**Лекция 3**

Следующие разделы курса посвящены различным программным технологиям, позволяющим реализовать распределенные системы. Наилучшей иллюстрацией использования этих технологий была бы программная система, созданная на их основе. Тут, однако, существует следующая опасность: если эта разрабатываемая система будет слишком сложна, то за ее сложностью может потеряться изучаемый предмет, если же она будет слишком проста, то может возникнуть вопрос, зачем для решения такой простой задачи применяются столь изощренные средства. Поэтому для иллюстрации основных рассматриваемых в курсе технологий выбрана достаточно простая система, позволяющая сконцентрироваться на важных для нас моментах, не углубляясь слишком сильно в предметную область. С другой стороны, рассматриваемая задача является достаточно сложной, чтобы проиллюстрировать на ее примере некоторые типичные проблемы, возникающие при разработке серьезных распределенных систем.

В качестве такой тестовой задачи рассмотрим задачу автоматизации некоторого отдельного контура работы сети столовых: необходимо разработать систему, которая автоматизирует участок работы, связанный с обслуживанием пластиковых карт. Дело в том, что по легенде в указанных столовых предусмотрен дополнительный сервис: постоянным клиентам выдают пластиковые карты, при предъявлении которых клиент получает существенную скидку. При этом на карту можно положить некоторую сумму денег и расплачиваться за обеды не наличными, а средствами, находящимися на карте. В целях экономии затрат на эмиссию карты закуплены не чиповые, а штриховые (такая карта несет только код - идентификатор клиента). Существенным является то обстоятельство, что карта, выданная в одной столовой, может использоваться в другой столовой. Между столовыми нет устойчивых защищенных каналов связи, в связи с этим встает проблема передачи баланса карт между столовыми. Кроме того, в будущем руководство предполагает изучать предпочтения клиентов, чтобы формировать более гибкую ценовую политику и более рационально управлять ассортиментом предлагаемых блюд.

В дальнейшем при реализации того или иного примера будут выдвигаться некоторые дополнительные условия и ограничения, позволяющие иллюстрировать те или иные аспекты рассматриваемых технологий.

**Первый пример**

Разобранный в предыдущем разделе пример уже достаточно сложен и содержит примеры всех необходимых нам технологий, для того чтобы послужить прототипом для первой версии нашей "ресторанной" системы. Чтобы программы-примеры не получались слишком большими, договоримся, что наша система будет поддерживать очень ограниченный набор операций:

* операцию выдачи клиенту новой карты;
* операцию пополнения счета;
* операцию оплаты картой покупки (снятия средств со счета);
* операцию получения остатка средств на карте.

Кроме того, практически не будут реализованы механизмы защиты от перерасхода средств, механизмы контроля действий пользователя и т.д., поскольку главной задачей рассматриваемых программ является демонстрация тех или иных механизмов взаимодействия, а не демонстрация реализации соответствующей бизнес-логики.

**Класс BillingService**

Первый серверный класс **BillingService** очень похож на класс **TCPServer** из предыдущего примера - так же как и **TCPServer**,он создает серверный сокет, после чего большую часть времени проводит в ожидании соединения клиентов. При подключении клиента создается экземпляр класса **BillingClientService**,в конструктор которого передаются, соответственно, **InputStream** и **OutputStream**,извлеченные из сокета. **BillingClientService** является потомком класса **Thread**,поэтому после запуска (вызов метода **start)** начинает обрабатывать клиентский запрос параллельно, в то время как **BillingService** возвращается к ожиданию соединения со следующим клиентом. Таким образом, предлагаемый класс **BillingService** способен одновременно обрабатывать соединения с несколькими клиентами.

Поскольку основной задачей системы является выполнение операций по обслуживанию клиентских карт, класс **BillingService** предлагает соответствующие методы - всего их четыре:

* **void addNewCard(String personName, String card)** - добавляет в систему новую карту, с идентификатором card и personName - ФИО пользователя;
* **void addMoney(String card, double money)** - увеличивает остаток на карте card на величину money (пополнение счета);
* **void subMoney(String card, double money)** - уменьшает остаток на карте card на величину money (оплата картой);
* **double getCardBalance(String card)** - для карты с идентификатором card возвращает значение баланса.

Для хранения "базы" карт применяется хэш-таблица, в которой ключом служит идентификатор карты, а в качестве значения выступает ее баланс. Соответственно, все операции над картой в качестве обязательного параметра содержат ее идентификатор, используя который в качестве ключа, получают доступ к балансу.

1 package com.asw.net.ex1;

2 import java.net.\*;

3 import java.util.Hashtable;

4 import java.io.\*;

5

6 public class BillingService extends Thread{

7 public static final int ADD\_NEW\_CARD = 1;

8 public static final int ADD\_MONEY = 2;

9 public static final int SUB\_MONEY = 3;

10 public static final int GET\_CARD\_BALANCE = 4;

11 public static final int EXIT\_CLIENT = 5;

12

13 private int serverPort = 7896;

14 private ServerSocket ss;

15 private Hashtable hash;

16

17 public static void main(String[] args) {

18 BillingService bs = new BillingService();

19 bs.start();

20 }

21

22 public BillingService(){

23 hash = new Hashtable();

24 }

25

26 public void run(){

27 try {

28 ss = new ServerSocket(serverPort);

29 System.out.println("Server started");

30 while(true){

31 Socket s = ss.accept();

32 System.out.println("Client accepted");

33 BillingClientService bcs = new BillingClientService(this,

 new DataInputStream(s.getInputStream()), new DataOutputStream(s.getOutputStream()));

34 bcs.start();

35 }

36 } catch (IOException e) {

37 e.printStackTrace();

38 }

39 }

40

41 public void addNewCard(String personName, String card) {

42 hash.put(card, new Double(0.0));

43 }

44 public void addMoney(String card, double money) {

45 Double d = (Double)hash.get(card);

46 if (d!=null) hash.put(card,new Double(d.doubleValue()+money));

47 }

48 public void subMoney(String card, double money) {

49 Double d = (Double)hash.get(card);

50 if (d!=null) hash.put(card,new Double(d.doubleValue()-money));

51 }

52 public double getCardBalance(String card) {

53 Double d = (Double)hash.get(card);

54 if (d!=null) return d.doubleValue();

55 return 0;

56 }

57 }

Листинг 4.1. Класс BillingService

**Класс BillingClientService**

Следующий серверный класс - **BillingClientService** - отвечает непосредственно за взаимодействие с клиентом ([пример 4.2](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=2#example.4.2)). Получив в конструкторе два потока - поток ввода и поток вывода, он способен с их помощью как принимать данные, отправляемые клиентом, так и посылать ему данные.

Клиентское приложение должно иметь возможность передать серверу информацию о том, какие действия должен выполнить сервер - создать новую карту, изменить баланс существующей и т.д. Причем для многих из этих операций нужно передать дополнительную информацию - об идентификаторе карты, о сумме, которую с нее предстоит списать, и т.д. Таким образом, поскольку все, что у нас есть - это потоки, нам предстоит разработать некое соглашение о том, как передавать информацию от клиента к серверу, или, говоря другими словами, создать собственный прикладной протокол для нашей системы.

Разработка таких протоколов, вообще говоря, задача не простая, поскольку от ее решения во многом зависит производительность и надежность будущей системы. В нашем же случае мы реализуем простейший протокол, сообщения в котором будут иметь следующий вид:

|  |  |
| --- | --- |
| Код операции  | Данные |

При этом количество и типы данных для каждой операции различны, а вот коды операции имеют целый тип, и их значения заранее известны.

Зафиксировав такое соглашение, мы можем построить обработку клиентского соединения следующим образом: в цикле читаем из потока целое число, которое интерпретируем как код операции, и, в зависимости от прочитанного кода операции, вызываем соответствующий метод. Вызываемый метод выполняет следующие действия:

* дочитывает из потока необходимые для работы параметры (поскольку это конкретный метод, например, увеличения баланса карты, список параметров для него известен);
* вызывает соответствующий метод **BillingService**,передавая ему прочитанные параметры, который и выполняет все необходимые действия.

Проиллюстрируем этот механизм на примере обработки операции по заведению новой карты.

Клиент посылает серверу данные следующего формата: код операции (в данном случае операция ADD\_NEW\_CARD имеет код 1), затем строку с именем владельца, затем строку, содержащую идентификатор карты. **BillingClientService** в методе **run** считывает код операции, после чего вызывает свой метод **addNewCard**.Этот метод "знает", что для операции заведения новой карты необходимо два строковых параметра. Он считывает их из потока, после чего вызывает метод **addNewCard**,который заносит новую запись в хэш-таблицу. Остальные операции выполняются аналогично. Единственный метод, о котором стоит сказать особо, - это метод **getCardBalance**,который отличается от других тем, что не только читает данные от клиента (идентификатор карты), но и отправляет ему данные, используя поток вывода **OutputStream**.