4.2. Простые протоколы на сокетах

Простейший чат с многопоточным сервером и асинхронными сообщениями

Рассмотрим простейший пример, в котором отсутствуют детали, дабы не загромождать сути. Приложение – сервер устанавливает постоянные соединения с клиентами, передающие ему строки сообщений. Принятое сообщение рассылается всем подключенным клиентам, кроме источника.

Основные технологические решения для сервера

Клиенты являются инициаторами соединений через объекты **Socket,** сервер использует **отдельный поток** для ожидания входящих запросов на соединения на объекте **ServerSocket**;

На каждое установленное соединение сервер запускает отдельный поток (в объекте класса соединения - **Connect**), который получает ссылку на открытый **Socket** и открывает на нем два **двоичных потока данных (is,os).** Строки текста передаются в потоках стандартными методами **writeUTF**, **readUTF** в виде последовательностей символов в формате **UTF** (формат кодирования, в сущности, не важен, важно само наличие этих методов в данных классах).

```
//------------------------------------------------------------------------------ 42_MultiThreadChat
   class Connect extends Thread{          // Класс - поток, управляющий соединением
      Socket sk=null;                     // Сокет соединения
      DataInputStream is=null;            // Двоичные потоки данных на сокете
      DataOutputStream os=null;
      Connect(Socket sk0){
         try {                            // Запомнить полученный сокет
            sk=sk0;                       // и открыть на нем потоки даннных
            is=new DataInputStream(sk.getInputStream());
            os=new DataOutputStream(sk.getOutputStream());
            synchronized (list){          // Добавить СЕБЯ в вектор
               list.add(this);
               }
            start();                      // Стартануть собственный поток - run
            out.append("Порт "+sk.getPort()+
                                 ", соединение установлено\n");
            } catch (Exception ex){out.append("1:"+ex.toString()+"\n");}
         }
      public void close(){                // Закрыть соединение
         synchronized (list){
            try {
               list.remove(this);         // Удалить себя из вектора
               out.append("Порт "+sk.getPort()+", соединение закрыто\n");
               sk.close();                // Закрыть сокет и потоки данных
               sk=null;
               is=null;
               os=null;
               } catch(Exception ee){}
            }
         }
```

Главный поток (анонимный класс со ссылкой **listen)** содержит цикл ожидания соединения на **ServerSocket**, для каждого открытого сокета создается объект-поток **Connect**, который получает ссылку на этот сокет. Все установленные соединения сохраняются в векторе **list** в виде ссылок на объекты **Connect;**

```
//------------------------------------------------------------------------------ 42_MultiThreadChat
```

```
Vector<Connect> list=new Vector();                  // Вектор объектов - соединений
   ServerSocket srv=null;                           // Объект для ожидания соединений
   Thread listen=new Thread(){                      // Поток для ожидания соединений
      public void run(){
         try {
            srv=new ServerSocket(Server.port);
            while(srv!=null){                       // Цикл ожидания
               Socket ss=srv.accept();              // Получить очередное соединение
               new Connect(ss);                     // Создать объект-соединение
               }                                    // и передать ему Socket
            } catch (Exception ex) {
                              out.append("3:"+ex.toString()+"\n"); }
                  }
      };
```

Каждый поток класса **Connect** ожидает получения строки от клиента через собственный двоичный поток данных, затем просматривает вектор **list,** выбирает в нем объекты соединений **Connect** и передает сообщение клиентам через двоичные потоки данных **os** в этих объектах;

```
//------------------------------------------------------------------------------ 42_MultiThreadChat
    public void run(){    // Функционал потока - прием и рассылка сообщений
      try {
         while(true){
                  String ss=is.readUTF();  // Читать строку из входного потока данных
            if (ss.length()==0){           // Пустая строка - сообщение о разрыве соединения
                  close();
                  break;
                  }
            ss="Порт "+sk.getPort()+","+ss;
            out.append(ss);
            synchronized (list){           // Для всех объектов - соединений
                  for (int i=0;i<list.size();i++){
                    Connect cc=list.get(i);
                    if (cc==this) continue;
                    cc.os.writeUTF(ss);    // Записать строку в поток соединения
                    }
                  }
            }
          } catch (Exception ex) {
            out.append("2:"+ex.toString()+"\n");
            close();
            }
      }
```

Поскольку несколько потоков **Connect** посылают строки в одни и те же потоки данных, то процессы записи в них необходимо синхронизировать (иначе вывод одной строки может быть прерван выводом другой по схеме «начало строки 1 – строка-2 – окончание строки 1»). В качестве объекта синхронизации логично использовать сам объект **Connect,** т.к. он является разделяемым ресурсом при выводе сообщений. Однако при создании и разрыве соединений необходимо также синхронизироваться к вектору **list** в целом. В целях упрощения разумно все события в сервере синхронизировать на объекте **list.** В результате возникнут необоснованные задержки (поток будет ждать освобождения вектора **list** в целом, хотя в данный момент времени выводом занят только один поток данных);

Пустая строка является сообщением о необходимости разорвать соединение. При закрытии приложения сервер рассылает всем клиентам пустую строку и закрывает все соединения. Напомним, что процедура закрытия является двунаправленной и симметричной: инициатор разрыва посылает сообщение о разрыве и закрывает

соединение, причем сообщение «уходит» в сеть. Приемная сторона, получив сообщение, закрывает соединение со своей стороны.

```
//-------------------------------------------------------------------------------- 42_MultiThreadChat
private void formWindowClosing(java.awt.event.WindowEvent evt) {
    synchronized (list){              // Для всех объектов - соединений
      while(list.size()!=0){
         Connect cc=list.get(0);
         try {             //
            cc.os.writeUTF("");
            cc.os.flush();
            }  catch(Exception ee){
         cc.stop();                   // Остановить поток
         cc.close();                  // Закрыть в объекте сокет и потоки данных
         }                            // и удалить из вектора
      }
    listen.stop();
    }
}
```

Технологические решения для клиента

Строки, получаемые от сервера, являются асинхронными, поскольку инициатором их передачи является сервер. Поэтому для их приема необходим отдельный поток на основе вложенного класса **Listen**, который в цикле ожидает ввода строки в потоке данных и выводит ее в текстовое поле формы.

```
//-------------------------------------------------------------------------------- 42_MultiThreadChat
   class Listen extends Thread{
      public void run(){
         String ss="";
         try {
            while(true){
              ss=is.readUTF();
              if (ss.length()==0) { close(); break; }
              else
                  synchronized (out) { out.append(ss); }
               }
            } catch (Exception ex)
               { out.append("1:"+ex.toString()+"\n"); close(); }
         }
      }
```

Оконный класс создает и уничтожает соединение на сокете и потки данных на нем, а также поток приема класса **Listen.** Вводимые в форме строки обработчиком события записываются в поток передаваемых данных.

```
//-------------------------------------------------------------------------------- 42_MultiThreadChat
private void button1ActionPerformed(java.awt.event.ActionEvent evt) {
      try {
         synchronized (out){
            String ss=in.getText();
            if (ss.length()==0) return;
            if (os==null) return;
            os.writeUTF(ss+"\n");
            os.flush();
            }
      } catch (Exception ex) { out.append(ex.getMessage()+"\n"); close(); }
         }
```

Для синхронизации потока приема и потока GUI (оконного класса, передающего сообщения) можно использовать любой объект головного (оконного класса), в данном случае это - поле вывода **out** формы.

Простейший чат с короткими соединениями и синхронным опросом сервера клиентами

В предыдущем примере использовались постоянные (долгоживущие) соединения, каждое из которых в сервере управлялось отдельным потоком. Вместо того, чтобы поддерживать большое число открытых сокетов и управляющих потоков с низкой интенсивностью обмена, можно предложить вариант, когда соединение устанавливается каждый раз для выполнения короткого обмена данными – **транзакции.** Тогда серверное приложение должно сохранять полученные от клиентов строки в порядке их поступления, а клиенты – использовать транзакции двух видов:

- для передачи собственного сообщения на сервер;
- для опроса сервера и получения от него сообщений, накопленных от момента предыдущего опроса. Опрос сервера с клиентом производится с частотой, необходимой для поддержки требуемой реактивности системы, например, 1 раз в минуту. Алгоритм опроса может быть усложнен, например, частота опроса может повышаться при поступлении сообщений и постепенно снижаться до минимума при их отсутствии.

Такой способ является единственно возможным, если сервер не имеет возможности посылать клиенту асинхронные сообщения, например, в протоколе **http**, который является полностью асимметричным.

Для реализации этой схемы сервер должен нумеровать и сохранять поступающие сообщения. Для этой цели можно использовать циклический буфер, в котором **последовательный номер** сообщения совпадает с его **индексом**. Клиенту необходимы следующие команды протокола:

- **сброс –** команда не имеет параметров, сервер возвращает индекс очередного ожидаемого сообщения и размер циклического буфера;
- **передача сообщения –** команда содержит передаваемую строку, в ответ сервер возвращает ее индекс в очереди;
- **опрос принятых сервером сообщений –** клиент передает индекс ожидаемого (следующего после принятого) сообщения, сервер возвращает счетчик (количество) принятых сообщений от указанного до последнего и последовательность из самих сообщений. Клиент, принимая сообщения, выводит их и корректирует у себя индекс ожидаемого.

Размер циклического буфера сообщений должен быть установлен таким, чтобы количество принимаемых сообщений за время максимального периода опроса клиентом не превышало его размерности (иначе возможны потери сообщений клиентом за счет «полного оборота» циклического буфера).

Как и ожидалось, функциональная простота сервера приводит к простой структуре программного кода: поток – цикл обработки транзакций – ожидание соединения - прием команды – выполнение действий – закрытие соединения. Обслуживание транзакций в отдельном потоке сделано с целью исключения зависания GUI.

```
//-------------------------------------------------------------------------------- 42_TransactionChat
public class Server extends javax.swing.JFrame {
   final static int qsize=1000;              // Размер циклической очереди
   final static int port=6100;
   final static String ip="127.0.0.1";
   String queue[]=new String[qsize];         // Циклическая очередь
   int idx=0;                                // Индекс номера очередного сообщения
   Socket sk=null;                           // Сокет соединения
   DataInputStream is=null;                  // Двоичные потоки данных на сокете
   DataOutputStream os=null;
   ServerSocket srv=null;                    // Объект для ожидания соединений
   Thread listen=new Thread(){               // Поток обработки транзакций
      public void run(){
         try {
            srv=new ServerSocket(Server.port);
            while(true){                     // Цикл ожидания
               sk=srv.accept();              // Получить очередное соединение
               is=new DataInputStream(sk.getInputStream());
               os=new DataOutputStream(sk.getOutputStream());
               byte code=is.readByte();
               out.append("Порт: "+sk.getPort()+" команда: "+code+"\n");
               switch(code){                 // Сброс - возвратить индекс ожидаемого
                                             // и размер очереди
case 0:           os.writeInt(idx);
                  os.writeInt(qsize);
                  break;
                                             // Прием сообщения от клиента
case 1:           queue[idx]=is.readUTF();
                  out.append(queue[idx]+"\n");
                  os.writeInt(idx);          // записать в очередь
                  idx++;                     // возвратить индекс записанного
                  if (idx==qsize) idx=0;
                  break;                     // Индекс - к следующему (в цикле)
                                             // Опрос накопленных сообщений
case 2:           int oidx=is.readInt();     // Индекс первого не принятого
                  int cnt=idx-oidx;          // Количество накопленных
                  if (cnt<0)                 // с учетом цикличности
                     cnt=qsize-oidx+idx;
                  os.writeInt(cnt);          // Количество накопленных
                  while(cnt--!=0){           // Передать накопленные
                        os.writeUTF(queue[oidx]);
                        oidx++;
                        if (oidx==qsize) oidx=0;
                     }
                  break;                     // Индекс - к следующему (в цикле)

               }
               sk.close();                   // Закрыть сокет и потоки данных
               sk=null;
               is=null;
               os=null;
               }
            }catch (Exception ex){out.append("1:"+ex.toString()+"\n"); }
         }
      };
```

В клиентском приложении исполнение транзакций вынесено в отдельный метод **transaction** для удобства их синхронизации между собой. Периодический опрос сервера реализован в отдельном потоке **Listen,** остальные транзакции выполняются непосредственно в потоке GUI.

```
//------------------------------------------------------------------------------- 42_TransactionChat
public class Client extends javax.swing.JFrame {
   int oidx=0;                      // Индекс первого неполученного сообщения
   int qsize=0;                     // Размер очереди на сервере
   int sidx=-1;                     // Индекс переданного сообщения в очереди сервера
   class Listen extends Thread{  // Поток периодического опроса
      Listen(){ start(); }
      public void run(){
         try {
            while(true){           // Вечный цикл опроса
               sleep(2000);        // Спать 2 сек.
               transaction(2);     // Выполнить команду 2 - опрос сервера
               }
            } catch (Exception ex)
               { out.append("1:"+ex.toString()+"\n"); close(); }
         }
      }
   //-----------------------------------------------------
   Listen listen=null;
   Socket sk=null;
   DataInputStream is=null;
   DataOutputStream os=null;
   //---------- Выполнить транзакцию - соединение + команда
   public void transaction(int code){
      synchronized (out){
         try {
            sk=new Socket(Server.ip,Server.port);
            is=new DataInputStream(sk.getInputStream());
            os=new DataOutputStream(sk.getOutputStream());
            os.writeByte(code);
            switch(code){
               // Команда 0  - получить индекс очередного принимаемого и размер буфера
case 0:         oidx=is.readInt();
               qsize=is.readInt();
               out.append("Сброс: id="+oidx+" size="+qsize+"\n");
               break;
               // Команда 1 - передать строку сообщения и получить ее индекс
case 1:         os.writeUTF(in.getText());
               sidx=is.readInt();
               out.append("Передано: id="+sidx+"\n");
               break;
               // Команда 2 - опрос сервера
case 2:        os.writeInt(oidx);               // передать индекс ожидаемого
               int cnt=is.readInt();            // получить количество накопленных
               while(cnt--!=0){                 // принять накопленные в цикле
                  if (oidx!=sidx)               // отображать только чужие
                     out.append(""+oidx+","+is.readUTF()+"\n");
                  oidx++;                       // корректировать индекс ожидаемого
                  if (oidx==qsize) oidx=0;
                  }
               break;
               }
            close();
            } catch (Exception ex)
               { out.append("1:"+ex.toString()+"\n"); close(); }
         }
      }
    public void close(){               // Закрыть соединение
```

```
 try {
    if (sk==null) return;
    sk.close();                    // Закрыть сокет и потоки данных
    sk=null;
    is=null;
    os=null;
    } catch(Exception ee){}
}
```