ЛАБОРАТОРНАЯ РАБОТА №2

По дисциплине: Распределенные информационно-аналитические системы.
Тема занятия: Разработка echo-клиента и сервера.
Цель занятия: Изучить основные принципы разработки клиент-серверных приложений с использованием технологии Java на базе потоковых сокетов TCP.
Количество часов: 4.

Введение

Echo-клиент – это простая программа, которая может установить TCP-соединение с определенным echo-сервером и обмениваться с ним текстовыми сообщениями. Клиент предоставляет интерфейс на основе командной строки, который фиксирует вводимые пользователем данные и контролирует взаимодействие с сервером. Помимо установления и разрыва соединения, пользователь может передавать сообщения на сервер. Эти сообщения, в свою очередь, возвращаются клиенту, где они отображаются пользователю. Последовательность взаимодействий показана ниже.

```
EchoClient> connect 127.0.0.1 5555
EchoClient> Connection to MSRG Echo server established: /127.0.0.1 / 5555
EchoClient> send hello world
EchoClient> hello world
EchoClient> disconnect
EchoClient> Connection terminated: 127.0.0.1 / 5555
EchoClient> send hello again
EchoClient> Error! Not connected!
EchoClient> quit
EchoClient> Application exit!
```

Детальное описание

Клиентская программа

Клиентская программа состоит из логики приложения, простого пользовательского интерфейса на основе командной строки (CLI) и логики связи для взаимодействия с сервером на основе потоковых сокетов TCP в Java. Для этого используются стандартные входные (*in*) и выходные (*out*) потоки (т.е. *System.in* и *System.out*). После запуска программы CLI выводит подсказку “`EchoClient>`” в каждой строке и предоставляет команды, перечисленные в следующей таблице.

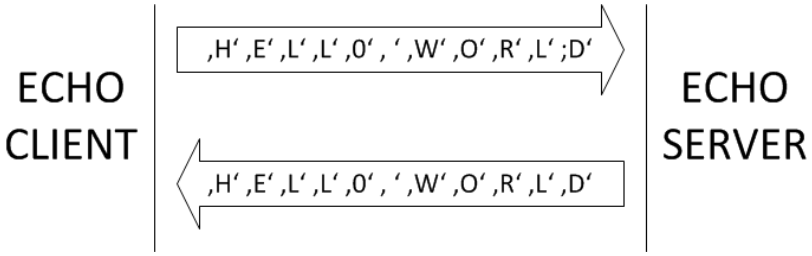
Команда	**Неформальное описание**	**Параметры**	**Вывод CLI**
`connect <address> <port>`	Пытается установить TCP-соединение с эхо-сервером на основе заданного адреса сервера и номера порта эхо-службы.	**address**: имя хоста или IP-адрес эхо-сервера. **port**: порт эхо-службы на соответствующем сервере.	**ответ сервера:** как только соединение установлено, эхо-сервер ответит сообщением подтверждения. Это сообщение должно отображаться пользователю. (*Обратите внимание, что, если установление соединения не удалось, клиентское приложение должно предоставить пользователю соответствующее сообщение об ошибке*).
`disconnect`	Пытается отключиться от	–	**отчет о состоянии:** после того, как клиент был отключен от сервера, он должен предоставить соответствующее уведомление пользователю.

	подключенного сервера.		(*Обратите внимание, что соединение также может быть потеряно из-за ошибки соединения или выхода из строя сервера*).
`send <message>`	Отправляет текстовое сообщение на эхо-сервер в соответствии с протоколом связи.	**message:** последовательность кодированных символов ASCII, соответствующих протоколу приложения. (см. более подробное описание ниже).	**ответ сервера:** как только эхо-сервер получит сообщение, он отправит то же сообщение клиенту. Это сообщение должно отображаться пользователю в новой строке. (*Обратите внимание на ситуацию, когда клиент не подключен к серверу*).
`logLevel <level>`	Устанавливает логгер на указанный уровень логирования.	**level**: Один из следующих уровень логирования log4j: (`ALL` | `DEBUG` | `INFO` | `WARN` | `ERROR` | `FATAL` | `OFF`)	**статусное сообщение:** распечатывает текущий статус логирования.
`help`		–	**текст справки:** показывает предполагаемое использование клиентского приложения и описывает его набор команд.
`quit`	Разрывает активное соединение с сервером и завершает выполнение программы.	–	**отчет о состоянии:** уведомляет пользователя о скором завершении работы программы.
`<прочее>`	Любой нераспознанный ввод в контексте этого приложения.	`<любые>`	**сообщение об ошибке:** неизвестная команда печатает текст справки.

Логика связи клиентской программы в основном включает потоковые сокеты (см. Java API [http://docs.oracle.com/javase/6/docs/api/java/net/Socket.html]). Методы, которые выполняют команды, введенные пользователем, являются частью логики приложения клиентской программы. Логика связи реализована в виде библиотеки, которая обеспечивает четко определенные функции (т.е. методы обработки соединений, взаимодействия с сервером, передачи сообщений и т.д.). Интерфейс позволяет приложениям получить доступ к этим функциям. Следовательно, клиентское приложение использует эту библиотеку, вызывая соответствующие методы в своем интерфейсе. Эта библиотека должна быть использована для последующих этапов.

Протокол связи и специальные сообщения приложения

Следующие описания должны помочь понять логику связи.



Сообщение

Сообщение в контексте этого проекта состоит из последовательности ASCII-символов. Символ возврата каретки или перевода строки (т.е. ASCII `13`, `0x0D`, ‘`\n`’) служит разделителем сообщений. Это означает, что сервер прекращает синтаксический анализ текущего сообщения, когда встречает возврат каретки. Максимальный размер сообщения, обрабатываемого сервером, составляет 128 Кбайт.

Протокол

Вообще говоря, протокол – это набор правил и форматов сообщений, которые описывают взаимодействие между различными процессами для выполнения конкретной задачи путем обмена сообщениями в соответствии с указанными правилами. Определение протокола включает в себя два важных этапа; (1) спецификация форматов сообщений и (2) порядок обмена сообщениями. Протокол связи для этого проекта довольно прост. Клиент отправляет текстовое сообщение, как описано выше, на эхо-сервер. В свою очередь, эхо-сервер анализирует сообщение и отвечает тем же сообщением. Исключением из этого правила является возврат каретки, который приводит к указанному выше поведению.

Marshaling и un-marshaling

Marshaling относится к преобразованию элементов данных в представление, которое обеспечивает возможность передачи содержимого сообщения (например, байтов) по сети связи. Обратная операция, un-marshaling, необходима для восстановления элементов данных после их передачи.

Этот проект направлен на обмен сообщениями посредством сокетов. Чтобы понять принципы сетевого программирования, мы используем коммуникационный интерфейс низкого уровня. Следовательно, клиентские методы для отправки и приема сообщений полагаются только на запись байтов или считывание байтов из сокета соответственно.

Сигнатуры метода выглядят следующим образом:

```
-    void send(byte[]);
-    byte[] receive();
```

Представьте, что сервер реализован на языке, отличном от Java, и не понимает формат сериализации объектов Java.

Логирование

В дополнение к реальной реализации необходимо ознакомиться и в дальнейшем использовать инструмент Java-логирования Apache Log4j. Скачайте jar-архив Log4j [http://logging.apache.org/log4j/2.x/], добавьте его в свой проект и инициализируйте логирование в консоли и в отдельный log-файл (`/logs/client.log`). Логирование является полезным механизмом для отслеживания поведения программы во время разработки и даже в процессе производства. Помимо любой другой информации, которую вы считаете полезной, регистрируйте как минимум все отправленные сообщения и все входящие ответы, полученные с сервера. Логирование должно быть динамически управляемым (`ALL` | `DEBUG` | `INFO` | `WARN` | `ERROR` | `FATAL` | `OFF`) командой `logLevel` (см. таблицу выше).

Корректная обработка ошибок

Жизненно важным условием для клиента является его способность корректно обрабатывать сбои. Убедитесь, что ваша программа устойчива к любому неправильному или непреднамеренному вводу пользователем (например, неправильные / неизвестные команды, количество и формат параметров и т.д.). В дополнение к этому, рассмотрите проблемы, которые могут возникнуть при обмене данными, то есть правильно обработайте исключения и следите за контролируемым разрывом соединения и потока данных. В дополнение к сообщениям об ошибках также распечатывайте сообщения о состоянии в CLI, если это имеет смысл (например, если клиент подключен или отключен от сервера).

Общие требования

- Правильно документируйте программу, используя JavaDoc-комментарии.
- Придерживайтесь общих Java-соглашений о кодировании [http://www.oracle.com/technetwork/java/codeconvtoc-136057.html].
- Обратите внимание на хороший дизайн программы (например, разделение пользовательского интерфейса и логики программы).

Структура проекта и библиотеки

- Ниже приведена рекомендуемая структура каталогов для проекта (если используется Eclipse, большая часть этого будет выполнена автоматически). **Обратите внимание**, что для приведенного Ant-сценария сборки (http://ant.apache.org/) потребуется настройка, предложенная ниже, с `main()` – метод с именем `ui.Application`.
- Добавьте необходимые библиотеки в проект и проинициализируйте логирование.
- Затем попытайтесь установить сокет-соединение с сервером.

```
- echoClient
        | - src
         |        | - ui
         |        |    | - Application.java (with main())
         |        |    | - ..
         |        | - <other packages>       .
         | - bin
         |        | - ui
         |        |    | - Application.class (with main())
         |        |    | - …
         |        | - <other packages>
         | - libs
         |        | - log4j.jar
         | - logs
         |        | - client.log
         | - build.xml
        | - echoClient.jar
```

Дополнительные ресурсы

- Integrated Development Environment IntelliJ IDEA: https://www.jetbrains.com/idea/.
- Integrated Development Environment Eclipse: http://www.eclipse.org/.
- Java SE API: http://docs.oracle.com/javase/6/docs/api/java/net/Socket.html.
- Java Coding Conventions: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.
- Log4j: http://logging.apache.org/log4j/2.x/.
- JUnit: http://www.junit.org/.
- Ant build tool: http://ant.apache.org/.
- ASCII format: http://tools.ietf.org/pdf/rfc20.pdf.