Технический Университет Молдовы.

Кафедра информационных технологий.

Отчет

по лабораторной работе №1

по предмету:

Программирование распределенных приложений.

на тему:

“Агент обмена сообщениями – Message Broker”

Подготовил: ст.гр.TI-144 Зубенко Дмитрий

Проверил преподаватель: Гавришко Александр

Кишинев 2017

Тема

Агент обмена сообщениями – Message Broker.

Задание

На рисунке 1 представлена диаграмма, которая иллюстрирует работу Message Broker-a.

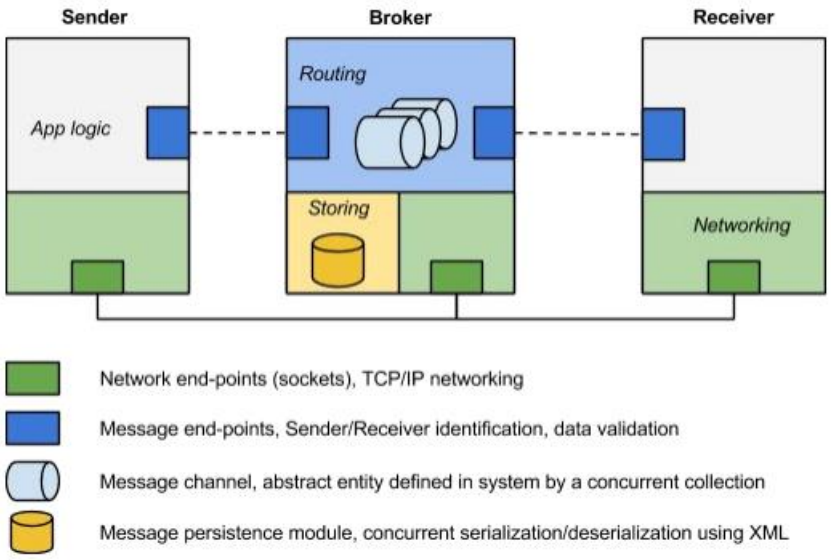

Рисунок 1 – Диаграмма Message Broker

1. Определение рабочего протокола агента обмена сообщениями

a) Формат (тип) отправляемых сообщений. Рекомендуется использовать формат XML;
b) Количество однонаправленных каналов (изменяемое/фиксированное, в зависимости от типа сообщений, etc.);
c) Структура общения, предоставляемая агентом (один к одному или один ко многим);
d) Политики доставки для различных случаев, определенные логикой работы агента (невалидные сообщения, падение агента и т.д.);

2. Разработка абстрактного слоя общения (на уровне сети), необходимого элементам для отправки/получения сообщений между отправителем, агентом и получателем

a) Транспортный протокол выбирается в зависимости от целей рабочего протокола;
b) Параллельная обработка запросов;

3. Разработка элементов, обеспечивающих хранение полученных сообщений

a) Переходный метод: сообщения будут храниться в потокобезопасных коллекциях в зависимости от выбранного языка программирования;
b) Постоянный метод: сообщения будут сериализоваться\десериализоваться асинхронными либо параллельными методами;

4. Разработка абстрактного слоя маршрутизации сообщений

Агент обмена сообщениями Message Broker

Агент обмена сообщениями (message broker eng.) это физическая компонента, которая управляет общение между компонентами распределенного приложения. Преимущество использования такой техники состоит в разъединении отправителя от получателя сообщений. Таким образом приложение-участник отправляет сообщения только агенты, указывая логическое название получателя. Агент может предложить различные интерфейсы приложениям для общения и может пересылать сообщения между ними, не обязывая использовать один общий интерфейс всем участникам во благо обеспечения лучшего взаимодействия. Ответственности и основные положения агента обменами сообщений приведены в таблице ниже. Рисунок 2 демонстрирует диаграмму Broker-а.

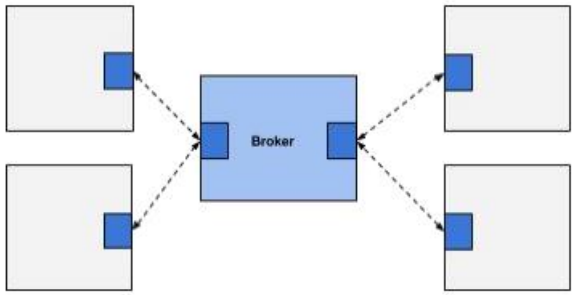

Рисунок 2 – Диаграмма Broker-а

Решение использовать агент обмена сообщениями для интеграции в приложения находится между гибкостью, полученной через разъединение участников, и усилиями, необходимыми для обслуживания агента:

1) Преимущества

a) Уменьшение связности участники общаются только с агентом, таким образом потенциальная группировка большего числа получателей под одним логическим именем может стать прозрачной для всех участников;

b) Увеличивает интегрируемость приложения, которые общаются с агентом не должны иметь один и тот же интерфейс, таким образом агент

может стать мостом между приложениями с разными слоями безопасности и качества сервисов QoS;

c) Улучшается расширяемость агент защищает компоненты от индивидуальных модификаций интегрированных приложений, часто предоставляя возможности для динамической конфигурации;

2) Ограничения

a) Растет сложность агент, общаясь со всеми участниками, должен реализовать множество интерфейсов (протоколов) и в перспективе производительности использует многопоточную обработку;

b) Растут необходимые для поддержки усилия все участники должны быть зарегистрированы (подписаны) у агента и необходим механизм их идентификации;

c) Снижается доступность одна компонента, которая является промежуточным слоем общения, является единственной точкой отказа (single point of failure eng.), ее падение приводит к блокировке деятельности всей системы; эта проблема исправляется репликацией агента и синхронизацией состояний основного агента и агента репликатора;

d) Снижается производительность агент обмена сообщениями добавляет один дополнительный шаг, который влечет за собой дополнительные затраты вычислительной мощности (overhead eng.).

Считая, что агент это обобщённая форма для посредничества между распределёнными компонентами, предлагаются реализации маршрутизации или построения сообщения в соответствии с шаблонами, упомянутыми Gregor Hohpe, объединенные в несколько групп:

a. Messaging Systems,

b. Messaging Channels,

c. Message Constructions,

d. Message Routing,

e. Message Transformation,

f. Messaging endpoints,

g. System management.

Message Channel (Messaging Systems)

Канал обмена сообщениями это логический элемент, использованный для соединения между приложениями. Одно приложение отправляет по этому каналу сообщения, а другое читает их. Таким образом этот метод соединения один из первоначальных, а очередь сообщений представляет собой модель реализации. На рисунке 3 изображена диаграмма данного шаблона.

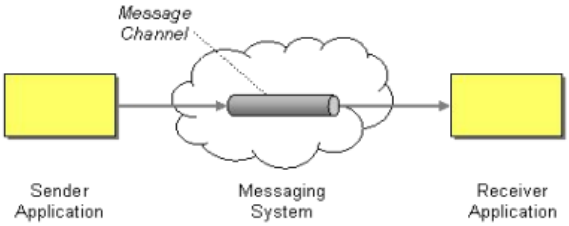

Рисунок 3 – Диаграмма Message Channel

Message Translator (Messaging Systems)

Транслятор сообщений используется для преобразования сообщений из одного формата в другой. например, одно приложение отправляет сообщения в формате XML, а другое получает только в формате JSON или в другом формате XML. На рисунке 4 изображена диаграмма данного шаблона.

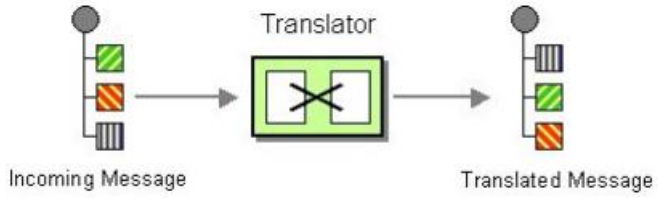

Рисунок 4 – Диаграмма Message Translator

Publish-Subscribe Channel (Messaging Channels)

Этот тип канала транслирует события или оповещает всех подписанных получателей. Очевидно противопоставляется каналу один к одному. Каждый подписчик получает один раз сообщение после чего он исключается из системы. На рисунке 5 изображена диаграмма данного шаблона.

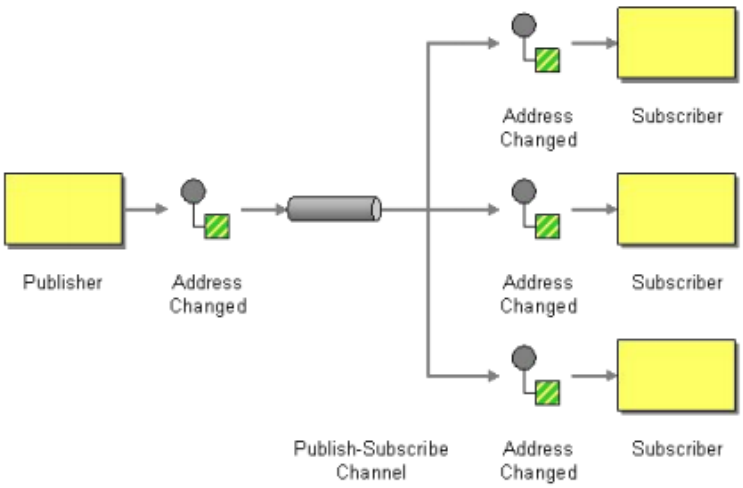

Рисунок 5 – Диаграмма Message Translator

Dead Letter Channel (Messaging Channels)

Канал Неотправленных Сообщений описывает сценарий, по которому система отправки сообщений определяет, как поступать в случае, если сообщение не может быть доставлено определённому получателю. Этот факт может быть вызван проблемой соединения или ошибкой нехватки места для хранения. В обычном режиме есть множество попыток отправки сообщения, следующие после его отправки в канал Неотправленных Сообщений. На рисунке 6 изображена диаграмма данного шаблона.

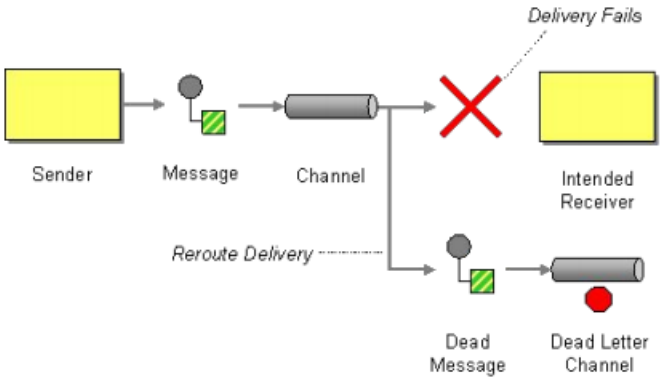

Рисунок 6 – Диаграмма Dead Letter Channel

Correlation Identifier (Message Construction)

Взаимосвязь идентификаторов предоставляет возможность сопоставлять сообщения запросов и ответов в системе асинхронной отправки сообщений при помощи добавления сообщения (или заголовка) идентификатора взаимосвязи. На рисунке 7 изображена диаграмма данного шаблона.

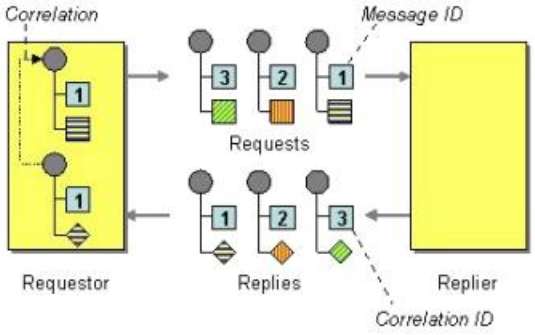

Рисунок 7 – Диаграмма Correlation Identifier

Content-Based Router (Message Routing)

Маршрутизатор, основанный на содержании, исследует сообщение и направляет его в зависимости от полученных данных из сообщения. На рисунке 8 изображена диаграмма данного шаблона.

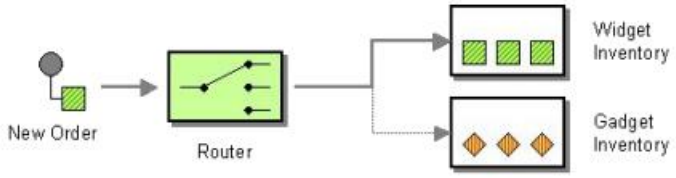

Рисунок 8 – Диаграмма Content-Based Router

Content Enricher (Message Transformation)

Этот тип преобразования заполняет сообщение недостающими данными, в таких случаях часто используются внешние источники данных. На рисунке 9 изображена диаграмма данного шаблона.

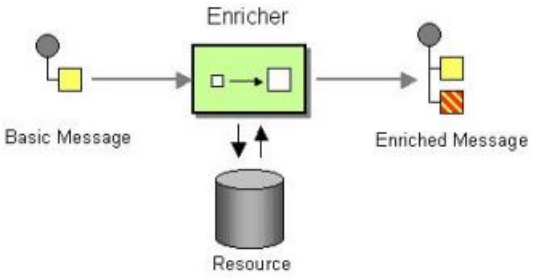

Рисунок 9 – Диаграмма Content Enricher

Event-Driven Consumer (Messaging Endpoints)

Потребитель, основанный на событиях, подразумевает существование деятельности, ассоциируемой каналу, по которому она пришел.

Асинхронность такой обработки особенно выделяется. На рисунке 10 изображена диаграмма данного шаблона.

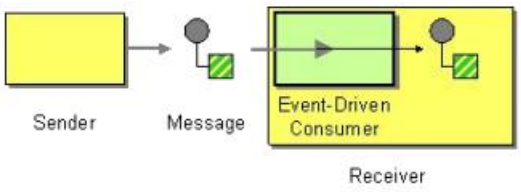

Рисунок 10 – Диаграмма Event-Driven Consumer

Polling Consumer (Messaging Endpoints)

Потребитель такого типа будет ждать сообщение, обработает его и перейдет к следующему сообщению. Таким образом этот шаблон синхронный по определению, т.к. будет блокировать канал до появления следующего сообщения. На рисунке 11 изображена диаграмма данного шаблона.

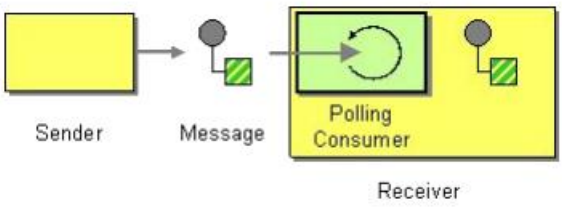

Рисунок 11 – Диаграмма Polling Consumer

Wire Tap (System Management)

Канал наблюдения копирует сообщение из канала и передает его в специально назначенный с целью его исследования или внешнего анализа. На рисунке 12 изображена диаграмма данного шаблона.

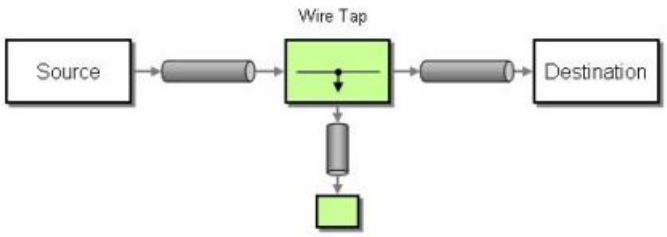

Рисунок 12 – Диаграмма Polling Consumer

Выполнение

Приложение было реализовано на языке JAVA в среде разработки INTELIJIDEA. Использовался TCP протокол для передачи сообщений. Был создан package xPlatfom, в который вошли интерфейс IOperation, который реализован был BrokerService-ом и TransportService-ом, класс DataParserManager для чтения из xml, класс Letter для будущего создания List из писем, полученных от Sender-ов, класс Receiver для будущего создания List из идентификаторов получателей, которые были подключены к Broker-у, а также ReceiverMessageReaderThread для чтения писем, посланных Broker-ом, от лица Receiver-а. Также реализованы 3 package-а:

- Broker, использующий BrokerService;
- Receiver, использующий TransportService;
- Sender, использующий TransportService.

TransportService и BrokerService реализуют 2 метода IOperation (readAsync() и writeAsync()).

Sender отправляет сообщения Broker-у в которых содержатся 1 или более имен receiver-ов и само сообщение в xml формате, которое в будущем поместится в Broker-е в letterList. Receiver отправляет сообщение Broker-у “connect name” для подключения и идентификации себя в receiverList. Для отключения от Broker-а, receiver-у необходимо отправить сообщение “disconnect name”. Broker определяет сообщение и распоряжается в соответствии с ним. Рисунки 13, 14, 15 демонстрируют пример работы Message Broker-а.

```
SENDER OPTIONS:
--Connection succeeded--
--Input the receivers--
john
bill
john
alan
.
Input the message: hello, 123456789
--Serialized data in XML--
<message text="hello, 123456789">
    <receiver>john</receiver>
    <receiver>bill</receiver>
    <receiver>john</receiver>
    <receiver>alan</receiver>
</message>

--Data succesfully transmitted--

Process finished with exit code 0
```

Рисунок 13 – Sender операции

```
RECEIVER OPTIONS:
--Input ur name--
john
--Input "connect" command to be connected to broker--
connect
Connected to broker
--Input "disconnect" command to be disconnected from broker--
hello, 123456789
hello, 123456789
disconnect
Disconnected from broker

Process finished with exit code 0
```

Рисунок 14 – Receiver операции

```
BROKER OPTIONS:
--Received data from buffer--
<message text="hello, 123456789"><receiver>john</receiver><receiver>bill</receiver><receiver>john</receiver><receiver>ala
--Parsed data--
Message from method : hello, 123456789
Receivers : [john, bill, john, alan]
---Receiver List:---
--Letters List--
Name:john, Message text: hello, 123456789
Name:bill, Message text: hello, 123456789
Name:john, Message text: hello, 123456789
Name:alan, Message text: hello, 123456789
--Broker loop--
--VALID MESSAGE--
--Received data from buffer--
connect john
parsed receiver: john to be connected
---Receiver List:---
Name: john, Socket: Socket[addr=/127.0.0.1,port=61842,localport=1488]
--Letters List--
Name:john, Message text: hello, 123456789
Name:bill, Message text: hello, 123456789
Name:john, Message text: hello, 123456789
Name:alan, Message text: hello, 123456789
--Broker loop--
--VALID MESSAGE--
--Receiver name and Letter name adress MATCH--
Receiver name: john
Letter name: john
Message: hello, 123456789
 was transmitted to john succesfully...
--Receiver name and Letter name adress MATCH--
Receiver name: john
Letter name: john
Message: hello, 123456789
 was transmitted to john succesfully...
```

Рисунок 15 – Broker операции

Как видно из скриншотов работы приложения, все сообщения, отправленные John-у были успешно доставлены ему, как receiver-у.

Вывод

Выполнив данную лабораторную работу, были изучены основы работы Message Broker-а с использованием TCP протокола передачи данных на примере, реализованном на JAVA приложения с использованием XML.

Приложение

IOperation.java

```
public interface IOperation {
    public String readAsync();
    public void writeAsync(String message);
}
```

BrokerService.java

```
public class BrokerService implements IOperation {
    private ServerSocket serverSocket;                    //Server Socket(broker)
    private List<Letter> letterList;                    //List of letters(name+message)
    private List<Receiver>receiversList;                  //List of active receivers
    public BrokerService() {
        letterList =new ArrayList<>();                  //List creating
        receiversList=new ArrayList<>();                  //List creating
        try {
            serverSocket = new ServerSocket(1488);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @Override
    public String readAsync() {
        Socket connectionSocket = null;
        Consumer<Receiver> styleRec = (Receiver p) -> System.out.println("Name:
"+p.getName() +", Socket: "+p.getSocket());
        Consumer<Letter> printLetterConsumer= (Letter l)->
System.out.print("Name:"+l.getName()+", Message text: "+l.getMessage());
        try {
            connectionSocket = serverSocket.accept();
        } catch (IOException e) {
            e.printStackTrace();
        }
        Socket finalConnectionSocket = connectionSocket;              //  Receiver's
socket
        ExecutorService executor = Executors.newSingleThreadExecutor(); //  Thread
Executor

        Callable<String> task=new Callable<String>() {
            public String call() {
                InputStream istream = null;
                StringBuffer result = new StringBuffer();
                try {
                    istream = finalConnectionSocket.getInputStream();
```

```
                BufferedReader receiveRead = new BufferedReader(new
InputStreamReader(istream));
                String partlyTransData;
                while(!(partlyTransData=receiveRead.readLine()).isEmpty())
                   result.append(partlyTransData.trim());
             } catch (IOException e) {
                e.printStackTrace();
             }
             String message= result.toString();
             String answer="valid";
             System.out.println("--Received data from buffer--");
             System.out.println(message);
             DataParserManager xml=new DataParserManager(message);
             if(message.length()>=9 && message.substring(0,8).equals("connect ")) {
                String name=message.substring(8,message.length());
                System.out.println("parsed receiver: "+name+" to be connected");
                receiversList.add(new Receiver(finalConnectionSocket,name));
             }else if(message.length()>=12 &&
message.substring(0,11).equals("disconnect ")) {
                String name=message.substring(11,message.length());
                System.out.println("parsed receiver: "+name+" to be disconnected");

                letterList.add(new Letter(name,"disconnect\n"));

             }else if(xml.CheckIfXml()) {
                System.out.println("--Parsed data--");
                String msg=xml.getMessage();
                System.out.println("Message from method : "+msg);
                List<String>rec=xml.getReceivers();
                System.out.println("Receivers : "+rec);
                for(int i=0;i<rec.size();i++)
                   letterList.add(new Letter(rec.get(i),msg+"\n"));
             }else{
                answer="invalid";
                System.out.println("Message is not valid");
                int port=finalConnectionSocket.getPort();
                for (int i=0;i<receiversList.size();i++)
                   if(receiversList.get(i).getSocket().getPort()==port)
                   {
                      letterList.add(new
Letter(receiversList.get(i).getName(),"IDIOT"+"\n"));
                      break;
                   }
             }
```

```
                System.out.println("---Receiver List:---");
                receiversList.forEach(styleRec);
                System.out.println("--Letters List--");
                letterList.forEach(printLetterConsumer);
                return answer;
            }
        };
        Future<String> future = executor.submit(task);
        String message = null;
        while(!future.isDone());
        try{
            message=future.get();
        } catch (InterruptedException ie) {
            ie.printStackTrace(System.err);
        } catch (ExecutionException ee) {
            ee.printStackTrace(System.err);
        }
        executor.shutdown();
        return message;
    }
    @Override
    public void writeAsync(String message) {
        BiConsumer<Receiver, Letter> recLetterMatch = (receiver, letter) -> {
            if(receiver.getName().equals(letter.getName()))
                try {
                    OutputStream ostream = receiver.getSocket().getOutputStream();
                    PrintWriter pwrite = new PrintWriter(ostream, true);
                    String messageToRec=letter.getMessage();
                    pwrite.println(messageToRec);    // sending to server
                    pwrite.flush();              // flush the data
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    letter.setSent(true);
                    System.out.println("--Receiver name and Letter name adress
MATCH--");
                    System.out.println("Receiver name: "+receiver.getName());
                    System.out.println("Letter name: "+letter.getName());
                    System.out.println("Message: "+letter.getMessage()+" was transmitted
to "+letter.getName()+" succesfully...");
                    if (letter.getMessage().equals("disconnect\n"))
                    {
```

```
                receiver.setDisconnected(true);
                receiver.getSocket().close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    };
    Runnable r=new Runnable() {
        @Override
        public void run() {
            for (int i=0;i<letterList.size();i++)
            {
                int finalI = i;
                receiversList.forEach(a ->
recLetterMatch.accept(a,letterList.get(finalI)));
            }
            letterList.removeIf(l-> l.isSent());
            receiversList.removeIf(r->r.isDisconnected());
        }
    };
    Thread t=new Thread(r);
    t.start();

  }
}
```

TransportService.java

```
public class TransportService implements IOperation{
   Socket transport;                    //127.0.0.1 port 1488
   public TransportService(Socket transport) {
      this.transport = transport;
   }
   @Override
   public String readAsync() {
      ExecutorService executor = Executors.newSingleThreadExecutor(); //  Thread
Executor
      Callable<String> task = new Callable<String>() {
         public String call() {
            InputStream istream = null;
            String partlyTransData;
            StringBuffer result = new StringBuffer();
            try {
               istream = transport.getInputStream();
```

```
                BufferedReader receiveRead = new BufferedReader(new
InputStreamReader(istream));
                if (!(partlyTransData = receiveRead.readLine()).isEmpty())
                    result.append(partlyTransData.trim());
            } catch (IOException e) {
                e.printStackTrace();
            }
            return result.toString();
        }
    };
    Future<String> future = executor.submit(task);
    String message="";
    while(!future.isDone());
    try{
        message=future.get();
    } catch (InterruptedException ie) {
        ie.printStackTrace(System.err);
    } catch (ExecutionException ee) {
        ee.printStackTrace(System.err);
    }
    executor.shutdown();
    return message;
  }
  @Override
  public void writeAsync(String message) {
    Thread thread =new Thread(new Runnable() {
        @Override
        public void run() {
            OutputStream ostream = null;
            try {
                ostream = transport.getOutputStream();
                PrintWriter pwrite = new PrintWriter(ostream, true);
                pwrite.println(message);    // sending to server
                pwrite.flush();             // flush the data
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
    thread.start();
  }
}
```