МИНИСТЕРСТВО образования РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

УНИВЕРСИТЕТ информатики и радиоэлектроники»

Факультет компьютерного проектирования

Индивидуальная практическая работа №1

По дисциплине: «Разработка распределенных программ на основе RMI»

Тема: «Учет выплат по кредитам»

Выполнил студент 4-го курса факультета инновационного непрерывного образования Группа [894351](https://iis.bsuir.by/#/schedule;groupName=493551)

Галкин Илья Викторович

Зачетная книжка №6535021

Дата сдачи: 13.01.2022

Проверил: Унучек Е.Н.

 Минск 2022

**Оглавление**

[1. Цель 3](#_Toc91872395)

[2. Выполнение работы 3](#_Toc91872396)

# Краткие теоретические сведения

RMI обозначает **дистанционный вызов метода**. Это механизм, который позволяет объекту, находящемуся в одной системе (JVM), получать доступ / вызывать объект, запущенный в другой JVM.

RMI используется для создания распределенных приложений; он обеспечивает удаленную связь между Java-программами. Это предусмотрено в пакете **java.rm**.

## Архитектура приложения RMI

В приложении RMI мы пишем две программы: **серверную** (на сервере) и **клиентскую** (на клиенте).

* Внутри серверной программы создается удаленный объект, и ссылка на этот объект становится доступной для клиента (с использованием реестра).
* Клиентская программа запрашивает удаленные объекты на сервере и пытается вызвать его методы.

Внутри серверной программы создается удаленный объект, и ссылка на этот объект становится доступной для клиента (с использованием реестра).

Клиентская программа запрашивает удаленные объекты на сервере и пытается вызвать его методы.

Следующая диаграмма показывает архитектуру приложения RMI.

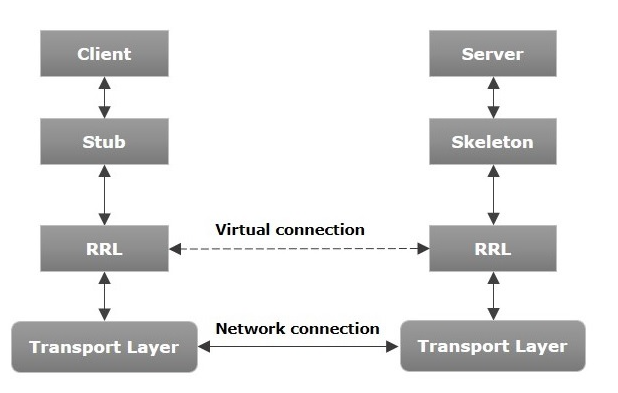


Рисунок 1 – архитектура приложения RMI

Давайте теперь обсудим компоненты этой архитектуры.

* **Транспортный уровень** — этот уровень связывает клиента и сервер. Он управляет существующим соединением, а также устанавливает новые соединения.
* **Заглушка** — это представление (прокси) удаленного объекта на клиенте. Он находится в клиентской системе; он действует как шлюз для клиентской программы.
* **Скелет** — это объект, который находится на стороне сервера. Заглушка связывается с этим скелетом для передачи запроса удаленному объекту.
* **RRL (Remote Reference Layer)** — это уровень, который управляет ссылками, сделанными клиентом на удаленный объект.

**Транспортный уровень** — этот уровень связывает клиента и сервер. Он управляет существующим соединением, а также устанавливает новые соединения.

Заглушка — заглушка — это представление (прокси) удаленного объекта на клиенте. Он находится в клиентской системе; он действует как шлюз для клиентской программы.

**Скелет** — это объект, который находится на стороне сервера. **заглушка** связывается с этим скелетом для передачи запроса удаленному объекту.

**RRL (Remote Reference Layer)** — это уровень, который управляет ссылками, сделанными клиентом на удаленный объект.

## **Работа приложения RMI**

Следующие пункты суммируют, как работает приложение RMI —

* Когда клиент выполняет вызов к удаленному объекту, он принимается заглушкой, которая в конечном итоге передает этот запрос в RRL.
* Когда RRL на стороне клиента получает запрос, он вызывает метод **invoke ()** объекта **remoteRef**  Он передает запрос в RRL на стороне сервера.
* RRL на стороне сервера передает запрос в Skeleton (прокси-сервер на сервере), который, наконец, вызывает требуемый объект на сервере.
* Результат передается обратно клиенту.

Когда клиент выполняет вызов к удаленному объекту, он принимается заглушкой, которая в конечном итоге передает этот запрос в RRL.

Когда RRL на стороне клиента получает запрос, он вызывает метод **invoke ()** объекта **remoteRef**. Он передает запрос в RRL на стороне сервера.

RRL на стороне сервера передает запрос в Skeleton (прокси-сервер на сервере), который, наконец, вызывает требуемый объект на сервере.

Результат передается обратно клиенту.

## **Маршаллинг и демаршаллинг**

Всякий раз, когда клиент вызывает метод, который принимает параметры на удаленном объекте, параметры объединяются в сообщение перед отправкой по сети. Эти параметры могут быть примитивного типа или объектов. В случае примитивного типа параметры объединяются и к нему прикрепляется заголовок. Если параметры являются объектами, они сериализуются. Этот процесс известен как **сортировка**.

На стороне сервера упакованные параметры разделяются, а затем вызывается требуемый метод. Этот процесс известен как **демаршаллинг**.

## **Реестр RMI**

Реестр RMI — это пространство имен, в котором размещены все объекты сервера. Каждый раз, когда сервер создает объект, он регистрирует этот объект в RMIregistry (используя методы **bind ()** или **reBind ()** ). Они зарегистрированы с использованием уникального имени, известного как **имя привязки**.

Чтобы вызвать удаленный объект, клиенту нужна ссылка на этот объект. В это время клиент выбирает объект из реестра, используя его имя привязки (используя метод **lookup ()**).

Следующая иллюстрация объясняет весь процесс

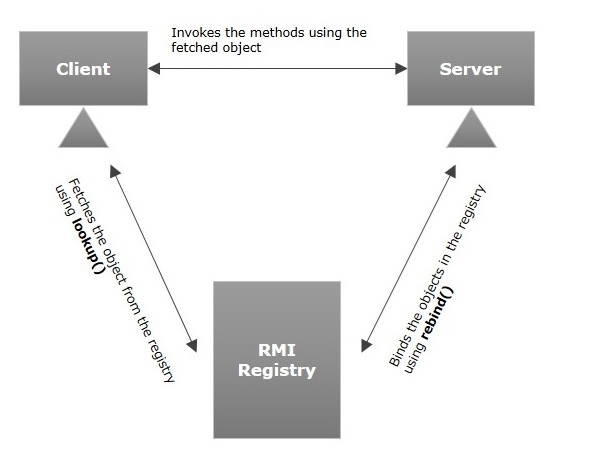


Рисунок 2 – схема работы RMI

## **Цели РМИ**

Ниже приведены цели RMI —

* Чтобы минимизировать сложность приложения.
* Чтобы сохранить безопасность типов.
* Распределенная сборка мусора.
* Минимизируйте разницу между работой с локальными и удаленными объектами.

# Скриншоты работы программы

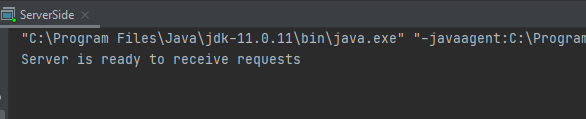


Рисунок 3 – запуск сервера

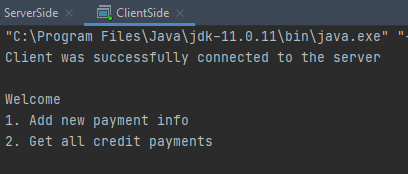


Рисунок 4 – запуск клиента

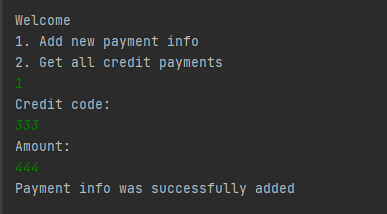


Рисунок 5 – добавление записи об оплате

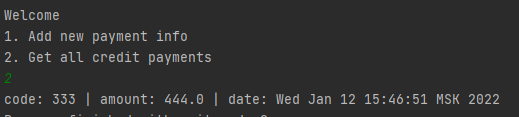


Рисунок 6 – получение списка записей

# Листинг программного кода

**ServerSide.java**

**package** com.company;

**import** **com.company.infrastructure.IRemoteObject**;

**import** **com.company.infrastructure.PaymentsRemoteObject**;

**import** **com.company.settings.Constants**;

**import** **java.rmi.registry.LocateRegistry**;

**import** **java.rmi.server.UnicastRemoteObject**;

**public** **class** **ServerSide** {

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

var remoteObject = **new** PaymentsRemoteObject();

**try**{

var stub = (IRemoteObject) UnicastRemoteObject.exportObject(remoteObject, **0**);

var registry = LocateRegistry.createRegistry(Constants.Port);

registry.bind(Constants.BindingName, stub);

System.out.println("Server is ready to receive requests");

}

**catch**(Throwable cause) {

System.err.println(cause.getMessage());

}

}

}

**ClientSide.java**

**package** com.company;

**import** **com.company.infrastructure.IRemoteObject**;

**import** **com.company.models.Payment**;

**import** **com.company.settings.Constants**;

**import** **java.io.IOException**;

**import** **java.rmi.NotBoundException**;

**import** **java.rmi.RemoteException**;

**import** **java.rmi.registry.LocateRegistry**;

**import** **java.util.Scanner**;

**public** **class** **ClientSide** {

**public** **static** **void** **main**(String[] args) **throws** RemoteException, NotBoundException {

var registry= LocateRegistry.getRegistry(Constants.Host);

var stub = (IRemoteObject) registry.lookup(Constants.BindingName);

System.out.println("Client was successfully connected to the server\n");

printMenu();

var scanner = **new** Scanner(System.in);

String choice = scanner.nextLine();

**switch** (choice) {

**case** "1" : {

handleInsertRecord(stub);

**break**;

}

**case** "2" : {

handleGetList(stub);

**break**;

}

**default**: {

System.out.println("Invalid command");

}

}

}

**private** **static** **void** **handleInsertRecord**(IRemoteObject remoteObject) {

var sc = **new** Scanner(System.in);

System.out.println("Credit code:");

var code = sc.nextLine();

System.out.println("Amount:");

var value = sc.nextLine();

**try** {

var parsedValue = Double.parseDouble(value);

remoteObject.insert(**new** Payment(code, parsedValue));

System.out.println("Payment info was successfully added");

} **catch**(NumberFormatException e) {

System.err.println("Incorrect data");

e.printStackTrace();

}

**catch** (IOException e) {

System.err.println("Error during data insert");

e.printStackTrace();

}

}

**private** **static** **void** **handleGetList**(IRemoteObject remoteObject) {

**try** {

var list = remoteObject.getList();

**for** (String s : list) {

System.out.print(s);

}

}

**catch** (IOException e) {

System.err.println("Error during data receiving");

e.printStackTrace();

}

}

**private** **static** **void** **printMenu**() {

System.out.println("Welcome");

System.out.println("1. Add new payment info");

System.out.println("2. Get all credit payments");

}

}

**PaymentsRemoteObject.java**

**package** com.company.infrastructure;

**import** **com.company.settings.Constants**;

**import** **java.io.\***;

**import** **java.rmi.RemoteException**;

**import** **java.util.ArrayList**;

**public** **class** **PaymentsRemoteObject** **implements** IRemoteObject {

**@Override**

**public** **void** **insert**(ISerializable model) **throws** RemoteException, IOException {

var fw = **new** FileWriter(Constants.FileName, **true**);

var bw = **new** BufferedWriter(fw);

bw.write(model.getFileRepresentation());

bw.close();

}

**@Override**

**public** ArrayList<String> **getList**() **throws** RemoteException, IOException {

var result = **new** ArrayList<String>();

var fr = **new** FileReader(Constants.FileName);

var br = **new** BufferedReader(fr);

**while** (br.ready()) {

result.add(br.readLine());

}

br.close();

fr.close();

**return** result;

}

}