UNIVERSITATEA TEHNICĂ A MOLDOVEI

FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

AGENT DE MESAGERIE – MESSAGE BROKER

LUCRARE DE LABORATOR

la disciplina ”Programarea Aplicațiilor Distribuite”

Autor:

studentul gr. TI 141,

învățământ cu frecvența redusă

MIROVSCHI Eugen

Profesor:

ANTOHI Ionel

Chișinău, 2019

I. Scopul lucrării

Integrarea bazată pe agenți de mesaje care ar permite o comunicare asincronă dintre componentele distribuite ale unui sistem.

II. Obiectivele specifice ale lucrării

1. Definirea protocolului de lucru al agentului de mesaje
   a. Formatul (tipul) mesajelor de transmis. Se recomandă utilizarea formatului XML
   b. Numărului de canale unidirecționale (variabil/fix, dependent de tipul mesajelor, etc.)
   c. Structura comunicației asigurată de agent (unul-la-unu sau unul-la-mulți)
   d. Politici de livrare pentru diverse cazuri definite de logica de lucru al agentului (mesaje invalide, căderea agentului, etc.)
2. Elaborarea nivelului abstract de comunicare (rețea) necesară elementelor pentru primirea/transmiterea mesajelor de către emițător-agent-receptor;
   a. Protocolul de transport se alege în dependență de obiectivele protocolului de lucru
   b. Tratarea concurentă a cererilor
3. Elaborarea elementelor ce asigură păstrarea mesajelor primite
   a. Metoda transientă: mesajele vor fi stocate în colecții concurente de date specifice limbajului selectat
   b. Metoda persistentă: mesajele vor fi serializate/deserializate utilizând metode de procesare asincronă sau concurentă
4. Elaborarea nivelului abstract de rutare a mesajelor.

III. Considerații generale

Agentul de mesaje (message broker - eng.) este o componentă fizică care gestionează comunicarea dintre componentele unei aplicații distribuite. Avantajul utilizării acestei tehnici constă în decuplarea receptorului de transmițătorul mesajelor. Prin urmare o aplicație participantă transmite mesaje doar agentului, indicând un nume logic al receptorului.

Agentul poate expune diverse interfețe aplicațiilor în colaborare și poate transfera mesajele între acestea, ne impunând o interfață comună tuturor participanților întru asigurarea interacțiunii. Responsabilitățile și colaborările esențiale ale unui broker de mesaje sunt prezentate în tabelul 1 de mai jos.

Tabel 1 – Responsabilitățile și colaborările esențiale ale unui broker de mesaje

Responsabilități	**Colaborări**
Primirea mesajelor	Expeditor: aplicații (componente) ce trimit mesaje agentului
Determinarea destinatarilor și efectuarea rutării	Receptori: aplicații (componente) ce primesc mesajele de la broker
Tratarea diferențelor între interfețe	
Transmiterea mesajelor	

Decizia de a utiliza brokerul de mesaje pentru integrarea aplicațiilor balansează între flexibilitatea primită prin decuplarea participanților și efortul de a menține brokerul:

1) Beneficii
    a) Reduce cuplarea - transmițătorii comunică doar cu brokerul, astfel o potențială grupare a mai multor receptori sub un nume logic comun poate deveni transparentă transmițătorilor;
    b) Mărește integrabilitatea - aplicațiile care comunică cu brokerul nu trebuie să aibă aceiași interfață, astfel brokerul poate deveni o punte dintre aplicații cu diferite nivele de securitate și calitate a serviciilor QoS;
    c) Mărește evolutivitatea - brokerul protejează componentele de modificările individuale ale aplicațiilor integrate, deseori oferind capacități de configurare dinamică;
2) Constrângeri
    a) Crește complexitatea - brokerul comunicând cu toți participanții trebuie să implementeze multiple interfețe (protocoale) și în perspectiva performanței utilizează multi-threading-ul;
    b) Crește efortul pentru mentenanță - toți participanții trebuie să fie înregistrați la broker și se cere un mecanism de identificare a acestora;
    c) Reduce disponibilitatea - o singură componentă care intermediază comunicarea este singurul punct de eșec (single point of failure - eng.), căderea acestuia implică blocarea activității întregului sistem; această problemă se remediază prin dublarea brokerului și sincronizarea stărilor agentului primar și secundar;

d) Reduce performanța - agentul de mesaje adaugă un pas adăugător, care implică cheltuieli suplimentare (overhead - eng.).

Există diferite variații pentru agentul de mesaje, tratat ca element esențial pentru stilul arhitectural hub-and-spoke. Construirea mesajelor, structura canalelor de mesaje și rutarea servesc drept subiecte ale deciziilor de implementare în cadrul lucrării de laborator.

Implementarea agentului poate lua în calcul stilul arhitectural Broker. Un alt șablon de proiectare cu intenții similare este Mediator.

IV. Sarcina lucrării

Considerând brokerul o formă generalizată a medierii dintre componentele distribuite, se propune implementări ale rutării sau construirii de mesaje în conformitate cu șabloanele menționate de Gregor Hohpe, care sunt grupate în câteva secțiuni:

1. Messaging Systems,
2. Messaging Channels,
3. Message Constructions,
4. Message Routing,
5. Message Transformation,
6. Messaging endpoints,
7. System management.

V. Efectuarea lucrării

Project Structure

Lucrarea dată constă din crearea mai multor componente care vor comunica între ele folosind sistemul de mesaje ce urmează a fi implementat. Pe lângă message broker, este necesar de creat o aplicație client care va folosi acest serviciu. Tematica acestor componente va fi o platforma de comunicare (chat).

Luând în considerarea aceste cerințe am definit următoarea structură a proiectului:

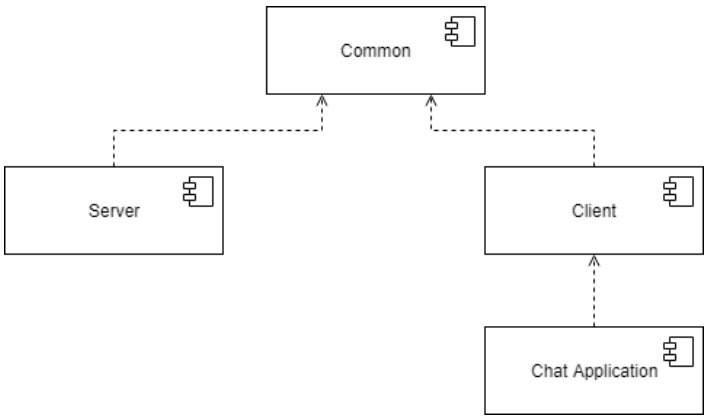

Figura 1 – Structura proiectului

Două cele mai importante module sunt însuși serverul (message broker) și clientul (component). Deoarece avem un mecanism comun de comunicare între server și client, vom crea o bibliotecă separată Common care va include toate clasele comune. În final avem aplicația chat care va utiliza biblioteca Client pentru a comunica cu serverul și eventual cu alte aplicații chat.

Network Layer

Primul pas în elaborarea acestei lucrări a fost proiectarea și elaborarea unui sistem de comunicare între message broker și aplicații (componente). Pentru a ușura crearea unui API, este necesar abstractizarea acestui nivel într-un serviciu comod de utilizat.

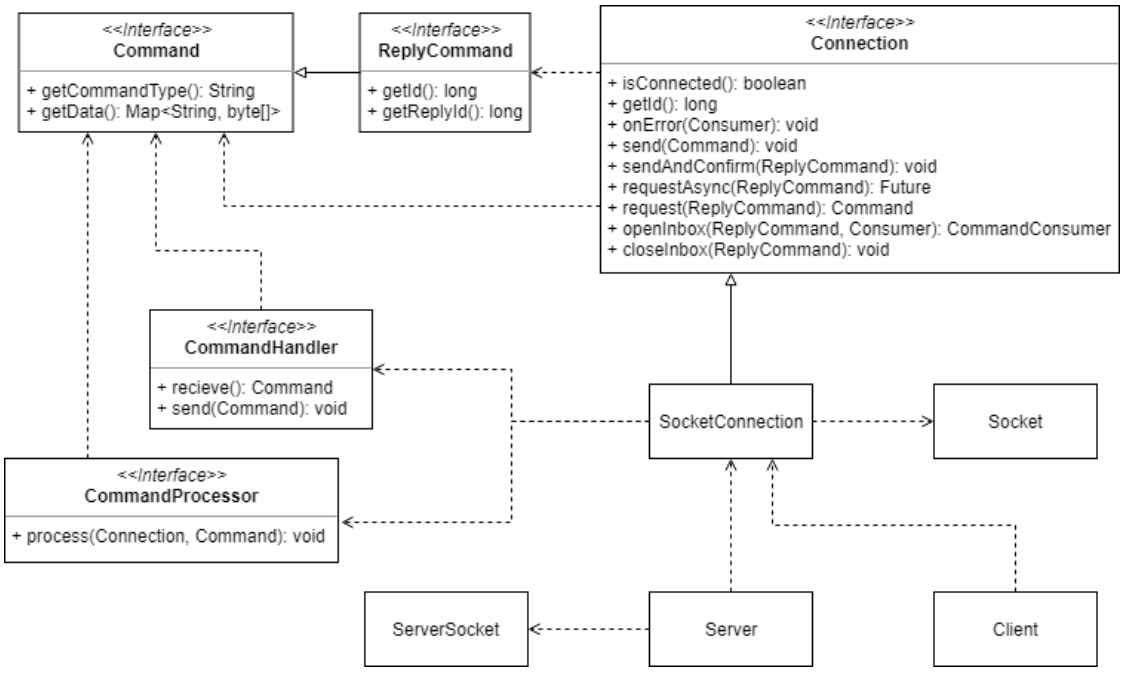

Figura 2 – Network Layer

Pentru comunicarea între componente vom folosi sockets cu protocolul TCP. Pentru a inițializa o conexiune, în limbajul de programare Java, este necesar de inițializat un obiect de tip ServerSocket și de ascultat pentru conexiuni noi din partea clienților.

```
private void internalRun() throws IOException
{
    serverSocket = new ServerSocket(port, MAX_CONNECTIONS);
    System.out.println("Server listening on port " + port);

    while (!serverSocket.isClosed())
    {
        accept();
    }
}
```

Pentru ca un client să se conecteze la server, trebuie să inițializăm un obiect de tip Socket specificând adresa IP și portul serverului.

```
public void connect() throws IOException
{
    try
    {
        final Socket socket = new Socket(serverIp, serverPort);
        connection = connectionFactory.create(socket);
        new Thread(connection).start();
    }
    catch (final Exception exception)
    {
        System.out.println("An error occurred with [" + connection + "]: " +
exception.getMessage());
        throw new IOException("Could not connect to the server", exception);
    }
}
```

Atunci când clientul reușește să se conecteze cu succes la server, acesta la rândul său creează un obiect de tip Socket care ulterior va fi folosit pentru comunicare.

```
private void accept()
{
    try
    {
        final Socket socket = serverSocket.accept();
        System.out.println("Client connected " + socket.toString());
        final Connection connection = connectionFactory.create(socket);

        try
        {
            executorService.execute(() -> handleConnection(connection));
        }
        catch (final RejectedExecutionException e)
        {
            System.out.println("Server full, rejecting:" + socket);
            connection.send(new ConnectResponseCommand(false));
        }
    }
    catch (final IOException e)
    {
        System.out.println("Client connection failed:" + e.getMessage());
    }
}
```

O dată ce am reușit să inițializăm conexiunea între broker și client, putem comunica folosind stream-urile din obiectele de tip Socket. În Java, stream-urile reprezintă niște fluxuri de octeți, respectiv avem nevoie de un mecanism de conversie a mesajelor în octeți și invers din octeți în mesaje. Pentru a simplifica acest proces, fiecare comandă va fi compusă din următoarele câmpuri: tipul comenzii și o listă de proprietăți, unde fiecare proprietate are nume și valoare. Acest mecanism este similar cu Remote Procedural Call, tipul comenzii ar putea fi văzut ca denumirea procedurii, iar proprietățile ca argumente ale procedurii.

```
public interface Command
{
    String getCommandType();

    Map<String, byte[]> getData();
}
```

Pentru a crea un mecanism aparte de Request-Reply, avem nevoie de un tip specific de comenzi care va păstra această corelație între comenzi. Cu acest scop avem definită următoarea interfață care ulterior va fi procesată într-un mod unic:

```
public interface ReplyCommand extends Command
{
    long getId();

    Long getReplyId();
}
```

Însuși scrierea și citirea comenzilor în fluxuri de date este abstractizată într-un API simplu care primește comenzi ce urmează a fi trimise și întoarce comenzile care au fost recepționate.

```
 public interface CommandHandler
{
    <T extends Command> T receive() throws IOException;

    void send(Command command) throws IOException;
}
```

Deoarece comenzile vor fi recepționate asincron, avem nevoie să definim o interfață care va permite procesarea acestor comenzi și rularea logicii specifice pentru fiecare comandă.

```
public interface CommandProcessor
{
    void process(final Connection connection, final Command command) throws IOException;
}
```

Combinând clasele CommandHandler, CommandProcessor și Socket, putem crea un API complet de comunicare între componente. Acest API are următoarea structură:

```
public interface Connection extends Runnable, Closeable
{
    boolean isConnected();

    long getId();
```

```
    void onError(Consumer<String> onError);

    void send(Command command) throws IOException;

    void sendAndConfirm(ReplyCommand command) throws IOException;

    <T extends ReplyCommand> CompletableFuture<T> requestAsync(ReplyCommand command)
throws IOException;

    <T extends ReplyCommand> T request(ReplyCommand command) throws IOException;

    <T extends ReplyCommand> CommandConsumer<T> openInbox(ReplyCommand command,
ThrowableConsumer<T, IOException> consumer) throws IOException;

    void closeInbox(ReplyCommand command) throws IOException;
}
```

Exemple de trimitere a unei comenzi:

```
connection.sendAndConfirm(new DeclareExchangeCommand(name));
```

Exemplu de recepționare a unei comenzi:

```
public class DeclareExchangeCommandProcessor extends
AbstractConfirmCommandProcessor<DeclareExchangeCommand>
{
    private final BrokerService brokerService;

    public DeclareExchangeCommandProcessor(final BrokerService brokerService)
    {
        this.brokerService = brokerService;
    }

    @Override
    public boolean canProcess(final Command command)
    {
        return command instanceof DeclareExchangeCommand;
    }

    @Override
    protected void processCommand(final Connection connection, final
DeclareExchangeCommand command) throws IOException
    {
        brokerService.declareExchange(command.getName());
    }
}
```

Message Channel (Messaging System)

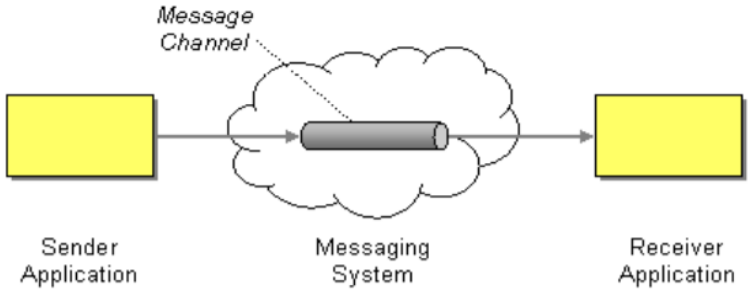

Figura 3 – Message Channel/Message System

Canalul de mesaje este un element logic utilizat pentru interconectarea aplicațiilor. O aplicație scrie mesaje în canal și alta le citește. Prin urmare această metodă de intermediere este una primară, iar coada de mesaje reprezintă forma sa de implementare.

Pentru a implementa un canal suficient de flexibil am introdus următoarele concepte în proiect:

- **Exchange** – Destinație abstractă care recepționează mesajele. Acest obiect nu stochează mesajele, doar le redirecționează conform unor reguli predefinite.
- **Queue** – Coadă de mesaje care are ca scop recepționarea și stocarea mesajelor. La fel are proprietatea de a livra mesajele către consumator.
- **Bind** – Oferă posibilitatea de a crea o regulă de redirecționare a mesajelor din Exchange în Queue. Acest obiect oferă posibilitatea de filtrare a mesajelor care va fi ulterior folosit pentru a crea mecanismul de rutare

Întreg sistem de procesare a mesajelor se va baza pe aceste obiecte de bază. Respectiv un client al serviciului broker va trebui să definească aceste obiecte. Astfel avem creată următoarea interfață:

```
public interface BrokerService
{
    void declareExchange(String name) throws IOException;
    void declareQueue(String name) throws IOException;
    void bind(String exchange, String queue, String routingKey) throws IOException;
    void deleteExchange(String name) throws IOException;
    void deleteQueue(String name) throws IOException;
    void deleteBind(String exchange, String queue, String routingKey) throws IOException;
    ...
}
```

Această interfață va avea două implementări: una pe client care va trimite comenzile către server și alta pe server care va procesa aceste comenzi și va gestiona obiectele date.

Implementarea pe client:

```
public class SocketClient implements Client
{
    private Connection connection;

    ...

    @Override
    public void declareExchange(final String name) throws IOException
    {
        connection.sendAndConfirm(new DeclareExchangeCommand(name));
    }

    @Override
    public void declareQueue(final String name) throws IOException
    {
        connection.sendAndConfirm(new DeclareQueueCommand(name));
    }

    @Override
    public void bind(final String exchange, final String queue, final String routingKey)
throws IOException
    {
        connection.sendAndConfirm(new BindCommand(exchange, queue, routingKey));
    }

    @Override
    public void deleteExchange(final String name) throws IOException
    {
        connection.sendAndConfirm(new DeleteExchangeCommand(name));
    }

    @Override
    public void deleteQueue(final String name) throws IOException
    {
        connection.sendAndConfirm(new DeleteQueueCommand(name));
    }

    @Override
    public void deleteBind(final String exchange, final String queue, final String
routingKey) throws IOException
    {
        connection.sendAndConfirm(new DeleteBindCommand(exchange, queue, routingKey));
    }
}
```

Implementarea pe server:

```
public class InMemoryBrokerService implements BrokerService
{
    private final Map<String, Exchange> exchanges;
    private final Map<String, Queue> queues;
    private final Collection<Route> routes;

    ...

    @Override
    public void declareExchange(final String name) throws IOException
```

```
    {
        if (exchanges.containsKey(name))
        {
            throw new IOException("Exchange already exists");
        }
        exchanges.put(name, new Exchange(name));
    }

    @Override
    public void declareQueue(final String name) throws IOException
    {
        if (queues.containsKey(name))
        {
            throw new IOException("Queue already exists");
        }
        queues.put(name, new Queue(name));
    }

    @Override
    public void bind(final String exchange, final String queue, final String routingKey)
throws IOException
    {
        if (!exchanges.containsKey(exchange))
        {
            throw new IOException("Unknown exchange");
        }
        if (!queues.containsKey(queue))
        {
            throw new IOException("Unknown queue");
        }

        try
        {
            final Route route = new Route(exchanges.get(exchange), queues.get(queue),
routingKey);
            route.bind();
            routes.add(route);
        }
        catch (final PatternSyntaxException exception)
        {
            throw new IOException("Invalid pattern: " + exception.getMessage());
        }
    }

    @Override
    public void deleteExchange(final String name) throws IOException
    {
        if (!exchanges.containsKey(name))
        {
            throw new IOException("Unknown exchange");
        }

        exchanges.get(name).remove();
        exchanges.remove(name);
    }

    @Override
    public void deleteQueue(final String name) throws IOException
    {
        if (!queues.containsKey(name))
        {
            throw new IOException("Unknown queue");
        }

        queues.get(name).remove();
```

```
        queues.remove(name);
    }

    @Override
    public void deleteBind(final String exchange, final String queue, final String
routingKey) throws IOException
    {
        deleteBind(Route.class, exchange, queue, routingKey);
    }
}
```

Content-Based Router (Message Routing)

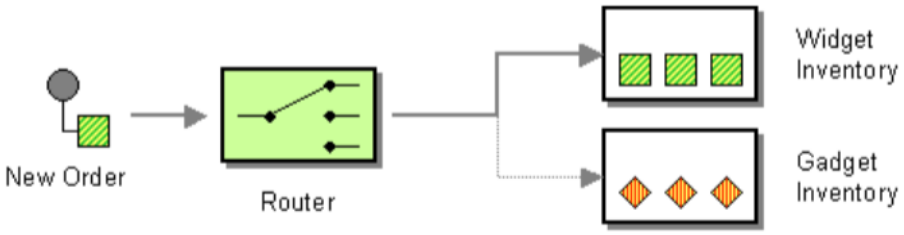

Figura 4 – Content-Based Router (Message Routing)

Ruterul bazat pe conținut examinează mesajul și îl rutează în dependență de datele conținute în mesaj. Pentru implementarea acestui mecanism vom folosi obiectele definite mai sus, în special Bind care definește modul în care mesajele sunt filtrate și redirecționate.

Fiecare obiect Bind are un atribut numit routingKey. Acest atribut conține în sine o expresie regulată. Fiecare mesaj la rândul său are un atribut numit routingKey. Acest atribut va fi procesat de expresiile regulate definite în obiectele Bind. Dacă routingKey din mesaj satisface expresia definită în Bind, atunci mesajul este redirecționat către coada de așteptare definită în Bind.

```
public void publish(final ByteMessage message, final String routingKey)
{
    final List<Queue> destinations = getDestinations(routes,
routingKey).collect(toList());

    if (destinations.size() > 0)
    {
        destinations.forEach(queue -> queue.add(message));
    }
}

private Stream<Queue> getDestinations(final Collection<Route> routes, final String
routingKey)
{
    return routes.stream()
        .filter(route -> route.matches(routingKey))
        .map(Route::getQueue)
        .distinct();
}
```

Dead Letter Channel (Messaging Channels)

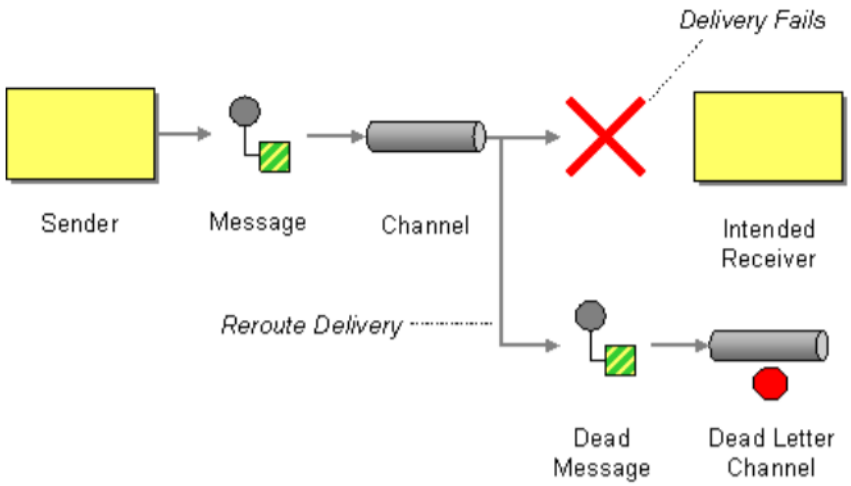

Figura 5 – Dead Letter Channel (Messaging Channels)

Canalul Scrisorilor Nelivrate descrie scenariul conform căruia sistemul de mesagerie definește ce de realizat în caz dacă mesajul nu poate fi livrat unui receptor specificat. Acest fapt poate fi cauzat de o problemă în conexiune sau de o excepție datorată lipsei spațiului. În mod obișnuit există multiple încercări de a transmite mesajul, succedate de livrarea acestuia spre canalul Scrisorilor Nelivrate.

În cazul acestui proiect, mesajele vor fi considerate nelivrate atunci când acestea sunt trimise către un Exchange, însă din cauza regulilor prezente în obiectele Bind, mesajul nu ajunge în nici o coadă de așteptare.

Canalele scrisorilor nelivrate nu vor fi nimic altceva decât alte cozi de așteptare definite de utilizator. Aceste cozi la fel vor fi legate de obiecte Exchange, însă vor folosi un tip specific de Bind. Pentru a permite declararea acestor obiecte vom adăuga următoarea metodă în BrokerService:

```
void bindDeadLetters(String exchange, String queue, String routingKey) throws
IOException;
void deleteDeadLettersBind(String exchange, String queue, final String routingKey) throws
IOException;
```

În logica de rutare definită anterior, vom adăuga un caz special atunci când nu se găsește nici o destinație pentru mesajul trimis.

```
public void publish(final ByteMessage message, final String routingKey)
{
    final List<Queue> destinations = getDestinations(routes,
routingKey).collect(toList());

    if (destinations.size() > 0)
    {
        destinations.forEach(queue -> queue.add(message));
    }
    else
    {
        getDestinations(deadLettersRoutes, routingKey).forEach(queue ->
queue.add(message));
    }
}
```

Publish-Subscribe Channel (Messaging Channels)

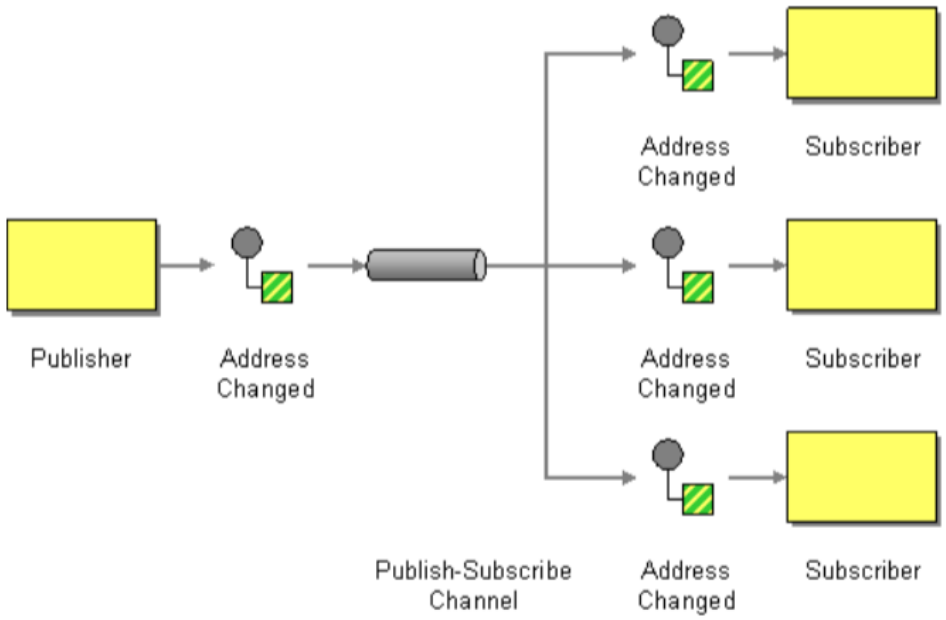

Figura 6 – Publish-Subscribe Channel (Messaging Channels)

Acest tip de canal difuzează un eveniment sau o notificare tuturor receptorilor abonați. Se contrapune în mod evident canalul punt-la-punct. Fiecare abonat va primi o dată mesajul după care acesta va fi eliminat din sistem.

Pentru a implementa acest sistem vom extinde interfața menționată anterior prin adăugarea unor metode noi ce permit trimiterea și recepționarea mesajelor. Vom avea două metode ce trimit mesajele, una implicită și alta care va conține o cheie de rutare. Această cheie ulterior va fi folosită de mecanismul de rutare descris mai jos. Pentru recepționarea mesajelor vom avea două metode:

recepționarea mesajelor asincron atunci când acestea apar în coada de așteptare sau sincron câte un mesaj.

```
public interface BrokerService
{
    ...

    void publish(String exchange, ByteMessage message) throws IOException;
    void publish(String exchange, ByteMessage message, final String routingKey) throws
IOException;
    void subscribe(String queue, String tag, ThrowableConsumer<ByteMessage, IOException>
consumer) throws IOException;
    void consume(String queue, ThrowableConsumer<ByteMessage, IOException> consumer)
throws IOException;
    void stopConsumer(String tag) throws IOException;
}
```

Similar ca la gestionarea obiectelor de bază, în implementarea din client trimiterea mesajului va fi doar trimiterea unei comenzi către server. Însă metodele de consum a mesajelor vor avea logică adițională care vor recepționa mesajele în mod asincron într-un thread. aparte vom avea în mare parte doar trimiterea comenzilor către server. Aici vom folosi mecanismul de mesaje definit în nivelul de rețea.

```
@Override
public void publish(final String exchange, final ByteMessage message) throws IOException
{
    connection.sendAndConfirm(new SendMessageCommand(exchange, message));
}

@Override
public void publish(final String exchange, final ByteMessage message, final String
routingKey) throws IOException
{
    connection.sendAndConfirm(new SendMessageCommand(exchange, message, routingKey));
}

@Override
public void subscribe(final String queue, final String tag, final
ThrowableConsumer<ByteMessage, IOException> consumer) throws IOException
{
    final SubscribeCommand subscribe = new SubscribeCommand(queue);
    this.<MessageCommand>registerConsumer(subscribe, tag, command ->
consumer.accept(command.getMessage()));
}

@Override
public void consume(final String queue, ThrowableConsumer<ByteMessage, IOException>
consumer) throws IOException
{
    final ConsumeMessageCommand subscribe = new ConsumeMessageCommand(queue);
    connection.<MessageCommand>requestAsync(subscribe).thenAccept(message -> {
        try
        {
            consumer.accept(message.getMessage());
            connection.send(new ConfirmActionCommand(message));
        }
        catch (IOException e)
        {
            try
```

```
            {
                connection.send(new ConfirmActionCommand(message, "Failed to consume
message"));
            }
            catch (IOException e1)
            {
                System.out.println("Connection lost");
            }
        }
    });
}

@Override
public void stopConsumer(final String tag) throws IOException
{
    if (commandConsumers.containsKey(tag))
    {
        commandConsumers.get(tag).stop();
        commandConsumers.remove(tag);
    }
}

private <T extends ReplyCommand> void registerConsumer(
    final ReplyCommand subscribe, final String tag, final ThrowableConsumer<T,
IOException> consumer) throws IOException
{
    commandConsumers.put(tag, connection.openInbox(subscribe, consumer));
}
```

Pe server metoda de publicare a mesajului va trimite mesajul către Exchange, care ulterior va fi redirecționat în cozile de așteptare existente. Metoda de consum asincronă a mesajelor va crea o instanță nouă care vor rula pe thread aparte și va consuma mesajele care vin în coada de așteptare. O data ce va citi un mesaj, acest obiect va trimite mesajul către client. Metoda de consum sincronă va citi direct mesajul din coada de așteptare și îl va trimite către client.

```
@Override
public void publish(final String exchange, final ByteMessage message) throws IOException
{
    if (!exchanges.containsKey(exchange))
    {
        throw new IOException("Unknown exchange");
    }

    exchanges.get(exchange).publish(message, "");
}

@Override
public void publish(final String exchange, final ByteMessage message, final String
routingKey) throws IOException
{
    if (!exchanges.containsKey(exchange))
    {
        throw new IOException("Unknown exchange");
    }

    exchanges.get(exchange).publish(message, routingKey);
}

@Override
public void subscribe(final String queue, final String tag, final
ThrowableConsumer<ByteMessage, IOException> consumer) throws IOException
```

```
{
    startConsumer(queue, tag, q -> new SingleMessageConsumer(q, consumer));
}

@Override
public void consume(final String queue, ThrowableConsumer<ByteMessage, IOException>
consumer) throws IOException
{
    if (!queues.containsKey(queue))
    {
        throw new IOException("Unknown queue");
    }

    ByteMessage message = null;
    try
    {
        message = queues.get(queue).getQueue().poll();

        if (message != null)
        {
            consumer.accept(message);
        }
    }
    catch (final IOException exception)
    {
        System.out.println("Couldn't consume message" + exception);
        queues.get(queue).getQueue().addFirst(message);
    }
}

private void startConsumer(final String queue, final String tag, final Function<Queue,
AbstractMessageConsumer> messageConsumerSupplier) throws IOException
{
    if (!queues.containsKey(queue))
    {
        throw new IOException("Unknown queue");
    }

    final AbstractMessageConsumer messageConsumer =
messageConsumerSupplier.apply(queues.get(queue));
    messageConsumers.put(tag, messageConsumer);
    queues.get(queue).add(messageConsumer);
    executorService.execute(messageConsumer);
}

@Override
public void stopConsumer(final String tag) throws IOException
{
    if (!messageConsumers.containsKey(tag))
    {
        throw new IOException("Unknown consumer");
    }

    messageConsumers.get(tag).stop();
    messageConsumers.remove(tag);
}
```

Event-Driven Consumer (Messaging Endpoints)

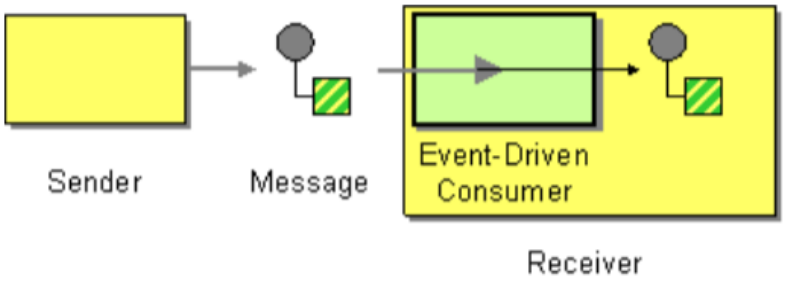

Figura 7 – Event-Driven Consumer (Messaging Endpoints)

Un consumator bazat pe evenimente presupune existența de activități asociate canalului la sosirea unui mesaj. Se menționează că această procesare este una asincronă.

Cum a fost menționat anterior, citirea mesajele din cozile de așteptare și trimiterea acestora către client are loc într-un obiect aparte care rulează în propriul său thread:

```
public class SingleMessageConsumer extends AbstractMessageConsumer
{
    private final ThrowableConsumer<ByteMessage, IOException> consumer;

    public SingleMessageConsumer(final Queue queue, final ThrowableConsumer<ByteMessage,
IOException> consumer)
    {
        super(queue);
        this.consumer = consumer;
    }

    @Override
    public void runInternal()
    {
        ByteMessage message = null;
        try
        {
            message = getQueue().getQueue().poll(10, TimeUnit.MILLISECONDS);

            if (message != null)
            {
                consumer.accept(message);
            }
        }
        catch (final InterruptedException | IOException exception)
        {
            System.out.println("Message consumer stopped " + exception);
            getQueue().getQueue().addFirst(message);
            stop();
        }
    }
}
```

Polling Consumer (Messaging Endpoints)

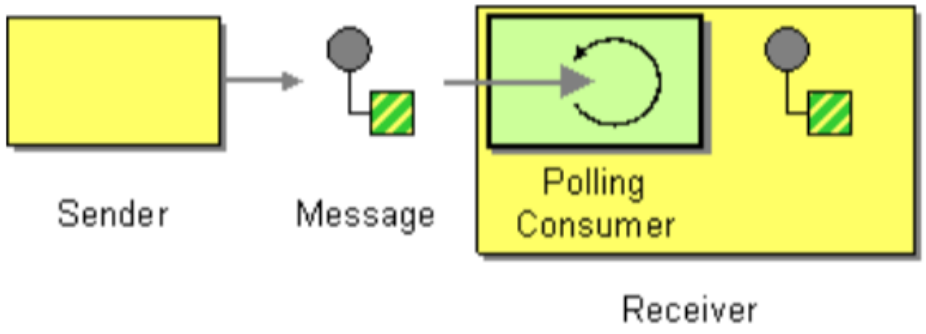

Figura 8 – Polling Consumer (Messaging Endpoints)

Un consumator de acest tip va aștepta un mesaj, îl va procesa și va trece spre următorul mesaj. Astfel acest șablon este prin definiție sincron, căci va bloca firul până la venirea următorului mesaj. Implementarea acestui șablon este descrisă mai sus în modulul Publish-Subscribe Channel.

Wire Tap (System Management)

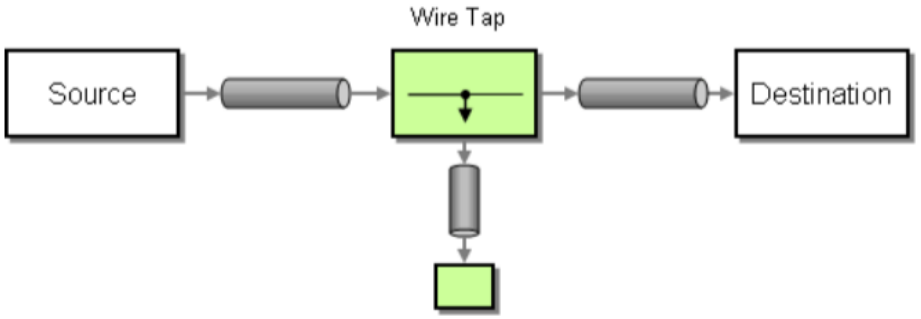

Figura 9 – Wire Tap

Canalul de supraveghere copie mesajul din canal și-l retransmite într-un special indicat în scopul inspectării mesajului sau analizei ulterioare.

Deși acest mecanism lucrează separat de consumatorul de mesaje descris anterior, unica diferență la nivel de implementare este obiectul de citire și trimite a mesajelor către client. Și anume faptul că mesajele nu sunt extrase din coada de așteptare, însă acest obiect va monitoriza coada și va trimite mesaje către client, atunci când coada de așteptare recepționează un mesaj nou.

În Broker Service vom adăuga o metodă nouă care va porni un Wire Tap:

```
void listen(String queue, String tag, ThrowableConsumer<ByteMessage, IOException>
consumer) throws IOException;
```

Implementarea acestei metode în client:

```
@Override
public void listen(final String queue, final String tag, final
ThrowableConsumer<ByteMessage, IOException> consumer) throws IOException
{
    final ListenCommand subscribe = new ListenCommand(queue, tag);
    this.<MessageCommand>registerConsumer(subscribe, tag, command ->
consumer.accept(command.getMessage()));
}
```

Implementarea acestei metode în server:

```
@Override
public void listen(final String queue, final String tag, final
ThrowableConsumer<ByteMessage, IOException> consumer) throws IOException
{
    if (!queues.containsKey(queue))
    {
        throw new IOException("Unknown queue");
    }
    messageConsumers.put(tag, new ListenMessageConsumer(queues.get(queue), consumer));
}
```

Consumatorul care va citi și trimite mesajele către client:

```
public class ListenMessageConsumer implements MessageConsumer
{
    private final Queue queue;
    private final Observer observer;
    private final ThrowableConsumer<ByteMessage, IOException> consumer;

    public ListenMessageConsumer(final Queue queue, final ThrowableConsumer<ByteMessage,
IOException> consumer)
    {
        this.queue = queue;
        this.consumer = consumer;
        this.observer = this::update;
        this.queue.getAddObservable().addObserver(this.observer);
    }

    private void update(final Observable observable, final Object object)
    {
        final ByteMessage message = (ByteMessage) object;
        try
        {
            consumer.accept(message);
        }
        catch (final IOException exception)
        {
            System.out.println("Message listener stopped " + exception);
            queue.getQueue().addFirst(message);
            stop();
        }
    }

    @Override
    public void stop()
    {
        queue.getAddObservable().deleteObserver(observer);
    }
}
```

Message Translator (Messaging Systems)

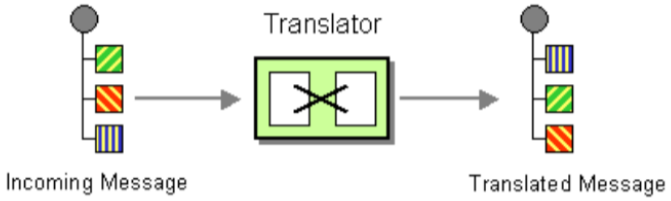

Figura 10 – Message Translator

Translatorul de mesaje este utilizat pentru a transforma mesajele dintr-un format în altul. De exemplu o aplicație trimite mesaje în XML, iar alta primește date doar în format JSON sau un alt XML.

Efectuând această lucrare am decis că serverul va răspunde doar de transmiterea, rutarea și recepționarea mesajelor fără a diferenția mesajele în bază de conținut. Astfel serverul va putea lucra cu orice tip de mesaj ce poate fi reprezentat printr-o serie de octeți, inclusiv fișiere binare cum ar fi imaginile.

Respectiv clientul își va asuma responsabilitatea de a transla mesajele. Deși acest mecanism de translare va putea lucra cu mai multe formate (JSON, XML, Yaml) scopul final nu va fi translarea dintr-un format în altul, ce însă transformarea obiectelor Java în unul din aceste formate și înapoi din text în obiect Java.

Pentru a nu afecta logica de bază care lucrează cu date binare, am decis să creez un serviciu aparte care va acoperi aceste cazuri:

```
public interface MessageService
{
    <T> void publish(String exchange, Message<T> message) throws IOException;

    <T> void publish(String exchange, Message<T> message, String routingKey) throws
IOException;

    <T> void subscribe(String queue, String tag, Consumer<Message<T>> consumer, Class<T>
type) throws IOException;

    <T> void consume(String queue, Consumer<Message<T>> consumer, Class<T> type) throws
IOException;

    <T> void listen(String queue, String tag, Consumer<Message<T>> consumer, Class<T>
type) throws IOException;
}
```

Acest API va avea o structură similară cu BrokerService, însă va avea posibilitatea de a lucra cu mesaje ce conțin obiecte Java în loc de date binare. Acest serviciu va folosi următoarea interfață pentru a transforma obiectele în date binare și invers:

```
public interface MessageTranslator
{
    ByteMessage serialize(final Message object);

    <T> Message<T> deserialize(ByteMessage message, Class<T> type);
}
```

Această interfață are câteva implementări care transformă obiectele în format JSON, XML, Yaml. Aceste implementări folosesc biblioteci terțe.

Pentru a suporta toate formatele existente, vom crea o implementare care va combina în sine toate implementările descrise mai sus. Această implementare va folosi Content-Type pentru a alege formatul de conversie. În caz ca nu este specific Content-Type sau are o valoare greșită, translatorul va folosi un format implicit (JSON) și va rescrie valoarea specificată pentru Content-Type.

```
public class CompositeMessageTranslator implements MessageTranslator
{
    private final MessageTranslator defaultTranslator;
    private final Map<String, MessageTranslator> translators;

    public CompositeMessageTranslator(final List<AbstractMessageTranslator>
messageTranslators)
    {
        this.defaultTranslator = messageTranslators.stream().findFirst().orElseThrow(() -
> new RuntimeException("No translators"));
        this.translators = messageTranslators.stream()
            .collect(toMap(AbstractMessageTranslator::getContentType, identity()));
    }

    @Override
    public ByteMessage serialize(final Message message)
    {
        return getMessageTranslator(message)
            .orElse(defaultTranslator)
            .serialize(message);
    }

    @Override
    public <T> Message<T> deserialize(final ByteMessage message, final Class<T> type)
    {
        return getMessageTranslator(message)
            .orElseThrow(() -> new RuntimeException("Unknown content type"))
            .deserialize(message, type);
    }

    private Optional<MessageTranslator> getMessageTranslator(final Message message)
    {
        return ofNullable(message.getProperties().get(CONTENT_TYPE))
            .map(Object::toString)
            .map(translators::get);
    }
}
```

Correlation Identifier (Messaging Channels)

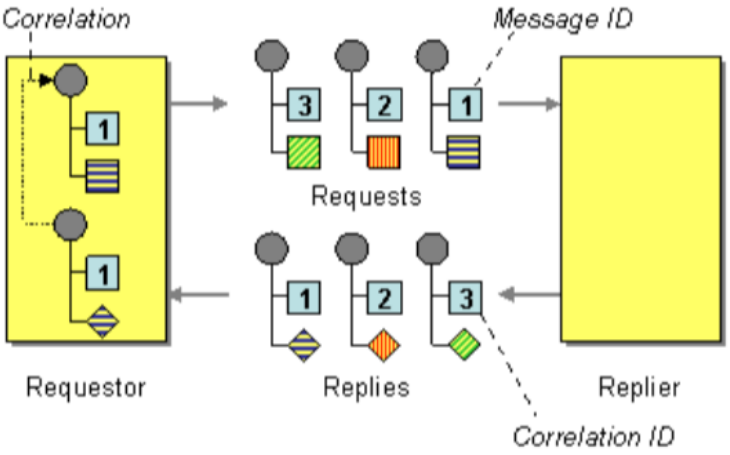

Figura 11 – Correlation Identifier

Corelarea Identificatorilor oferă posibilitatea de potrivit mesajele cererii și răspunsului într-un sistem de mesagerie asincronă prin atașare mesajului unui identificator de corelare.

În sistemul implementat la această lucrare, fiecare mesaj poate avea o listă de proprietăți. Respectiv corelația poate fi efectuată manual folosind oricare din aceste proprietăți. Însă pentru a ușura utilizatorilor acestui sistem, am adăugat un constructor nou în clasa mesaj care eventual va adăuga identificatorul de corelație în lista de proprietăți a mesajului.

```
public class Message<T>
{
    private static final String CORRELATION = "X-Correlation";

    private final T message;
    private final Properties properties;

    public Message(final T message)
    {
        this.message = message;
        this.properties = new Properties();
    }

    public Message(final T message, final Properties properties)
    {
        this.message = message;
        this.properties = properties;
    }

    public Message(final T message, final Properties properties, final String
correlation)
    {
        this.message = message;
        this.properties = properties;

        correlate(correlation);
    }
```

```
    public T getMessage()
    {
        return message;
    }

    public Properties getProperties()
    {
        return properties;
    }

    public String getCorrelation()
    {
        return getProperties().getProperty(CORRELATION);
    }

    public void correlate(final String correlationId)
    {
        getProperties().put(CORRELATION, correlationId);
    }
}
```

Content Enricher (Message Transformation)

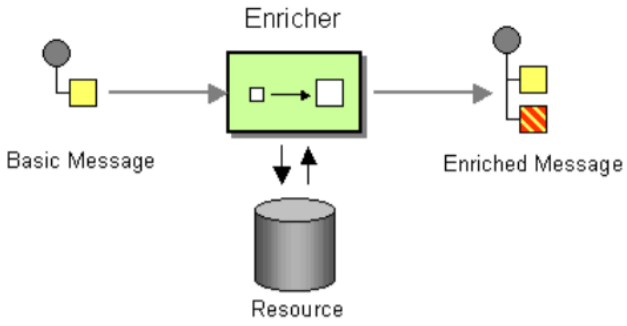

Figura 12 – Content Enricher

Acest tip de transformare completează mesajul cu date lipse, frecvent în astfel de cazuri sunt utilizate surse externe de date.

Similar ca modulul de translare, am decis că serverul nu va avea responsabilitatea de a modifica în careva mod mesajul. Doar clientul va răspunde de conținutul mesajelor și formatul acestora. Respectiv completarea mesajului, la fel va avea loc în client și anume în MessageService definit anterior.

Inițial vom defini o interfață care va permite completarea mesajelor:

```
public interface MessageEnricher
{
    boolean shouldApply(Message message);

    Message apply(Message message);
}
```

Apoi vom adăuga o metodă nouă la MessageService care va permite adăugarea unui MessageEnricher în sistem:

```
<T> void enrich(final MessageEnricher messageEnricher);
```

Implementarea acestei metode

```
private List<MessageEnricher> messageEnrichers;

@Override
public <T> void enrich(final MessageEnricher messageEnricher)
{
    messageEnrichers.add(messageEnricher);
}
```

Utilizarea acestor obiecte are loc atunci când se translează și se trimite un mesaj:

```
private <T> ByteMessage serialize(final Message<T> message) throws IOException
{
    final Iterator<MessageEnricher> iterator = messageEnrichers.stream()
        .filter(enricher -> enricher.shouldApply(message))
        .iterator();

    return Stream.<Message>iterate(message, m -> iterator.next().apply(m))
        .filter(m -> !iterator.hasNext())
        .findFirst()
        .map(messageTranslator::serialize)
        .orElseThrow(() -> new IOException("Could not send null message"));
}
```

Chat Application

În acest moment avem sistemul finalizat cu toate șabloane necesare implementate. Acum urmează propriu zis integrarea acestui sistem într-un produs real. În această lucrare voi implementă o aplicații de comunicare (chat).

Deoarece sistemul de mesaje nu are nici un mecanism de autentificare, vom permite oricui să acceseze orice canal de comunicare și să folosească orice nume de utilizator. Pentru a identifica fiecare utilizator în parte, vom genera câte un ID unic folosind Universal Unique Identifier. Prima fereastră va permite utilizatorilor să-și aleagă un nume. Exemplu de astfel de fereastră este reprezentat în figura 13.

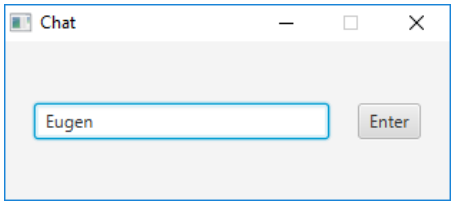

Figura 13 – Fereastra de autentificare

În momentul când utilizatorul se autentifică, aplicația creează un Exchange numit ”private-messages”, apoi un Queue cu același identificator ca și al utilizatorului. În final va crea un obiect de tip Bind între Exchange ”private-messages” și coada personală de așteptare, iar ca routingKey se va folosi același Id unic al utilizatorului. Acest Bind ne va permite să acceptăm mesajele private. În final, aplicația se va conecta la coadă și va începe să citească mesajele de intrare.

```
public void connect() throws IOException
{
    client.connect();
    client.declareQueue(id);

    try
    {
        client.declareExchange(PRIVATE_EXCHANGE);
    }
    catch (final IOException exception)
    {
        System.out.println("Private exchange is already created");
    }

    client.bind(PRIVATE_EXCHANGE, id, id);

    messageService.subscribe(id, id, onMessage::notifyObservers, String.class);
}
```

Următoare fereastră este afișată în figura 14 și reprezintă însuși aplicația de bază. La acest moment, utilizatorul are opțiunea de a se conecta la un anumit canal prin specificarea denumirii canalului.

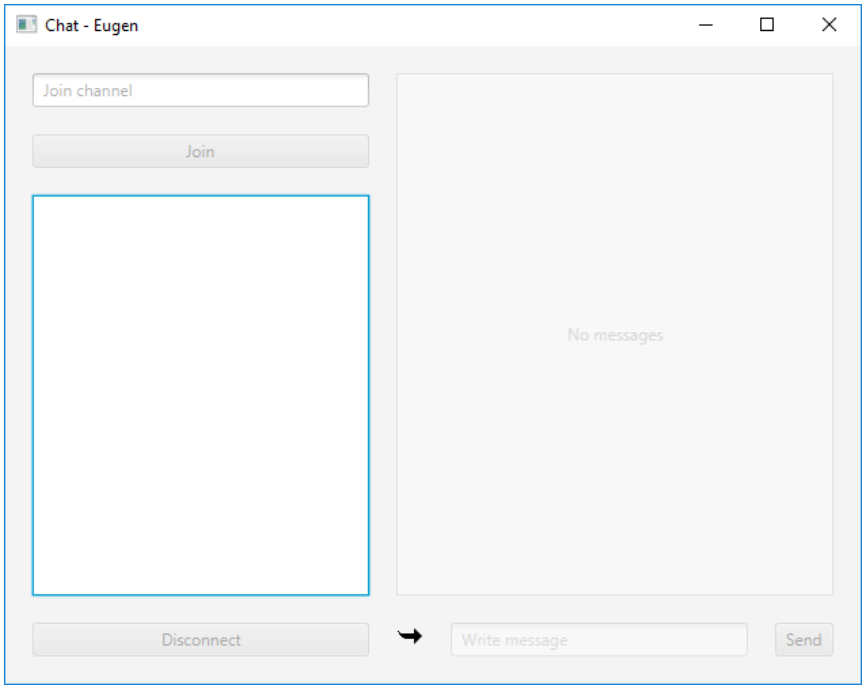

Figura 14 – Fereastra de bază a aplicației

În momentul când utilizatorul se conectează la un canal, se creează un Exchange nou cu aceeași denumire ca cea a canalului. Apoi se crează alt obiect de tip Bind între exchange-ul creat anterior și coada de așteptare a utilizatorului. Astfel toate mesajele trimise în acest canal vor fi redirecționate și către utilizatorul curent.

```
 public void joinChannel(final String channel) throws IOException
{
    try
    {
        client.declareExchange(channel);
    }
    catch (final IOException exception)
    {
        System.out.println("Exchange already created");
    }

    client.bind(channel, id, ".*");
}
```

După ce utilizatorul se conectează cu succes la canal, va avea posibilitatea să trimită și să primească mesaje. Exemple de mesaje sunt reprezentate în figura 15.

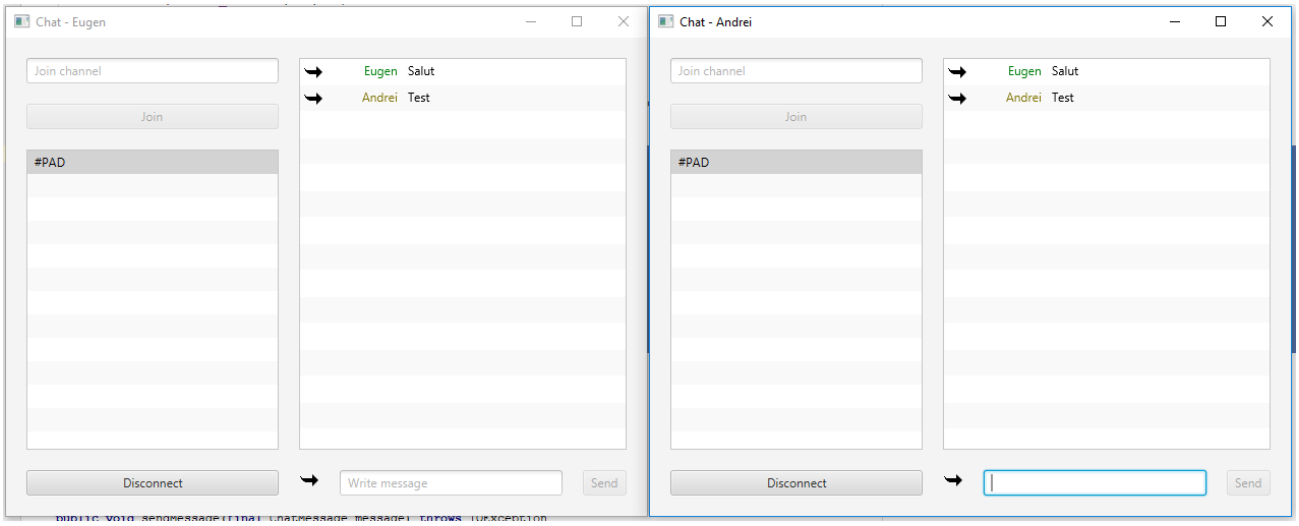

Figura 15 – Exemplu de mesaje în canal public

Mesajele sunt de trimise ca text folosind TextMessageTranslator, iar numele utilizatorului și id-ul acestuia sunt trimise folosind Enricher.

Trimiterea unui mesaj:

```
public void sendMessage(final ChatMessage message) throws IOException
{
    final String replyTo = message.getReplyTo() != null ? message.getReplyTo() :
UUID.randomUUID().toString();

    final Properties properties = new Properties();
    properties.setProperty(CONTENT_TYPE, APPLICATION_TEXT);
```

```
    properties.setProperty(CHANNEL, message.getChannel());

    final Message<String> request = new Message<>(message.getText(), properties,
replyTo);

    try
    {
        messageService.publish(message.getChannel(), request);
    }
    catch (final IOException exception)
    {
        messageService.publish(PRIVATE_EXCHANGE, request, id);

        if (!id.equals(message.getChannel()))
        {
            properties.setProperty(CHANNEL, id);
            messageService.publish(PRIVATE_EXCHANGE, request, message.getChannel());
        }
    }
}
```

Codul sursă al translatorului:

```
public class TextMessageTranslator extends AbstractMessageTranslator
{
    public TextMessageTranslator()
    {
        super("application/text");
    }

    @Override
    protected <T> byte[] translateData(final T object)
    {
        return object.toString().getBytes();
    }

    @Override
    @SuppressWarnings("unchecked")
    protected <T> T translateData(final byte[] data, final Class<T> type)
    {
        if (type != String.class)
        {
            throw new RuntimeException("Text translator can convert only string
objects");
        }
        return (T) new String(data, Charset.forName("utf8"));
    }
}
```

Codul sursă al MessageNameEnricher care populează numele și id-ul utilizatorului:

```
public class MessageNameEnricher extends GenericMessageEnricher<String>
{
    public static final String USER_ID = "user_id";
    public static final String USER_NAME = "user_name";

    private final String id;
    private final String name;

    public MessageNameEnricher(final String id, final String name)
    {
        super(String.class);
        this.id = id;
        this.name = name;
    }
```

```
    @Override
    protected Message<String> applyInternal(final Message<String> message)
    {
        message.getProperties().put(USER_ID, id);
        message.getProperties().put(USER_NAME, name);
        return message;
    }
}
```

Utilizatorii au posibilitatea de a răspunde la un mesaj prin apăsarea pe iconița mesajului. Această funcționalitate va folosi mecanismul Correlation definit anterior în cadrul lucrării. Exemplu de mesaje legate este reprezentat în figura 16. Se observă că iconița are diferite culori pentru fiecare grup de mesaje.

Figura 16 – Exemplu de corelație

La fel utilizatorii au posibilitatea de a inițializa un chat privat cu unul din utilizatori. Atunci când utilizatorul face click pe numele altei persoane, aplicația deschide un canal privat. Exemplu al acestei funcționalități este reprezentat în figura 17.

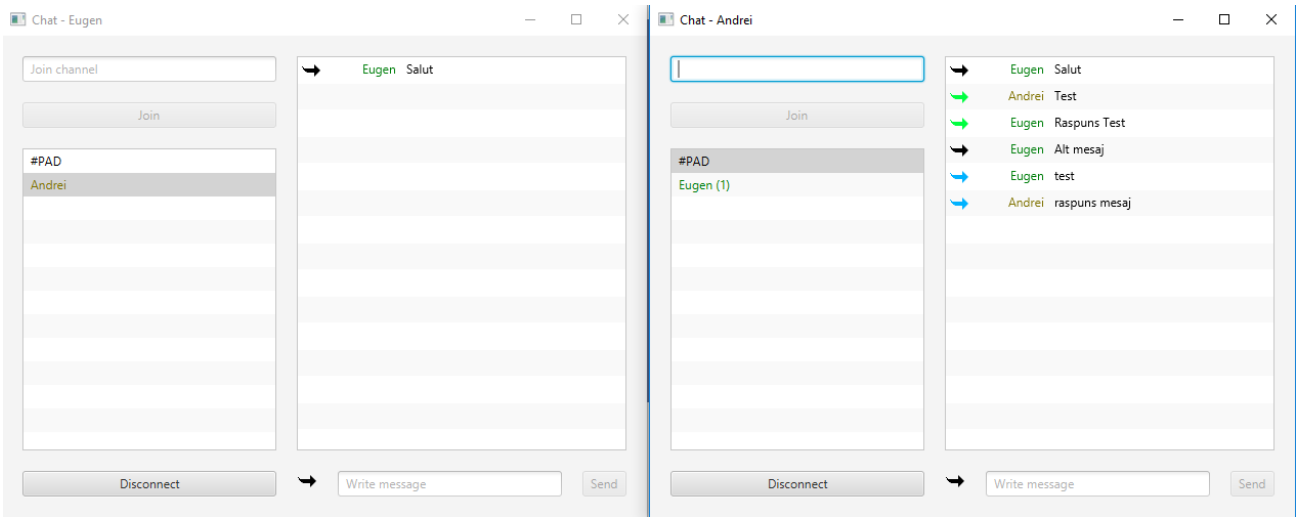

Figura 17 – Crearea unui canal privat.

În final utilizatorul poate părăsi un canal prin acționarea butonului Disconnect. Această operație va șterge obiectul Bind între exchange și coada personală a utilizatorului.

VI. Concluzii

Efectuând această lucrare de laborator am creat un sistem de mesaje (message broker) care permite comunicarea asincronă dintre componentele unui sistem distribuit.

Sistemul de mesaje lucrează cu două tipuri de clienți: producători care creează și trimit mesajele către sistem și consumatori care se conectează la cozile de așteptare și citesc mesajele. De obicei în practică o componentă a unui sistem distribuit poate avea ambele roluri, respectiv consumă mesajele generate de alte componente și la rândul său publică mesaje pentru alte componente.

Un sistem de mesaje își asumă următoarele responsabilități: primește mesajele produse de clienți, redirecționează mesajele în dependență de o configurare prestabilită, salvează mesajele în cozi de așteptare, trimite mesajele din cozi către clienți atunci când aceștia se abonează la o coadă de așteptare, asigură faptul că mesajul a fost consumat cu succes de client și îl exclude din coada de așteptare.

De obicei un sistem de mesaje definește un protocol de comunicare între server și clienți, respectiv ar trebui să aibă un SDK care suportă acest protocol. Această bibliotecă eventual va fi integrată în componentele sistemului și va permite comunicarea cu serverul. Tot aici putem implementa și careva șabloane noi care ar adăuga responsabilități sistemului, fără a afecta propriu zis serverul. Careva din aceste responsabilități ar fi translarea mesajelor sau completarea acestora în dependență de conținut.