikan Manager
    for c in connections:
        ip = c.get("ip_local")
        port = c.get("port")
        # Mengecek agar tidak terkoneksi ke diri sendiri
        mine = ip == mip and port == mport
        if(not mine):
            connection_with_another_relay, username_relay, usernames_in_another_relay = connect_to_another_relay(ip, port, my_username)
            if(not username_relay):
                continue
            # Menyimpan koneksi dengan relay lainnya ke dalam connection_relay
            connection_relay[username_relay] = connection_with_another_relay.socket
            # Menyimpan user yang terkoneksi dengan relay yang baru saja disimpan
            for uname in usernames_in_another_relay:
                user_in_another_relay[uname] = username_relay
        print("========================================")
    print(f"Connection Relay Konfigurasi: {connection_relay}")
    print(f"User in Another Relay Konfigurasi: {user_in_another_relay}")
    print("========================================")
```

Gambar B. 27 Relay - Melakukan Konfigurasi Relay Awal

C.4 Menerima Koneksi TCP dari User

Input: {username: user_init, message: hello!}.

Output: menyimpan koneksi antara keduanya, memberikan pesan NCIR kepada manager, dan memberikan pesan “new user” kepada relay lainnya.

```
if(message.get("username")):
    print(f"Mendapatkan koneksi baru dari user {msg}")
    username = message.get("username")
    # Menyimpan user ke dalam daftar koneksinya
    connections[username] = connection
    # Membuat layanan eksklusif untuk mengelola komponen user
    threading.Thread(target=handle_component_user, args=(connection, my_username, username), daemon=True).start()
    objek = {
        "message": "ncir", # new connection in relay
        "username_relay": my_username,
        "tipe": "relay"
    }
    send_message(relay_to_manager, objek)
    send_new_user_to_another_relay(username, my_username)
```

Gambar B. 28 Relay - Menerima Koneksi TCP dari User

C.5 Menerima Koneksi TCP dari Relay

Input: {ip_local: ip, port: port_num, type: relay, is_private: Boolean, relay_username: username_relay}.

Output: {relay_username: username_relay, username_users: connection_user_in_another_relay}.

```
elif(message.get("type") == "relay"):
    print(f"Mendapatkan koneksi baru dari relay junior {msg}")
    uname_relay = message.get("relay_username")
    # Memanggil fungsi untuk menyimpan relay yang baru terkoneksi
    config_new_relay(uname_relay, connection)
    usernames = list(connections.keys())
    # Membuat objek yang akan dikirimkan kepada relay yang baru terkoneksi
    objek = {
        "relay_username": my_username,
        "username_users": usernames
    }
    send_message(connection, objek)
    # Membuat layanan eksklusif untuk mengelola komponen relay
    threading.Thread(target=handle_component_relay, args=(connection, my_username, ), daemon=True).start()
    print('========================================')
```

Gambar B. 29 Relay - Menerima Koneksi TCP dari Relay

C.6 Melayani Stanza Message dari User

Input: {stanza: message, from: user_target, to: user_init, lang: id, body: pesan}

Output: pesan tersampaikan.

```
elif(message.get("stanza") and message.get("stanza") == "message"):
    print('========================================')
    print(f'Menerima packet stanza message dari user')
    print(message)
    target = message.get("to")
    connection_target = connections.get(target)
    # Target dalam relay yang sama
    if(connection_target):
        print("Mengirim pesan kepada target pada relay yang sama")
        send_message_to_target(connection_target, message)
    # Target dalam relay yang berbeda
    else:
        relay_user = user_in_another_relay.get(target)
        print(f"Meneruskan pesan kepada relay {relay_user}")
        if(not relay_user):
            print("Terjadi error karena user yang dituju tidak terdapat di dalam sistem")
            initiate_user = message.get("from")
            # Mengirim packet error kepada user
            send_packet_error(communicate, 404, initiate_user)
            print('========================================')
            continue
        socket_relay = connection_relay.get(relay_user)
        send_message_to_target(socket_relay, message)
    print('========================================')
```

Gambar B. 30 Relay - Melayani Stanza Message dari User

C.7 Melayani Stanza Message dari Relay

Input: seperti C.6.

Output: meneruskan pesan ke user_target.

```
# Mengelola packet yang masuk adalah stanza message
elif(message.get("stanza") and message.get("stanza") == "message"):
    print('========================================')
    print(f"Relay mendapatkan stanza message dari relay {connected_relay_username}")
    print(message)
    target = message.get("to")
    connection_target = connections.get(target)
    # Memanggil fungsi untuk mengirim stanza message kepada target
    result = send_message_to_target(connection_target, message)
    # Membuat objek untuk mengembalikan hasil dari pengiriman pesan, dengan nilai activity adalah "message from another relay"
    objek = {"from": message.get("from"), "to": message.get("to"), "activity": "message from another relay"}
    # Validitas apakah error atau success
    if(not result):
        objek["error"] = 1
    else:
        objek["success"] = 1
    send_message(communicate, objek)
    print('========================================')
```

Gambar B. 31 Relay - Melayani Stanza Message dari Relay

C.8 Melayani Message “new user”

Input: {message: “new user”, username_user: user_terkoneksi, relay_username: username_relay}.

Output: user tersebut akan dicatat sebagai bagian dari koneksi relaynya.

```
elif(message.get("message") and message.get("message").lower() == "new user"):
    print('========================================')
    print("Relay mendapatkan pesan 'new user' dari relay lainnya")
    print(message)
    relay_uname = message.get("relay_username")
    username_user = message.get("username_user")
    print(f"Koneksi {username_user} kepada {relay_uname} telah tersambung!")
    # Mendaftarkan user ke dalam daftar koneksi relay pengirim
    user_in_another_relay[username_user] = relay_uname
    print(f"User didalam relay {relay_uname} adalah {user_in_another_relay}")
    print('========================================')
```

Gambar B. 32 Relay - Melayani Message "new user"

C.9 Melayani Message “end user”

Input: {message: “end user”, username_user: user_terkoneksi, relay_username: username_relay}.

Output: user tersebut akan dihapus dari bagian dari koneksi relaynya.

```
# Mengelola packet dengan nilai message "new user"
elif(message.get("message") and message.get("message").lower() == "end user"):
    print('========================================')
    print("Relay mendapatkan pesan 'end user' dari relay lainnya")
    print(message)
    relay_uname = message.get("username_relay")
    username_user = message.get("username_user")
    print(f"Koneksi {username_user} kepada {relay_uname} telah terputus!")
    # Menghapus user dari daftar koneksi relay pengirim
    del user_in_another_relay[username_user]
    print('========================================')
```

Gambar B. 33 Relay - Melayani Message "end user"

C.10 Menerima Packet Activity “message from another relay”

Input: {from: user_target, to: user_init, activity: 'message from another relay', success: null/1, error: null/1} .

Output: mengetahui pengiriman pesan berhasil atau tidak pada relay lainnya.

```
# Mengelola jika packet yang diterima nilai error adalah 1 dan nilai activity adalah "message from another relay"
elif(message.get("error") and message.get("activity") == "message from another relay"):
    initiate_user = message.get("from")
    communicate = connections[initiate_user]
    send_packet_error(communicate, 404, initiate_user)
elif(message.get("success") and message.get("activity") == "message from another relay"):
    print('========================================')
    print("Relay mendapatkan packet nilai success 1 dan nilai activity 'messsage from another relay'")
    print(message)
    # initiate_user = message.get("from")
    # communicate = connections[initiate_user]
    # send_packet_error(communicate, 404, initiate_user)
    print('========================================')
```

Gambar B. 34 Relay - Menerima Packet Activity "message from another relay"

DAFTAR PUSTAKA

Amira, K. (2021). *Pengertian Jaringan Komputer: Jenis-Jenis, Cara Kerja, dan Manfaat*. Gramedia. https://www.gramedia.com/literasi/pengertian-jaringan-komputer

Ben Gorman. (2023). *TCP vs UDP: What's the Difference and Which Protocol Is Better?* Gen Digital Inc. https://www.avast.com/c-tcp-vs-udp-difference

Biznet News. (2023a). *5 Keunggulan dan Fungsi VPS, Apa Saja?* PT Biznet Gio Nusantara. https://www.biznetgio.com/news/5-keunggulan-dan-fungsi-vps

Biznet News. (2023b). *Perbedaan Antara Public IP dan Private IP*. PT Biznet Gio Nusantara. https://www.biznetgio.com/news/perbedaan-public-ip-dan-private-ip

Breje, A. R., Gyorödi, R., Gyorödi, C., Zmaranda, D., & Pecherle, G. (2018). Comparative study of data sending methods for XML and JSON models. *International Journal of Advanced Computer Science and Applications*, *9*(12), 198–204. https://doi.org/10.14569/IJACSA.2018.091229

Faiz. (2023). *Ketahui Perbedaan IP Private dan IP Public*. Telkom University. https://dte.telkomuniversity.ac.id/ketahui-perbedaan-ip-private-dan-ip-public/

Faradilla A. (2023). *Apa Itu IP Address? Pengertian, Jenis, dan Fungsinya*. Hostinger.Co.Id. https://www.hostinger.co.id/tutorial/apa-itu-ip-address

Genç, R. (2017). The Importance of Communication in Sustainability & Sustainable Strategies. *Procedia Manufacturing*, *8*(October 2016), 511–516. https://doi.org/10.1016/j.promfg.2017.02.065

Gupta, D., Shivankar, J., & Gugulothu, S. (2021). Instant messaging using xmpp. *Journal of Physics: Conference Series*, *1913*(1), 8–12. https://doi.org/10.1088/1742-6596/1913/1/012126

Muhammad Ridho Rizqillah. (2024). *Improvisasi Crawling Pada Peta Web Menggunakan Algoritma Terdistribusi Dengan Model Koordinasi Berbasis Socket Programming*. Universitas Negeri Jakarta.

Oikarinen, J., & Reed, D. (1993). Internet Relay Chat Protocol. *Internet Engineering Task Force, May*, 1–65. https://www.rfc-editor.org/info/rfc1459

Python. (2024). *About Python*. Python Software Foundation. https://www.python.org/about/

Rekhter, Y., Karrenberg, D., J., de G. G., Lear, E., & Moskowitz, R. G. (1996). *Request for Comments: 1918 - Address Allocation for Private Internets*. 1–9.

Revou. (2024). *Middleware*. PT Revolusi Cita Edukasi. https://revou.co/kosakata/middleware

Roberts, P. A., & Challinor, S. (2000). IP address management. *BT Technology Journal*, *18*(3), 127–136. https://doi.org/10.1023/A:1026749131441

Saint-Andre, P., Smith, K., & Tronçon, R. (2009). *XMPP : The Definitive Guide*.

Shafiei, B. M., Idiranmanesh, F., & Iranmanesh, F. (2012). Socket programming. *Advances in Environmental Biology*, *6*(5), 1812–1822. https://doi.org/10.1016/b978-155860826-9/50016-1

Verma, K. (2023). *Socket Programming in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/socket-programming-python/

Virga, J. (2020). *UDP Communication*. MoinMoin Wiki Engine. https://wiki.python.org/moin/UdpCommunication