

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**К. В. Попов, Н. С. Карамышева,
С. А. Зинкин**

**МЕТОДЫ И ТЕХНОЛОГИИ РАЗРАБОТКИ
СЕТЕВЫХ ПРИЛОЖЕНИЙ
ДЛЯ РАСПРЕДЕЛЕННЫХ
ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ**

Учебно-методическое пособие

ПЕНЗА 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Пензенский государственный университет» (ПГУ)

К. В. Попов, Н. С. Карамышева,
С. А. Зинкин

Методы и технологии разработки
сетевых приложений для распределенных
вычислительных систем

Учебно-методическое пособие

Пенза
Издательство ПГУ
2022

УДК 681.324.066

П58

Р е ц е н з е н т

кандидат технических наук, доцент кафедры
«Математическое обеспечение и применение ЭВМ»
Пензенского государственного университета
И. Ю. Балашова

Попов, Константин Владимирович.

П58

Методы и технологии разработки сетевых приложений для распределенных вычислительных систем : учеб.-метод. пособие / К. В. Попов, Н. С. Карамышева, С. А. Зинкин. – Пенза : Изд-во ПГУ, 2022. – 182 с.

Содержится описание сетевых технологий для распределенных вычислительных систем, охватывающих дисциплины «Программирование сетевых приложений», «Распределенные вычисления», «Разработка клиент-серверных приложений». Рассмотрены фундаментальные вопросы сетевого программирования, требующие знания различий между протоколами, требующими установления логического соединения (*connection-oriented protocols*), и протоколами, не требующими этого (*connectionless protocols*). Пособие предназначено для овладения навыками работы со стеком сетевых протоколов TCP/IP, изучения методов работы с сокетами, а также методов доступа к распределенным и параллельным базам данных в TCP/IP сетях. Приведены примеры сетевых приложений. Используемые языки программирования – C/C++, C#, Java.

Издание подготовлено на кафедре «Вычислительная техника» ПГУ и предназначено для обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» при изучении дисциплин, связанных с современными технологиями обработки информации в компьютерных сетях.

УДК 681.324.066

© Пензенский государственный
университет, 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
ОБЩИЕ СВЕДЕНИЯ	8
Основные модели распределенной обработки данных.....	8
Концептуальная модель распределенных сетевых вычислений.....	14
Классификация облачно-сетевых вычислений и систем.....	19
Цель и задачи лабораторного практикума	25
Перечень вопросов, рекомендуемых к изучению для выполнения лабораторных и самостоятельных работ	26
Часть 1. СЕТЕВЫЕ КЛИЕНТ-СЕРВЕРНЫЕ ТЕХНОЛОГИИ	28
Тема 1.1. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP	28
Тема 1.2. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP	30
Тема 1.3. Использование методов криптографии для закрытия данных, передаваемых между программами клиента и сервера.....	32
Тема 1.4. Комплексная отладка программ клиента и сервера	34
Тема 1.5. Передача и прием пакетов с подтверждением через протокол UDP	35
Тема 1.6. Создание и закрытие сокета в проекте Visual C++	35
Тема 1.7. Установление соединения для сокета, созданного в проекте Visual C++	41
Тема 1.8. Передача данных для сокета, созданного в проекте Visual C++	45
Тема 1.9. Использование библиотеки MFC для работы с сокетами.....	48
Тема 1.10. Использование асинхронных функций класса CSocket	52

Т е м а 1.11. Сокеты TCP/IP в системе UNIX.....	53
Т е м а 1.12. Создание системы «клиент-сервер» в ОС Linux при использовании СУБД MySQL	60
Ч а с т ь 2. ИНТЕГРАЦИЯ СЕТЕВЫХ, ИНФОРМАЦИОННЫХ И КЛАСТЕРНЫХ ТЕХНОЛОГИЙ В TCP/IP СЕТЯХ	73
Т е м а 2.1. Работа с базами данных в операционной системе Linux.....	74
Т е м а 2.2. Использование C API для доступа к базе данных	79
Т е м а 2.3. Технология сокетов в операционной системе Linux и распределенные базы данных.....	83
Т е м а 2.4. Библиотека MPI Linux и распределенные базы данных	90
Т е м а 2.5. Использование технологии ODBC для работы с базами данных в операционной системе Windows	98
Т е м а 2.6. Технология сокетов в операционной системе Windows и распределенные базы данных.....	102
Т е м а 2.7. Библиотека MPI Windows и распределенные базы данных	108
Ч а с т ь 3. ИСПОЛЬЗОВАНИЕ ЯЗЫКА C# ДЛЯ РАЗРАБОТКИ СЕТЕВЫХ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ.....	111
Т е м а 3.1. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP	111
Т е м а 3.2. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP	115
Т е м а 3.3. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP	118
Т е м а 3.4. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP	120
Т е м а 3.5. Разработка интерфейса серверного приложения с базой данных	122

Часть 4. РАЗРАБОТКА КЛИЕНТ-СЕРВЕРНОГО ПРИЛОЖЕНИЯ С ПАРАЛЛЕЛЬНЫМ ДОСТУПОМ К БАЗЕ ДАННЫХ И РАСПРЕДЕЛЕННОГО ПРИЛОЖЕНИЯ ДЛЯ РАБОТЫ С НЕСКОЛЬКИМИ БАЗАМИ ДАННЫХ НА ЯЗЫКАХ С# И JAVA	125
Тема 4.1. Разработка программы-сервера с параллельным доступом к базе данных в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP	125
Тема 4.2. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP	134
Тема 4.3. Разработка распределенного серверного приложения с доступом к нескольким базам данных с использованием семейства протоколов TCP/IP	141
Тема 4.4. Разработка распределенного клиентского приложения с доступом к нескольким базам данных с использованием семейства протоколов TCP/IP	150
Тема 4.5. Разработка программы-сервера с параллельным доступом к базе данных в архитектуре взаимодействия «клиент-сервер» на языке Java с использованием семейства протоколов TCP/IP	156
Тема 4.6. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» на языке Java с использованием семейства протоколов TCP/IP	163
Часть 5. ЗАДАНИЯ НА САМОСТОЯТЕЛЬНУЮ РАБОТУ И КОНТРОЛЬНЫЕ ВОПРОСЫ.....	173
5.1. Общие требования.....	173
5.2. Примерный перечень заданий на самостоятельную работу.....	174
5.3. Контрольные вопросы по сетевому программированию	175
5.4. Примерный перечень вопросов для тестирования	177
СПИСОК ЛИТЕРАТУРЫ	180

ВВЕДЕНИЕ

Большинство технологий распределенных вычислений основано на модели «Клиент-сервер». Клиент-сервер (англ. *Client-server*) – вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Физически клиент и сервер – это программное обеспечение. Обычно они взаимодействуют через компьютерную сеть посредством сетевых протоколов и находятся на разных вычислительных машинах, но могут выполняться также и на одной машине. Программы, расположенные на сервере, ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных или сервисных функций.

Преимущества архитектуры клиент-сервер:

- 1) отсутствие дублирования кода программы-сервера программами-клиентами;
- 2) так как все вычисления выполняются на сервере, то требования к компьютерам, на которых установлен клиент, снижаются;
- 3) все данные хранятся на сервере, который, как правило, защищен гораздо лучше большинства клиентов. На сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа.

Недостатки:

- 1) неработоспособность сервера может сделать неработоспособной всю вычислительную сеть. Неработоспособным сервером следует считать сервер, производительности которого не хватает на обслуживание всех клиентов, а также сервер, находящийся на ремонте, профилактике и т.д.;
- 2) поддержка работы данной системы требует отдельного специалиста – системного администратора;
- 3) высокая стоимость оборудования.

Одним из основных протоколов, предназначенных для управления передачей данных в сетях и подсетях TCP/IP, является TCP (Transmission Control Protocol).

Механизм TCP предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета. TCP осуществляет надежную передачу потока байтов от одной программы на некотором компьютере к другой программе на другом компьютере.

Альтернативой протоколу TCP является протокол UDP (User Datagram Protocol). UDP использует простую модель передачи, без неявных «рукопожатий» для обеспечения надежности, упорядочивания или целостности данных. Таким образом, UDP предоставляет ненадежный сервис, и дейтаграммы могут прийти не по порядку, дублироваться или не дойти совсем. UDP подразумевает, что проверка ошибок и исправление либо не нужны, либо должны выполняться в приложении.

Протоколы TCP и UDP являются базовыми для организации протоколов прикладного уровня в сложных системах распределенной обработки и хранения для передачи данных между узлами сети.

ОБЩИЕ СВЕДЕНИЯ

Основные модели распределенной обработки данных

Хорошо известная на практике модель клиент-сервер соответствует распределенной структуре приложения, которая разделяет задачи или рабочие нагрузки между поставщиками ресурса или услуги, называемыми серверами, и инициаторами запросов на услуги, называемыми клиентами. Часто клиенты и серверы обмениваются данными по компьютерной сети на отдельном оборудовании. Появление архитектуры «клиент-сервер», как и многих других новых компьютерных технологий, послужило развитием соответствующей терминологии [5, 12, 27, 29, 30, 33]. *Клиент-хост* и *сервер-хост* имеют несколько иное значение, чем *клиент* и *сервер*. *Хостом* называют любой компьютер, подключенный к сети. В то время как слова *сервер* и *клиент* могут относиться либо к компьютеру, либо к компьютерной программе, *сервер-хост* и *клиент-хост* всегда относятся к компьютерам. *Клиенты* и *серверы* – это просто программы, которые работают на хостах. Далее будут использоваться следующие термины:

Прикладной программный интерфейс (Application Programming Interface, API) – набор функций и подпрограмм, обеспечивающих взаимодействие клиентов и серверов.

Промежуточное программное обеспечение – набор драйверов, прикладных программных интерфейсов и прочего программного обеспечения, позволяющего улучшить взаимодействие между клиентским приложением и сервером.

Реляционная база данных – база данных, в которой доступ к информации ограничен выбором строк, удовлетворяющих определенным критериям поиска

Язык структурированных запросов (Structured Query Language, SQL), разработанный корпорацией IBM и стандартизованный институтом ANSI, – язык для создания, управления и изменения баз данных.

Двухзвенная и трехзвенная архитектуры «клиент-сервер» наиболее распространены (рис. 1). В двухзвенной архитектуре первым звеном является клиентское приложение, а второе звено образуют сервер БД и сама БД.

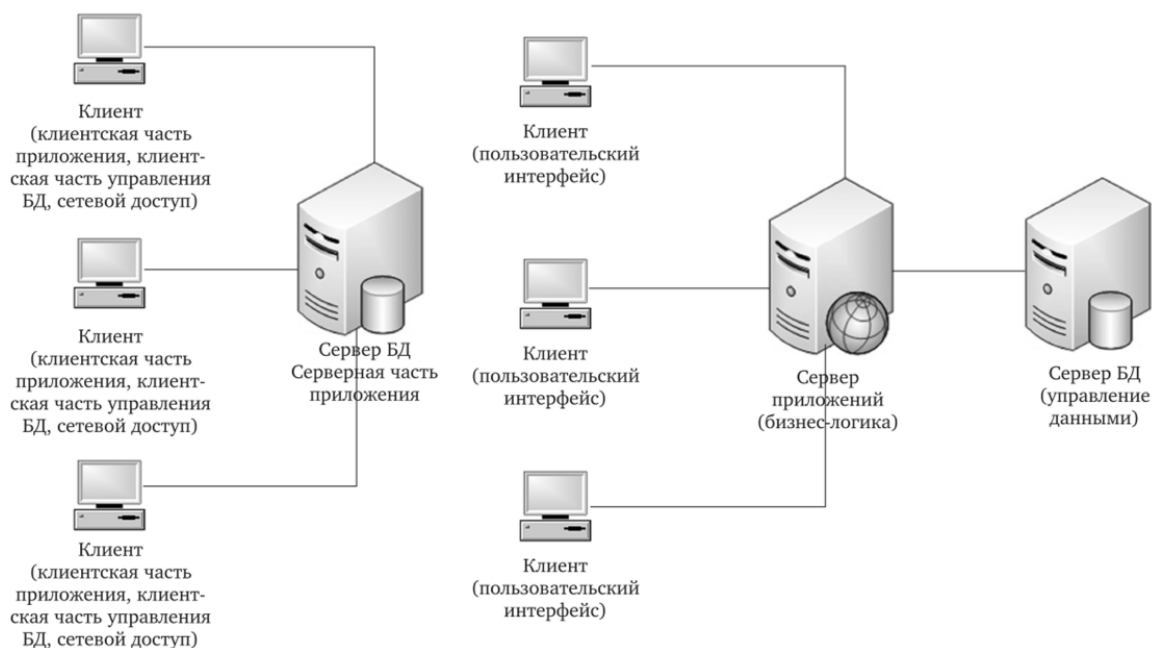


Рис. 1. Двух- и трехзвенная архитектура «клиент-сервер» с базой данных

Работа этой архитектуры обычно организуется в компьютерной сети следующим образом [5, 27, 28, 30]:

1. База данных (БД) в виде набора файлов находится на жестких дисках специально выделенного компьютера (сервера сети).

2. СУБД (система управления базой данных) располагается также на сервере сети.

3. Существует локальная сеть, состоящая из клиентских компьютеров, на каждом из которых установлено клиентское приложение для работы с БД.

4. На каждом из клиентских компьютеров пользователи имеют возможность запустить приложение. Используя предоставляемый пользовательский интерфейс, приложение инициирует обращение к СУБД, расположенной на сервере, на выборку или обновление хранимой в БД информации. Для общения используется специальный язык запросов (чаще всего это язык SQL), т.е. по сети от клиента к серверу передается лишь текст запроса.

5. СУБД содержит и может использовать все сведения о физической структуре БД, расположенной на сервере.

6. СУБД инициирует обращения к данным, находящимся на сервере, в результате которых на сервере осуществляется вся обработка данных и лишь результат выполнения запроса копируется на клиентский компьютер. Таким образом СУБД возвращает результат в приложение.

7. Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Функции между сервером и клиентом распределяются следующим образом.

Функции приложения-клиента:

- 1) посылка запросов серверу;
- 2) интерпретация результатов запросов, полученных от сервера;
- 3) представление результатов пользователю в некоторой форме, соответствующей интерфейсу пользователя.

Функции серверной части:

- 1) прием запросов от приложений-клиентов;
- 2) интерпретация запросов;
- 3) оптимизация и выполнение запросов к БД;
- 4) отправка результатов приложению-клиенту;
- 5) обеспечение системы безопасности и разграничение доступа;
- 6) управление целостностью БД;
- 7) реализация стабильности многопользовательского режима работы.

В двухзвенной архитектуре «клиент-сервер» бизнес-логика (хранимые SQL процедуры) может размещаться как на стороне клиента, так и на стороне сервера. Бизнес-логика реализуется в коде классов и их методов, в случае использования объектно ориентированных языков программирования, или процедур и функций, в случае применения процедурных языков, или в виде хранимых процедур языка SQL. Термин «бизнес» в данном контексте можно заменить на понятие предметная область (*domain*), не обязательно имеющее отношение к коммерции.

В трехзвенной архитектуре вся бизнес-логика (деловая логика, хранимые SQL процедуры), ранее входившая в клиентские приложения, выделяется в отдельное звено, называемое *сервером приложений*. При этом в клиентских приложениях остается лишь пользовательский интерфейс. Например, клиентским приложением может быть Web-браузер. При использовании данной организации при изменении бизнес-логики уже нет необходимости изменять клиентские приложения и обновлять их у всех пользователей. Кроме того, максимально снижаются требования к аппаратуре пользователей.

Работа трехзвенной архитектуры «клиент-сервер» организуется в компьютерной сети следующим образом:

1. База данных в виде набора файлов находится на жестком диске специально выделенного компьютера (сервера сети).

2. СУБД располагается также на сервере сети.

3. Существует специально выделенный сервер приложений, на котором располагается программное обеспечение (ПО) делового анализа (бизнес-логика, хранимые SQL процедуры).

4. Существует множество клиентских компьютеров, на каждом из которых установлен так называемый «тонкий клиент» – клиентское приложение, реализующее интерфейс пользователя.

5. На каждом из клиентских компьютеров пользователи имеют возможность запустить приложение-клиент. Используя предоставляемый этому приложению пользовательский интерфейс, оно инициирует обращение к ПО делового анализа, расположенному на сервере приложений.

6. Сервер приложений анализирует требования пользователя и формирует запросы к БД и по сети от сервера приложений к серверу БД передается текст запроса.

7. СУБД инкапсулирует внутри себя все сведения о физической структуре БД, расположенной на сервере.

8. СУБД инициирует обращения к данным, находящимся на сервере, в результате которых результат выполнения запроса копируется на сервер приложений.

9. Сервер приложений возвращает результат в клиентское приложение (пользователю).

10. Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Взаимодействие двух компьютеров при реализации сетевого приложения в архитектуре «клиент-сервер» иллюстрирует рис. 2.

Существуют также и другие технологии распределенного программирования в сетях.

Файл-серверная (ФС) технология – это технология, обеспечивающая работу в сетевом пространстве с доступом к файлам СУБД, хранящимся на сервере. При обработке запроса одного пользователя производится обращение к БД и последующая перекачка данных с блокировкой доступа других пользователей. Затем производится обработка данных на компьютере пользователя.

Недостатком такой организации является то, что для выборки полезных данных в общем случае необходимо просмотреть на стороне клиента весь соответствующий файл целиком, т.е. в файл-

серверной архитектуре почти вся работа выполняется на стороне клиента, а от сервера требуется только обеспечение хранения и передачи файла. Блокировка данных при редактировании одним пользователем делает невозможной работу с этими данными других пользователей и, кроме того, пользователь может повредить считанный «большой» файл, хотя ему нужна только небольшая часть данных.



Рис. 2. Взаимодействие двух компьютеров при реализации сетевого приложения в архитектуре «клиент-сервер»

Пиринговая (P2P – peer-to-peer networking) технология – это технология организации одноранговых сетей, часто называемая технологией P2P, является одной из самых полезных. В дополнение к модели клиент-сервер распределенные вычислительные приложения часто используют P2P-архитектуру. Пиринговая организация работы группы пользователей основана на понятии «равноправного участника». Каждая отдельная группа считается «хорошо соединенной», если соблюдено хотя бы какое-то одно из следующих условий:

– между каждой парой равноправных участников существует соединение, позволяющее каждому участнику подключаться к другому равноправному участнику по сети;

– между каждой парой равноправных участников существует относительно небольшое количество соединений, по которым они могут связываться;

– удаление одного равноправного участника из группы не лишает остальных равноправных участников возможности взаимодействия друг с другом.

Хотя при описании пиринговых приложений терминология архитектуры «клиент-сервер» не используется, можно дать следующее пояснение к организации P2P-приложений: сетевая программа пользователя может выступать как в роли сервера, так и в роли клиента. Такая сетевая программа может работать как на прием, так и на выдачу данных по запросу.

В системах электронной торговли получили распространение технологии B2B и B2C. Аббревиатура B2B происходит от английского «business-to-business», что в переводе означает «*бизнес – бизнесу*». Это продажи, в которых заказчиками выступают одни юридические лица, а поставщиками или подрядчиками – другие юридические лица. Одной из эффективных организаций B2B-торговли может быть пиринговая P2P-архитектура. Аббревиатура B2C происходит от английского «business-to-consumer» – «*бизнес – розничному потребителю*». Такого рода электронная торговля может быть основана на обычной трехзвенной архитектуре с Web-сервером в качестве сервера приложений и браузером в качестве клиентского приложения. Основное базовое отличие B2B и B2C архитектур с правовой точки зрения состоит в том, что B2C-договор заключается между юридическим и физическим лицом, а B2B-договор – между юридическими лицами.

Существуют и другие распределенные архитектуры, близкие к клиент-серверным. Например, многозвенные архитектуры «клиент-сервер», смешанные архитектуры систем электронной торговли типа B2B2C или B2B2B, в которых продукт на пути к конечному потребителю проходит больше стадий купли-продажи и даже переработки, распределенные приложения, создаваемые на основе парадигм передачи сообщений или согласованных взаимодействий сетевых процессов через пространство памяти, агентно-ориентированные технологии, современные Web-технологии. Для многих сетевых приложений характерна также архитектура *master-slave* – «*мастер-слуги*», при реализации которой размещенное на клиентском

хосте приложение-мастер «раздает» однотипные задания одинаковым приложениям-слугам, размещенным на других сетевых хостах.

Концептуальная модель распределенных сетевых вычислений

Для предварительного описания распределенных приложений предлагается простой графический сервис-ориентированный метаязык MPL – MetaProgramming Language, предназначенный для разработки грид-приложений. При его использовании считается, что все ресурсы распределенной вычислительной грид-системы составляют единое логическое пространство, где каждый хост содержит ресурс в форме службы, реализованной программными модулями-агентами, а приложение решает проблему, работая с этими службами. Метаязык MPL позволяет программистам сконцентрироваться на логике своих приложений, такой как логика реализации сервиса, логика вызова сервиса, логика связей и передач управления. Детали нижнего уровня, такие как распределение ресурсов, привязка ресурсов и развертывание служб, поддерживаются программным обеспечением промежуточного уровня (*middleware*), в качестве которого выбирается платформа сетевого программирования на основе языков C++, C#, Python или Java. Может быть выбрана и агентно-ориентированная платформа. Метаязык MPL помогает повысить продуктивность программистов при разработке распределенных приложений.

Конструкции метаязыка MPL в целом соответствуют дейкстровским принципам структурного программирования:

1. *Последовательность* – данная конструкция представлена в двух вариантах. На приведенном ниже рисунке (рис. 3) дано графическое представление фразы «оператор A_0 , размещенный на узле Y_0 , передает сообщение оператору A_1 , размещенному на узле Y_1 ».

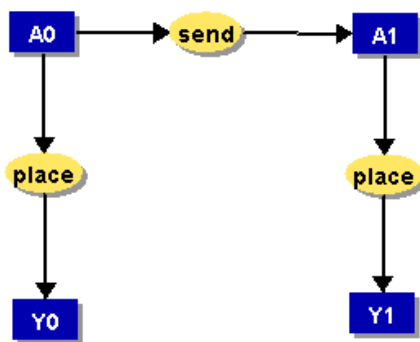


Рис. 3. Оператор A_0 , размещенный на узле Y_0 , передает сообщение оператору A_1 , размещенному на узле Y_1

С помощью сообщений в распределенном алгоритме передается управление от одного оператора другому; сообщения могут содержать результаты вычислений, запросы к базам данных и результаты выполнения данных запросов. Концептуальные графы являются графической формой представления формул в исчислении предикатов первого порядка. Графическому представлению на указанном рисунке соответствует следующая формула:

$$place(A_0, Y_0) \& place(A_1, Y_1) \& send(A_0, A_1).$$

Графу соответствуют следующие факты, составляющие экстенциональную базу данных в программе на языке Пролог:

$$\begin{aligned} place(a0, y0). \\ place(a1, y1). \\ send(a0, a1). \end{aligned}$$

Второй вариант реализации может быть получен путем логического вывода из первого, т.е. для того, чтобы передача управления от оператора A_0 оператору A_1 состоялась, необходимо, чтобы узлы Y_0 и Y_1 были связаны двунаправленными каналами (рис. 4).

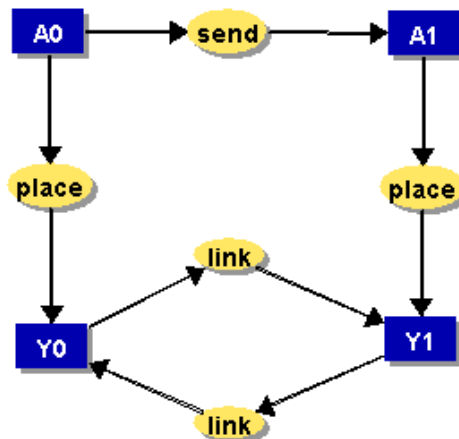


Рис. 4. Узлы Y_0 и Y_1 , связанные двунаправленными каналами

Для данного концептуального графа новые связи *link* находятся с помощью следующих правил:

$$\begin{aligned} link(X, Y) &:- send(A, B), place(A, X), place(B, Y). \\ link(Y, X) &:- send(A, B), place(A, X), place(B, Y). \end{aligned}$$

Здесь переменные A и B принимают значения в множестве имен операторов, а переменные X и Y – в множестве имен узлов распределенной вычислительной системы. Рекурсивное правило $link(Y, X) :- link(X, Y)$ не использовано по причине появления

в ответах копий кортежей в связи с особенностью выполнения рекурсивных правил в различных версиях Пролога.

2. *Ветвление*, или *альтернатива*. Особенность конструкции состоит в следующем (рис. 5). По умолчанию полагается, что в результате выполнения оператора A_6 определяется и проверяется значение логического условия Z_6 . Если $Z_6 = true$, то сообщение передается оператору A_7 , а при $Z_6 = false$ – оператору A_8 .

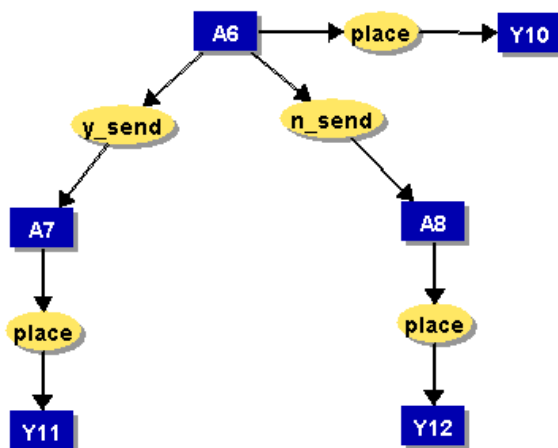


Рис. 5. Принцип ветвления

3. *Цикл*. Во фрагменте «цикл» оператор A_5 выполняется циклически следующим образом: после каждого выполнения данного оператора A_5 по умолчанию вычисляется значение логической переменной Z_5 (рис. 6). Если $Z_5 = true$, то осуществляется выход из цикла путем передачи сообщения оператору A_6 . При $Z_5 = false$ управление передается вспомогательному оператору A'_5 , размещенному на том же узле Y_8 , что и оператор A_5 , и далее оператор A'_5 передает управление оператору A_5 .

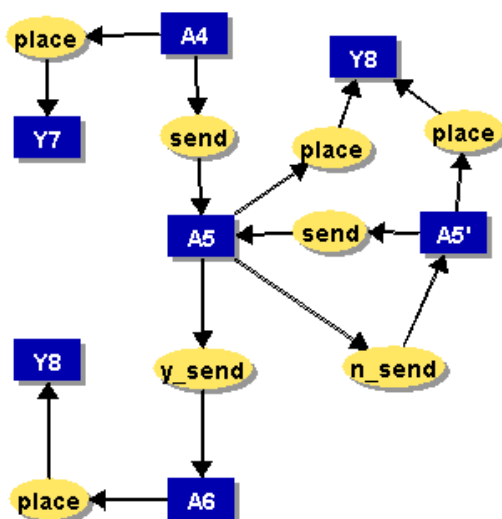


Рис. 6. Фрагмент «цикл»

4. *Fork-Join*. Фрагмент "fork-join" описывает передачу сообщений по параллельным ветвям распределенного алгоритма (рис. 7).

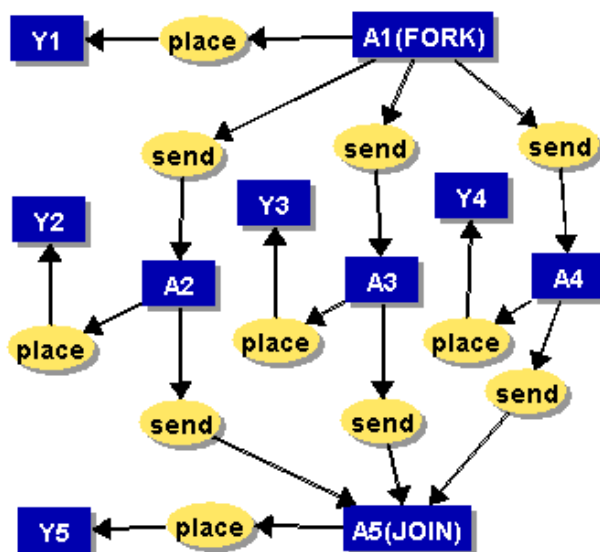


Рис. 7. Передача сообщений по параллельным ветвям распределенного алгоритма

Здесь по завершении выполнения всех параллельных ветвей сообщения объединяются в одно результирующее сообщение. В распределенном варианте выполнения ветви алгоритма не обязательно выполняются параллельно. Эти ветви выполняются «возможно одновременно», независимо друг от друга: во-первых, они запускаются на выполнение поочередно выдаваемые оператором A_1 сообщениями, во-вторых, их выполнение происходит в различных фрагментах вычислительной сети. В зависимости от ситуации, сложившейся в вычислительной сети, операторы, размещенные на различных узлах, могут выполняться с пересечением временных интервалов или вовсе в непараллельном режиме.

Возможны другие варианты реализации. Например, оператором A_5 вместо оператора соединения *join* может использоваться оператор *gather*, осуществляющий барьерную синхронизацию всех сообщений, выходящих из параллельных ветвей и их последующую передачу следующему оператору. При реализации другого варианта сообщения, выходящие из параллельных ветвей, могут передаваться по независимым путям другим операторам.

5. *Switch-Or*. Фрагмент "switch-or" реализует передачу сообщения по одной из ветвей распределенного алгоритма по значению переменной k . Возможен вариант реализации, при котором каждое результирующее сообщение передается различным операторам (рис. 8).

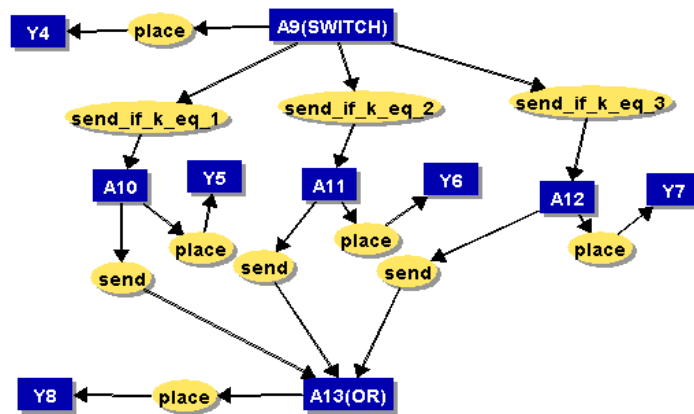


Рис. 8. Передача сообщения по одной из ветвей распределенного алгоритма по значению переменной k

Пример концептуального графа распределенного алгоритма, описывающего декларативные и процедурные знания о распределенном приложении, выполняемом в сетевой грид-среде, представлен на рис. 2.

Каждый оператор здесь может быть реализован на основе ПО узла (ячейки) пиринговой P2P-системы либо на основе совмещения функций приложения-клиента (при передаче сообщений) и клиента-сервера (при приеме сообщений). Для приведенного на рис. 9 концептуального графа, описывающего декларативные и процедурные знания о распределенном приложении, выполняемом в сетевой грид-среде, нетрудно разработать ее логическое представление и программу (например, на языке Пролог) для размещения распределенного алгоритма в физической сети, представленной на рис. 10.

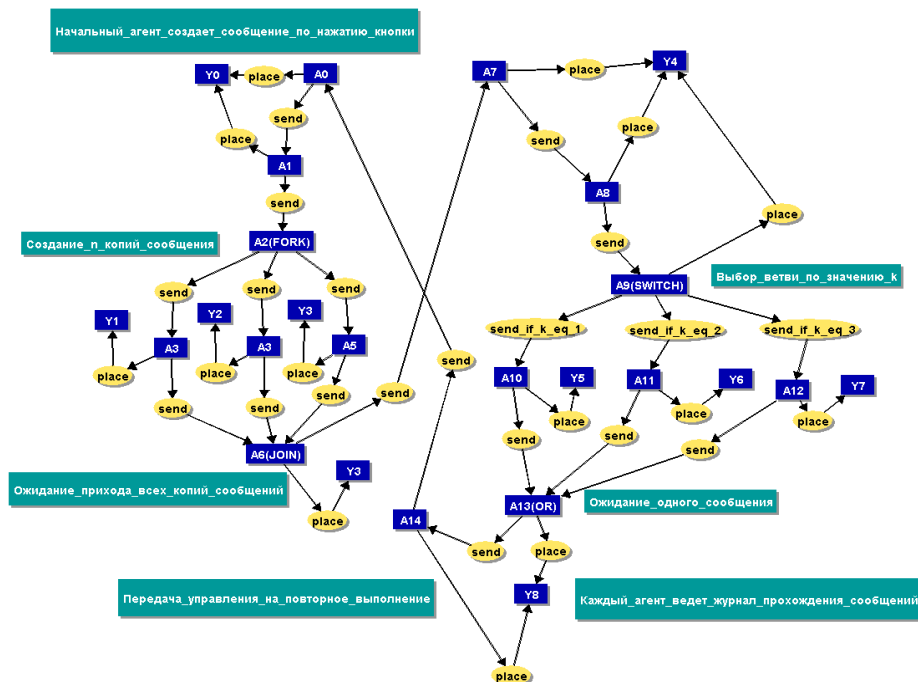


Рис. 9. Концептуальный граф распределенного алгоритма

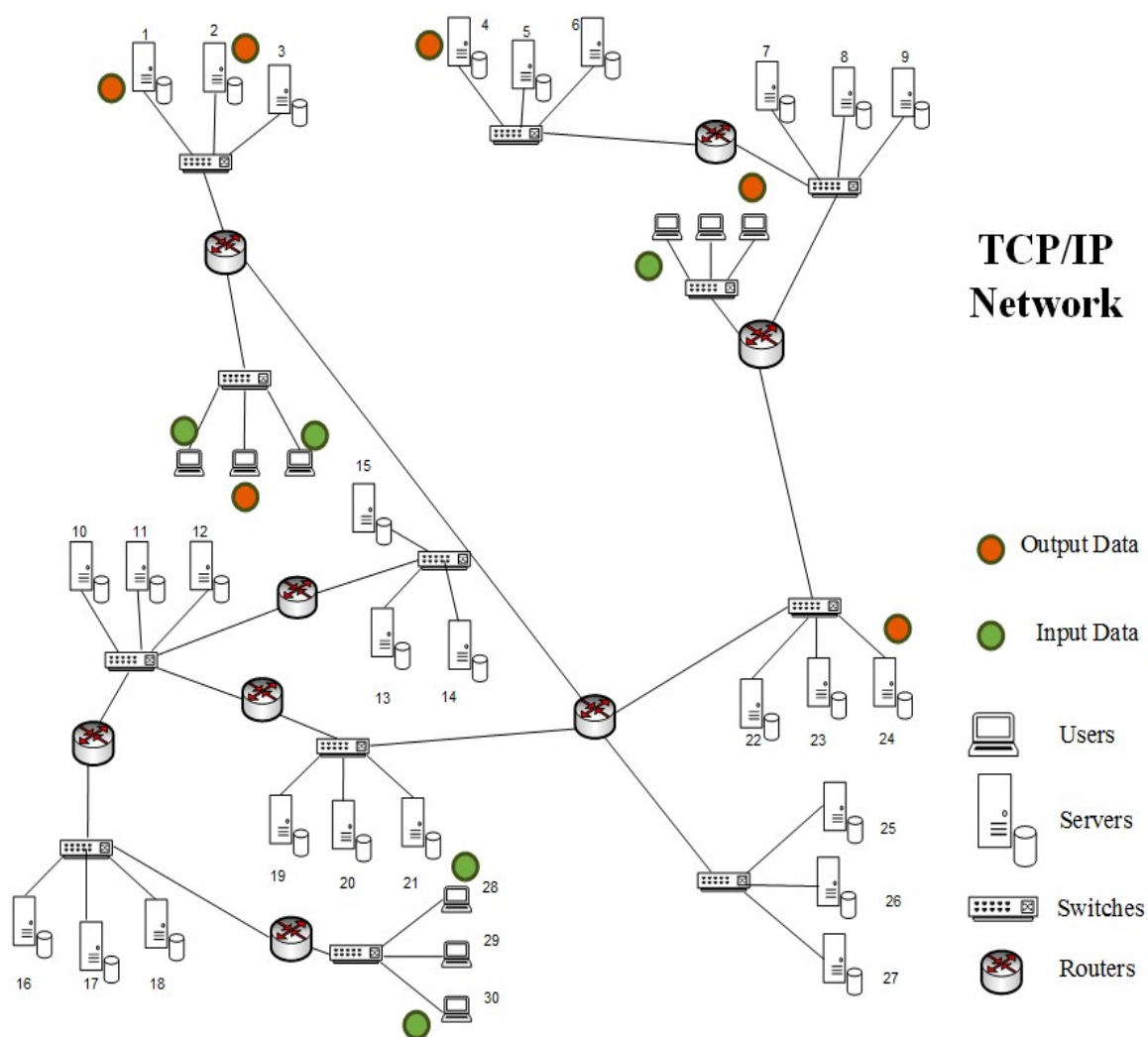


Рис. 10. Распределенная обработка данных
в компьютерной сети

Классификация облачно-сетевых вычислений и систем

Современный подход к организации распределенных вычислений предполагает выбор сетевой архитектуры *Grid Computing and Networking*, что дословно переводится как сетевой компьютерный грид (*grid* – решетка). Для использования практических преимуществ *Grid Computing and Networking* организациям необходимо обосновать переход на эту относительно новую вычислительную парадигму, современная реализация которой предполагает также взаимодействие грид-системы с клиентом при посредстве облачной среды, которой соответствует термин *Cloud Computing* – облачные вычисления. Несмотря на наличие в названии термина

«вычисления», собственно вычисления, как правило, не являются глобальными. При наличии спроса на вычисления, в том числе высокопроизводительные вычисления, через «облако» чаще всего организуется связь с отдельной грид-системой, кластером компьютеров или с суперкомпьютером.

Концепции, как и приложения грид-вычислений, не всегда являются ясными и понятными для нетехнических специалистов. При выборе распределенных грид-вычислений необходимо объяснить заказчику лежащий в основе грид-сетевой механизм и ответить на важные для бизнеса вопросы.

В настоящей работе распределенная обработка структурированных в виде таблиц данных связана с выполнением заданий многочисленных пользователей, организацией запросов к распределенным по вычислительной сети базам данных, вычислениями значений атрибутов. На рис. 10 представлена структура вычислительной сети, в которой могут быть реализованы распределенные вычисления. Например, это может быть компьютерная TCP/IP сеть. Эта сеть содержит клиентские компьютеры, серверы, коммутаторы, маршрутизаторы и каналы связи.

Зелеными кружками на рис. 10 обозначены запросы (возможно, содержащие данные) к распределенной базе данных, вводимые с пользовательских компьютеров. Коричневыми кружками обозначены конечные результаты, сохраняемые в локальных базах данных или принимаемые клиентами.

Грид-вычисления являются развивающейся областью и связаны с несколькими другими инновационными вычислительными системами, некоторые из которых являются частными случаями грид-систем.

Совместно используемые вычисления обычно относятся к совокупности компьютеров, которые совместно используют вычислительную мощность для выполнения определенной задачи. В этой связи в облачных средах реализуется сервис «программное обеспечение как услуга» (SaaS), известная как служебные вычисления, в которой компания предлагает определенные услуги (например, хранение данных или увеличение мощности процессора) за определенную плату. Облачные вычисления – это система, в которой приложения и хранилище являются ресурсами сети, а не компьютера пользователя.

На рис. 11 представлена классификация распределенных вычислений и реализующих эти вычисления систем. Данная

классификация учитывает современные тенденции развития распределенных вычислений и систем, рассматриваемых в качестве коммунальных услуг, предоставляемых пользователю.

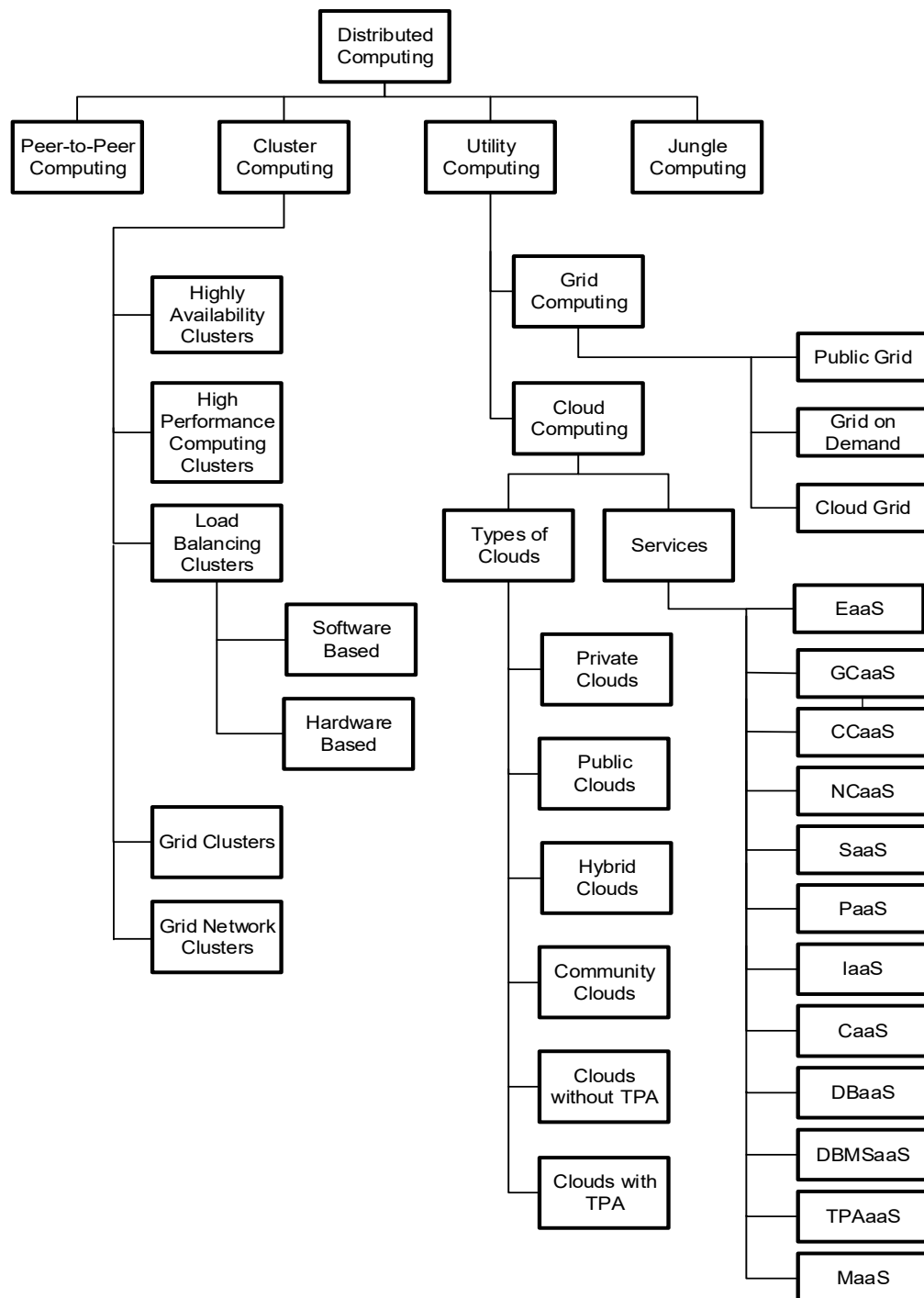


Рис. 11. Классификация распределенных вычислений и систем

Ниже описаны используемые термины и аббревиатуры, а также даны краткие комментарии. Для терминов на английском языке

дан перевод. При составлении классификации были приняты во внимание литературные источники, приведенные в работе [4].

- Distributed Computing (DC) – вычисления, организуемые в распределенной (возможно, параллельной) вычислительной системе.

- Peer-to-Peer Computing (PtPC, или P2PC) – пиринговые вычисления, в реализации которых участвуют, как правило, конечные компьютеры вычислительной сети, образующие масштабируемую в широких пределах одноранговую децентрализованную вычислительную среду, основанную на равноправии участников.

- Cluster Computing (CC) – кластеры (группы) компьютеров, связанные посредством высокоскоростных коммутаторов.

- Highly Available Clusters (HAC) – кластеры компьютеров, обеспечивающие высокий уровень надежности и готовности к использованию.

- High Performance Computing Clusters (HPCC) – высокопроизводительные кластеры.

- Load Balancing Clusters (LBC) – кластеры LBC-компьютеров, обеспечивающих балансировку вычислительной нагрузки.

- Software Based LBC (SB LBC) – программно-управляемые кластеры LBC-компьютеров.

- Hardware Based LBC (HB LBC) – аппаратно-управляемые кластеры LBC-компьютеров.

- Grid Clusters (GC) – кластеры компьютеров на основе высокоскоростных коммутаторов или локальных сетей, образующих грид-систему, т.е. «решетку» компьютеров.

- Grid Networking Clusters (GNC) – организуемые в грид-системах виртуальные сетевые кластеры, содержащие в виде узлов отдельные компьютеры физической вычислительной сети (рис. 12).

- Utility Computing (UC) – вычисления, при реализации которых вычислительные возможности предоставляются как коммунальные услуги.

- Grid Computing (GC) – грид-вычисления; для реализации распределенных и, возможно, параллельных вычислений инфраструктура грид-системы образует виртуальный суперкомпьютер, доступный, как правило, нескольким организациям или проектным группам; виртуальный компьютер при этом может содержать как отдельные взаимосвязанные компьютеры, так и целые кластеры компьютеров, связанных высокоскоростными коммутаторами.

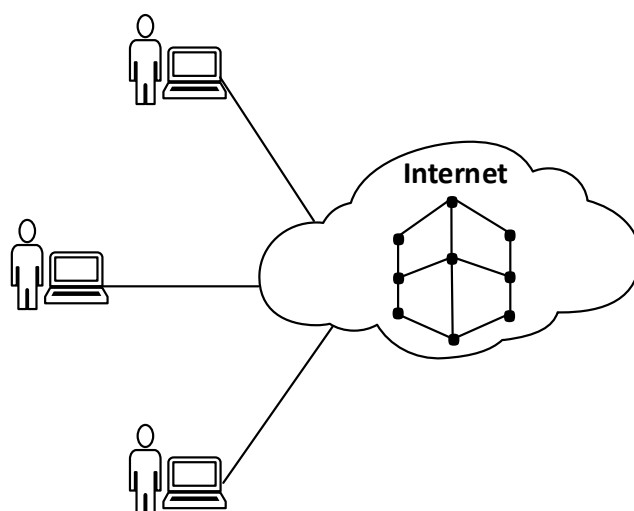


Рис. 12. Концептуальная модель Grid Networking Cluster (виртуального сетевого кластера)

– Public Grid Computing (PGC) – грид-системы, открытые для коллективного пользования и образуемые свободно предоставляемыми вычислительными и коммуникационными ресурсами.

– Grid Computing on Demand (GCoD) – грид-системы типа GCoD; данные системы обычно являются коммерческими приложениями, организуемыми по требованию клиентов.

– Cloud Grid Computing (CGC) – грид-системы, организуемые в облачных предметно-ориентированных средах (рис. 13).

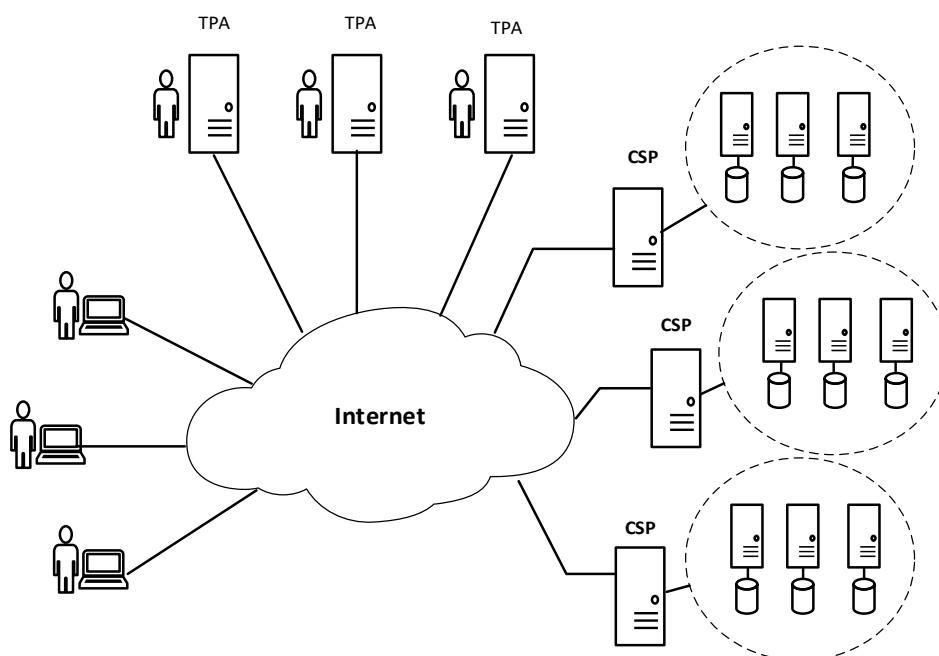


Рис. 13. Концептуальная модель облачной грид-системы CGC (Cloud Grid Computing) с несколькими сторонними аудиторами TPA (Third Party Auditors) и провайдерами облачных сервисов CSP (Cloud Services Provider)

– Cloud Computing (CC) – облачные вычисления, реализуемые как частный случай коммунальных вычислений и предполагающие использование вычислительных ресурсов по типу использования ресурсов сети Интернет.

– Types of Clouds – разновидности облачных вычислений.

– Private Clouds (PrC) – «приватные, или частные, облака», т.е. облачная среда, реализуемая в вычислительной инфраструктуре, принадлежащей клиенту.

– Public Clouds (PubC) – общедоступная облачная среда.

– Hybrid Clouds (HC) – гибридная облачная среда, включающая как общедоступные, так и частные облачные ресурсы.

– Community Clouds (ComC) – облачная среда, коллективно используемая сообществом организаций.

– Clouds without TPA (CwoTPA) – облачная среда, предоставляющая вычислительные услуги или услуги хранения без участия стороннего аудитора (TPA – Third Party Auditor, аудитор третьей стороны, или сторонний аудитор).

– Clouds with TPA (CwTPA) – облачная среда, предоставляющая вычислительные услуги или услуги хранения при участии стороннего аудитора TPA.

– Services – сервисы облачной среды.

– Everything as a Service (EaaS) – «все» как сервис.

– Grid Computing as a Service (GCaaS) – грид-вычисления как сервис.

– Cluster Computing as a Service (CCaaS) – кластерные вычисления как сервис (рис. 14).

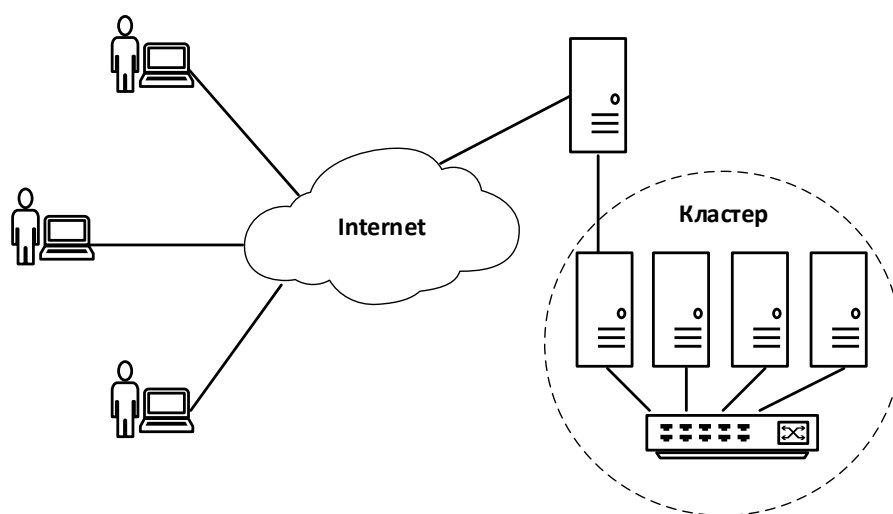


Рис. 14. Концептуальная модель CCaaS – Cluster Computing as a Service (кластерные вычисления как сервис)

- Network Computing as a Service (NCaaS) – сетевые вычисления как сервис.
- Software as a Service (SaaS) – программное обеспечение как сервис.
- Platform as a Service (PaaS) – платформа как сервис.
- Infrastructure as a Service (IaaS) – инфраструктура как сервис.
- Communications as a Service (CaaS) – коммуникации как сервис.
- Data Bases as a Service (DBaaS) – базы данных как сервис.
- Data Base Management System as a Service (DBMSaaS) – системы управления базами данных как сервис.
- Third Party Auditing as a Service (TPAaaS) – сторонний аудит как сервис.
- Monitoring as a Service (MaaS) – мониторинг использования облачной системы как сервис.
- Jungle Computing (JC) – форма организации распределенных вычислений, реализуемая в кластерных, грид и облачных средах на основе произвольной имеющейся инфраструктуры (Jungle в переводе с английского означает дебри, джунгли).

Цель и задачи лабораторного практикума

Лабораторный практикум и приведенные в нем примеры распределенных приложений позволяют получить опыт реализации большинства распределенных архитектур, поскольку почти все из них можно свести к частным случаям реализации архитектур P2P и «клиент-сервер». В результате изучения дисциплины бакалавры должны:

- знать основные принципы проектирования клиент-серверных и подобных приложений, в том числе с использованием технологий баз данных; классификацию и сравнительные характеристики распределенных вычислительных систем; базовые понятия клиент-серверных и родственных технологий;
- уметь проектировать сетевые прикладные программы, используя клиент-серверную и родственные ей технологии; самостоятельно обучаться использованию современных визуальных объектно ориентированных средств программирования клиент-серверных баз данных;
- владеть понятийным аппаратом теории моделирования клиент-серверных баз данных и уметь использовать методы анализа предметной области на инфологическом, логическом и физическом

уровнях, понимая основные закономерности функционирования в клиент-серверной среде.

Перечень вопросов, рекомендуемых к изучению для выполнения лабораторных и самостоятельных работ

1. Концепция и особенности объектно ориентированного подхода при использовании языков программирования для разработки сетевых приложений.

2. Основные принципы, методы и перспективы разработки объектно ориентированных программ и сетевых приложений на основе технологий ведущих фирм.

3. Фундаментальные методы и свойства сетевой архитектуры и механизмы ее программной реализации в Windows и Web-приложениях.

4. Понятие о сетевой архитектуре. Общие представления о процессе передачи данных по сети.

5. Понятие протокола и механизмы взаимодействия системы «клиент-сервер».

6. Обзор общих принципов построения многоуровневых приложений.

7. Работа с сетью на уровне сокетов и потоков.

8. Способы доступа к ресурсам сети из программных приложений при помощи URL.

9. Концептуальные основы одного из языков программирования для разработки сетевых приложений (C++, C#, Java, Python).

10. Средства языка для организации работы в сети. Основные классы и интерфейсы реализации сетевого взаимодействия.

11. Распределенная обработка данных. Основы работы в сети. Клиент-сервер. Прокси-серверы. Адресация Internet. Сетевые классы и интерфейсы.

12. Обзор сокетов. Зарезервированные сокет. Сокеты TCP/IP клиентов. Сокеты TCP/IP серверов. Дейтаграммы.

13. Использование URL. Основные классы и интерфейсы реализации сетевого взаимодействия.

14. Проектирование и разработка приложений в архитектуре «клиент-сервер» с организацией взаимодействия с базой данных.

15. Понятия и терминология ODBC-JDBC. Связь JDBC и ODBC. Настройка базы данных. DriverManager. Создание соединения с источником данных. URL и ODBC-JDBC.

16. Более сложные соединения. Драйвер JDBC-ODBC Bridge. Получение метаданных для множества результатов.

17. Концепция распределенной обработки данных и технологии удаленной обработки данных. Понятие и архитектура распределенной системы.

18. Протоколы и программная реализация удаленного вызова процедур. Объектно ориентированные вызовы удаленных методов.

19. Архитектура решений, основанных на Web Services. Протоколы и стандарты.

20. Публикация и развертывание служб.

Часть 1

СЕТЕВЫЕ КЛИЕНТ-СЕРВЕРНЫЕ ТЕХНОЛОГИИ

Тема 1.1. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP

При работе над темой изучаются протоколы семейства TCP/IP, функции и методы стандарта WINSOCK, определяющего сетевой интерфейс для программирования сокетов в ОС Microsoft Windows, а также принципы разработки программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP.

После изучения данной темы необходимо разработать программу клиента, которая должна:

- запрашивать у пользователя адрес программы-сервера;
- устанавливать соединение с сервером;
- передавать на сервер данные;
- принимать ответ от сервера и выводить его на экран;
- закрывать соединение с сервером.

Основные сведения

Для того чтобы установить связь с процессом, выполняющимся на другой ЭВМ, клиентская программа должна создать сокет. Сокет в любой современной системе представляет собой особый вид файла, из которого можно читать и в которой можно записывать двоичные данные. При операциях обмена с сокетом нет никакого контроля типов, эта задача возлагается на приложения. На схеме алгоритма (рис. 15) блок «Создание сокета» подразумевает вызов функции `socket`, который в случае успеха создает сокет – потоковое или дейтаграммное – и семейство протоколов. В данном случае создается потоковый сокет и используется семейство протоколов TCP/IP.

Установка соединения с другим процессом заключается в обмене специальными пакетами и, возможно, только тогда, когда тот процесс ожидает приема соединений. В противном случае результат операции будет неудачным и будет получено сообщение о том, что либо не удалось установить соединение, либо оно было разорвано (зависит от реализации).

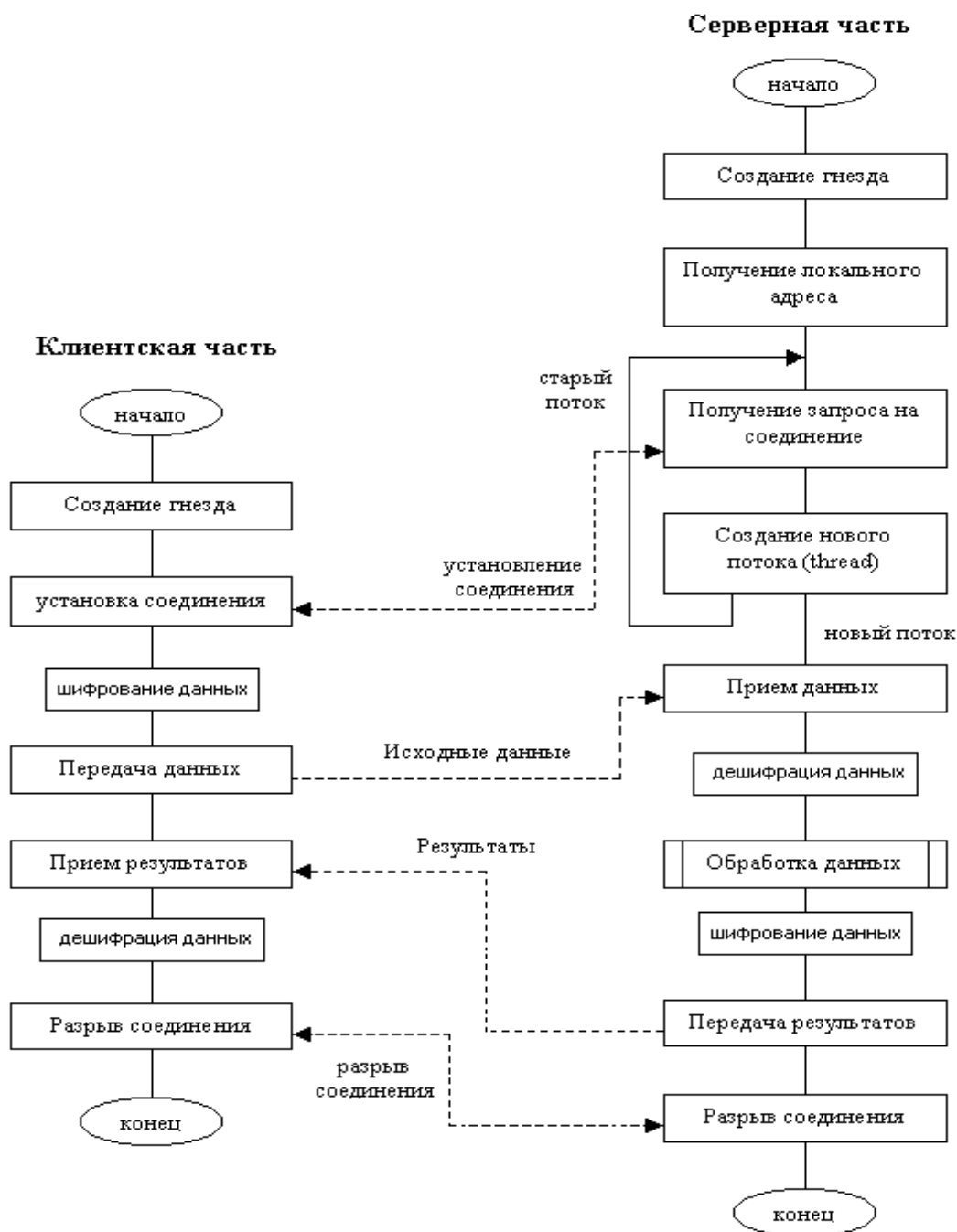


Рис. 15. Алгоритм работы клиент-серверного приложения

Для установки соединения требуется указать ЭВМ по IP-адресу или по доменному имени, которое обязательно должно быть преобразовано в IP-адрес, и процесс на этой ЭВМ (по целочисленному идентификатору, называемому портом). Все это реализуется с помощью функции `connect`. Если соединение успешно установлено, то сразу после вызова этой функции можно вести обмен с гарантированной доставкой пакетов. В противном случае работа невозможна.

Передача и прием данных, т.е. обмен с сокетом, производятся всеми доступными в системе средствами обмена с файлами, например, системные вызовы *read* и *write* в UNIX и Windows, библиотечные функции *fprintf*, *fgets* и др. Перед осуществлением передачи данные, если это необходимо, шифруются каким-либо алгоритмом. На принимающей стороне полученные данные дешифруются, и определяется (или опровергается) их подлинность, от результата чего зависит дальнейшая работа с этим клиентом.

Разрыв соединения означает обмен специальными пакетами и может производиться с помощью системного вызова *close*. Функция *close* уничтожает сокет, делая его непригодным к использованию (любая операция с ним будет заканчиваться неудачей).

Указание: для использования сокетов в проекте на языке C++ в системе программирования Microsoft Visual Studio 2010 необходимо подключить библиотеку *winsock2*, добавив в заголовок исходного файла следующие директивы:

```
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")
```

Перед началом работы с сокетами необходимо инициализировать библиотеку. Это делается с помощью функции *WSAStartup()*. Прототип функции следующий:

```
int WSAStartup (WORD wVersionRequested, LPWSADATA
lpWSADATA )
```

Здесь *wVersionRequested* – требуемая версия WinSock. Например, для задания версии 2.0 следует указать в качестве этого параметра *MAKEWORD(2, 0)*. *lpWSADATA* – указатель на данные среды WinSock. Необходимо объявить переменную типа *WSADATA* и передать ее адрес. Если функция выполнилась успешно, возвращается 0, иначе – значение *SOCKET_ERROR*. Код ошибки можно получить с помощью функции *WSAGetLastError()*.

Т е м а 1.2. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP

При работе над темой изучаются принципы разработки программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP и библиотеки WinSock.

Программа сервера должна:

- ожидать запросы от программ клиентов на соединение;
- устанавливать соединение с клиентами; принимать данные от клиентов и выполнять их обработку;
- пересылать результат обработки клиенту.

Основные сведения

Основной задачей серверной части является обработка данных. Обмен данными с клиентскими процессами есть важная составляющая часть этой задачи.

Для того чтобы процессы-клиенты могли связаться с сервером, сервер создает сокет для обмена данными. На схеме алгоритма (см. рис. 15) это представлено блоком «Создание сокета» (гнезда). Производится так же, как и в клиентской программе.

Следующий блок – «Получение локального адреса» – принципиально важен. Он служит для того, чтобы все запросы на соединение, приходящие на данную ЭВМ и обращающиеся к указанному порту, операционная система направляла данному процессу. Операция производится посредством системного вызова *bind*, в котором указывается созданный ранее сокет, IP-адрес ЭВМ (как правило, это константа 0) и идентификатор процесса, т.е. порт. После этого, в случае успеха, программа сервера вызывает функцию *listen*, которая говорит операционной системе о том, что процесс ожидает поступления запросов на соединение к данному сокету и что эти запросы нужно ставить в очередь указанной длины.

Получение запроса на соединение происходит тогда, когда клиентский процесс вошел в блок «Установка соединения», т.е. вызвал функцию *connect*. ОС сервера при этом создает копию сокета, чтобы программа могла на первом экземпляре продолжить работу, а на другом – вести обмен с подключившимся клиентом. Следующий блок – «Создание нового потока» – подразумевает порождение новой параллельной ветки программы, которая будет вести обработку данных. В системах Windows это обычно нить (*thread*), создаваемая с помощью функции *_beginthread*, в UNIX-системах это новый процесс, создаваемый с помощью вызова *fork*.

Передача и прием данных, т.е. обмен с сокетом, производятся всеми доступными в системе средствами обмена с файлами, например: системные вызовы *read* и *write* в UNIX и Windows, библиотечные функции *fprintf*, *fgets* и др. После приема данных они дешифрируются с целью, во-первых, получить открытый текст и, во-вторых, чтобы определить подлинность данных. Затем, если

установлена подлинность данных и получен корректный открытый текст, производится обработка данных. Перед отправкой клиенту результатов работы данные снова шифруются.

По окончании работы с клиентом серверный процесс закрывает свою копию сокета и уничтожается.

Указание: для использования сокетов в проекте на языке C++ в системе программирования Microsoft Visual Studio 2010 необходимо подключить библиотеку winsock2, добавив в заголовок исходного файла следующие директивы:

```
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")
```

Т е м а 1.3. Использование методов криптографии для закрытия данных, передаваемых между программами клиента и сервера

Изучаются методы обеспечения закрытия данных (шифрования) при передаче их по сети между программами клиента и сервера.

Функциональная возможность при взаимодействии в системе «клиент-сервер» должна:

- обеспечить закрытие данных при передаче их по сети;
- обеспечить корректное расшифровывание передаваемых данных.

Основные сведения

Для обеспечения закрытия данных, передаваемых между клиентом и сервером по сети, выполняется их шифрование при передаче и расшифровка при приеме.

Для шифрования используется алгоритм гаммирования. Алгоритм заключается в наложении гаммы на открытый текст (побитовое сложение открытого текста с некоторой гаммой по модулю 2), в результате чего получается зашифрованный текст, который и передается по сети.

Данный алгоритм является симметричным. Поэтому для получения открытого текста из зашифрованного на принимающей стороне необходимо наложить путем сложения по модулю 2 на принятый зашифрованный текст ту же самую гамму.

Гамма, используемая в алгоритме шифрования, получается путем преобразования открытого ключа (пароля) в соответствии

с разработанным алгоритмом. Для того, чтобы клиент и сервер могли шифровать и расшифровывать полученные данные, им необходимо знать открытый ключ. Ключ должен передаваться между клиентом и сервером без использования сети, чтобы исключить возможность его перехвата и нарушения секретности передаваемой информации.

Пример:

```
/* CRYPT.CPP Функции шифрования */

#include <string.h>

// функция преобразования строки пароля в гамму шифра
unsigned short gamma(char *pwd)
{
    char buf[20];
    int i;
    unsigned long flag;
    static unsigned long g;

    if(pwd)
    {
        memset(buf,0x55,20); // UUUUUUUUUUUUUUUUUUUUUUUUUU
        memcpy(buf,pwd,strlen(pwd)); //passwordUUUUUUUUUUUUUUU
        for(i=0, g=0;i<20;i++) // свертка пароля
            g += (unsigned long)(buf[i]<<(i%23));
    }

    for(i=5;i>0;i--) // циклический сдвиг на 5 разрядов вправо
    {
        flag = g & 1;
        g = g>>1;
        if(flag) g |= 0x80000000;
    }

    return g; // вернуть значение гаммы
}

// шифрует открытый текст source по паролю pwd
// и записывает зашифрованный текст в dest
// *** шифрование симметричное!
int crypt(char *source, char *dest, char *pwd)
{
    int i, len, numblk;
    unsigned short *px, *py, g;

    px = (unsigned short *)source; // указатель на открытый текст
    py = (unsigned short *)dest; // указатель на буфер для зашифрованного текста
    g = gamma(pwd); // получить гамму шифра
    len = strlen(source); // сосчитать длину открытого текста
    numblk = (len+1)/4; // сосчитать число 32-разрядных блоков
```

```

for(i=0; i < numblk; i++, py++, px++) // цикл гаммирования открытого
*py = *px ^ gamma(0); // текста блоками по 32 бита

dest[numblk*4] = 0; // вставить завершающий символ 0

return numblk*4+1; // вернуть число байт в зашифрованном тексте
}

```

Т е м а 1.4. Комплексная отладка программ клиента и сервера

При работе над темой изучаются методы комплексной отладки программ клиента и сервера и проверки их работоспособности.

Проверка работоспособности приложений «клиент-серверной» технологии включает:

- проверку приложений на одной машине ЛВС;
- проверку приложений на разных машинах ЛВС.

Основные сведения

Система состоит из двух частей: серверной и клиентской. Они могут работать как на одной ЭВМ, так и на разных. Обе написаны на языке C на основе библиотеки WinSock и работают в системе Windows. Для простоты анализа они выполнены в виде консольных приложений и поэтому могут быть легко перенесены в среду UNIX. Ниже приводится схема алгоритма работы системы, на которой отражены все основные потоки информации.

Серверная часть представляет собой программу, которая в бесконечном цикле ожидает поступления от клиентов запросов на соединение, и при поступлении такого запроса она порождает поток, который обслуживает нового клиента: принимает от него данные, обрабатывает их и посылает обратно результаты. После этого он разрывает соединение и прекращает свое существование. Старый же поток продолжает «слушать» сеть и ожидать запросы на соединения. Остановка серверной программы не предусмотрена и предполагается путем принудительного уничтожения процесса.

Клиентская часть представляет собой программу, которая устанавливает соединение с сервером, передает ему свои данные и ожидает получения результатов. После приема результатов соединение разрывается.

Т е м а 1.5. Передача и прием пакетов с подтверждением через протокол UDP

При работе над темой изучаются принципы разработки программы-сервера и программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP и библиотеки WinSock. В процессе изучения темы необходимо разработать программу сервера и программу клиента, которые должны устанавливать соединение между собой и обмениваться данными.

Основные сведения

Для того чтобы процессы-клиенты могли связаться с сервером, сервер создает сокет для обмена данными с помощью системного вызова *socket*. Данная операция также производится и в клиентской программе.

Далее выполняется операция получения локального адреса. Это нужно для того, чтобы все запросы на соединения, приходящие на данную ЭВМ и обращающиеся к указанному порту, операционная система направляла данному процессу. Операция производится посредством системного вызова *bind*, в котором указывается созданный ранее сокет, IP-адрес ЭВМ (как правило, это константа 0) и идентификатор процесса.

Далее в цикле происходит прием данных через сокет от клиента с помощью системного вызова *recvfrom*. Принятая строка выводится на экран и затем отсылается обратно клиенту с помощью системного вызова *sendto*. Работа программы-сервера завершается в том случае, если строка начинается с символа 'x'.

Работа программы-клиента во многом схожа с работой программы-сервера. Вначале создается сокет для обмена данными и вызывается функция *bind* для получения локального адреса. Далее в цикле пользователь набирает строки, которые отсылаются на сервер (системный вызов *sendto*) и принимает данные, присланные с сервера (системный вызов *recvfrom*). Цикл выполняется до тех пор, пока не получена строка, начинающаяся с символа 'x'.

По окончании работы процесс-клиент и процесс-сервер закрывают свои копии сокетов (системный вызов *closesocket*) и уничтожаются.

Т е м а 1.6. Создание и закрытие сокета в проекте Visual C++

Цель работы над темой – овладение навыками создания сокетов в среде Visual C++ пакета Microsoft Visual Studio 2010 для различных протоколов (IPX, SPX, TCP).

- В процессе работы необходимо:
- инициализировать среду WinSock;
 - создать по одному сокету для каждого протокола (IPX, SPX, TCP);
 - добиться правильного выполнения функции `bind()`;
 - закрыть сокеты;
 - завершить работу со средой WinSock.

Основные сведения

Сокеты представляют собой удобный способ передачи данных от одного приложения к другому по сети. Под сокетом понимается логический порт, через который сетевая станция ведет обмен данными с другими сетевыми станциями. Для идентификации сокетов используются числовые значения. На одной станции может быть создано несколько сокетов, причем некоторые из них могут использоваться для передачи, а некоторые – для приема пакетов. Последовательность действий для приема-передачи данных в сети по протоколам транспортного уровня показана на рис. 16.

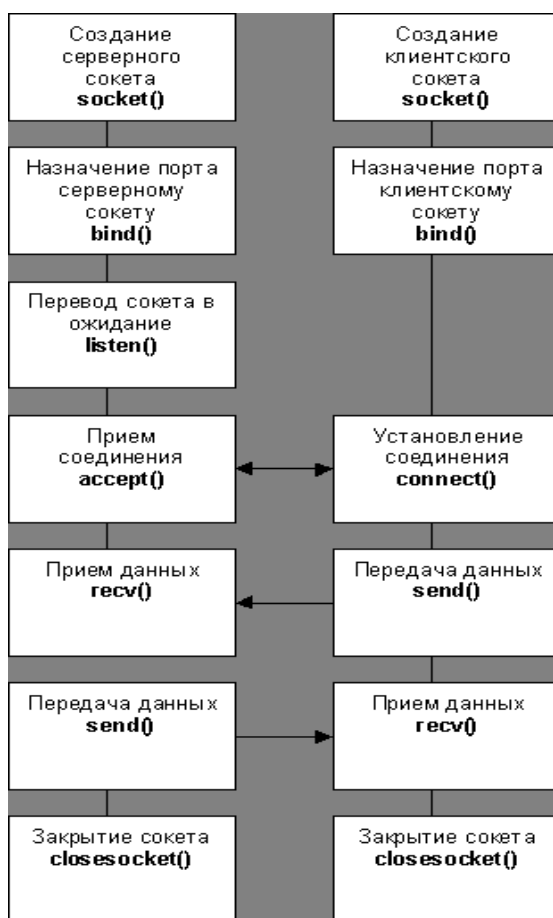


Рис. 16. Последовательность действий при обмене данными

При обмене данными по протоколам сетевого уровня последовательность действий несколько иная (рис. 17).

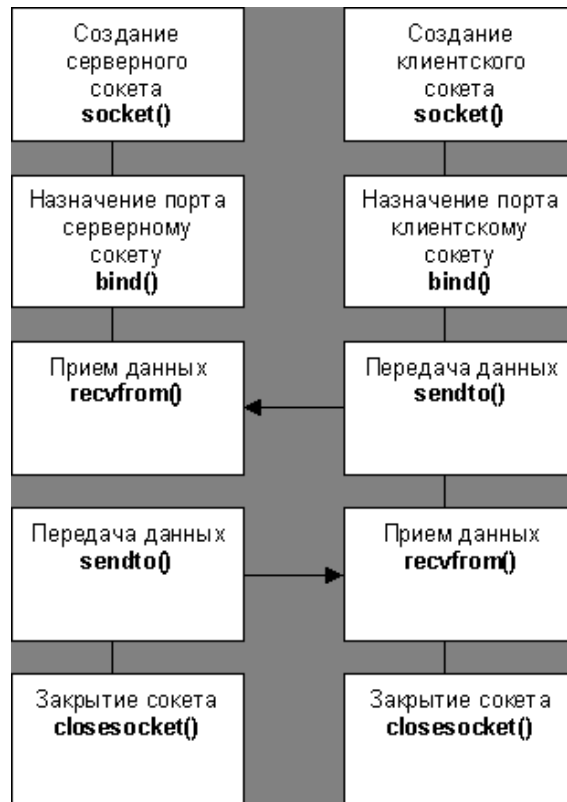


Рис. 17. Обмен данными по протоколам сетевого уровня

Пример:

```
WSADATA          IpWSAData; // структура для WSASStartup
if(WSASStartup(MAKEWORD(2, 0), & IpWSAData) != 0) {
    // не удалось инициализировать библиотеку WinSock
    cout<<"WSAStartup(...)"
error"<<WSAGetLastError()<<endl;
}
```

Для создания сокетов в Visual C++ используется функция `socket()`.

Прототип этой функции следующий:
`SOCKET socket (int af,int type,int proto);`

Параметры этой функции следующие:

`Af` - семейство адресов. Наиболее часто используемые семейства:

`AF_INET` - протоколы Интернет (TCP/IP, UDP)

`AF_IPX` - протоколы Novell Netware (IPX/SPX)

Примечание. Для работы с протоколами IPX/SPX необходимо включить заголовочный файл `<wsipx.h>`

`Type` – тип сокета

Обычно используются значения:

SOCK_STREAM – для передачи потока данных, используется в протоколах SPX и TCP. SOCK_DGRAM – для передачи дэйтограмм, используется в протоколах IP, UDP, IPX. SOCK_SEQPACKET – для передачи последовательных пакетов данных. Используется протоколом SPX. Отличается от SOCK_STREAM тем, что ожидается установка специального бита окончания передачи сообщения. Proto – протокол передачи данных.

Часто используемые протоколы следующие:

IPPROTO_IP - протокол IP;
IPPROTO_TCP - протокол TCP;
IPPROTO_UDP - протокол UDP;
NSPROTO_IPX - протокол IPX;
NSPROTO_SPX - протокол SPX;
NSPROTO_SPXII - протокол SPXII;

Создание сокета заключается в вызове функции socket() с нужными параметрами. Если сокет успешно создан, то возвращается его идентификатор (хэндл), в противном случае возвращается значение INVALID_SOCKET. Значение кода ошибки получается функцией WSAGetLastError().

Пример:

```
SOCKET      s_tcp, s_ipx, s_spx; // сокеты для трех протоколов
s_tcp = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
           // создать сокет TCP
if(s == INVALID_SOCKET) {
    cout<<"socket(...) for TCP error"
    <<WSAGetLastError()<<endl;
}
s_ipx = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);
           // создать сокет IPX
if(s == INVALID_SOCKET) {
    cout<<"socket(...) for IPX error"
    <<WSAGetLastError()<<endl;
}
s_spx = socket(AF_IPX, SOCK_STREAM, NSPROTO_SPX);
           // создать сокет TCP
if(s == INVALID_SOCKET) {
    cout<<"socket(...) for SPX error"
    <<WSAGetLastError()<<endl;
}
```

После создания сокета необходимо настроить его на определенный порт (для TCP/IP) или номер сокета (для IPX/SPX).

Для этого используется функция `bind()`. Прототип функции следующий:

```
int bind (SOCKET s,const struct sockaddr FAR* name,int namelen);
```

Параметры этой функции:

S – хэндл сокета.

Name – указатель на структуру типа `SOCKADDR`

Namelen – длина структуры `SOCKADDR`

Структура `SOCKADDR` предназначена для описания сетевого адреса в различных протоколах.

```
struct sockaddr {  
    u_short    sa_family;  
    char sa_data[14];  
};
```

sa_family – семейство адресов (то же, что при создании сокета)

sa_data – данные адреса.

В чистом виде эта структура почти никогда не используется. Для работы с конкретными протоколами используются похожие структуры:

Для IP-адресов:

```
struct sockaddr_in {  
    short                sin_family;  
    u_short              sin_port;  
    struct in_addr sin_addr;  
    char                sin_zero[8];  
};
```

В этой структуре

sin_family - `AF_INET`;

sin_port - используемый порт (байты располагаются в обратном порядке. Необходимо использовать функцию `htol()`).

sin_addr - IP адрес, его структура следующая:

```
struct in_addr {  
    union {  
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;  
        struct { u_short s_w1,s_w2; } S_un_w;  
        u_long S_addr;  
    } S_un;  
};
```

Таким образом, можно заполнять IP адрес как 32-разрядным числом, так и двумя 16-разрядными числами или 4 байтами. sin_zero – нули, нужны для дополнения структуры до размера `SOCKADDR`.

Для IPX-адресов:

```
struct sockaddr_ipx
{
    short sa_family;
    char sa_netnum[4];
    char sa_nodenum[6];
    unsigned short sa_socket;
}
sa_family - AF_IPX
sa_netnum - номер сети
sa_nodenum - номер узла
sa_socket - номер сокета
```

При использовании функции *bind()* для создания серверного сокета в передаваемой структуре адреса должно быть заполнено поле *sa_family*, а остальные поля должны содержать все нули, кроме номера порта (для TCP) или номера сокета (для IPX и SPX). При создании клиентского сокета номер порта или номер сокета также должны быть равны 0. Если функция выполнена успешно, возвращается 0, иначе – `SOCKET_ERROR`.

Пример:

```
sockaddr_in          sa;      // структура данных серверного
сокета TCP
unsigned short        port = 2000; // порт сервера
sa.sin_family = AF_INET; // заполнение структуры адреса
sa.sin_port = htons(port);
sa.sin_addr.s_addr = htonl(INADDR_ANY);
// получение локального адреса
if (bind(s_tcp,(sockaddr*)&sa,sizeof(sa))==SOCKET_ERROR) {
    // не удалось
    cout<<endl<<"bind(...) for TCP error"
        <<WSAGetLastError()<<endl;
    closesocket(s_tcp);
}
sockaddr_ipx          sa; // структура адреса серверного сокета IPX
unsigned short         socketnum = 2000; // Номер сокета
memset(&sa, 0, sizeof(sa)); // Обнуление структуры адреса
sa.sa_family = AF_IPX; // заполнение структуры адреса
sa.sa_socket = socketnum;
// получение локального адреса
if (bind(s_ipx,(sockaddr*)&sa,sizeof(sa))==SOCKET_ERROR) {
    // не удалось
    cout<<endl<<"bind(...) for IPX error"
        <<WSAGetLastError()<<endl;
    closesocket(s_ipx);
}
sockaddr_ipx          sa; // структура адреса серверного сокета SPX
```

```

unsigned short          socketnum = 2000; // Номер сокета
memset(&sa, 0, sizeof(sa)); // Обнуление структуры адреса
sa.sa_family = AF_IPX; // заполнение структуры адреса
sa.sa_socket = socketnum;
// получение локального адреса
if (bind(s_spx,(sockaddr*)&sa,sizeof(sa))==SOCKET_ERROR) {
    // не удалось
    cout<<endl<<"bind(...) for SPX error"
        <<WSAGetLastError()<<endl;
    closesocket(s_spx);
}

```

Заккрытие сокета выполняется функцией `closesocket()`.
Прототип функции следующий:

```

int closesocket (
    SOCKET s
);

```

В качестве параметра передается хэндл сокета. Если закрытие произошло успешно, то возвращается 0. В противном случае возвращается значение `SOCKET_ERROR`. Код ошибки получается функцией `WSAGetLastError()`.

Пример:

```

if(closesocket(s_tcp)==SOCKET_ERROR) {
    cout<<endl<<"closesocket(...) for TCP error"
        <<WSAGetLastError()<<endl;
    WSACleanup();
    return 0;
}

```

Перед завершением работы необходимо закончить использование DLL WinSock. Это делается с помощью функции `WSACleanup()`. Функция не требует параметров. Если функция выполнялась успешно, возвращается 0, иначе – значение `SOCKET_ERROR`. Код ошибки можно получить с помощью функции `WSAGetLastError()`.

Пример:

```

if(WSACleanup() == SOCKET_ERROR) {
    cout<<endl<<"WSACleanup() error"
        <<WSAGetLastError()<<endl;
}

```

Т е м а 1.7. Установление соединения для сокета, созданного в проекте Visual C++

В процессе изучения темы необходимо научиться устанавливать соединения для сокетов, работающих по протоколам

транспортного уровня (TCP и SPX). При установлении соединения следует:

- создать сокет для протокола TCP, перевести его в «слушающее» состояние;
- вызвать для этого сокета функцию `accept()`;
- в другом приложении создать сокет на порте 0;
- установить соединение со «слушающим» сокетом;
- получить на обоих концах соединения данные о противоположной стороне;
- закрыть сокеты;
- выполнить пункты 1–6 для протокола SPX.

Основные сведения

Соединения устанавливаются только для протоколов транспортного уровня, таких как TCP или SPX. Для протоколов сетевого уровня соединение установить невозможно. Установление соединения в Visual C++ производится двумя типами сокетов, один из которых является серверным, а другой – клиентским.

Серверный сокет создается с указанием конкретного порта (для TCP) или номера сокета (для SPX). Далее для него вызывается функция `listen()`. Эта функция имеет следующий прототип:

```
int listen (SOCKET s, int backlog );
```

Здесь `s` – созданный сокет, `backlog` – количество одновременно обрабатываемых соединений. При использовании блокирующих функций, таких как `accept`, `send` и `recv`, значение этого параметра разумно установить равным 1.

Если функция выполнилась успешно, возвращается 0, иначе – значение `SOCKET_ERROR`. Код ошибки можно получить с помощью функции `WSAGetLastError()`. Данная функция переводит сокет в «слушающее» состояние. После ее вызова будут обрабатываться все устанавливаемые извне соединения. Сама функция является неблокирующей, т.е. завершается сразу после вызова.

Пример:

```
// Объявления переменных обычные
if (listen(s_tcp,5)==SOCKET_ERROR) {
// не удалось перевести сокет в состояние ожидания
cout<<endl<<"listen(...) for TCP error"
<<WSAGetLastError()<<endl;
closesocket(s_tcp);
}
```

Для приема соединения на серверной стороне должна вызываться функция `accept()`. Ее прототип следующий:

`SOCKET accept (SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen)`. Здесь `s` – «слушающий» сокет, `addr` – указатель на структуру типа `SOCKADDR`, в которую запишется адрес вызывающего компьютера, `addrlen` – указатель на длину этой структуры. Функция возвращает идентификатор нового сокета, который следует использовать в качестве параметра функций обработки данных. В случае ошибки возвращается значение `INVALID_SOCKET`.

Функция `accept` является блокирующей, т.е. выполнение программы (или потока, в котором вызвана функция) прекращается до поступления запроса на соединение извне. При поступлении такого запроса соединение устанавливается и выполнение программы продолжается.

Пример:

```
SOCKET sw; // сокет, на который принимаются соединения
sockaddr_in sa; // структура IP адреса
int a; // для служебных целей
a = sizeof(sockaddr_in);
sw = accept(s_tcp, (sockaddr*)&sa, &a);
// принять соединение
if(sw == INVALID_SOCKET) {
    cout<<endl<<"accept(...) for TCP error"
    <<WSAGetLastError()<<endl;
    closesocket(s_tcp);
}
```

Клиентский сокет создается с указанием в качестве номера порта или номера сокета значения 0. В противном случае возможно появление неожиданных ошибок. Для установления соединения с сервером используется функция `connect()`. Прототип этой функции:

`int connect (SOCKET s, const struct sockaddr FAR* name, int namelen)`.

Здесь `s` – идентификатор созданного сокета, `name` – указатель на структуру типа `SOCKADDR`, задающую адрес компьютера, с которым устанавливается соединение. `namelen` – длина этой структуры.

В структуре адреса необходимо указать все данные сервера, включая номер порта или номер сокета.

Если функция выполнялась успешно, возвращается 0, иначе – значение `SOCKET_ERROR`. Код ошибки можно получить с помощью функции `WSAGetLastError()`. Функция является полублокирующей, т.е. завершается либо при установлении соединения, либо когда ОС посчитает, что это невозможно.

Пример:

```
// Подключение к локальному компьютеру
unsigned short port = 2000; // порт сервера
sockaddr_in sa; // структура IP адреса
sa.sin_family = AF_INET; // заполнить структуру адреса TCP
sa.sin_port = htons(port);
sa.sin_addr.s_addr = inet_addr("127.0.0.1");
if(connect(s_tcp, (sockaddr*)&sa, sizeof(sa)) == SOCKET_ERROR) {
// не удалось установить соединение
cout<<"connect(...) for TCP error"
<<WSAGetLastError()<<endl;
closesocket(s_tcp);
}
```

Как видно из примера, для удобного заполнения структуры IP-адреса используется функция `inet_addr()`. В качестве параметра этой функции передается строка с IP или URL адресом в стандартной нотации (например, "127.0.0.1" или "microsoft.com"). Функция возвращает 32-разрядное число, соответствующее этому адресу. В случае ошибки функция возвращает значение `INADDR_NONE`.

После установления соединения данные о подсоединенном компьютере можно получить с использованием функции `getpeername()`. Прототип этой функции:

```
int getpeername (SOCKET s, struct sockaddr FAR* name, int FAR* namelen).
```

Здесь `s` – идентификатор локального сокета (для клиентской стороны это идентификатор, возвращенный функцией `accept()`);

`name` – указатель на структуру `sockaddr`, в которую будет записан адрес подсоединенного компьютера;

`namelen` – указатель переменной, куда запишется реальная длина этой структуры.

Если функция выполнялась успешно, возвращается 0, иначе – значение `SOCKET_ERROR`. Код ошибки можно получить с помощью функции `WSAGetLastError()`.

Пример:

```
sockaddr_in sa; // структура IP адреса
if(getpeername(s_tcp, (sockaddr*)&sa, sizeof(sa)) == SOCKET_ERROR) {
// не удалось получить информацию
cout<<"getpeername(...) for TCP error"
<<WSAGetLastError()<<endl;
}
```

Т е м а 1.8. Передача данных для сокета, созданного в проекте Visual C++

При изучении темы необходимо научиться передавать данные по протоколам сетевого и транспортного уровней с использованием сокетов. Для реализации процесса передачи данных следует:

- создать «слушающий» сокет для протокола TCP;
- в другом приложении создать сокет для TCP с нулевым портом;
- установить соединение;
- передать некоторые данные, убедиться в их приеме на противоположной стороне;
- закрыть сокеты;
- повторить пункты 1–5 для протокола SPX;
- создать сокет с ненулевым портом для протокола IPX;
- в другом приложении создать сокет для IPX с нулевым портом;
- передать данные, убедиться в их приеме;
- закрыть сокеты.

Основные сведения

Передача для сокетов с установлением соединения и без установления соединения выполняется разными функциями.

Для сокетов с установлением соединения (для протоколов TCP и SPX) используются функции `send` и `recv`.

Для передачи данных по установленному соединению используется функция `send`. Прототип этой функции следующий:

```
int send (SOCKET s, const char FAR * buf, int len, int flags )
```

Здесь `s` – идентификатор локального сокета (для серверной стороны это идентификатор возвращенный функцией `accept()`);

`buf` – указатель на буфер данных для отправки;

`len` – длина буфер данных;

`flags` – флаги передачи. Для простой передачи в качестве этого параметра следует указать 0.

Функция возвращает количество реально переданных байт. В случае ошибки возвращается значение `SOCKET_ERROR`. Код ошибки получается с помощью функции `WSAGetLastError()`. Следует отметить, что успешное завершение этой функции не означает, что данные были получены противоположной стороной.

Для подтверждения приема следует использовать механизм квитирования.

Пример:

```
// Объявление переменных соответствует предыдущим программам.
char strtosend[]="Пакет от клиента";
if(send(s_tcp, strtosend, strlen(strtosend), 0)== SOCKET_ERROR) {
// не удалось отправить данные
cout<<"send(...) for TCP error!"
<<WSAGetLastError()<<endl;
closesocket(s_tcp);
}
```

Для приема данных по установленному соединению используется функция `recv()`. Прототип функции следующий:

`int recv (SOCKET s, char FAR* buf, int len, int flags).`

Здесь `s` – идентификатор сокета (для серверной стороны это идентификатор возвращенный функцией `accept()`);

`buf` – указатель на буфер, куда запишутся данные;

`len` – длина буфера;

`flags` – флаги приема. Обычно имеют значение 0.

Функция возвращает количество реально принятых байт. В случае ошибки возвращает `SOCKET_ERROR`. Код ошибки получается с помощью функции `WSAGetLastError()`.

Пример:

```
int len; // длина полученного пакета
char buf[100]; // буфер для данных
len = recv(s_tcp, buf, 99, 0); // прием данных от клиента
if(len == SOCKET_ERROR) {
// ошибка приема
cout<<"recv(...) for TCP error"
<<WSAGetLastError()<<endl;
len=0;
}
buf[len] = 0;
```

Для протоколов, работающих без установления соединения, таких как IPX, используются другие функции: `sendto` и `recvfrom`. Их использование ничем не отличается от функций `send` и `recv`, за исключением того, что используются дополнительные параметры, определяющие адрес.

Прототип функции `sendto()`:

```
int sendto (SOCKET s,
const char FAR * buf,
```

```
int len,
int flags,
const struct sockaddr FAR * to,
int tolen )
```

Параметры здесь имеют тот же смысл, что и в функции `send`, но добавлены еще два:

`to` – адрес, по которому отправляется сообщение;
`tolen` – длина адреса.

Пример:

```
sockaddr_ipx sa; // структура адреса серверного сокета SPX
unsigned short socketnum = 2000; // Номер сокета
char strtosend[]="Пакет от клиента";
memset(&sa, 0, sizeof(sa)); // Обнуление структуры адреса
sa.sa_family = AF_IPX; // заполнение структуры адреса
sa.sa_socket = socketnum;
// Номер сети и номер узла равны 0
if(send(s_ipx,strtosend,strlen(strtosend),0,(sockaddr*)&sa,sizeof(sa))==
SOCKET_ERROR) { // не удалось отправить данные
cout<<"send(...) for IPX error!"
<<WSAGetLastError()<<endl;}
```

Прототип функции `recvfrom` следующий:

```
int recvfrom (SOCKET s,
char FAR* buf,
int len,
int flags,
struct sockaddr FAR* from,
int FAR* fromlen )
```

Параметры здесь имеют тот же смысл, что и в функции `recv`, но добавлены еще два:

`from` – адрес, откуда пришло сообщение;
`fromlen` – указатель на переменную, куда будет записана длина адреса.

Функции `recv` и `recvfrom` являются блокирующими.

Пример:

```
int len; // длина полученного пакета
char buf[100]; // буфер для данных
sockaddr_ipx sa; // структура адреса серверного сокета SPX
len = recv(s_tcp,buf,99,0, // флаги
(sockaddr*)&sa,sizeof(sa)); // прием данных от клиента
if(len == SOCKET_ERROR) { // ошибка приема
cout<<"recv(...) for TCP error"<<WSAGetLastError()<<endl;len=0;}
buf[len] = 0;
```

Т е м а 1.9. Использование библиотеки MFC для работы с сокетами

В процессе работы над темой необходимо использовать для работы с сокетами библиотеку классов MFC. Далее необходимо выполнить все пункты предыдущей темы, но используя классы библиотеки MFC. Для протокола TCP установить соединение с использованием функции Connect, передавая в качестве параметра имя хоста.

Основные сведения

В библиотеке MFC для работы с сокетами существуют два специальных класса: CAsyncSocket и CSocket, причем второй унаследован от первого. CAsyncSocket содержит основные функции для работы с сокетами, однако он не поддерживает блокирующие функции. Для работы с блокирующими функциями используется класс CSocket.

Перед началом работы необходимо подключить DLL WinSock с помощью функции WSAStartup(). Если при создании проекта был установлен флажок для использования Windows Sockets, то инициализация и подключение DLL производятся автоматически. Для работы необходимо либо создать производный класс от CSocket и использовать объекты этого класса, либо использовать объекты класса CSocket. В первом случае возможно отслеживание возникающих событий с использованием асинхронных функций.

Создание сокета может производиться с помощью функций – членов класса Create и Socket. Функция Create используется для создания TCP сокетов.

Прототип функции: BOOL Create(UINT nSocketPort = 0,
int nSocketType = SOCK_STREAM,
LPCTSTR lpszSocketAddress = NULL)

Пример:

```
// 1. Пример создания серверного сокета для TCP
CSocket Server; // Сокет
UINT Port=2000; // Порт сервера
if (!Server.Create(Port)) {
// не удалось создать сокет
MessageBox("Ошибка при создании сокета");
}
// 2. Пример создания клиентского сокета для TCP
```

```

CSocket Client; // Сокет
if (!Client.Create()) {
// не удалось создать сокет
MessageBox("Ошибка при создании сокета");
}

```

Для создания серверного сокета нужно указать порт, для создания клиентского сокета можно вызвать функцию без параметров. Функция возвращает TRUE в случае успеха, иначе – FALSE. Код ошибки получается с помощью функции GetLastError(). При создании сокета с помощью функции Create вызывать функцию Bind не требуется. Более универсальный способ, позволяющий создавать сокеты для любого протокола, с помощью функции – члена класса Socket. Прототип этой функции:

```

BOOL Socket(int nSocketType,
long lEvent,
int nProtocolType,
int nAddressFormat)

```

Здесь nSocketType - тип сокета (SOCK_STREAM, SOCK_DGRAM...);
lEvent - отслеживаемые события. Должно быть равно (FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE)
nProtocolType - используемый протокол (см. тему №1.1)
nAddressFormat - семейство адресов (см. тему №1.1)

Функция возвращает TRUE в случае успеха, иначе – FALSE. Код ошибки получается с помощью функции GetLastError().

Пример:

```

// Создание IPX сокета
CSocket Server; // Сокет
long lEvent=FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT |
FD_CLOSE; // Отслеживаемые события
if (!Server.Socket( SOCK_DGRAM,lEvent,NSPROTO_IPX,AF_IPX)) {
// не удалось создать сокет
MessageBox("Ошибка при создании сокета");
}

```

После вызова функции Socket требуется вызвать функцию – член класса Bind(). Основные члены класса имеют имена, схожие с именами обычных функций для работы с сокетами, они также имеют сходные прототипы. Отличие в том, что не используется первый параметр – идентификатор сокета. Ниже перечислены такие функции:

```

Bind
Listen
Accept

```

GetPeerName
Connect
Send
Receive
SentTo
ReceiveFrom

Пример 1:

```
// Прием данных от клиента по протоколу SPX
CSocket Server, Work; // Два сокета (основной и рабочий)
UINT SockNum=2000;
long lEvent=FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT |
FD_CLOSE; // Отслеживаемые события
sockaddr_ipx Addr; // Структура адреса IPX
char Buf[100]; // Буфер принимаемых данных
// Создание сокета
if (!Server.Socket( SOCK_STREAM,lEvent,NSPROTO_SPX,AF_IPX)) {
// не удалось создать сокет
MessageBox("Ошибка при создании сокета");}
// Подсоединение сокета
memset(Addr, 0, sizeof(Addr)); // Обнуление структуры адреса
sa.sa_family = AF_IPX; // заполнение структуры адреса
sa.sa_socket = SockNum;
if (!Server.Bind( (sockaddr*)&Addr,sizeof(Addr))) {
// не удалось подключить сокет
MessageBox("Ошибка при подключении сокета");
}
// Перевод в ожидающее состояние
if (!Server.Listen(1)) {
// не удалось подключить сокет
MessageBox("Ошибка при переводе сокета в ожидание");
}
// Прием соединения
if (!Server.Accept(Work)) {
// не удалось соединиться
MessageBox("Ошибка при соединении");
}
// Прием данных
if (Work.Receive(Buf, sizeof(Buf)-1)<=0) {
// не удалось принять данные
MessageBox("Ошибка при принятии данных");
}
// Вывод данных
MessageBox(Buf, "Приняты данные!");
//Закрытие сокетов
Work.Close();
Server.Close();
```

Пример 2:

```
// Передача данных на сервер по протоколу SPX
CSocket Client; // Сокет клиента
UINT SockNum=2000;
char ServNetnum[4]={12,34,0,62}; // Номер сети сервера
char ServNodenum[6]={0,0,0,0,34,21}; // Номер узла сервера
long IEvent=FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT |
FD_CLOSE; // Отслеживаемые события
sockaddr_ipx Addr, // Структура адреса IPX
ServAddr; // Адрес сервера
char Buf[]="Сообщение от клиента"; // Сообщение
// Создание сокета
if (!Client.Socket( SOCK_STREAM,IEvent,NSPROTO_SPX,AF_IPX)) {
// не удалось создать сокет
MessageBox("Ошибка при создании сокета");
}
// Подсоединение сокета
memset(Addr, 0, sizeof(Addr)); // Обнуление структуры адреса
sa.sa_family = AF_IPX; // заполнение структуры адреса
if (!Client.Bind( (sockaddr*)&Addr,sizeof(Addr))) {
// не удалось подключить сокет
MessageBox("Ошибка при подключении сокета");
}
// Соединение с сервером
memset(ServAddr, 0, sizeof(ServAddr)); // Обнуление структуры адреса
сервера
sa.sa_family = AF_IPX; // заполнение структуры адреса
for (int i=0; i<6; i++) {
ServAddr.sa_netnum[i]=ServNetnum[i];
}
for (int i=0; i<4; i++) {
ServAddr.sa_nodenum[i]=ServNodenum[i];
}
ServAddr.sa_socket=SockNum;
if (!Client.Connect( (sockaddr*)&ServAddr,sizeof(ServAddr))) {
// не удалось соединиться
MessageBox("Ошибка при соединении");
}
// Передача данных
if (Client.Send(Buf, sizeof(Buf))<=0) {
// не удалось отправить данные
MessageBox("Ошибка при отправке данных");
}
//Закрытие сокетов
Work.Close();
Server.Close();
```

Функция Connect имеет, кроме того, еще одно представление, используемое для TCP сокетов:

`BOOL Connect(LPCTSTR lpszHostAddress, UINT nHostPort)`.

Здесь `lpszHostAddress` – строка, представляющая интернет-адрес. Допустимые адреса могут быть, например, такими: "123.45.67.89", "www.microsoft.com"; `nHostPort` – номер порта TCP.

Аналогичные представления имеют функции `SendTo`, `ReceiveFrom`, хотя эти функции не рекомендуется использовать при работе по TCP.

Заккрытие сокета производится автоматически при удалении объекта.

Т е м а 1.10. Использование асинхронных функций класса CSocket

При изучении темы необходимо научиться использовать асинхронные функции для приема и обработки данных в системах «клиент-сервер», а затем создать простую систему «клиент-сервер», используя классы библиотеки MFC. Для обработки данных на серверной стороне использовать асинхронные функции, а также обеспечить поддержку протоколов TCP, IPX и SPX.

Основные сведения

Асинхронные функции позволяют решить на серверной части проблему блокирующих функций. Проблема состоит в следующем: для успешного управления приложением блокирующие функции должны быть в другом потоке по отношению к главному окну. Но тогда уничтожение сокета производится вместе с уничтожением потока, при этом сокет нормально не закрывается. Удаление сокета вне потока, в котором он был создан, приводит к недопустимой операции.

Проблема решается с использованием асинхронных функций. Это функции, которые вызываются автоматически при возникновении определенных событий. Для работы серверной части наиболее полезны события прихода заявки на подключение для протоколов TCP и SPX, а также событие прихода пакета по протоколу IPX. За эти события отвечают функции – члены класса CSocket `OnAccept` и `OnReceive` соответственно.

Для работы с этими функциями необходимо создать класс – наследник от класса CSocket. В этом классе следует переписать функции `OnAccept` и `OnReceive` в соответствии со следующими прототипами:

```
void OnAccept( int nErrorCode );  
void OnReceive( int nErrorCode ).
```

Внутри функции OnAccept должна вызываться функция Асcept, а затем могут использоваться функции Send и Receive.

Внутри функции OnReceive должна вызываться функция ReceiveFrom, далее могут выполняться другие функции.

Таким образом, достаточно создать сокет описанным в предыдущей теме способом и перевести его в «слушающее» состояние функцией Listen. Далее при возникновении запросов от клиента они будут обрабатываться практически независимо от основной программы.

Т е м а 1.11. Сокеты TCP/IP в системе UNIX

В процессе изучения темы необходимо получить навыки работы с сокетами UNIX. Далее требуется создать простую систему «клиент-сервер», используя сокеты TCP/IP в системе UNIX.

Основные сведения

В качестве примера использования сокетов Unix рассмотрим программу «клавиатурное эхо». Сервер возвращает данные, которые были набраны на клавиатуре клиента.

Эта программа открывает поточный сокет TCP, привязывает сокет, устанавливает очередь на прослушивание и принимает любые запросы на соединение, поступающие на сокет. После установления сеанса связи дочерний процесс оперирует с вводом/выводом сокета. Эта программа сервера представляет модель *параллельного сервера (concurrent server)*, в которой родительский процесс слушает запросы на соединение с помощью accept(), затем использует вилку (fork()) для порождения дочернего процесса обработки задач передачи соединения. В этом примере дочерний процесс отражает все данные, принятые на соquete, обратно клиенту. Данная программа клиента может быть скомпилирована с помощью клиентских библиотек TCP/IP PATHWORKS Version 2.0. Стек TCP, использованный с ней, является стеком фирмы 3Com, хотя нет жесткой привязки к данной версии продукта.

Данное клиентское приложение весьма простое. Оно открывает сокет, затем соединяет его с адресом сервера. Когда соединение установлено, клиент набирает строку на клавиатуре и посылает эти данные на сервер.

Затем клиент использует вызов `ioctl()` для определения входящих на гнездо данных и использует `recv()` для их захватывания. Как и в программе сервера, операторы `printf()` помещаются в стратегические места работы программы, что облегчает отладку и использование приложения.

Функция `inet_addr()` преобразует межсетевой номер в виде последовательности знаков в 32-битный межсетевой номер, который должен быть послан вместе с пакетом. Эта функция межсетевой поддержки сокета имеется также в комплекте средств разработчика программного обеспечения сокетов, версия V5 фирмы DEC. Это стандартная функция поддержки UNIX.

Функция `htons()` превращает 16-битовый номер порта из хостового байтового порядка в сетевой байтовый порядок. Протокол TCP требует, чтобы его номера портов передавались как Big Endian; в компьютерах Intel эта работа выполняется оператором `htons()`. Этот номер помещается в структуру адресов гнезд в виде короткого (16 бит) целого числа без знака в сетевом байтовом порядке. Затем клиент открывает поточное гнездо TCP и осуществляет соединение со следующей кодовой структурой:

```
if(connect(sockfd,&serv_addr,sizeof(serv_addr))<0)
{
    printf("client:cannot connect to TCP server");
    return;
}
```

Отметим, что оператор адреса подается на переменную `serv_addr`. Этот адрес указывает на межсетевую структуру адресов сокетов `sock_addr_in`. Функция `sizeof(serv_addr)` определяет длину структуры имени получателя или структуру `sockaddr_in`.

Входной код программы с клавиатуры клиента является прямым. Он вызывает свою собственную функцию `getline()`, которая использует стандартную функцию `getchar()`. Информация помещается во входной буфер.

Затем используется оператор `send()` для отсылки данных на сервер с использованием файлового дескриптора сокета, полученного вызовом `socket()`. Оператор `send()` требует адреса буфера, его длину, включая пустой символ, и параметр `flags`.

В этом случае используется флаг `MSG_PUSH`, который немедленно сбрасывает буфер. Для того, чтобы получить данные, «отраженные» от сервера в режиме «эхо», клиент вводит программу приема с помощью `recv()`. Однако сначала он использует вызов

ioctl() для определения количества байтов, которые ждут приема. Этот вызов осуществляется с помощью параметра запроса FION-READ, для которого argp содержит адрес, который заполняется величиной количества байтов (коротким целым числом), ожидающих приема на сокете.

Если этот вызов не происходит, сокет закрывается, и управление возвращается из main(). В противном случае данные сокета считываются с помощью recv() и печатаются с помощью printf().

Пример:

Client.c

```
/*          * Example of a client using TCP/IP protocol stream sockets
* Include files have been given complete paths to show where
* default directories are upon installation of the PC software.*/
/* Version 1.1 includes... */
#include      <\tcip\tools\include\sockdefs.h>
#include      <\tcip\tools\include\sock_err.h>
#include      <\tcip\tools\include\netdb.h>
#include      <stdio.h>
#include      <errno.h>
#include      <\tcip\tools\include\sys\socket.h>
#include      <\tcip\tools\include\netinet\in.h>
#include      <varargs.h>
/* Version 2.0 includes ... Same filenames; sockets subdirectory takes place
of tools subdirectory on install floppies
*
*#include      <\tcip\sockets\include\sockdefs.h>
*#include      <\tcip\sockets\include\sock_err.h>
*#include      <\tcip\sockets\include\netdb.h>
*#include      <stdio.h>
*#include      <errno.h>
*#include      <\tcip\sockets\include\sys\socket.h>
*#include      <\tcip\sockets\include\netinet\in.h>
*#include      <varargs.h>
*/
/*
* Server's address and port number are hardcoded into program,
* since this program only attempts to demonstrate
* socket-to-socket communications. The server program chose
* arbitrary port number 6000, since this is not reserved under
* ULTRIX...
*/
#define SERV_TCP_PORT  6000
#define SERV_HOST_ADDR  "16.72.32.196"
char          *pname;
main()
{
    char      buf[128],c;
    int       sockfd;
```

```

        long    retcode;
        long    far *argp;
        struct  sockaddr_in    serv_addr;
        struct  sockaddr *from;

/*
* Fill in the 'serv_addr' structure with the address
* and port number (hardcoded) of the
* server with which we want to "talk" ...
*/
        serv_addr.sin_family    = AF_INET;
        serv_addr.sin_addr.s_addr =
inet_addr(SERV_HOST_ADDR);
        serv_addr.sin_port      = htons(SERV_TCP_PORT);
/* Open a TCP stream socket ... This call on the PC
* is the same as a Unix BSD call; with this version of
* the 3COM 3+Open software, we are limited to SOCK_STREAM
* and SOCK_DGRAM for the second argument ...
*/
        if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0)
        {
            printf("client: cannot open TCP stream socket");
            return;
        }
        if(connect(sockfd,&serv_addr,sizeof(serv_addr))<0)
        {
            printf("client: cannot connect to TCP server");
            return;
        }
/* Server needs a newline character attached to end of
* buffer received; thus, '\n' is placed in buf[len]
* MSG_PUSH parameter on send() call pushes data out
* immediately ... */
        printf("Starting the Client- type a message to send or 'q' to
quit:\n ");

        for (;;)
        {
            int    len;
            int    bytesrec;
            printf("Client> ");
            gets(buf);
            if (buf[0] == 'q') return;
            len = strlen(buf);
            printf("[%d](%s)\n",len,buf);
            buf[len]='\n';
            buf[len+1]='\000';
            if(send(sockfd, buf, len+1, MSG_PUSH) != (len+1))
            {
                printf("write socket error(%d)\n",errno);
                close_socket(sockfd);
                return;
            }
        }

```

```

printf("Press any key to read...(q to quit)\n");
c=getch();
if ('q'==tolower(c))
{
    close_socket(sockfd);
    return;
}
len = 128;
argp = & retcode;
if(ioctl(sockfd,FIONREAD,argp))
{
    printf(" ioctl failed(%d)\n",errno);
    close_socket(sockfd);
    return;
}
printf("ioctl says %ld bytes waiting\n",*argp);
if((bytesrec = recv(sockfd, buf, len, 0 )) < 0)
{
    printf("read socket error %d\n",errno);
    close_socket(sockfd);
    return;
}
printf("recv'd: (%s)\n",buf);
}
}

```

Server.c

/* FUNCTION to read a stream socket one line at a time, and write each line
* back to the sender.

```

*/
#define MAXLINE 512
str_echo(sockfd)
int sockfd;
{
    int n;
    char line[MAXLINE];
    for ( ; ; ) {
        n = readline(sockfd, line, MAXLINE);
        if (n == 0)
            return;
        else if (n < 0)
            printf("str_echo: readline error\n");
        if (writen(sockfd, line, n) != n)
            printf("str_echo: writen error\n");
    }
}

```

/* FUNCTION: Read a line from a file descriptor. Read the line
* one byte at a time,
* looking for the newline. We store the newline in the buffer,
* then follow it with a null.
* We return the number of characters up to, but not including,
* the null..

```

*/
int
readline(fd, ptr, maxlen)

```

```

register          int      fd;
/*'register' means value is stored in CPU register*/
register          char     *ptr;
register          int      maxlen;
{
    int          n, rc;
    char         c;
    printf("Enter Readline\n");
    for (n = 1; n < maxlen; n++) {
        if ( (rc = read(fd, &c, 1)) == 1) {
            printf("rec:%d: %c\n", c, c);

/* print decimal value,
 * then the character */

            *ptr++ = c;
            if (c == '\n')
                break;

/* leave loop now; end-of-line */
        } else if (rc == 0) {
            if (n == 1)
                { printf("EOF, no data read\n");
return(0);          /* EOF, no data read */
}

                else
                break; /* EOF, some data was read */
            } else
                return(-1);          /* error */
        }
        *ptr = 0;
        printf("returning from readline(%d)\n",n);
        return(n);
    }
}

/* FUNCTION: Write 'n' bytes to a (network 'file') descriptor.
 * Use in place of the ULTRIX write() call when the
 * descriptor is a stream socket.
 */
int
writen(fd, ptr, nbytes)
register int      fd;
register char     *ptr;
register int      nbytes;
{
    int          nleft, nwritten;
    printf("writen: (%d) [%s]\n",nbytes,ptr);
    nleft = nbytes;
    while (nleft > 0) {
        nwritten = write(fd, ptr, nleft);
        if (nwritten <= 0)
            {
                printf(" Error (%d)\n",nwritten);
                return(nwritten); /* error */
            }
        nleft -= nwritten;
    }
}

```

```

        ptr += nwritten;
    }
    printf("string written returning(%d)\n",nbytes-nleft);
    return(nbytes - nleft);
}

/* Definitions for TCP server programs. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <varargs.h>
#include <syslog.h>
#define SERV_TCP_PORT 6000 /* port number, hardcoded */
#define SERV_HOST_ADDR "16.72.32.196"
/* server address hardcoded*/
char *pname;
/* Example of a server using TCP protocol, with socket programming... */
/* #include </usr/include/arpa/inet.h> */
main(argc,argv)
int argc;
char *argv[];
{
    int sockfd,newsockfd,clilen,childpid;
    struct sockaddr_in cli_addr,serv_addr;
    pname = argv[0];
    printf("Enter Server Program\n");
    /* Open a TCP socket (an Internet stream socket)*/
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("server: cannot open stream socket\n");
        return(-1);
    }
    /* Bind our local address so that the client can send to us...*/
    bzero((char *) &serv_addr,sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("server: cannot bind local address\n");
    }
    return(-1);
}
listen(sockfd, 5);
for ( ; ; ) {
    /* Wait for a connection from a client process.
     * This is an example of a concurrent server, in which the
     * parent forks a child to handle the socket I/O after
     * connection is established ... */
    clilen = sizeof(cli_addr);
    printf("starting the accept ...\n");
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

```

```

printf("after accept ...\n");
if (newsockfd < 0)
    printf("server: accept error\n");
if ((childpid = fork()) < 0)
    printf("server: fork error\n");
else if (childpid == 0) {          /* child process */
    close(sockfd);
    str_echo(newsockfd);          /*
        * read line from socket,
        * then write it back
        * to client          */
    exit(0);
}
close(newsockfd);                /* parent process */
}

```

Т е м а 1.12. Создание системы «клиент-сервер» в ОС Linux при использовании СУБД MySQL

В процессе освоения темы необходимо научиться использовать MYSQL C API для написания клиентских приложений, а также закрепить навыки работы с сокетами BSD. В процессе выполнения предлагается:

- создать простую систему «клиент-сервер», используя сокеты TCP/IP;
- передачу данных от клиента к серверу осуществить с помощью SQL-запросов.

Основные сведения

В качестве примера рассмотрим программу, состоящую из двух частей: клиента и сервера. Клиент соединяется с сервером по TCP/IP и отправляет серверу данные, вводимые пользователем с клавиатуры (каковыми являются SQL-запросы). Сервер в свою очередь является клиентом СУБД, в качестве которой выбрана СУБД MySQL. Данная СУБД в основном используется в UNIX-системах, хотя существуют версии и под другие известные ОС. MySQL обеспечивает быструю обработку данных при достаточно широком наборе функциональных возможностей. На рис. 18 изображена схема информационных потоков в системе. Сервер устанавливает соединение с сервером БД, выполняет принятые от клиента запросы и возвращает результаты клиенту.

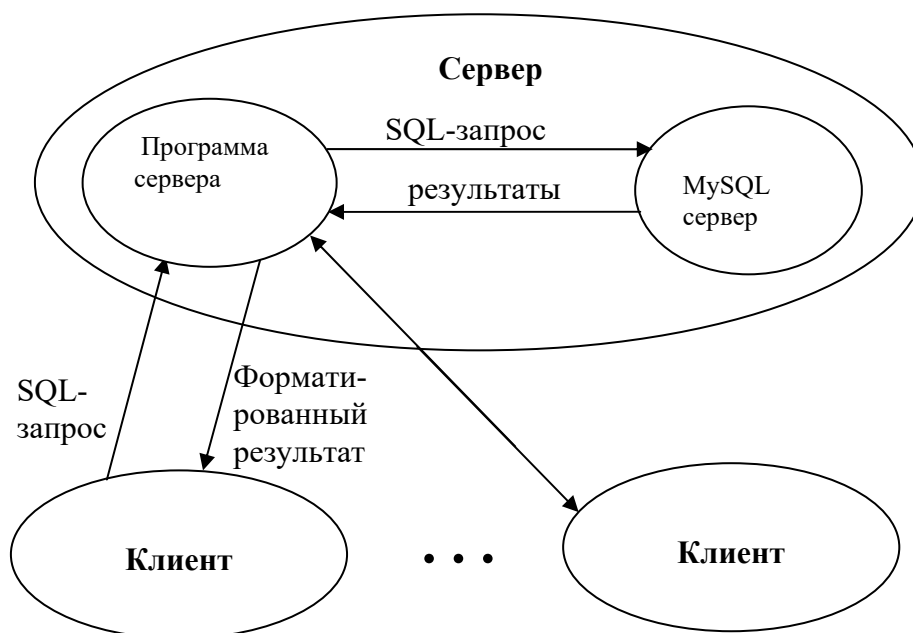


Рис. 18. Схема информационных потоков в системе

Для уяснения возможностей MySQL сервера приведем краткий обзор функций MySQL API для C/C++ (табл. 1).

Таблица 1

Обзор основных функций MySQL API для C/C++

Функция	Возможности функции
Mysql_affected_rows()	Возвращает количество строк, полученных последним UPDATE, DELETE или INSERT запросом
mysql_close()	Закрывает соединение с сервером
mysql_connect()	Устанавливает соединение с MySQL сервером (также используется mysql_real_connect())
mysql_change_user()	Смена пользователя для соединения
mysql_character_set_name()	Возвращает текущую кодовую таблицу
mysql_create_db()	Создает БД
mysql_drop_db()	Удаляет БД
mysql_errno()	Возвращает номер ошибки
mysql_error()	Возвращает текст ошибки
mysql_real_escape_string()	Преобразует строки в строку с Esc последовательностью, учитывает кодовую страницу
mysql_escape_string()	Преобразует строки в строку с Esc последовательностью
mysql_fetch_field()	Возвращает тип поля
mysql_fetch_field_direct()	Возвращает тип поля по его номеру
mysql_fetch_fields()	Возвращает структуру, содержащую все поля
mysql_fetch_lengths()	Возвращает длину всех столбцов в строке
mysql_fetch_row()	Возвращает следующую строку из результата
mysql_field_seek()	Установка курсора
mysql_field_count()	Возвращает количество столбцов

Функция	Возможности функции
mysql_field_tell()	Возвращает позицию курсора для последнего
mysql_free_result()	Освобождает память от результата
mysql_get_client_info()	Возвращает клиентскую информацию
mysql_get_host_info()	Возвращает строку, описывающую соединение
mysql_get_proto_info()	Возвращает версию протокола для соединения
mysql_get_server_info()	Возвращает версию сервера
mysql_init()	Получает или инициализирует MySQL структуру
mysql_kill()	Уничтожает заданную нить
mysql_list_dbs()	Возвращает имена баз данных, соответствующих выражению
mysql_list_fields()	Возвращает имена полей, соответствующих выражению
mysql_list_processes()	Возвращает имена всех процессов на сервере (MySQL)
mysql_list_tables()	Возвращает имена таблиц, соответствующих выражению
mysql_num_fields()	Возвращает количество столбцов в результате запроса
mysql_num_rows()	Возвращает количество строк в результате
mysql_options()	Устанавливает опции соединения для mysql_connect()
mysql_ping()	Проверяет соединение, соединяется заново при необходимости
mysql_query()	Выполняет SQL запрос, заданный как null-terminated строка
mysql_real_connect()	Соединяется с MySQL сервером
mysql_real_query()	Выполняет SQL запрос
mysql_reload()	Перезагрузка таблиц прав
mysql_row_seek()	Ищет строку, используя значения mysql_row_tell()
mysql_row_tell()	Возвращает позицию курсора для строки
mysql_select_db()	Выбор базы данных
mysql_store_result()	Получает полный результат запроса
mysql_use_result()	Инициализирует построчное извлечение результата

Настройка и запуск программы сервера

Для запуска программы может потребоваться ее перекомпиляция, так как в разных системах часто используются разные библиотеки. Для этого необходимо набрать файл инициализации serv.ini, имеющий следующий вид:

```
#serv.ini -log file
#ip address is host's
serv_host_addr = 127.0.0.1
```

```
#TCP/IP port
serv_port = 6000
#name base data
szDB = clsv
#name user's
nmDB = user
# password user's
psDB = password
#the end
```

Следует обратить внимание на конфигурацию базы данных. Необходимо указать имя БД, имя и пароль для соединения с MySQL сервером. Данный пользователь должен обладать только ограниченным набором прав на данную БД из соображения безопасности.

Настройка и запуск программы клиента

Для запуска программы в файле client.cc необходимо прописать IP-адрес и порт программы – сервера. Для запуска программы может потребоваться ее перекомпиляция, так как в разных системах часто используются разные библиотеки. Для этого необходимо набрать

```
c++ client.cc mcl.cc -L'/usr/lib/mysql' -lmysqlclient -o client
```

После перекомпиляции программа готова к эксплуатации. Достаточно запустить ее, и она попытается установить соединения с сервером. После установки соединения пользователь может писать SQL-запросы, которые будут передаваться серверной части системы. Получив запрос, сервер обработает его, а результат будет передан клиентской части системы и выведен на экран. Выход осуществляется путем ввода символа 'q'.

Пример:

```
/*****|server.cc|*****/
* This program is TCP/IP server. It recieves *
* SQL query and executes it on mySQL server. *
* It requires UNIX (Tested on Linux SuSE, *
* ASP Linux) *
\*****/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/socket.h>
```

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <varargs.h>
#include <unistd.h>
#include <syslog.h>
#include <errno.h>
#include <signal.h>
#include <mysql/mysql.h>
#include "mcl.h"

#define MAXLINE 2048
//-----
// a few mysql variables
MYSQL *myData;
MYSQL_RES *res;
MYSQL_FIELD *fd;
MYSQL_ROW row;
char szSQL[MAXLINE];
//this is TCP/IP configuration
int SERV_TCP_PORT=6000;
char SERV_HOST_ADDR[MAXSTRLEN];

int readline(register int fd,register char *ptr,register int maxlen);
int written(register int fd,register char *ptr,register int nbytes);
mcl *ut;

int childpid,sockfd,newsockfd;
struct sockaddr_in cli_addr, serv_addr;
socklen_t clilen;
bool ready=true;
//-----
void connect_to_mysql()
{
char szDB[50];
strcpy( szDB, "clsv" );
if ((myData = mysql_init((MYSQL*) 0)) && mysql_real_connect( myData, NULL,"",
"", NULL, MYSQL_PORT, NULL, 0 ) )
{
if ( mysql_select_db( myData, szDB ) < 0 ) {
ut->PutToLog("serv.log","Can't perform connection(error on
db selecting).");
mysql_close( myData ) ;
exit(-1);
}
}
else {
ut->PutToLog("serv.log","Can't perform connection on a
specified port.");
mysql_close( myData ) ;
exit(-1);
}
}

```

```

ut->PutToLog("serv.log","mySQL connection set.");
}

//-----
int SqlQuery()
{
    int i=0;
    if (!mysql_query(myData, szSQL)) {
        if (strstr(szSQL,"select")>0) {
            res = mysql_store_result(myData);
            i = (int) mysql_num_rows(res);
        }
        return(i);
    }
    else {
        ut->PutToLog("serv.log","SQL Query couldn't be
executed.");
        return(-1);
    }
}

//-----
void SqlFree()
{
    strcpy(szSQL,"");
    mysql_free_result(res);
}

//-----
void Signal_handler1(int signo)
{
    ready=false;
    int *stat;
    if ((childpid>0)&&(signo==SIGCHLD)) {
        wait(stat);
        ut -> PutToLog("serv.log","Session Complete.");
    }
    ready=true;
}

//-----
int readline(register int fd,register char *ptr,register int maxlen)
{
    int n,rc;
    char c;
    for (n = 1; n < maxlen; n++) {
        if ((rc = read(fd, &c,1))==1) {
            *ptr++=c;
            if (c=='\0') //Stopping Character
                break;
        }
        else
            if (rc == 0) {
                if (n==1) { //no data read
                    return(0);
                }
            }
    }
}

```

```

        else
            break;
    } else
        return(-1);
    }
    *ptr = 0;
    return(n);
}

//-----
int writen(register int fd,register char *ptr,register int nbytes)
{
    int nleft,nwritten;
    nleft=nbytes;
    while (nleft > 0) {
        nwritten = write(fd,ptr,nleft);
        if (nwritten<=0) {
            ut -> PutToLog("serv.log","writen error");
            return(nwritten);
        }
        nleft-=nwritten;
        ptr+=nwritten;
    }
    return(nbytes-nleft);
}

//-----
char *pname;
//-----
void main(int argc,char *argv[])
{
    // initialization begins...
    char tmp2[MAXSTRLEN];
    pname = argv[0];
    int accepterrors=0;
    int n=0,i=0,j=0,m;
    char line[MAXLINE];
    int state;
    struct sigaction act, oact;
    act.sa_handler = Signal_handler1;
    act.sa_flags = SA_NOMASK;
    sigemptyset(&act.sa_mask);
    sigaction(SIGCHLD, &act, &oact);
    ut = new mcl;
    ut -> LogActive=true;
    ut -> PutToLog("serv.log","Started");
    ut ->
    iniLoad("serv.ini","serv_host_addr",SERV_HOST_ADDR,"serv_port",tmp2,NULL);
    SERV_TCP_PORT = atoi(tmp2);
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))<0) //AF_INET ?
        ut -> PutToLog("serv.log","server: cannot open stream socket");
}

```

```

bzero((char *) &serv_addr,sizeof(serv_addr));
serv_addr.sin_family = PF_INET; //AF_INET ?
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_TCP_PORT);
if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr))<0)
    ut -> PutToLog("serv.log","server:cannot bind local address");
// initialization ends.
listen(sockfd, 5);
//listening for clients(max 5 clients simultaniaslly)
for (;;) {
    clilen = sizeof(cli_addr);
    ut -> PutToLog("serv.log","starting the accept");
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);
    if ((newsockfd < 0)&&ready) { //Retry
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);
        if (newsockfd < 0) {
            ut -> PutToLog("serv.log","accept error");
            accepterrors++;
            if (accepterrors<10)
                continue;
            else {
                ut -> PutToLog("serv.log","To many accept
errors");
                exit(-1);
            }
        }
    }
    accepterrors=0;
    if ((childpid = fork())<0) {
        ut -> PutToLog("serv.log","fork error");
        exit(-1);
    }
    if (childpid == 0) {
        close(sockfd);
        connect_to_mysql();
        while (1) {
            n=readline(newsockfd, line, MAXLINE);
            if (n==0) {
                ut->PutToLog("serv.log","no query
recieved. Client disconnected");
                break;
            }
            ut->PutToLog("serv.log","query recieved.");
            //Now we shell perform SQL query
            ut -> PutToLog("queries.log",line);
            strcpy(szSQL, line);
            m=SqlQuery();
            if (m==0) {

```

```

affected",22)!=22)
{
    if (written(newsockfd,"OK. (0) rows
        ut -> PutToLog("serv.log","written error");
        }
        }
        if (m==-1) {
            if (written(newsockfd,"Invalid
                {
                    ut -> PutToLog("serv.log","written error");
                }
            }
            if (m>0) {
                strcpy(line,"");
                for (i=0;i<m;i++) {
                    row = mysql_fetch_row(res);
                    strcat(line,"\n");
                    int cols = (int) mysql_num_fields(res);
                    for (j=0;j<cols;j++) {
                        strcat(line,row[j]);
                        strcat(line,"|");
                    }
                }
                //The only thing to do is answer to client.

                if(written(newsockfd,line,strlen(line)+1)!=strlen(line)+1)
                {
                    ut -> PutToLog("serv.log","written error");
                }
                SqlFree();
            }
            strcpy(line,"");
            ut -> PutToLog("serv.log","request
served.");
        }
        exit(0);
    }
    close(newsockfd);
}
}

```

```

//----- client.c -----
#define MAXLINE 512
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

```

```

#include <varargs.h>
#include <syslog.h>
#include <sys/ioctl.h>

#define SERV_TCP_PORT 6000
#define SERV_HOST_ADDR "127.0.0.1"

char *pname;

void main()
{
    char buf[128],c,*tmp2;
    int sockfd;
    long retcode;
    long *argp;
    struct sockaddr_in serv_addr;
    struct sockaddr_in *from;
    //          getline(SERV_HOST_ADDR,"IP>");
    //          getline(tmp2,"PORT>");
    //          SERV_TCP_PORT = atoi(tmp2);
    serv_addr.sin_family =PF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))<0) {
        printf("client:cannot open TCP stream socket");
        return;
    }
    if (connect(sockfd,&serv_addr,sizeof(serv_addr))<0) {
        printf("client: cannot connect to TCP server");
        return;
    }
    for (;;) {
        int len;
        int bytesrec;
        printf("Sending a new msg - type 'q' to quit:\n");
        getline(buf,">");
        if (buf[0] == 'q') return;
        len = strlen(buf);
        printf("send [%d](%s)\n",len,buf);
        if (send(sockfd, buf, len+1, MSG_DONTWAIT)!= (len+1)) {
            printf("write socket error(%d)\n",errno);
            close(sockfd);
            return;
        }
        len = 128;
        argp = &retcode;
        if ((bytesrec=recv(sockfd,buf,len,0))<0) {
            printf("read socket error %d\n",errno);
            close(sockfd);
            return;
        }
    }
}

```

```

        printf("recived: (%s)\n",buf);
    }
}
getline(char *line,char *prompt)
{
    char ch;
    printf("%s",prompt);
    while ((ch = getchar())!='\n' && ch!=EOF)
        *line++=ch;
    *(line--]='\0';
    if (ch == EOF)
        return(EOF);
}

```

```
//-----
```

Mcl.cc

```
/*-----
```

CGI procedures

```
*/
```

```
//-----
```

```
#ifndef mcl_c
```

```
#define mcl_c
```

```
//-----
```

```
#include "mcl.h"
```

```
#include <iostream.h>
```

```
#include <string.h>
```

```
#include <stdio.h>
```

```
#include <stdarg.h>
```

```
#include <time.h>
```

```
#include <stdlib.h>
```

```
#include <malloc.h>
```

```
//-----
```

```
mcl::mcl(void)
```

```
{
```

```
    LogActive = false;
```

```
}
```

```
//-----
```

```
bool mcl::iniLoad(char *filename, ...) //needs "const",result[maxlen],...,NULL
```

```
{
```

```
FILE *inf;
```

```
char
```

```
st[MAXSTRLEN],nosp[MAXSTRLEN],param[MAXSTRLEN],value[MAXSTRLEN];
```

```
char *str,*strout;
```

```
int count=0;
```

```
//lets check given arguments and prepare outstr...
```

```
    str = filename;
```

```
    va_list par;
```

```
    va_start(par,filename);
```

```
    while ((str = va_arg(par,char *)) != 0) {
```

```
        count++;
```

```
        if (count%2 == 0)
```

```

        strcpy(str,"");
    }
    if (count%2 != 0) {
        return false; //bad args
    }
    count=count/2;
    va_start(par,filename);
    if ((inf=fopen(filename,"rt"))==NULL) {
        return false;
    }
    unsigned int i,m,j;
    while (!feof(inf))
    {
        fgets(st,MAXSTRLEN,inf);
        for (i=0; i<strlen(st);i++)
        {
            if (st[i]!=' ') { // Deleting spaces at the
beginning of line

                for (j=i,m=0; j<=strlen(st);j++,m++)
                    nosp[m]=st[j];
                nosp[m+1]='\0';
                break;
            }
        }
        if ((nosp[0]!='\n')&&(nosp[1]!='\n')) // if it is not a
comment

        {
            char *m;
            strcpy(param,nosp);
            if ((m=strchr(param,'='))==NULL) continue;
            else *m='\0';
            strcpy(value,strchr(nosp,'=')+1);
            char ch=10,*pos;
            pos=strchr(value,ch);
            if (pos!=NULL) *pos='\0';
            ch=13;
            pos=strchr(value,ch);
            if (pos!=NULL) *pos='\0';
            va_start(par,filename);
            for (int c=0;c<count;c++) {
                str = va_arg(par,char *);
                strout = va_arg(par,char *);
                if (strcmp(param,str)==0) {
                    strcpy(strout,value);
                }
            }
        }
    }
    if (feof(inf)) break;
}
va_end(par);
fclose(inf);

```

```

        return true;
    }
    //-----
    bool mcl::PutToLog(char *filename, char *text)
    {
        time_t tim;
        FILE *logf;
        if (!LogActive) return true;
            time(&tim);
            char *tm=ctime(&tim);
            tm[strlen(tm)-1]='\0';
            if ((logf=fopen(filename,"at"))==NULL) {
                return false;
            }
        fprintf(logf,"[%s] : %s\n", tm, text);
        fclose(logf);
        return true;
    }
    //-----
#endif

/*-----
                                Mcl.h
-----*/
#ifndef mcl_h
#define mcl_h
//-----
#define MAXSTRLEN 512
//-----

class mcl
{
private:
public:
    bool LogActive;        //if log is needed set to true
    mcl(void);
    bool iniLoad(char *filename, ...); //needs filename,"const",result[maxlen],...,NULL
    bool PutToLog(char *filename, char *text);
};
//-----
#endif

Содержимое файла для компиляции:
c++ server.cc mcl.cc -L'/usr/lib/mysql' -lmysqlclient -o serv2 strip serv2

```

В результате изучения тем, представленных в **части 1**, должен быть получен практический опыт проектирования «клиент – серверных» приложений на основе протоколов TCP/IP и UDP, также получены навыки работы в среде Visual C и в использовании стандартных средств MFC.

Часть 2

ИНТЕГРАЦИЯ СЕТЕВЫХ, ИНФОРМАЦИОННЫХ И КЛАСТЕРНЫХ ТЕХНОЛОГИЙ В ТСП/IP СЕТЯХ

Общие сведения

При изучении первых четырех тем данной части используется операционная система (ОС) Linux, а для остальных – ОС Windows.

При практическом освоении первых двух тем в ОС Linux необходимо создать новую БД. Существующая БД с именем `workdb` предназначена для выполнения лабораторных работ с использованием сокетов и библиотеки `MPI`. База данных `workdb` доступна любому пользователю и не требует ввода пароля.

В домашнем каталоге пользователя располагается папка `Info` содержащая справочную информацию. В папке `Info/MySQL` находится информация по языку `SQL` и работе с СУБД `MySQL`. В папке `Info/MPI` находится информация по функциям библиотеки `MPI` и пакету `MPICH`. Справочная информация представляет собой файлы формата `html` и `pdf`. Для их просмотра можно использовать пакеты и программы, предоставляемые из графического интерфейса пользователя. Для запуска этого интерфейса используется команда `startx`. Работа с СУБД `MySQL`, редактирование и компиляция программ может осуществляться из графической оболочки. Для этого необходимо запустить одну или несколько консолей.

Для чтения или записи данных на флеш-накопитель сначала командой `mount/dev/sdb0` выполняется монтирование файловой системы, а затем собственно операции чтения или записи. Для размонтирования файловой системы используется команда `umount/dev/sdb0`. Вместо `sdb0` может использоваться другое имя в зависимости от наличия других накопителей в системе. Список устройств можно посмотреть командой `mount`.

Запуск файлового менеджера `Midnight Commander` выполняется командой `mc`. Для правильного выхода из ОС Linux необходимо использовать команды `reboot` (перезагрузка) или `halt` (останов).

При выполнении работ в ОС Windows для создания БД используется `MS Access`. Для разработки программного обеспечения используется `Visual Studio 2010`.

Т е м а 2.1. Работа с базами данных в операционной системе Linux

Цель работы над темой – изучение СУБД MySQL для ОС Linux и назначения входящих в нее утилит. Изучение методов работы с БД в СУБД MySQL.

СУБД MySQL включает в себя большое число разнообразных программ и скриптов, предназначенных для эффективной работы и администрирования БД. К числу основных относятся:

- mysqld;
- mysql;
- mysqlaccess;
- mysqladmin;
- mysqldump;
- mysqlshow.

Сервером для клиентских приложений служит программа mysqld, которая запускается в системе как демон. Она принимает подключение, выполняет запросы и возвращает результаты. Программа многопоточная, поэтому допускается одновременная работа нескольких клиентских приложений.

Программа mysql предназначена для соединения с сервером MySQL и интерактивной обработки команд пользователя. Общий формат команды запуска mysql следующий:

```
shell> mysql [-h host_name] [-u user_name] [-pyour_password]
[name_database]
```

Альтернативным вариантом использования ключей -h, -u и -p является следующая запись:

```
shell> mysql [--host=host_name] [--user=user_name] [--
password=your_password] [name_database]
```

По умолчанию соединение осуществляется с

host_name=localhost, user_name

таким же, как имя пользователя и password="". Примеры запуска клиентской части MySQL:

```
shell> mysql -h localhost -u spec1 -pmy_pass spec1db
shell> mysql --user=spec1 --password=my_pass
shell> mysql
```

Хранить пароль в командной строке небезопасно, поэтому рекомендуется для запуска `mysql` использовать команду следующего вида:

```
shell> mysql -p name_database,
```

где `name_database` – имя базы данных, с которой предполагается работать. Пароль будет запрашиваться каждый раз при выполнении этой команды.

Скрипт `mysqlaccess` используется, чтобы установить привилегии пользователю для доступа к БД. После внесения изменений в БД, чтобы изменения вступили в силу, необходимо выполнить команду `mysqladmin reload`.

Утилита `mysqladmin` используется для администрирования СУБД MySQL. Она позволяет создавать и удалять БД, получить информацию о версии СУБД и статусе сервера.

Утилита `mysqldump` используется для создания дампа БД, резервного копирования или переноса данных в другой SQL сервер. Дамп содержит операторы SQL для создания и заполнения таблиц.

Утилита `mysqlshow` используется для получения информации о существующих БД, таблицах, колонках и индексах. Примеры использования утилиты:

```
shell> mysqlshow
shell> mysqlshow test
```

Создание новой БД может быть выполнено несколькими способами: с помощью утилиты `mysqladmin` или соответствующей командой в программе `mysql`. В первом случае используется команда `create`. Например, следующая команда создаст новую БД с именем `db_jobs`:

```
shell> mysqladmin create db_jobs
```

Для удаления существующей БД предназначена команда `drop`. Следующая команда удалит БД с именем `db_jobs`:

```
shell> mysqladmin drop db_jobs
```

При создании БД из программы `mysql` используется оператор `CREATE DATABASE [IF NOT EXISTS] db_name`, где `db_name` – имя создаваемой БД. Выполнение оператора будет неудачно, если БД с таким именем уже существует и не определено предложение `IF NOT EXISTS`. Пример создания новой БД с именем `db_jobs`:

```
mysql> CREATE DATABASE db_jobs;
```

Удаление существующей БД выполняется оператором DROP DATABASE [IF EXISTS] db_name. Выполнение оператора будет неудачно, если БД с таким именем не существует и не определено предложение IF EXISTS. Пример удаления существующей БД с именем db_jobs:

```
mysql> DROP DATABASE db_jobs;
```

Заполнение БД информацией может осуществляться в интерактивном режиме из командной строки программы mysql или в пакетном режиме с использованием файла, содержащего SQL запросы. Для добавления в БД новых строк из программы mysql используется команда

```
INSERT INTO tbl_name [(col_name,...)] VALUES (expression,...),
```

где tbl_name – имя существующей таблицы, в которую будет добавлена запись, col_name – имя столбца, expression – значение столбца. Пример добавления новой записи в таблицу с именем table1, состоящую из трех столбцов:

```
mysql> INSERT INTO table1 (Color, Form, Mass)
> VALUES ("Red", "Cube", 120);
mysql> INSERT INTO table1 VALUES ("Yellow", "Globe", 340);
```

Если данные должны быть загружены из файла, то используется команда LOAD DATA INFILE 'file_name.txt' INTO TABLE tbl_name, где file_name.txt – имя файла с данными, а tbl_name – имя существующей таблицы, в которую будут заноситься эти данные. Столбцы в файле должны отделяться знаком табуляции (\t), а кортежи – переходом на новую строку (\n). Пример использования этой команды:

```
mysql> USE db_jobs;
mysql> LOAD DATA INFILE 'jobs.txt' INTO TABLE
name_jobs;
```

СУБД MySQL поддерживает множество разнообразных типов атрибутов, которые могут быть сгруппированы в три категории: числовые типы, дата и время, строковые (символьные) типы.

Рассмотрим пример создания БД для поддержки работы склада, состоящей из четырех сущностей: склад (Storage), продукт (Product), потребитель (Customer) и поставщик (Provisor). На логическом уровне БД можно представить ER-диаграммой (рис. 19).

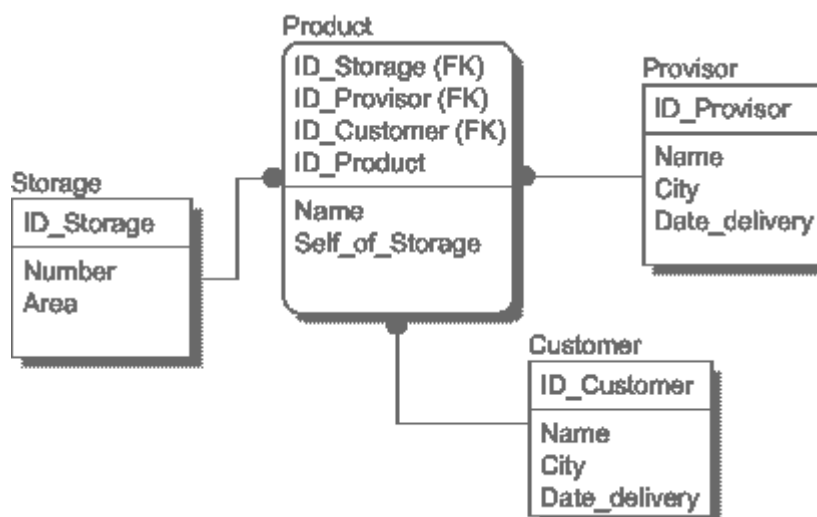


Рис. 19. ER-диаграмма БД

Сначала создадим новую БД с именем `storage_db`, затем – таблицы. Для заполнения таблиц новыми данными будем использовать оператор `INSERT INTO`.

Пример:

```

mysql> CREATE DATABASE storage_db;
mysql> USE storage_db;
mysql> CREATE TABLE Storage (
  > ID_Storage      INTEGER    NOT NULL,
  > Number          INTEGER    NOT NULL,
  > Area            INTEGER    NOT NULL,
  > PRIMARY KEY (ID_Storage));
mysql> CREATE TABLE Customer (
  > ID_Customer     INTEGER    NOT NULL,
  > Name            VARCHAR(20),
  > City            VARCHAR(20),
  > Date_delivery   DATE NOT NULL,
  > PRIMARY KEY (ID_Customer));
mysql> CREATE TABLE Provisor (
  > ID_Provisor     INTEGER    NOT NULL,
  > Name            VARCHAR(20),
  > City            VARCHAR(20),
  > Date_delivery   DATE NOT NULL,
  > PRIMARY KEY (ID_Provisor));
mysql> CREATE TABLE Product (
  > ID_Product      INTEGER    NOT NULL,
  > Provisor        INTEGER    NOT NULL,
  > Customer        INTEGER    NOT NULL,
  > Storage         INTEGER    NOT NULL,
  > Name            VARCHAR(50),
  > Self_of_Storage INTEGER    NOT NULL,
  
```

```

> PRIMARY KEY (ID_Product),
> FOREIGN KEY (Provisor) REFERENCES Provisor,
> FOREIGN KEY (Customer) REFERENCES Customer,
> FOREIGN KEY (Storage) REFERENCES Storage);
mysql> INSERT INTO Storage VALUES (101, 2, 1000);
mysql> INSERT INTO Customer
> VALUES (1001, "ET", "Penza", "2005-12-04");
mysql> INSERT INTO Provisor
> VALUES (2003, "Platan", "Moskou", "2005-10-12");
mysql> INSERT INTO Product
> VALUES (1, 2003, 1001, 101, "ADSP2196", 10);

```

Задания для освоения темы

1. Создать базу данных «Поставки», состоящую из таблиц «Поставщик», «Деталь» и «Поставка». Таблица «Поставка» должна содержать информацию о поставщике и поставляемых им деталях.

2. Создать базу данных «Издательство», состоящую из таблиц «Автор», «Издатель» и «Книга». Таблица «Книга» должна содержать информацию о том, каким издательством выпущена книга и кто ее автор.

3. Создать базу данных «Библиотека», состоящую из таблиц «Книга», «Место», «Читатель». Таблица «Книга» должна содержать информацию о том, кто читает книгу, какая это книга и ее месторасположение в библиотеке.

4. Создать базу данных «Магазин», состоящую из таблиц «Поставщик», «Товар», «Отдел». Таблица «Товар» должна содержать информацию о поставщике, поставляемом им товаре и отделе, в котором продается этот товар.

5. Создать базу данных «Производство», состоящую из таблиц «Продукт», «Потребитель» и «Поставщик». Таблица «Продукт» должна содержать информацию о продукте, его потребителе и поставщике.

6. Создать базу данных «Музыка», состоящую из таблиц «Музыкальное произведение», «Звукозаписывающая компания», «Музыкальный альбом» и «Исполнитель». Таблица «Музыкальное произведение» должна содержать информацию об исполнителе, звукозаписывающей компании и музыкальном альбоме, в который входит музыкальное произведение.

7. Создать базу данных «Университет», состоящую из таблиц «Студент», «Кафедра» и «Куратор». Таблица «Студент» должна содержать информацию о студентах, кафедре на которой они учатся и кто является их куратором.

8. Создать базу данных, состоящую из трех таблиц «Служащий», «Руководитель» и «Отдел». Таблица «Служащий» должна содержать информацию о служащем, отделе, в котором он работает и его руководителе.

Т е м а 2.2. Использование C API для доступа к базе данных

Цель работы над темой – изучение функций C API для доступа к БД в архитектуре «клиент-сервер».

C API код распространяется вместе с MySQL. Он включен в библиотеку `mysqlclient` и позволяет получить доступ к БД из программы, написанной на языке программирования C. Функции, входящие в состав библиотеки, обеспечивают широкие возможности для работы с БД, как с локальной, так и с удаленной вычислительной системой. Для использования этих функций необходимо подключить библиотеку **`mysql.h`** в начале программы.

Данные от сервера MySQL поступают в программу через коммуникационный буфер, размер которого первоначально равен 16 Кбайт и увеличивается по мере необходимости до своего максимального размера (16 Мбайт).

Чтобы соединиться с сервером, вызывается функция `mysql_real_connect()` (рис. 20). Одним из множества входных параметров функции является указатель на экземпляр структуры `MYSQL`, который должен быть инициализирован до попытки соединения. Инициализация выполняется вызовом функции `mysql_init()`. После успешного соединения клиент может посылать SQL-запросы серверу, используя для этого функции `mysql_query()` и `mysql_real_query()`. Чаще всего используется первая функция. Она вычисляет длину SQL-запроса, а затем вызывает функцию `mysql_real_query()`. Функция `mysql_real_query()` позволяет указать длину SQL-оператора и вызывается в том случае, когда запрос может содержать двоичные данные.

Обе функции позволяют выполнить один SQL-оператор и не требуют добавления в конце строки символа «;» или «\g». Если запросы фиксированы, то `mysql_real_query()` выполняется быстрее, так как не требует обращения к `strlen()` для вычисления длины строки. Функции возвращают ноль при успешном выполнении запроса или код ошибки.

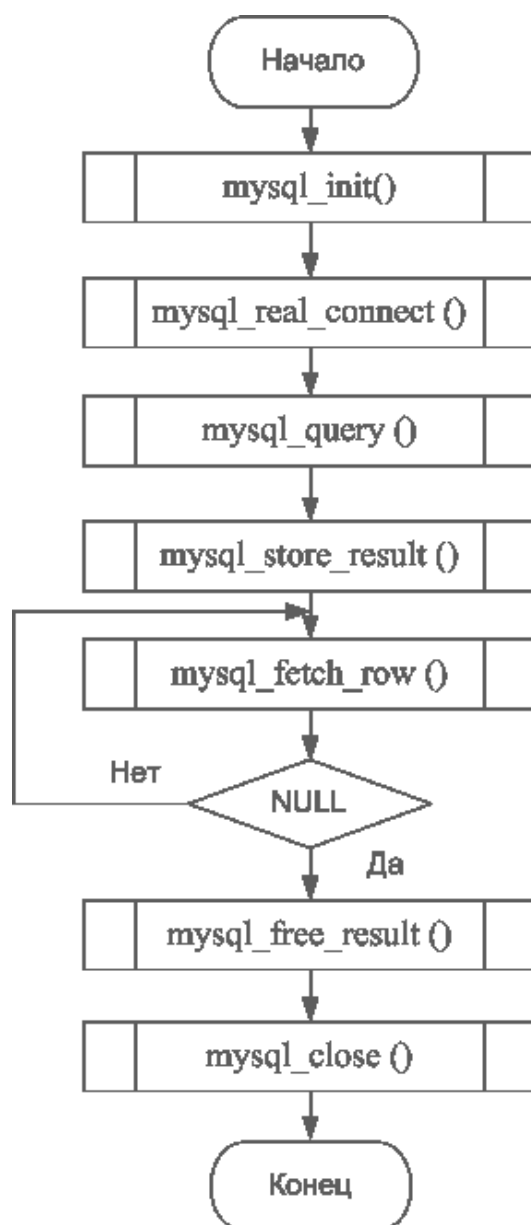


Рис. 20. Блок-схема алгоритма соединения с сервером MySQL

При выполнении SELECT подобных запросов, когда предполагается наличие результирующего множества строк, для организации выборки данных используется несколько функций. Первая из них `mysql_store_result()` извлекает весь результирующий набор данных в буфер клиента. Эта функция должна вызываться после каждого успешно выполненного запроса, который должен вернуть результаты, например, SELECT или SHOW. Единственным аргументом функции является указатель на экземпляр структуры MYSQL, а возвращаемое значение – экземпляр структуры MYSQL_RES. После получения от сервера результата для выборки строки результирующего множества используется функция

`mysql_fetch_row()`, аргументом которой является инициализированная вызовом функции `mysql_store_result()` структура `MYSQL_RES`. После вызова `mysql_fetch_row()` данные одной строки размещаются в структуре `MYSQL_ROW`, которая является массивом указателей на значения столбцов этой строки. Количество столбцов в строке возвращает функция `mysql_num_fields()`, аргументом которой является указатель на экземпляр структуры `MYSQL_RES`.

Выборка строк может продолжаться до тех пор, пока `mysql_fetch_row()` не возвратит `NULL`. Можно определить количество строк в результирующем множестве с помощью функции `mysql_num_rows()`, аргументом которой является указатель на экземпляр структуры `MYSQL_RES`.

Обязательным этапом является освобождение памяти, выделенной для результата функцией `mysql_store_result()`. Это осуществляется вызовом функции `mysql_free_result()`, аргументом которой является указатель на экземпляр структуры `MYSQL_RES`. После завершения работы клиента освобождение выделенной памяти и закрытие соединения с сервером осуществляется вызовом функции `mysql_close()`, аргументом которой является инициализированный `mysql_init()` экземпляр структуры `MYSQL`.

При выполнении не-`SELECT` подобных запросов, например `INSERT`, `UPDATE` или `DELETE`, для определения числа измененных строк используется функция `mysql_affected_rows()`, аргументом которой является указатель на экземпляр структуры `MYSQL`.

Рассмотрим пример программы для соединения с сервером MySQL и выборки данных из БД для поддержки работы склада. Блок-схема алгоритма работы программы представлена на рис. 20.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#include "mysql.h"

int main()
{
    MYSQL mysql;
    MYSQL_RES *res;
    MYSQL_ROW row;

    unsigned int num_fields;
    unsigned int i;
```

```

const char host[] = "host";           // Имя хоста
const char user[] = "user";           // Имя пользователя
const char passwd[] = "passwd";       // Пароль
const char dbase[] = "storage_db";    // Имя БД
const char query[] = "SELECT * FROM Product"; // Запрос

// Инициализация структуры MYSQL
if(NULL == mysql_init(&mysql))
{
    printf("Error init.\n");
    return 0;
}

// Соединение с сервером
if(NULL == mysql_real_connect(&mysql, host, user, passwd,
                              dbase, 0, NULL, 0))
{
    printf("Error connect.\n");
    return 0;
}

// Выполнение запроса
if(0 != mysql_query(&mysql, query))
{
    // Error
    printf("Error query.\n");
}
else
{
    // Сохранение результата
    res = mysql_store_result(&mysql);
    if(NULL != res)
    {
        // Вывод данных на экран
        printf("Results query:\n");
        num_fields = mysql_num_fields(res);
        while((row = mysql_fetch_row(res)))
        {
            for(i = 0; i < num_fields; i++)
                printf("%s\t", row[i]);
            printf("\n");
        }
    }
}

// Освобождение памяти
mysql_free_result(res);
mysql_close(&mysql);
return 0;
}

```

При сборке программы необходимо использовать `mysqlclient` библиотеки. Для этого компоновка выполняется со следующими параметрами (`in.c` – входной файл; `out` – выходной файл):

```
g++ -I/usr/include/mysql -L/usr/lib/mysql in.c -o out -lmysqlclient.
```

при условии, что включаемые файлы размещены в `/usr/include/mysql`, а библиотеки в `/usr/lib/mysql`. Если это не так, то необходимо указать другой путь.

Задания для освоения темы

Установить соединение с базой данных, созданной в результате выполнения предыдущей лабораторной работы. Обеспечить возможность ввода SQL запроса с клавиатуры и структурированного вывода результата на экран монитора.

1. Вывести на экран информацию о поставщиках и деталях, которые они поставляют в формате «Поставщик – Деталь».

2. Вывести на экран информацию об авторах и издательствах, в которых они печатаются в формате «Автор – Издательство».

3. Вывести на экран информацию о читателях и книгах, которые они читают в формате «Читатель – Книга».

4. Вывести на экран информацию о товарах и отделах, в которых он продается в формате «Товар – Отдел».

5. Вывести на экран информацию о потребителях и поставщиках в формате «Потребитель – Поставщик».

6. Вывести на экран информацию об исполнителях и звукозаписывающих компаниях в формате «Исполнитель – Компания».

7. Вывести на экран информацию о студентах и кафедрах в формате «Студент – Кафедра».

8. Вывести на экран информацию о служащих и отделах в формате «Служащий – Отдел».

Т е м а 2.3. Технология сокетов в операционной системе Linux и распределенные базы данных

Цель работы над темой – изучение трехуровневой архитектуры «клиент-сервер» для реализации доступа к БД; изучение методов работы с распределенными базами данных.

Распределенная БД представляет собой набор логически связанных между собой разделяемых данных и их описаний, которые

физически распределены в некоторой компьютерной сети. Такая база данных управляется распределенной СУБД и образована путем «горизонтального» и (или) «вертикального» деления исходной БД с целью использования преимуществ данной технологии. Одной из разновидностей распределенных СУБД являются мультибазовые системы, в которых управление локальными данными осуществляется автономно, а необходимая функциональность реализуется дополнительным уровнем ПО. В качестве такого дополнительного уровня можно использовать компоненты трехуровневой архитектуры «клиент-сервер».

Трехуровневая архитектура «клиент-сервер» предполагает, что система с распределенной БД состоит из клиента, сервера приложения и сервера БД (рис. 21). Клиентское программное обеспечение (ПО) функционирует на стороне пользователя РБД. Серверная часть системы состоит из сервера приложений и сервера БД.

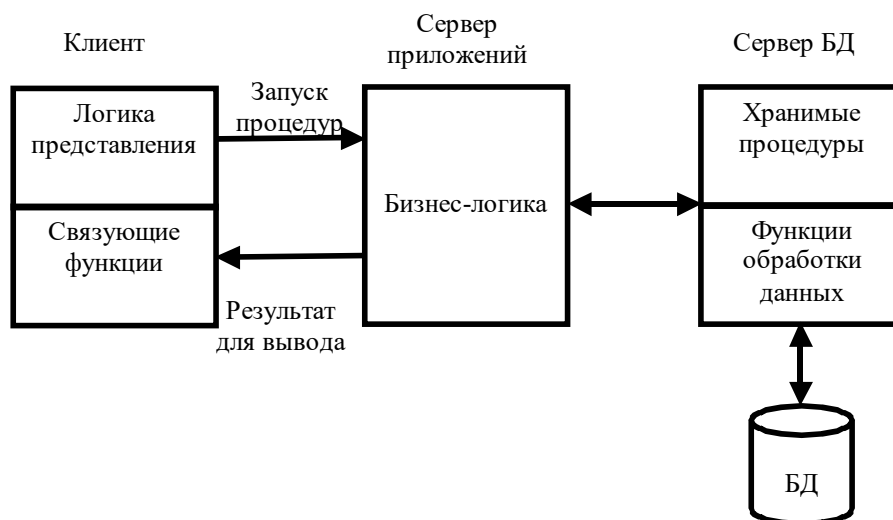


Рис. 21. Трехуровневая архитектура «клиент-сервер»

Сервер приложений реализует коммуникации с клиентским ПО и СУБД. Клиентское ПО реализует логику представления, связанную с отображением данных на экране, работой с формами, клавиатурой и манипуляторами, а также набор связующих функций необходимых для передачи и приема информации от сервера приложений. Сервер приложений реализует бизнес-логику, которая представляет собой алгоритмы решения конкретных задач обработки. Хранимые процедуры и функции обработки данных реализует СУБД.

Взаимодействие между клиентом и сервером приложений можно реализовать с помощью технологии сокетов. В этом случае

на каждой рабочей станции распределенной системы запускается серверное ПО, которое взаимодействует с СУБД с целью передачи SQL-запросов и выборки результата. Сервер принимает соединение от клиента на серверный сокет и ожидает поступления SQL-запросов. Клиент после соединения со всеми известными ему серверами с помощью вызова функции `send()` последовательно передает каждому из них SQL-запросы. После передачи с помощью вызова функции `recv()` клиент последовательно принимает результаты с последующим их выводом на экран или в файл. Вариант развертывания системы показан на рис. 22.

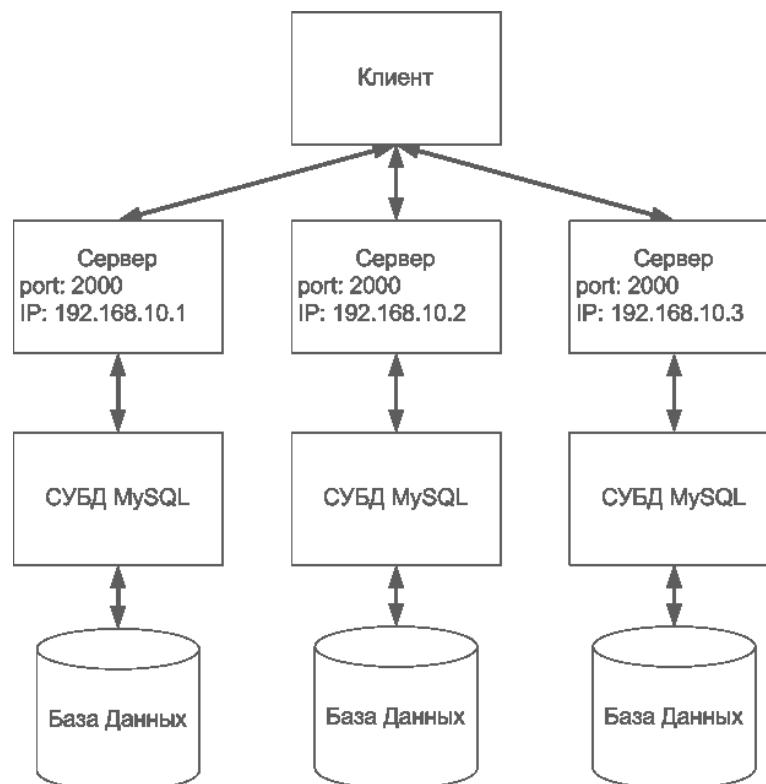


Рис. 22. Вариант развертывания мультибазовой системы «клиент-сервер»

Пример:

```

/***** Клиентское ПО *****/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

```

```

#define SIZE_BUF 1024          // Буфер для данных
#define COUNT_SERV 3          // Кол-во серверов
int main()
{
    char buf[SIZE_BUF];
    int sock[COUNT_SERV];
    const int port[COUNT_SERV] = { 2000, 2000, 2000};
    const char* host[COUNT_SERV] = {"192.168.10.1", "192.168.10.2",
                                     "192.168.10.3"};

    sockaddr_in addr[COUNT_SERV];
    int i;
    for(i = 0; i < COUNT_SERV; i++)
    {
        addr[i].sin_family = AF_INET;
        addr[i].sin_addr.s_addr = inet_addr(host[i]);
        addr[i].sin_port = htons(port[i]);
        // Создание сокета
        sock[i] = socket(AF_INET, SOCK_STREAM, 0);
        if(0 > sock[i])
        {
            printf("Error. Cannot open socket #%d for client.\n", i);
            return 0;
        }
        // Соединение
        if(0 > connect(sock[i],(sockaddr*)&addr[i], sizeof(addr[i])))
        {
            printf("Error. Cannot connect to server #%d.\n", i);
            return 0;
        }
    }
    printf("Connect is OK.\n");
    printf("Enter SQL-query. (quit - exit)\n");

    while(true)
    {
        printf(">");
        gets(buf);
        // Пересылка данных серверам
        for(i = 0; i < COUNT_SERV; i++)
            send(sock[i], buf, strlen(buf) + 1, 0);
        if(!strcmp(buf, "quit"))
            break;
        // Прием результатов
        for(i = 0; i < COUNT_SERV; i++)
        {
            recv(sock[i], buf, SIZE_BUF, 0);
            printf("#%d\n%s\n", i, buf);
        }
    }
}

```

```

        // Заккрытие сокетов
        for(i = 0; i < COUNT_SERV; i++)
            close(sock[i]);
        return 0;
    }

    /***** Серверное ПО *****/
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>

    #include <sys/socket.h>
    #include <unistd.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>

    #include "mysql.h"

    #define SERV_PORT    2000        // Порт серверного сокета
    #define SIZE_BUF    1024        // Размер буфера

    int main()
    {
        MYSQL mysql;
        MYSQL_RES* res;
        MYSQL_ROW row;

        unsigned int num_fields;
        unsigned int i;
        char buf[SIZE_BUF];
        char Err[] = "Error";

        char user[] = "";            // Имя пользователя
        char passwd[] = "";          // Пароль
        char dbase[] = "storage_db"; // Имя БД

        int server_sock;
        int client_sock;
        socklen_t client_len;
        sockaddr_in server_addr;
        sockaddr_in client_addr;

        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        server_addr.sin_port = htons(SERV_PORT);

        if(NULL == mysql_init(&mysql))
        {
            printf("Error mysql init.\n");
            return 0;
        }
    }

```

```

// Создание серверного сокета
server_sock = socket(AF_INET, SOCK_STREAM, 0);
if(0 > server_sock)
{
    printf("Error. Cannot open socket for server.\n");
    return 0;
}
// Назначение порта
if(0 > bind(server_sock, (sockaddr*)&server_addr, sizeof(server_addr)))
{
    printf("Error. Cannot bind address to server.\n");
    return 0;
}
// Перевод сокета в «слушающее» состояние
listen(server_sock, 1);
client_len = sizeof(client_addr);
// Прием подключения на новый сокет
client_sock = accept(server_sock, (sockaddr*)&client_addr, &client_len);

if(NULL == mysql_real_connect(&mysql, "", user,
                                passwd, dbase, 0, NULL, 0))
{
    printf("Error connect mysql.\n");
    close(client_sock);
    close(server_sock);
    return 0;
}

while(true)
{
    // Прием SQL-запроса от клиента
    recv(client_sock, buf, SIZE_BUF, 0);
    if(!strcmp(buf, "quit"))
        break;
    if(0 != mysql_query(&mysql, buf))
        send(client_sock, Err, strlen(Err) + 1, MSG_CTRUNC);
    else
    {
        strset(buf, '\0');
        res = mysql_store_result(&mysql);
        if(NULL != res)
        {
            num_fields = mysql_num_fields(res);
            while((row = mysql_fetch_row(res)))
            {
                for(i = 0; i < num_fields; i++)
                {
                    strcat(buf, row[i]);
                    buf[strlen(buf)] = '\t';
                }
                buf[strlen(buf)] = '\n';
            }
        }
    }
}

```

```

    }
    buf[strlen(buf)] = '\0';
    // Передача результата клиенту
    send(client_sock, buf, strlen(buf) + 1, MSG_CTRUNC);

}
else
    send(client_sock, Err, strlen(Err) + 1, MSG_CTRUNC);
mysql_free_result(res);
}
}
mysql_close(&mysql);
close(client_sock);
close(server_sock);
return 0;
}

```

В примере серверное программное обеспечение функционирует на рабочих станциях со следующими IP-адресами: 192.168.10.1, 192.168.10.2 и 192.168.10.3. Каждый сервер создает сокет и ожидает подключения клиента на порт с номером 2000. После того как пользователь ввел SQL-запрос, в цикле производится его передача серверам и прием результата. Выход из программы происходит после ввода пользователем команды «quit».

При сборке программы-сервера необходимо использовать mysqlclient библиотеки. Для этого компоновка выполняется со следующими параметрами (in.c – входной файл, out – выходной файл):

```
g++ -I/usr/include/mysql -L/usr/lib/mysql in.c -o out -lmysqlclient
```

при условии, что включаемые файлы размещены в /usr/include/mysql, а библиотеки в /usr/lib/mysql. Если это не так, то необходимо указать другой путь. При сборке программы-клиента используется следующая команда (in.c – входной файл, out – выходной файл):

```
g++ in.c -o out
```

Задания для освоения темы

Разработать два компонента трехуровневой архитектуры «клиент-сервер» для доступа к БД. Первый компонент (клиент) передает SQL-запросы второму компоненту (серверу). Сервер передает их СУБД MySQL, а результаты возвращает клиенту. Серверов

в системе должно быть не менее трех. Прием и передачу реализовать с использованием сокетов ОС Linux.

Т е м а 2.4. Библиотека MPI Linux и распределенные базы данных

Цель работы над темой – изучение функций библиотеки MPI Linux для реализации доступа к БД. Реализация трехуровневой архитектуры «клиент-сервер» на платформе MPI Linux.

Message Passing Interface (MPI) переводится как интерфейс передачи сообщений. Этот интерфейс реализуется набором библиотек функций для программирования на языках C, C++ и Fortran. Библиотеки позволяют разрабатывать эффективное ПО для высокопроизводительных вычислительных систем, особенностью которых являются параллельная обработка и взаимодействие между программами. Эффективность достигается большим количеством разнообразных функций передачи и приема данных с механизмами буферизации и синхронизации, позволяющих реализовать различные варианты коммуникаций между рабочими станциями или потоками. Возможность использовать функции MPI для коммуникаций между рабочими станциями позволяет использовать их и для организации доступа к хранимой на них информации, например, к базам данных.

Все MPI-программы находятся в области взаимодействия, которая защищает от возможных ошибок и несанкционированного доступа. Область взаимодействия образуется процессами, запущенными средствами загрузчика `mpirun` в рамках одной сессии.

Обычно работа процессов начинается с инициализации, которая предполагает определение задачи, возлагаемой на каждую ветвь. Инициализация может быть произведена или каким-то одним процессом, как это показано на рис. 23, посредством передачи некоторых начальных данных, или самостоятельно каждым приложением. После отработки задачи результат может быть собран и обобщен.

В трехуровневой архитектуре «клиент-сервер» один из таких процессов может выполнять функции клиента, а остальные процессы функции сервера, т.е. предоставлять некоторые услуги клиенту. При работе с БД серверные процессы принимают запросы клиента и передают их СУБД с последующим возвратом результата.

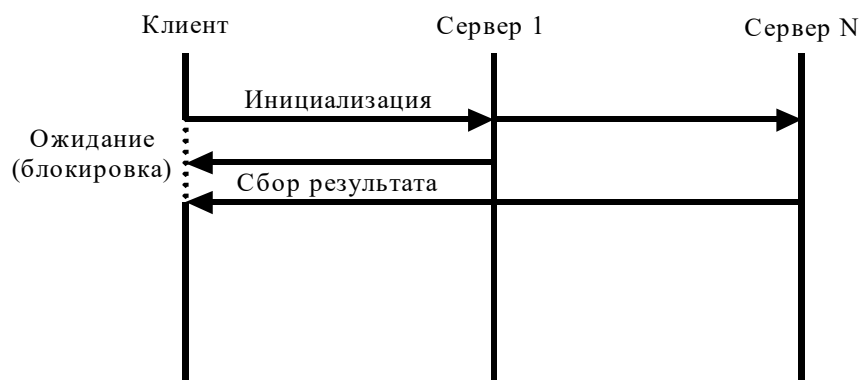


Рис. 23. Инициализация процессов

Отличие от традиционных систем состоит в использовании для коммуникаций интерфейса передачи сообщений, что разрабатывать эффективное ПО для сложных распределенных систем, таких как системы управления распределенной БД.

Существует несколько реализаций MPI. Одна из них MPICH Аргонской национальной лаборатории может быть взята с сайта www.mcs.anl.gov. Эта реализация является свободно распространяемой и существует для большинства операционных систем, включая ОС Linux и ОС Windows.

Чтобы использовать функций библиотеки MPI, необходимо в начале программы подключить заголовочный файл `mpi.h`. Любая MPI-программа должна иметь в начале функции `main` вызов `MPI_Init()`, который инициализирует библиотеку. Вызов этой функции обязательно должен предшествовать вызовам остальных функций библиотеки MPI. В конце программы должна вызываться функция `MPI_Finalize()`, которая осуществляет закрытие библиотеки.

После успешной инициализации можно использовать функции для передачи и приема сообщений, получения информации о процессах и области взаимодействия. Функция `MPI_Send()` позволяет передать данные другому процессу. Для приема данных используют функцию `MPI_Recv()`. Эти функции являются блокирующими. Функции `MPI_Isend()` и `MPI_Irecv()` являются не блокирующими. Их вызов инициирует передачу или прием с немедленным выходом из функции, что может быть использовано для продолжения вычислений. Определить окончания передачи или приема можно вызовом функции `MPI_Test()`.

Для определения количества процессов в области взаимодействия используется функция `MPI_Comm_size()`. Функция `MPI_Comm_rank()` возвращает порядковый номер вызвавшего ее процесса. Блок-схема возможного алгоритма работы клиента

в трехуровневой архитектуре «клиент–сервер» с использованием функций MPI представлена на рис. 24.

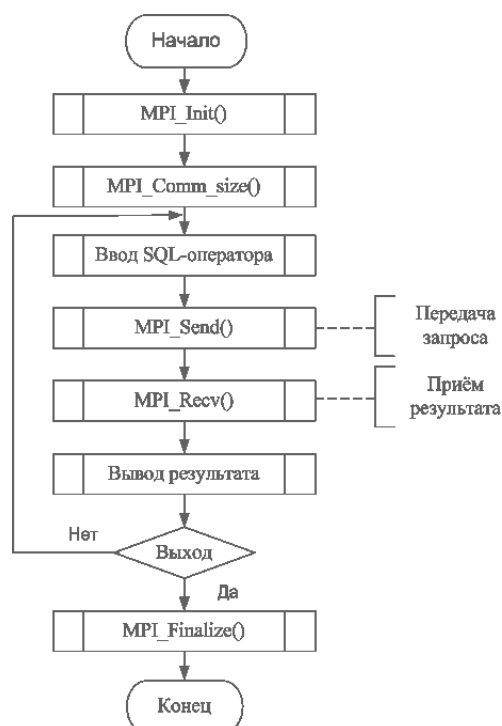


Рис. 24. Блок-схема алгоритма работы программы-клиента

Блок-схема возможного алгоритма работы сервера представлена на рис. 25.

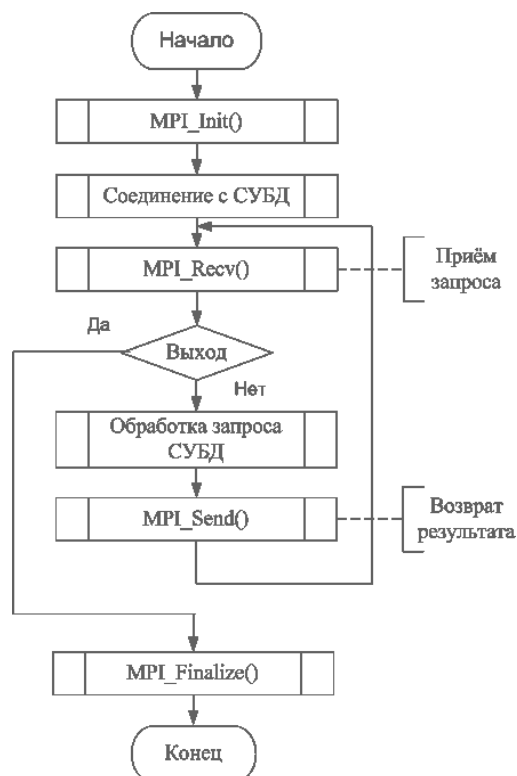


Рис. 25. Блок-схема алгоритма работы программы-сервера

Для компоновки программы используются командные файлы с именами `mpicc` и `mpiCC`. Первый компилирует программы на языке программирования C. Второй на языке программирования C++. Команда `mpiCC file1.c file2.c -o out` создаст исполняемый файл `out` из файлов `file1.c` и `file2.c`. Указать путь к командным файлам и загрузчику можно следующим образом:

```
shell> export PATH=/путь до MPI.../mpich/bin:$PATH
```

Следующая команда позволит использовать справку `man` для функций MPI и порядку работы с системой MPI:

```
shell> export MANPATH=/путь до MPI.../mpich/man:$MANPATH
```

Для запуска MPI программы используется командный файл-загрузчик `mpirun`. Есть два способа запустить программу. Первый способ: явное указание числа процессов и имени программы. Второй способ: создание конфигурационного файла.

При явном указании формат запуска загрузчика следующий `mpirun -np <число процессов> <имя программы>`. В этом случае будет запущено столько процессов, сколько укажет пользователь. Пример запуска трех программ с именем `cr1`:

```
shell> mpirun -np 3 ./cr1
```

Каждый процесс будет функционировать на том компьютере, имя или IP-адрес которого указан в файле `machines.<архитектура>` (для ОС Linux `machines.LINUX`). Этот файл расположен в `/путь до MPI... /mpich/util/machines/` и состоит из строк вида `имя_хоста:число_процессоров`. Недостатком такого подхода является невозможность запуска разных программ на разных машинах. Для запуска разных процессов на удаленных рабочих станциях используется конфигурационный файл, в котором указываются имя рабочей станции, директория, имя программы и число копий этой программы. В этом случае загрузчик `mpirun` выполняется с ключом `-p4pg`, после которого указывается имя конфигурационного файла и имя исполняемой программы на текущей рабочей станции. Конфигурационный файл содержит строки вида `<имя компьютера> <число процессов> <имя программы> [<имя пользователя>]`.

Пример:

192.168.10.1	1	/home/spec1/myprog
192.168.10.2	1	/home/spec1/prog

192.168.10.3	3	/home/spec1/relese
vt320-04	2	/home/spec1/new

Рекомендуется следующее содержание конфигурационного файла для запуска программы клиента (client) на Host_0 и нескольких серверов (server) на Host_1, Host_2 и Host_3:

Host_0	0	<путь до программы на Host_0>/server
Host_1	1	<путь до программы на Host_1>/server
Host_2	1	<путь до программы на Host_2>/server
Host_3	1	<путь до программы на Host_3>/server

Запуск системы осуществляется следующей командой:

```
shell> mpirun -p4pg ./pgfile ./client.,
```

где pgfile – имя конфигурационного файла. Такой способ гарантирует, что процесс client получит идентификатор внутри области взаимодействия равный нулю, а серверы другие, отличные от нуля идентификаторы. Эта информация используется для назначения идентификаторов процессов при организации коммуникаций в программах.

В каталоге /путь до MPI.../mpich/example/basic расположены примеры программ на языке C и Fortran.

Для запуска процесса на удаленной рабочей станции загрузчик mpirun использует сервис rsh или ssh в зависимости от конфигурации при установке пакета MPICH. Особенностью использования ssh является то, что при запуске каждого приложения необходимо вводить пароль для доступа к удаленной рабочей станции. Этому неудобства можно избежать, если заранее устанавливать переменные среды окружения, в которых указать имя пользователя и пароль для сервиса ssh. Сервис rsh ввода пароля не требует.

Пример:

```

/***** Клиент *****/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "mpi.h"

#define SIZE_BUF 1024

int main(int argc, char* argv[])

```

```

{
    char buf[SIZE_BUF];
    int i;
    int num_procs;
    int my_id;
    MPI_Status status;

    // Инициализация библиотеки MPI
    if(MPI_SUCCESS != MPI_Init(&argc, &argv))
    {
        printf("Error init MPI.\n");
        return 0;
    }
    // Определение числа процессов в области взаимодействия
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    if(2 > num_procs)
    {
        printf("Servers not find. Exit.\n");
        MPI_Finalize();
        return 0;
    }
    // Определение своего идентификатора в области взаимодействия
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

    printf("Enter SQL-query. (quit - exit)\n");

    while(true)
    {
        printf(">");
        gets(buf);
        for(i = 1; i < num_procs; i++)
            MPI_Send(buf, strlen(buf) + 1, MPI_CHAR,
                    i, 0, MPI_COMM_WORLD);

        if(!strcmp(buf, "quit"))
            break;
        for(i = 1; i < num_procs; i++)
        {
            MPI_Recv(buf, SIZE_BUF, MPI_CHAR,
                    i, 0, MPI_COMM_WORLD,
                    &status);
            printf("#%d\n%s\n", i, buf);
        }
    }
    // Заккрытие библиотеки MPI
    MPI_Finalize();
    return 0;
}

```

/***** Сервер *****/

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "mysql.h"
#include "mpi.h"

#define SIZE_BUF 1024
#define CLIENT_ID 0

int main(int argc, char* argv[])
{
    MYSQL mysql;
    MYSQL_RES* res;
    MYSQL_ROW row;
    unsigned int num_fields;
    unsigned int i;
    MPI_Status status;

    char buf[SIZE_BUF];
    char Err[] = "Error";

    char user[] = ""; // Имя пользователя
    char passwd[] = ""; // Пароль
    char dbase[] = "storage_db"; // Имя БД

    // Перенаправление вывода в стандартный поток
    setvbuf(stdout, NULL, _IONBF, 0);

    // Инициализация библиотеки MPI
    if(MPI_SUCCESS != MPI_Init(&argc, &argv))
    {
        printf("Error init MPI.\n");
        return 0;
    }

    if(NULL == mysql_init(&mysql))
    {
        printf("Error mysql init.\n");
        MPI_Finalize();
        return 0;
    }
    if(NULL == mysql_real_connect(&mysql, "", user, passwd,
                                dbase, 0, NULL, 0))
    {
        printf("Error connect mysql.\n");
        return 0;
    }

    while(true)
    {

```

```

// Прием SQL-запроса от клиента
MPI_Recv(buf, SIZE_BUF, MPI_CHAR,
          CLIENT_ID, 0, MPI_COMM_WORLD, &status);
if(!strcmp(buf, "quit"))
    break;
if(0 != mysql_query(&mysql, buf))
{
    MPI_Send(Err, strlen(Err) + 1, MPI_CHAR,
             CLIENT_ID, 0, MPI_COMM_WORLD);
}
else
{
    for(i = 0; i < SIZE_BUF; i++)
        buf[i] = 0;
    res = mysql_store_result(&mysql);
    if(NULL != res)
    {
        num_fields = mysql_num_fields(res);
        while((row = mysql_fetch_row(res)))
        {
            for(i = 0; i < num_fields; i++)
            {
                strcat(buf, row[i]);
                buf[strlen(buf)] = '\t';
            }
            buf[strlen(buf)] = '\n';
        }
        buf[strlen(buf)] = '\0';
        // Передача результата клиенту
        MPI_Send(buf, strlen(buf) + 1, MPI_CHAR,
                 CLIENT_ID, 0, MPI_COMM_WORLD);
    }
    else
    {
        // Передача сообщения об ошибке
        MPI_Send(Err, strlen(Err) + 1, MPI_CHAR,
                 CLIENT_ID, 0, MPI_COMM_WORLD);
    }
    mysql_free_result(res);
}
}
mysql_close(&mysql);
// Закрытие библиотеки MPI
MPI_Finalize();
return 0;
}

```

Задания для освоения темы

Разработать два компонента трехуровневой архитектуры «клиент–сервер» для доступа к БД. Первый компонент (клиент) передает SQL-запросы второму компоненту (серверу). Сервер передает

их СУБД MySQL, а результаты возвращает клиенту. Серверов в системе должно быть не менее трех. Прием и передачу реализовать с использованием функций библиотеки MPI Linux.

Т е м а 2.5. Использование технологии ODBC для работы с базами данных в операционной системе Windows

Цель работы над темой – изучение функций ODBC для работы с БД в ОС Windows. Построение клиентских приложений для обработки данных.

Одним из способов обработки данных, хранимых в БД, является разработка программы с использованием функций ODBC. Механизмы ODBC (Open Database connectivity – открытая связь с базами данных) существенно облегчают работу с данными и делают ее универсальной благодаря использованию драйверов для различных типов БД. Технология ODBC реализуется набором API-функций, на основе которых строятся такие ODBC-классы, как CDatabase, CRecordset и CRecordView.

Помимо готовых классов можно использовать и сами API-функции. Для этого необходимо подключить библиотеки sql.h и sqlext.h в начале программы. В этом случае работа с БД начинается с вызова инициализирующих функций ODBC. Первой вызывается функция SQLAllocEnv(). Она осуществляет распределение памяти для идентификатора среды типа HENV. Далее вызовом функции SQLAllocConnect() распределяется память для идентификатора соединения типа HDBC. Соединение с источником данных осуществляется вызовом функции SQLConnect(), входными параметрами которой являются идентификатор соединения, имя источника данных, имя пользователя и пароль. В случае успешного выполнения функции возвращают код SQL_SUCCESS.

После успешного соединения для выполнения SQL-операторов необходимо функцией SQLAllocStmt() распределить память для идентификатора оператора типа HSTMT. Оператор может быть выполнен двумя способами: последовательным вызовом функций SQLPrepare() и SQLExecute() или вызовом функции SQLExecDirect().

Выборка следующей строки результирующего набора осуществляется вызовом функции SQLFetch(). Получить значения столбцов результата можно с помощью функции SQLGetData() или связать их с буфером программы функцией SQLBindCol().

В первом случае после вызова SQLFetch() необходимо использовать SQLGetData() для каждого столбца. Во втором случае достаточно однократного вызова SQLBindCol() и после каждого выполнения SQLFetch() очередные данные будут в буфере программы. Типовой алгоритм использования функций ODBC API представлен на рис. 26.

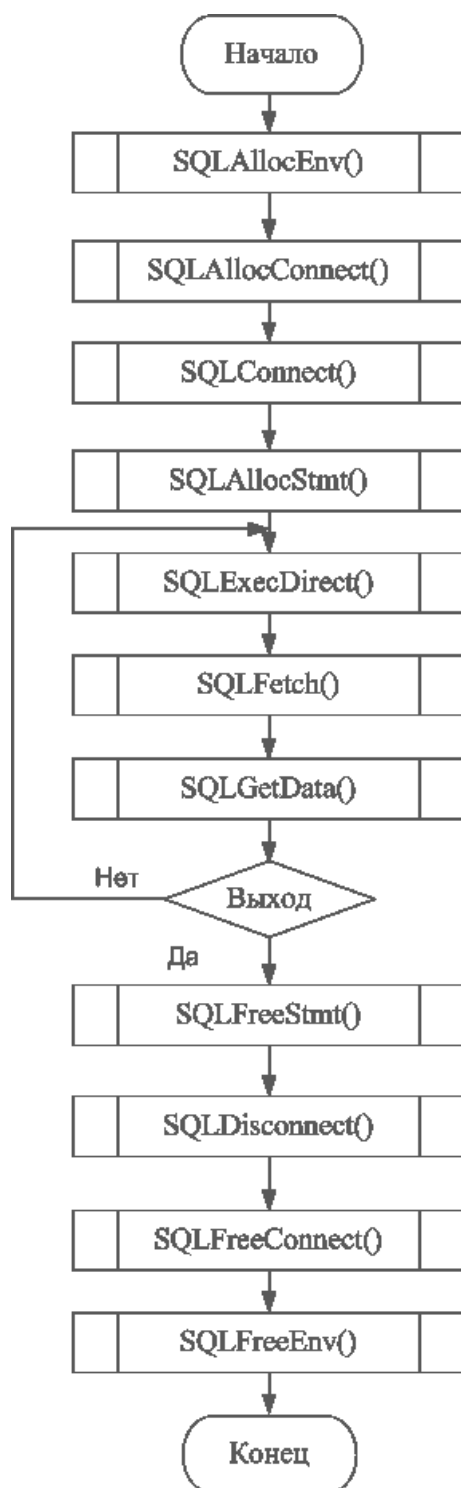


Рис. 26. Типовой алгоритм использования функций ODBC API

Перед началом работы с созданной БД ее необходимо зарегистрировать как источник данных. Это осуществляется программой odbcad32.exe. В программе необходимо указать драйвер источника данных, а затем и сам источник данных (новую БД).

Пример:

```
#include "stdafx.h"
#include "windows.h"
#include "sql.h"
#include "sqlext.h"

int main(int argc, char* argv[])
{
    HSTMT      hstmt; // идентификатор оператора
    HDBC hdbc; // идентификатор соединения
    HENV henv; // идентификатор среды

    SQLRETURN retcode;

    char  dbase[] = "Storage";      // имя БД
    char  user[] = "";              // имя пользователя
    char  passwd[] = "";            // пароль пользователя
    char  query[] = "SELECT * FROM STORAGE"; // SQL-оператор

    SQLINTEGER sStorageID; // поле Storage_ID
    SQLINTEGER sNumber; // поле Number
    SQLINTEGER sArea;      // поле Area

    SQLINTEGER cbStorageID;
    SQLINTEGER cbNumber;
    SQLINTEGER cbArea;

    // назначение идентификатора среды
    retcode = SQLAllocEnv(&henv);
    if(SQL_SUCCESS != retcode)
    {
        printf("Не удалось назначить идентификатора среды.\n");
        return 0;
    }

    // назначение идентификатора соединения
    retcode = SQLAllocConnect(henv, &hdbc);
    if(SQL_SUCCESS != retcode)
    {
        printf("Не удалось назначить идентификатора соединения.\n");
        SQLFreeEnv(henv);
        return 0;
    }
}
```

```

// соединение с источником данных
retcode = SQLConnect(hdbc, (SQLCHAR*)dbase,
                    SQL_NTS, (SQLCHAR*)user,
                    SQL_NTS, (SQLCHAR*)passwd,
                    SQL_NTS);

if(SQL_SUCCESS != retcode)
{
    printf("Не удалось соединиться с БД.\n");
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    return 0;
}

// назначение идентификатора оператора
retcode = SQLAllocStmt(hdbc, &hstmt);
if(SQL_SUCCESS != retcode)
{
    printf("Не удалось назначить идентификатора оператора.\n");
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    return 0;
}

// выполнение SQL-оператора
retcode = SQLExecDirect(hstmt, (SQLTCHAR*)(LPCTSTR)query,
                        strlen(query));

if(SQL_SUCCESS == retcode)
{
    while(TRUE)
    {
        // выборка следующей строки
        retcode = SQLFetch(hstmt);
        if(SQL_SUCCESS == retcode)
        {
            // выборка данных из колонок
            SQLGetData(hstmt, 1, SQL_C_LONG, &sStorageID,
                      0,
                      &cbStorageID);
            SQLGetData(hstmt, 2, SQL_C_LONG, &sNumber,
                      0,
                      &cbNumber);
            SQLGetData(hstmt, 3, SQL_C_LONG, &sArea,
                      0, &cbArea);

            // вывод результата
            printf("%d\t%d\t%d\n", sStorageID, sNumber, sArea);
        }
        else
            break;
    }
}

```

```

else
{
    printf("Ошибка выполнения SQL-оператора.\n");
}

SQLFreeStmt(hstmt, SQL_DROP);
SQLDisconnect(hdbc);    // закрытие соединения
SQLFreeConnect(hdbc);   // освобождение памяти
SQLFreeEnv(henv);       // освобождение памяти

return 0;
}

```

Задания для освоения темы

С помощью программы MS Access создать базу данных аналогичную созданной в процессе выполнения предыдущего задания. Разработать программу на языке Visual C++ для работы с этой БД.

Т е м а 2.6. Технология сокетов в операционной системе Windows и распределенные базы данных

Цель работы над темой – использование технологии сокетов ОС Windows для передачи и приема информации в трехуровневой архитектуре «клиент-сервер».

Для доступа к информации, хранимой на разных узлах (физически разнесенной), но логически связанной, необходимо два приложения: приложение-клиент и приложение-сервер. Клиент передает команды обработки данных серверу. Сервер интерпретирует команды в SQL-запросы и передает их серверу БД. В ОС Windows в качестве интерфейса между сервером и БД может использоваться технология ODBC, а для взаимодействия клиента и сервера различные коммуникационные протоколы и технологии. Наиболее часто используется технология сокетов, которая отличается простотой использования и наличием достаточного количества примитивов для организации коммуникаций. В приложениях типа «клиент–сервер» сокет можно непосредственно использовать для передачи и приема данных между программами.

Для использования сокетов в проекте на языке C++, созданном в системе программирования Microsoft Visual Studio 2010, необходимо подключить библиотеку winsock2, добавив в заголовок исходного файла следующие директивы:

```

#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")

```

Перед началом работы с сокетами необходимо инициализировать библиотеку. Это делается с помощью функции `WSAStartup()`. Если функция выполнилась успешно, возвращается 0, иначе – значение `SOCKET_ERROR`. Код ошибки можно получить с помощью функции `WSAGetLastError()`.

Пример:

```
/****** Клиент *****/
#include "stdafx.h"
#include <winsock2.h>
#include <ws2tcpip.h>

#pragma comment(lib, "Ws2_32.lib")

#define SIZE_BUF 1024
#define COUNT_SERV 3

int main(int argc, char* argv[])
{
    char buf[SIZE_BUF];
    const int port[COUNT_SERV] = { 2000, 2001, 2002};
    const char* host[COUNT_SERV] = { "127.0.0.1", "127.0.0.1", "127.0.0.1"};

    sockaddr_in addr[COUNT_SERV];

    WSADATA wsaData;
    SOCKET sock[COUNT_SERV];

    if(WSAStartup(MAKEWORD(2, 0), &wsaData))
    {
        printf("Не удалось инициализировать библиотеку winsock.\n");
        return 0;
    }

    for(int i = 0; i < COUNT_SERV; i++)
    {
        addr[i].sin_family = AF_INET;
        addr[i].sin_addr.s_addr = inet_addr(host[i]);
        addr[i].sin_port = htons(port[i]);

        sock[i] = socket(AF_INET, SOCK_STREAM, 0);
        if(0 > sock[i])
        {
            printf("Не удалось создать %d клиентский сокет.\n", i);
            return 0;
        }
        if(0 > connect(sock[i],(sockaddr*)&addr[i], sizeof(addr[i])))
        {

```

```

        printf("Не удалось соединиться с %d сервером.\n", i);
        return 0;
    }
}

printf("Соединение выполнено успешно.\n");
printf("Введите SQL-оператор для таблицы Storage.\n");
printf("(quit - выход из приложения)\n");

while(true)
{
    printf(">");
    gets(buf);
    for(int i = 0; i < COUNT_SERV; i++)
        send(sock[i], buf, strlen(buf) + 1, 0);
    if(!strcmp(buf, "quit"))
        break;
    for(int i = 0; i < COUNT_SERV; i++)
    {
        recv(sock[i], buf, SIZE_BUF, 0);
        printf("#%d\n%s\n", i, buf);
    }
}

for(int i = 0; i < COUNT_SERV; i++)
    closesocket(sock[i]);

WSACleanup();
return 0;
}

```

```

/***** Сервер *****/
#include "stdafx.h"
#include <winsock2.h>
#include <ws2tcpip.h>

```

```

#pragma comment(lib, "Ws2_32.lib")

```

```

#include "windows.h"
#include "sql.h"
#include "sqlext.h"

```

```

#define SERV_PORT    2002
#define SIZE_BUF 5120 // 5K

```

```

int main()
{
    HSTMT hstmt; // идентификатор оператора
    HDBC hdbc; // идентификатор соединения
    HENV henv; // идентификатор среды

```

```

SQLRETURN retcode;

char buf[SIZE_BUF];
char conv[100];
char Err[] = "Error query!";

char   dbase[] = "Storage";      // имя БД
char   user[] = "";              // имя пользователя
char   passwd[] = "";            // пароль пользователя

SQLINTEGERsStorageID;
SQLINTEGERsNumber;
SQLINTEGERsArea;

SQLINTEGERcbStorageID;
SQLINTEGERcbNumber;
SQLINTEGERcbArea;

WSADATA wsaData;
SOCKET ServerSock;
SOCKET ClientSock;
int client_len;

sockaddr_in server_addr;
sockaddr_in client_addr;

server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(SERV_PORT);

if(WSAStartup(MAKEWORD(2, 0), &wsaData))
{
    printf("Не удалось инициализировать библиотеку winsock.\n");
    return 0;
}

// назначение идентификатора среды
retcode = SQLAllocEnv(&henv);

if(SQL_SUCCESS != retcode)
{
    printf("Не удалось назначить идентификатора среды.\n");
    return 0;
}

// назначение идентификатора соединения
retcode = SQLAllocConnect(henv, &hdbc);

if(SQL_SUCCESS != retcode)
{

```

```

        printf("Не удалось назначить идентификатора соединения.\n");
        SQLFreeEnv(henv);
        return 0;
    }

    // соединение с источником данных
    retcode = SQLConnect(hdbc, (SQLWCHAR*)dbase,
                        SQL_NTS, (SQLWCHAR*)user,
                        SQL_NTS, (SQLWCHAR*)passwd,
                        SQL_NTS);

    if(SQL_SUCCESS != retcode)
    {
        printf("Не удалось соединиться с БД.\n");
        SQLFreeConnect(hdbc);
        SQLFreeEnv(henv);
        return 0;
    }

    ServerSock = socket(AF_INET, SOCK_STREAM, 0);
    if(0 > ServerSock)
    {
        printf("Не удалось создать серверный сокет.\n");
        return 0;
    }

    if(0 > bind(ServerSock, (sockaddr*)&server_addr, sizeof(server_addr)))
    {
        printf("Не удалось назначить порт для сервера.\n");
        return 0;
    }

    listen(ServerSock, 1);
    client_len = sizeof(client_addr);
    ClientSock = accept(ServerSock, (sockaddr*)&client_addr, &client_len);

    while(true)
    {
        recv(ClientSock, buf, SIZE_BUF, 0);
        if(!strcmp(buf, "quit"))
            break;
        printf("%s\n", buf);
        // назначение идентификатора (ид.) оператора
        retcode = SQLAllocStmt(hdbc, &hstmt);
        if(SQL_SUCCESS != retcode)
        {
            printf("Не удалось инициализировать ид. оператора.\n");
            SQLFreeStmt(hstmt, SQL_DROP);
            continue;
        }
    }

```

```

        // выполнение SQL-оператора
        retcode = SQLExecDirect(hstmt, (SQLTCHAR*)(LPCTSTR)buf,
                                strlen(buf));

        if(SQL_SUCCESS != retcode)
        {
            send(ClientSock, Err, strlen(Err) + 1, 0);
        }
        else
        {
            strset(buf, "\0");
            while(TRUE)
            {
                // выборка следующей строки
                retcode = SQLFetch(hstmt);
                if(SQL_SUCCESS == retcode)
                {
                    // выборка данных из колоно
                    SQLGetData(hstmt, 1, SQL_C_LONG,
                                &sStorageID, 0, &cbStorageID);
                    SQLGetData(hstmt, 2, SQL_C_LONG,
                                &sNumber, 0, &cbNumber);
                    SQLGetData(hstmt, 3, SQL_C_LONG,
                                &sArea, 0, &cbArea);
                    // формирование результата
                    sprintf(conv, "%d\t%d\t%d\n",
                            sStorageID, sNumber, sArea);
                    strcat(buf, conv);
                }
                else
                {
                    // передача результата
                    send(ClientSock, buf, strlen(buf) + 1, 0);
                    break;
                }
            }
            SQLFreeStmt(hstmt, SQL_DROP);
        }

    }

    SQLFreeStmt(hstmt, SQL_DROP);
    SQLDisconnect(hdbc); // закрытие соединения
    SQLFreeConnect(hdbc); // освобождение памяти
    SQLFreeEnv(henv); // освобождение памяти

    // закрытие сокетов
    closesocket(ClientSock);
    closesocket(ServerSock);
    return 0;
}

```

Задания для освоения темы

Созданную при работе над темой 2.5 БД скопировать для нескольких рабочих станций, объединенных в локальную сеть. Зарегистрировать копии БД на каждой станции. Разработать два компонента трехуровневой архитектуры «клиент-сервер» для доступа к БД. Первый компонент (клиент) передает SQL-запросы второму компоненту (серверу). Клиент осуществляет передачу команд пользователя с помощью технологии сокетов серверам. Сервер, используя механизмы ODBC, выполняет запрос к БД и возвращает результат или код ошибки.

Т е м а 2.7. Библиотека MPI Windows и распределенные базы данных

Цель работы над темой – изучение функций библиотеки MPI Windows для реализации доступа к БД. Реализация трехуровневой архитектуры «клиент-сервер» на платформе MPI Windows.

В теме 2.4 рассматривался вопрос использования библиотек MPI на платформе Linux, схожим образом они используются в Windows. В данной лабораторной работе будем использовать пакет mpich2-1.4-win-ia32.msi. По умолчанию файлы пакета расположены в каталоге "C:\Program Files\MPICH2\". В подкаталоге bin располагается загрузчик mpiexec.exe, сервис запуска smpd.exe и другие системные файлы. В подкаталогах include и lib располагаются соответственно заголовочные файлы и библиотеки для компиляции. Разработка проекта проходит следующие этапы:

1. В Visual C++ создаем новый проект, являющийся консольным приложением (Visual C++ → New → Win32 Console Application).

2. Для использования MPI функций подключаем заголовочный файл mpi.h.

3. Для того чтобы не копировать заголовочные и библиотечные файлы MPI, нужно указать путь к ним во вкладке Project -> Settings.

4. Указываем путь к заголовочным файлам: Project → Properties. Выбираем в поле Configuration – All configurations. Выбираем в левом окне дерева свойств Configuration Properties → C/C++ → General → Additional Include Directories. Добавляем в это поле путь "C:\Program Files\MPICH2\include".

5. Указываем путь к файлам библиотек: Project → Properties. Выбираем в поле Configuration – All configurations. Выбираем

в левом окне дерева свойств Configuration Properties → Linker → General → Additional Library Directories. Добавляем в это поле путь "C:\Program Files\MPICH2\lib".

6. Необходимо добавить библиотеки в проект, для каждой конфигурации "Debug" или "Release" указываем: Configuration Properties → Linker → Input → Additional Dependencies. Добавляем в список библиотеки mpi.lib, sxx.lib.

7. Убедиться, что в настройках установлена многопоточная библиотека: Configuration Properties → C/C++ → Code Generation → Runtime Library → Multithreaded Debug DLL.

8. На данном этапе запускаем компиляцию проекта и, если ошибок нет, можно начинать писать программу с использованием функций MPI.

Для запуска MPI программы используется приложение mpiexec. Есть два способа запустить программу. Первый способ: явное указание числа процессов и имени программы. Второй способ: использование настроечного файла, созданного специальным приложением wmpiconfig.

При явном указании формат запуска загрузчика следующий: mpiexec -n <число процессов> <имя программы>. В этом случае будет запущено столько процессов, сколько укажет пользователь. Если перед этим не производилось никакой настройки mpiexec, то все процессы будут функционировать на одном компьютере. Для того чтобы запустить процессы на разных машинах, необходимо создать конфигурационный файл. При первичном запуске программа попросит ввести имя пользователя и пароль аутентификации, которые должны быть одинаковыми на всех машинах, участвующих в эксперименте. В конфигурационном файле указывается список рабочих станций, на которых необходимо запустить параллельную программу, и другие параметры. Эти параметры следует указать в конфигурационном окне программы wmpiconfig. При запуске в домене нажатием кнопки Add Hosts можно добавить в список все рабочие станции, входящие в домен. На каждой рабочей станции должен быть установлен пакет mpiexec той же версии, что и на сервере.

Задание для освоения темы

Разработать два компонента трехуровневой архитектуры «клиент-сервер» для доступа к БД. Первый компонент (клиент) передает SQL-запросы второму компоненту (серверу). Клиент осуществляет передачу команд пользователя с помощью функций

библиотеки MPI Windows серверам. Сервер, используя механизмы ODBC, выполняет запрос к БД и возвращает результат или код ошибки.

В результате изучения тем части 2 должен быть получен практический опыт проектирования приложений для доступа к распределенным базам данных с использованием сокетов TCP/IP, библиотек MPI и ODBC.

Часть 3

ИСПОЛЬЗОВАНИЕ ЯЗЫКА C# ДЛЯ РАЗРАБОТКИ СЕТЕВЫХ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ

Тема 3.1. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP

Цель работы над темой – изучение протоколов семейства TCP/IP. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP на языке C#.

Задание: разработать программу клиента, которая должна:

- запрашивать у пользователя адрес программы-сервера;
- устанавливать соединение с сервером;
- передавать на сервер данные;
- принимать ответ от сервера и выводить его на экран;
- закрывать соединение с сервером.

Основные сведения

Для того чтобы установить связь с процессом, выполняющимся на другой ЭВМ, клиентская программа должна создать сокет. Сокет в любой современной системе представляет собой особый вид файла, из которого можно читать и в которой можно записывать двоичные данные. При операциях обмена с сокетом нет никакого контроля типов, эта задача возлагается на приложения. На схеме алгоритма (рис. 27) блок «Создание сокета» подразумевает создание экземпляра класса Socket, который в случае успеха создает сокет – потоковое или дейтаграммное – и семейство протоколов. В данном случае создается потоковый сокет и используется семейство протоколов TCP/IP.

Установка соединения с другим процессом заключается в обмене специальными пакетами и возможно только тогда, когда тот процесс ожидает приема соединений. В противном случае результат операции будет неудачным и будет получено сообщение о том, что либо не удалось установить соединение, либо оно было разорвано (зависит от реализации).

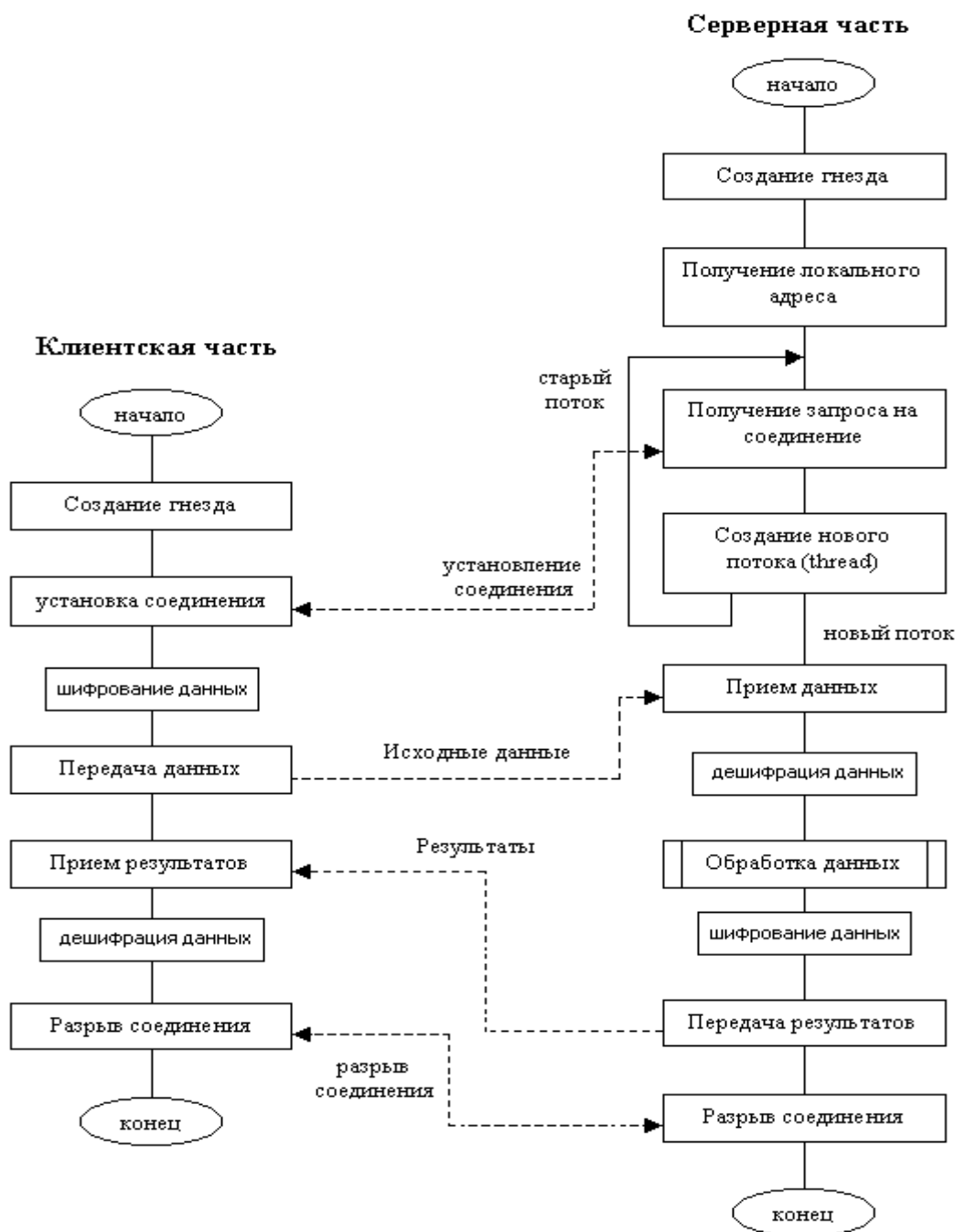


Рис. 27. Алгоритм работы «клиент-серверных» приложений

Для установки соединения требуется указать ЭВМ по IP-адресу или по доменному имени, которое обязательно должно быть преобразовано в IP-адрес, и процесс на этой ЭВМ (по целочисленному идентификатору, называемому портом). Все это реализуется с помощью метода Connect класса Socket. Если соединение успешно установлено, то сразу после вызова этой функции можно вести обмен с гарантированной доставкой пакетов. В противном случае работа невозможна.

Передача и прием данных производятся с помощью методов Send и Receive класса Socket. Перед осуществлением передачи данные, если это необходимо, шифруются каким-либо алгоритмом. На принимающей стороне полученные данные дешифруются, и определяется (или опровергается) их подлинность, от результата зависит дальнейшая работа с этим клиентом.

Разрыв соединения означает обмен специальными пакетами и может производиться с помощью метода Close класса Socket. Функция Close уничтожает сокет, делая его непригодным к использованию (любая операция с ним будет заканчиваться неудачей).

Пример кода клиентского приложения на языке C# приведен ниже.

```
using System.Net;
using System.Net.Sockets;

namespace Client
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private IPEndPoint ipHost;
        private IPAddress ipAddr;
        private IPEndPoint ipEndPoint;
        private Socket sSender;

        private void checkBoxPsw_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBoxPsw.Checked)
                textBoxPsw.ReadOnly = false;
            else
                textBoxPsw.ReadOnly = true;
        }

        private void buttonConnect_Click(object sender, EventArgs e)
        {
            //Буфер для входящих данных
            byte[] bytes = new byte[256];

            //Соединяемся с удаленным устройством
            try
            {
                //Считываем введенные пользователем ip-адрес и номер порта
                ipHost = Dns.Resolve(textBoxIp.Text);
            }
        }
    }
}
```

```

        ipAddr = ipHost.AddressList[0];
        ipEndPoint = new IPEndPoint(ipAddr, Convert.ToInt32(textBoxPort.Text));

        //Создаем сокет
        sSender = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);

        //Соединяем сокет с удаленной конечной точкой
        sSender.Connect(ipEndPoint);

        textBoxResult.Text = textBoxResult.Text + "Соединение установлено" +
        Environment.NewLine;
    }
    catch (Exception ex)
    {
        textBoxResult.Text = textBoxResult.Text + ex.ToString() + Environment.NewLine;
    }
}

private void SendMsg(string mess)
{
    //Получаем массив байт из строки
    byte[] msg = Encoding.UTF8.GetBytes(mess);

    //Шифруем сообщение, если введен пароль
    if (textBoxPsw.Text != "")
    {
        byte[] psw = Encoding.UTF8.GetBytes(textBox4.Text);
        for (int i = 0; i < (msg.Length); i++)
        {
            msg[i] = (byte)(msg[i] + psw[i % psw.Length]);
        }
    }
    try
    {
        //Отправляем данные через сокет
        int bytesSent = sSender.Send(msg);
    }
    catch (Exception ex)
    {
        textBoxResult.Text = textBoxResult.Text + ex.ToString() + Environment.NewLine;
    }
}

private void ReseiveData()
{
    //Буфер для входящих данных
    byte[] bytes = new byte[512];
    try
    {
        //Получаем ответ от удаленного устройства

```

```

int bytesRec = sSender.Receive(bytes);
//Расшифровываем сообщение, если введен пароль
if (textBoxPsw.Text != "")
{
    byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
    for (int i = 0; i < (bytes.Length); i++)
    {
        bytes[i] = (byte)(bytes[i] - psw[i % psw.Length]);
    }
}

//Получаем строку из массива байт и выводим ее в текстовое поле
textBoxResult.Text = textBoxResult.Text + Encoding.UTF8.GetString(bytes, 0,
bytesRec) + Environment.NewLine;

//закрываем соединение
sSender.Shutdown(SocketShutdown.Both);
//Уничтожаем сокет
sSender.Close();
textBoxResult.Text = textBoxResult.Text + "Соединение закрыто" + Environ-
ment.NewLine;
}
catch (Exception ex)
{
    textBoxResult.Text = textBoxResult.Text + ex.ToString() + Environment.NewLine;
}
}

private void buttonSend_Click(object sender, EventArgs e)
{
    string mess = textBox4.Text;
    SendMsg(mess);
    ReseiveData();
}
}
}

```

Т е м а 3.2. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP

Цель работы над темой – разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP на языке C#.

Задание: разработать программу сервера, которая должна:

- ожидать запросов от программ-клиентов на соединение;
- устанавливать соединение с клиентами;

- принимать данные от клиентов и выполнять их обработку;
- пересылать результат обработки клиенту.

Основные сведения

Основной задачей серверной части является обработка. Обмен данными с клиентскими процессами есть важная составляющая часть этой задачи.

Для того, чтобы процессы-клиенты могли связаться с сервером, сервер создает сокет для обмена данными. На схеме алгоритма это представлено блоком «Создание сокета» и производится так же, как и в клиентской программе.

Следующий блок – «Получение локального адреса» – принципиально важен. Он служит для того, чтобы все запросы на соединения, приходящие на данную ЭВМ и обращающиеся к указанному порту, операционная система направляла данному процессу. Операция производится посредством метода Bind класса Socket. После этого, в случае успеха, программа сервера вызывает функцию Listen, которая говорит операционной системе о том, что процесс ожидает поступления запросов на соединение к данному сокету и что эти запросы нужно ставить в очередь указанной длины (в штуках).

Получение запроса на соединение происходит тогда, когда клиентский процесс вошел в блок «Установка соединения», т.е. вызвал функцию Connect. ОС сервера при этом создает копию сокета, чтобы программа могла на первом экземпляре продолжить работу, а на другом – вести обмен с подключившимся клиентом.

Передача и прием данных производятся с помощью методов Send и Receive класса Socket. После приема данных они дешифруются с целью, во-первых, получить открытый текст и, во-вторых, чтобы определить подлинность данных. Затем, если установлена подлинность данных и получен корректный открытый текст, производится обработка данных. Перед отправкой клиенту результатов работы данные снова шифруются.

По окончании работы с клиентом серверный процесс закрывает свою копию сокета и уничтожается.

Пример кода приложения-сервера на языке C# приведен ниже.

```
private void TcpServer()  
{  
    //Устанавливает для сокета локальную конечную точку  
    IPHostEntry ipHost = Dns.Resolve("localhost");
```

```

IPAddress ipAddr = ipHost.AddressList[0];
IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);

//Создаем сокет TCP/IP
Socket sListener = new Socket(AddressFamily.InterNetwork, Socket-
Type.Stream, ProtocolType.Tcp);

//Назначаем сокет локальной конечной точке и "слушаем" входящие сокеты
try
{
    sListener.Bind(ipEndPoint);
    sListener.Listen(10);

//Начинаем "слушать" соединение
    while (true)
    {
        SetTextBox("Сервер ожидает" + Environment.NewLine);

        //Программа приостанавливается, ожидая входящее соединение
        Socket handler = sListener.Accept();

        //Мы дождались клиента, пытающегося с нами соединиться
        string data = null;
        int bytesRec;

        byte[] bytes = new byte[256];

        //Получаем данные, посылаемые клиентом
        bytesRec = handler.Receive(bytes);

        //Дешифруем, если введен пароль
        if (textBoxPsw.Text != "")
        {
            byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
            for (int i = 0; i < (bytes.Length); i++)
            {
                bytes[i] = (byte)(bytes[i] - psw[i % psw.Length]);
            }
        }

        //Получаем строку из массива байт
        data = Encoding.UTF8.GetString(bytes, 0, bytesRec);

//Выводим данные на экран
        SetTextBox("Полученный запрос: " + data + Environment.NewLine);

        string theReply = "Полученное сообщение: " + data;

        //Получаем массив байт из строки
        byte[] msg = Encoding.UTF8.GetBytes(theReply);

```

```

//Зашифровываем отправляемое сообщение
if (textBoxPsw.Text != "")
{
    byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
    for (int i = 0; i < (msg.Length); i++)
    {
        msg[i] = (byte)(msg[i] + psw[i % psw.Length]);
    }
}

//Отправляем сообщение
handler.Send(msg);

//Разрываем соединение с клиентом
handler.Shutdown(SocketShutdown.Both);
handler.Close();
if (stopReceive == true) break;
}
}
catch (Exception ex)
{
    SetTextBox(ex.ToString() + Environment.NewLine);
}

//Прекращаем прослушивание соединения
sListener.Close();
SetTextBox("Сервер остановлен" + Environment.NewLine);
}

```

Т е м а 3.3. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP

Цель работы над темой – разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP на языке C#.

Задание: разработать программу-сервер, которая должна получать сообщения от клиентов и отправлять им ответы с использованием протокола UDP.

Основные сведения

Для того, чтобы процессы-клиенты могли связаться с сервером, сервер создает сокет для обмена данными. Для этого создается либо экземпляр класса `Socket`, либо экземпляр класса `UdpClient`. Класс `UdpClient` построен на классе `Socket`, но скрывает излишние члены, которые не требуются для реализации приложения, базирующегося на UDP.

Далее в цикле происходит прием данных через сокет от клиента с помощью функции Receive. Принятая строка выводится на экран и затем отсылается обратно клиенту с помощью метода Send.

По окончании работы процесс-сервер закрывает сокет с помощью функции Close.

Пример кода приложения-сервера на языке C# приведен ниже.

```
private void UdpServer()
{
    //Создание экземпляра класса UdpClient,
    //который предоставляет методы для отправки и получения UDP-
    //дейтаграмм
    UdpClient udp = new UdpClient(11000);
    SetTextBox("Серверзапущен" + Environment.NewLine);

    //Цикл прослушивания порта в ожидании входящих сообщений
    while (true)
    {
        try
        {
            IPEndPoint ipendpoint = null;

            //Получаем сообщения от клиента,
            //сохраняем адрес отправителя в переменной ipendpoint
            byte[] message = udp.Receive(ref ipendpoint);

            //Расшифровка, если введен пароль
            if (textBoxPsw.Text != "")
            {
                byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
                for (int i = 0; i < message.Length; i++)
                {
                    message[i] = (byte)(message[i] - psw[i % psw.Length]);
                }
            }

            //Получение строки из массива байт
            string data = Encoding.UTF8.GetString(message);
            string text = "Полученное сообщение: " + data + Environment.NewLine;
            SetTextBox(text);

            string theReply = text;

            //Преобразуем ответ сервера из строки в массив байт
            byte[] msg = Encoding.UTF8.GetBytes(theReply);

            //Шифруем, если введен пароль
            if (textBoxPsw.Text != "")
            {
                byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
```

```

        for (int i = 0; i < msg.Length; i++)
        {
            msg[i] = (byte)(msg[i] + psw[i % psw.Length]);
        }
    }
    //Отсылаем сообщение, используя адрес, с которого пришел запрос
    int sended = udp.Send(msg, msg.Length, ipendpoint);
    //Если данные были переданы, то с большой долей вероятности можно
    сказать,
    //что они дошли до адресата
    if (sended != msg.Length)
    {
        SetTextBox("Ошибка: Не все данные были переданы!" + Environ-
ment.NewLine);
    }
}
catch (Exception ex)
{
    SetTextBox(ex.ToString() + Environment.NewLine);
}
if (stopReceive == true) break;
}
try
{
    //Удаляем экземпляр класса UdpClient
    udp.Close();
    SetTextBox("Сервер остановлен" + Environment.NewLine);
}
catch (Exception ex)
{
    SetTextBox(ex.ToString() + Environment.NewLine);
}
}
}

```

Т е м а 3.4. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP

Цель работы над темой – разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием протокола UDP на языке C#.

Задание: разработать программу-клиент, которая должна отправлять серверу сообщение и получать ответ с использованием протокола UDP.

Основные сведения

Работа программы-клиента во многом схожа с работой программы-сервера. Вначале создается сокет для обмена данными.

Для этого создается либо экземпляр класса Socket, либо экземпляр класса UdpClient.

Сообщение отправляется серверу с помощью метода Send. Далее происходит прием данных от сервера с помощью функции Receive. Принятая строка выводится на экран. По окончании работы процесс закрывает сокет с помощью функции Close.

Пример кода приложения-клиента на языке C# приведен ниже.

```
private void buttonSend_Click(object sender, EventArgs e)
{
    try
    {
        //Создание экземпляра класса UdpClient,
        //который предоставляет методы для отправки и получения UDP-
        //дейтаграмм
        UdpClient udp = new UdpClient(
            buttonSend.Text, Convert.ToInt32(textBoxPort.Text));

        string mess = textBoxMessage.Text;

        //Преобразуем сообщение из строки в массив байт
        byte[] msg = Encoding.UTF8.GetBytes(mess);

        //Шифруем, если введен пароль
        if (textBoxPsw.Text != "")
        {
            byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
            for (int i = 0; i < msg.Length; i++)
            {
                msg[i] = (byte)(msg[i] + psw[i % psw.Length]);
            }
        }

        //Отсылаем сообщение серверу
        int send = udp.Send(msg, msg.Length);
        //Если данные были переданы, то с большой вероятностью можно сказать,
        //что они дошли до адресата
        if (send != msg.Length)
        {
            textBoxResult.Text = textBoxResult.Text + "Ошибка: Не все данные были
            переданы!" + Environment.NewLine;
        }

        //Прием
        IPEndPoint ipendpoint = null;
        //Получение ответа сервера
        byte[] data = udp.Receive(ref ipendpoint);
        //Расшифровка, если введен пароль
```

```

        if (textBoxPsw.Text != "")
        {
            byte[] psw = Encoding.UTF8.GetBytes(textBoxPsw.Text);
            for (int i = 0; i < data.Length; i++)
            {
                data[i] = (byte)(data[i] - psw[i % psw.Length]);
            }
        }

textBoxResult.Text = textBoxResult.Text + Encoding.UTF8.GetString(data) +
Environment.NewLine;

        //Удаляем экземпляр класса UdpClient
        udp.Close();
    }
    catch (Exception ex)
    {
        textBoxResult.Text = textBoxResult.Text + ex.ToString() + Environment.NewLine;
    }
}

```

Т е м а 3.5. Разработка интерфейса серверного приложения с базой данных

Цель работы над темой – изучение интерфейса ODBC. Разработка интерфейса серверного приложения с базой данных на языке C#.

Задание: разработать программу-сервер, которая должна отправлять запрос базе данных и обрабатывать полученную из БД информацию.

Основные сведения

Взаимодействие сервера с базой данных осуществляется с помощью программного интерфейса ODBC. Для соединения с базой данных используется метод Open класса OdbcConnection. После окончания работы с БД соединение закрывается с помощью функции Close. Запрос к БД формируется с помощью экземпляра класса OdbcCommand. В переменную CommandText этого класса должна быть занесена строка, содержащая sql-запрос. Для чтения из БД используется класс OdbcDataReader. Метод Read используется для получения следующей строки из БД:

```

using System.Data.Odbc;

private string GetFromDB(string sqlcom)

```

```

{
    // prepare ODBC database connection
    string strConnect =
"DSN=MySQL;UID=root;PWD=wiwi;DATABASE=bookstore;CharSet=cp1251";
    OdbcConnection dbMySQL = new OdbcConnection(strConnect);
    try
    {
        dbMySQL.Open();
        OdbcCommand sqlCommand1 = dbMySQL.CreateCommand();
        sqlCommand1.CommandText = sqlcom;
        OdbcDataReader sqlReader1 = sqlCommand1.ExecuteReader();
        string table = "";

        while (sqlReader1.Read())
        {
            table = table + sqlReader1.GetString(0) + " " + sqlReader1.GetString(1) + "
" + sqlReader1.GetString(2) + " " + sqlReader1.GetString(3) + Environ-
ment.NewLine;
        }

        sqlReader1.Close();
        dbMySQL.Close();
        if (table != "")
        {
            return table;
        }
        else
        {
            return "Ничего не найдено";
        }
    }
    catch (OdbcException ex)
    {
        textBox1.Text = textBox1.Text + ex.ToString() + Environment.NewLine;
        return "Ошибка базы данных";
    }
    finally
    {
        if (dbMySQL != null) dbMySQL.Close();
    }
}

```

Для базы данных со структурой, представленной на рис. 28, sqlcom может принимать следующие значения:

– select book_name, aname, pub_name, price from book b,
publisher p, author a where b.pub_id=p.pub_id and
b.author_id=a.author_id;

– select book_name, aname, pub_name, price from book b, publisher p, author a where b.pub_id=p.pub_id and b.author_id=a.author_id and book_name = '<название>';

– select book_name, aname, pub_name, price from book b, publisher p, author a where b.pub_id=p.pub_id and b.author_id=a.author_id and price > <нижняя граница диапазона> and price < <верхняя граница>.

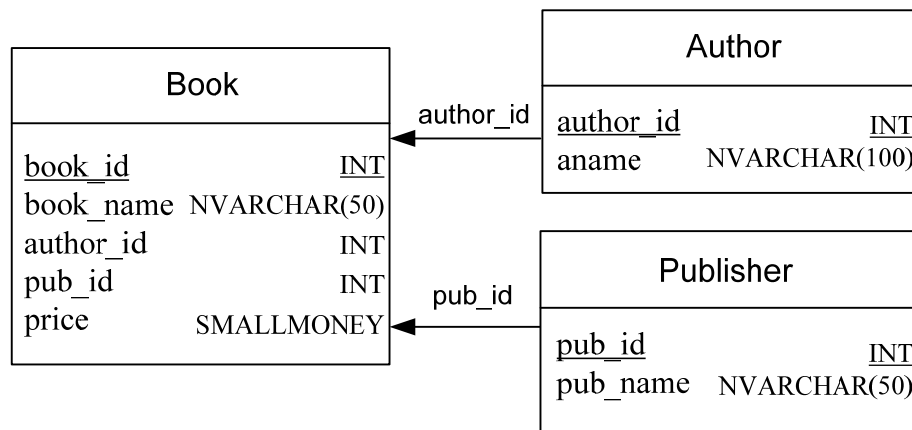


Рис. 28. Физическая ER-диаграмма

В результате изучения тем части 3 должен быть получен практический опыт проектирования приложений на языке С# для доступа к распределенным базам данных с использованием сокетов TCP/IP и библиотек ODBC.

Часть 4

РАЗРАБОТКА КЛИЕНТ-СЕРВЕРНОГО ПРИЛОЖЕНИЯ С ПАРАЛЛЕЛЬНЫМ ДОСТУПОМ К БАЗЕ ДАННЫХ И РАСПРЕДЕЛЕННОГО ПРИЛОЖЕНИЯ ДЛЯ РАБОТЫ С НЕСКОЛЬКИМИ БАЗАМИ ДАННЫХ НА ЯЗЫКАХ C# И JAVA

Т е м а 4.1. Разработка программы-сервера с параллельным доступом к базе данных в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP

Цель работы над темой – изучение способов параллельной обработки информации. Разработка программы-сервера в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP на языке C#.

Задание: разработать программу сервера, которая должна выполнять:

- прием соединения (соединение с клиентом);
- прием SQL-запроса;
- обработку и отправку результата SQL-запроса;
- закрывать соединение с клиентом.

Основные сведения

В данном проекте взаимодействие между клиентом и сервером основано на использовании сокетов. С помощью сокетов на стороне клиента и сервера устанавливается соединение и у них появляется возможность отправлять и принимать сообщения. Сервер создает слушающий сокет, привязывает его к какому-нибудь порту и начинает принимать соединения, в качестве которых выступают клиенты.

Этот слушающий сокет находится в цикле ожидания и «просыпается», когда на соquete появляется новое соединение, т.е. подключается новый клиент.

Клиент тоже создает сокет и в качестве параметров подключения указывается IP-адрес сервера и порт. Отправляя запрос на соединение, клиент пытается установить связь и начать работать с сервером.

После установления соединения клиент начинает отправлять сообщение со сформированным SQL-запросом. Сервер в свою очередь ожидает сообщения от клиента, получает его, обрабатывает и отправляет полученный результат обратно.

На рис. 29 представлена взаимосвязь клиента и сервера.

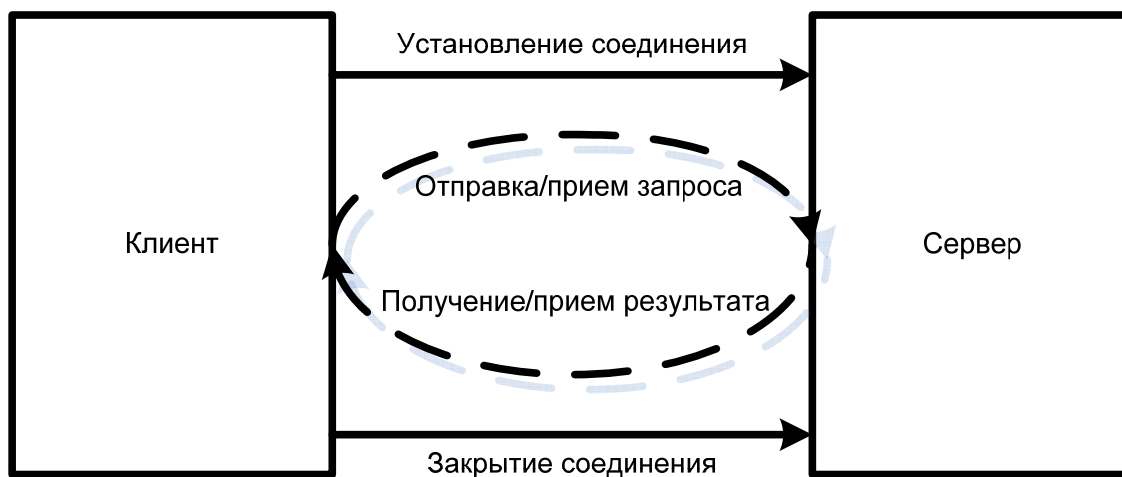


Рис. 29. Схема взаимодействия клиента с сервером

Ниже представлены основные функции, реализуемые в серверном приложении:

- `selectDB()` – функция, обрабатывающая нажатие кнопки, которая представляет оператору сервера диалоговое окно для выбора базы данных, с которой будут работать клиенты;

- `connectDB()` – функция, обрабатывающая нажатие кнопки, которая выполняет подключение к выбранной базе данных;

- `addClient()` – функция добавления нового клиента к работе с базой данных. Прием соединений происходит в цикле. Как только устанавливается новое соединение, сервер создает две нити на прием запроса и передачу результата клиенту;

- `recvMsg()` – функция, отвечающая за прием сообщений от клиента;

- `sendMsg()` – функция, отвечающая за отправку сообщений клиенту;

- `execQuery()` – функция, выполняющая обработку запросов, принятых от клиентов, как на модификацию данных, так и на выборку;

- `setText()` – функция, предназначенная для добавления записей в элемент интерфейса "listBox1" из разных нитей приложения;

– closeClientSocket() – функция, отвечающая за разрыв соединения с клиентом и закрытия его;

– formClose() – функция, которая срабатывает при закрытии формы, разрывая соединение с базой данных, сокет сервера и освобождая все используемые ресурсы.

Ниже, на рис. 30, представлена схема программы сервера.

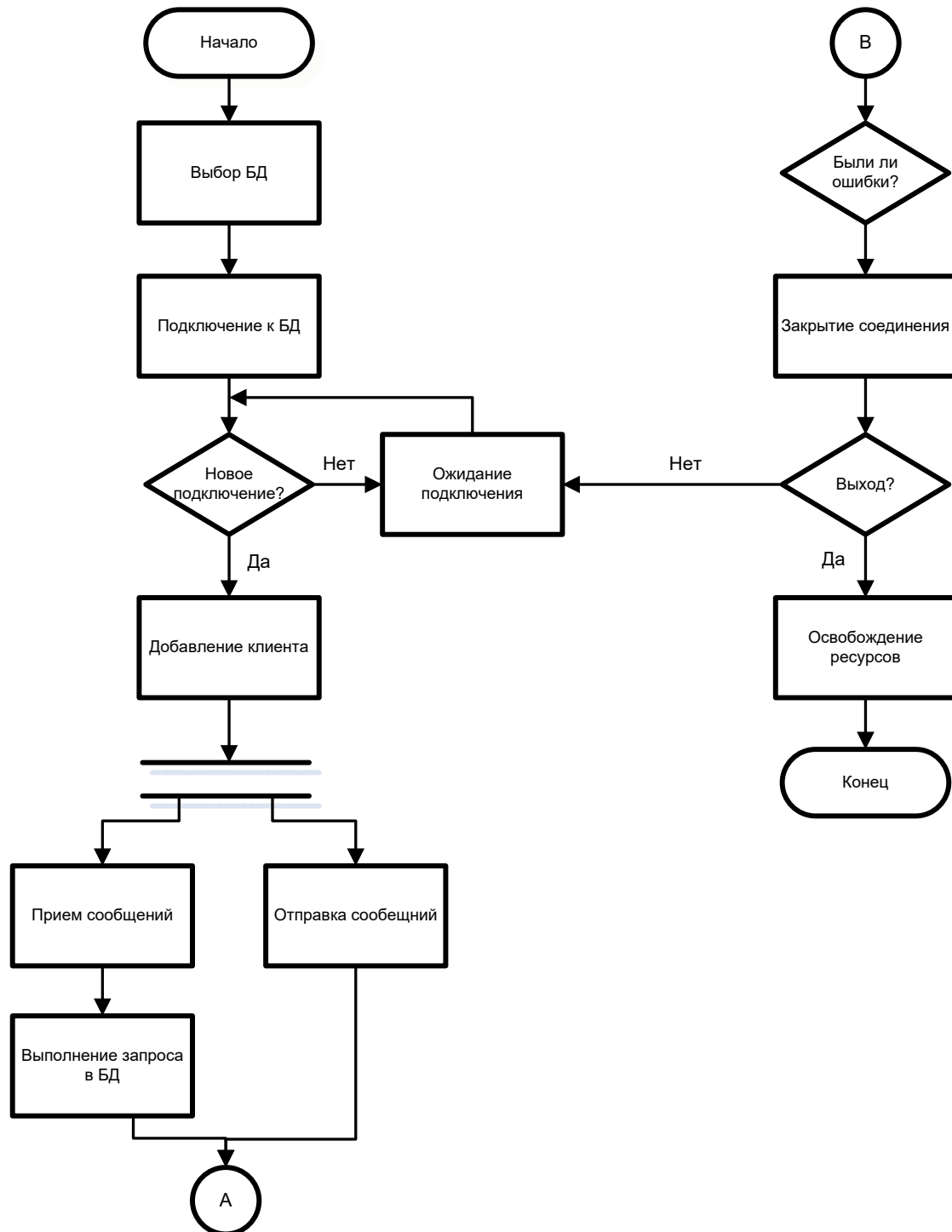


Рис. 30. Схема программы-сервера

Пример кода приложения-сервера на языке C# приведен ниже.

```
namespace server
{
    public partial class Form1 : Form
    {
        struct infoClient
        {
            public Socket sck;
            public int numberFlag;
            public infoClient(Socket sck, int flag)
            {
                this.sck = sck;
                this.numberFlag = flag;
            }
        }
        string query = "";
        string pathDB = "";
        string connDB = "";
        string excepMsg = "";
        int port = 0;
        int countSize = 0;
        OdbcConnections sampleConnect = null;
        OdbcCommand command = null;
        OdbcDataReader reader = null;
        Socket sck = null;
        Thread thread = null;
        object locker = new object();
        bool connectedDB = false;
        bool goFlag = false;
        bool sockAdd = false;
        bool sqlFlag = false;
        bool modifyFlag = false;
        List<Thread[]> threads = new List<Thread[]>();
        List<bool> chekingFlag = new List<bool>();

        delegate void setTextCallback(string text1);

        public Form1()
        {
            InitializeComponent();
            button1.Enabled = false;
            listBox1.Enabled = true;
        }

        //Функция записи текста в textBox
        private void setText(string text1)
        {
            if (this.listBox1.InvokeRequired)

```

```

    {
        setTextCallback d = new setTextCallback(setText);
        //Вызов делегата
        this.Invoke(d, new object[] { text1 });
    }
    else
    {
        this.listBox1.Items.Add(text1);
    }
}

//Функция закрытия клиентского сокета
private void closeClientSocket(Socket clientSck)
{
    clientSck.Shutdown(SocketShutdown.Both);
    clientSck.Close();
}

//Функция подключения к БД
private void connectDB(object sender, EventArgs e)
{
    //Проверка на имеющееся подключение
    if (!connectedDB)
    {
        if (textBox2.Text != "" && textBox1.Text != "")
        {
            try
            {
                port = Convert.ToInt32(textBox2.Text);
            }
            catch (FormatException ex)
            {
                //Вывод сообщений о неверном формате введенных данных
                MessageBox.Show("Неверный формат ввода порта соединения", "Ошибка",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }
            //Создание строки подключения к БД
            //Подключение
            //Открытие соединения
            connDB = "Driver={Microsoft Access Driver (*.mdb)};Dbq=" + pathDB;
            sampleConnect = new OdbcConnection(connDB);
            sampleConnect.Open();
            setText("Соединение с БД установлено");
            button1.Text = "Отключиться от БД";
            connectedDB = true;
        }
    }
    else
    {
        //Вывод сообщения о невведенных значениях или отсутствия целевой БД
    }
}

```

```

MessageBox.Show("Введите порт соединения и выберите БД", "Внимание",
MessageBoxButtons.OK, MessageBoxIcon.Warning);
return;
    }

//Создание нити и начало ее работы
thread = new Thread(addClient);
thread.Start();
    }
else
    {
//Закрытие соединения с БД
setText("Соединение с БД разорвано");
button1.Text = "Соединение с БД";
connectedDB = false;
        button2.Enabled = true;

sampleConnect.Close();
    }
}

//Функция выбора БД
private void selectDB(object sender, EventArgs e)
{
//Вызов диалогового окна и выбор файла
OpenFileDialog openFileDialog1 = new OpenFileDialog();
    openFileDialog1.Filter = "mdb files (*.mdb)|*.mdb";
if(openFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        textBox1.Text = openFileDialog1.FileName;
pathDB = openFileDialog1.FileName;
    }
    button1.Enabled = true;
    button2.Enabled = false;
}

//Функция добавление клиента
private void addClient()
{
    {
if (!sockAdd)
    {
//Создание сокета
sck = new Socket(AddressFamily.InterNetwork, SocketType.Stream, Proto-
colType.Tcp);
IPEndPoint endPoint = new IPEndPoint(0, port);

sck.Bind(endPoint);
sck.Listen(10);

sockAdd = true;
    }
}
}

```

```

//Цикл соединения с новыми клиентами
//Добавление информации о них
//Создание нитей на прием и отправку запроса и результатов
while (true)
{
    SocketclientSck = sck.Accept();
    setText("Установлена связь с новым клиентом");

    chekingFlag.Add(goFlag);

    infoClientinfoClient = new infoClient(clientSck, countSize);
    countSize++;

    Thread[] newClient = new Thread[2];

    newClient[0] = new Thread(recvMsg);
    newClient[1] = new Thread(sendMsg);

    threads.Add(newClient);

    threads[threads.Count - 1][0].Start(infoClient);
    threads[threads.Count - 1][1].Start(infoClient);
}

//Функция приема сообщения
private void recvMsg(Object info)
{
    infoClientiClient = (infoClient)info;
    Socket sClient = iClient.sck;
    intnumberFlag = iClient.numberFlag;
    while (true)
    {
        byte[] msgBuffer = new byte[sClient.ReceiveBufferSize];
        intrecv = 0;
        try
        {
            //Прием запроса от клиента
            recv = sClient.Receive(msgBuffer, 0, msgBuffer.Length, SocketFlags.None);
        }
        catch (SocketExceptionsEx)
        {
            setText("Клиент отключился");
            closeClientSocket(sClient);
            return;
        }

        if (recv == 0)
        {
            //Закрытие сокета, если приходит неверный запрос от клиента

```

```

setText("Принят неверный запрос или клиент отключился");
closeClientSocket(sClient);
return;
    }

byte[] newBuf = new byte[recv];
//Декодирование принятого запроса

for (int i = 0; i <recv; i++)
    {
newBuf[i] = msgBuffer[i];
    }

query = Encoding.Default.GetString(newBuf);

setText("Получен запрос от клиента:");
setText(query);

//Вызов функции исполнения запроса
execQuery(numberFlag);
    }
}

//Функция вызова запроса
private void execQuery(intnumFlag)
{
//Проверка на нахождении по крайней мере одной нити в условии
//Возможно только атомарное выполнение запроса
lock (locker)
{
//Создание строки запроса
command = new OdbcCommand(query, sampleConnect);
query = query.ToUpper();
try
{
if (query.StartsWith("INSERT") || query.StartsWith("UPDATE") || query.StartsWith("DELETE"))
{
//Исполнение запроса
command.ExecuteNonQuery();
modifyFlag = true;
}
else
{
//Чтение результата в reader
reader = command.ExecuteReader();
}
}
catch (OdbcException ex)
{

```

```

    excepMsg = oEx.Message;
    sqlFlag = true;
    }
    chekingFlag[numFlag] = true;
    }
    }

//Функция отправки результата
private void sendMsg(Object info)
{
    infoClientiClient = (infoClient)info;
    Socket sClient = iClient.sck;
    intnumberFlag = iClient.numberFlag;
    while (true)
    {
        //Проверка на выполнение запроса
        if (chekingFlag[numberFlag])
        {
            while (true)
            {
                stringmsgString = "";
                byte[] msgBuffer;

                //Формирование результата запроса
                if (!sqlFlag&& !modifyFlag)
                {
                    while (reader.Read())
                    {
                        //Кодирование запроса
                        //Через + идут слова
                        //Через ~ идут строки
                        for (int i = 0; i < reader.FieldCount; i++)
                        {
                            msgString += reader[i].ToString();
                            msgString += "+";
                        }
                        msgString += "~";
                    }
                    msgBuffer = Encoding.Default.GetBytes(msgString);
                }
                //Проверка на правильность запроса SQL
                else if (sqlFlag)
                {
                    msgString = "Был отправлен неверный SQL запрос " + excepMsg;

                    sqlFlag = false;

                    msgBuffer = Encoding.Default.GetBytes(msgString);
                }
                else
                {

```

```

msgString = "Изменения в таблицу занесены";

modifyFlag = false;

msgBuffer = Encoding.Default.GetBytes(msgString);
    }
//Отправка результата запроса
sClient.Send(msgBuffer, 0, msgBuffer.Length, SocketFlags.None);

cheekingFlag[numberFlag] = false;

break;
    }
    }
    }
}
//Функция закрытия формы
private void formClose(object sender, FormClosingEventArgs e)
{
//Завершение работы нитей
for (int i = 0; i < threads.Count; i++)
{
threads[i][0].Abort();
threads[i][1].Abort();
}
try
{
thread.Abort();
//Закрытие соединения с БД и клиентами
sampleConnect.Close();
sck.Close();
}
catch (NullReferenceException nrEx)
{
//Освобождение ресурсов и закрытие формы
this.Dispose();
this.Close();
}
}
}
}

```

Т е м а 4.2. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием семейства протоколов TCP/IP

Цель работы над темой – разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» с использованием протокола TCP/IP на языке C#.

Задание: разработать программу клиента, которая должна:

- запрашивать у пользователя адрес программы-сервера;
- устанавливать соединение с сервером;
- передавать на сервер данные;
- принимать ответ от сервера и выводить его на экран;
- закрывать соединение с сервером.

Основные сведения

Весь основной функционал программы представляет собой описание всех основных компонентов, располагающихся на интерфейсе, и реализацию основных функций, которые обрабатывают те или иные манипуляции пользователя.

Ниже представлены основные функции, реализуемые в клиентском приложении:

- `connectServer()` – функция обрабатывающая кнопку, которая выполняет подключение к серверу, данные которого пользователь указывает в текстовых полях формы приложения;

- `connectDB()` – функция обрабатывающая кнопку, которая выполняет подключение к выбранной базе данных;

- `recvMsg()` – функция, отвечающая за прием сообщений от сервера;

- `sendMsg()` – функция, отвечающая за отправку сообщений серверу;

- `sqlToServerFromServer()` – функция, отвечающая за создание двух нитей на прием результатов от сервера и отправку запросов серверу;

- `setText()` – функция, предназначенная для добавления записей в элемент интерфейса "listBox1" из разных нитей приложения;

- `setColumnCount()` – функция, предназначенная для установки количества столбцов в компонент "dataGridView1" из разных нитей приложения;

- `addRows()` – функция, с помощью которой заносится результат ответа сервера в компонент "dataGridView1" из разных нитей приложения;

- `disconnectServer()` – функция, отвечающая за разрыв соединения с сервером и закрытие его.

Блок-схема программы-клиента представлена на рис. 31.

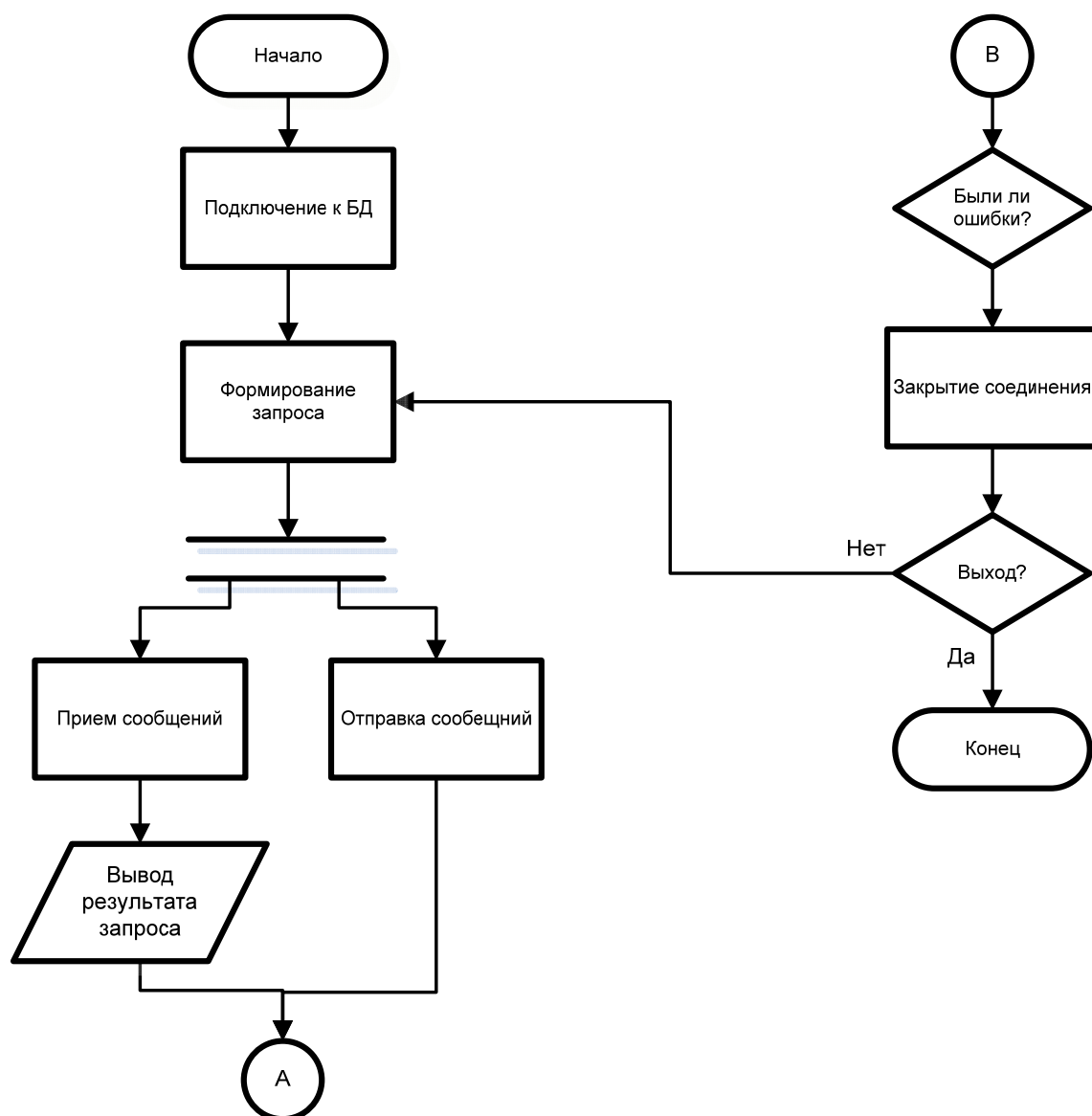


Рис. 31. Схема программы клиента

Пример кода приложения-клиента на языке C# приведен ниже.

```

namespace client
{
    public partial class Form1 : Form
    {
        //Объявление переменных
        Boolean isPressed = false;
        int port = 0;
        IPAddress Addr = null;
        Socket sck = null;
        Thread recvThread = null;
        Thread sendThread = null;

        //Объявление делегатов
        private delegate void setTextCallback(string text1);
    }
}

```

```

private delegate void setColumnDataGridViewCallback(int columnCount);
private delegate void addRowsDataGridViewCallback(string[] str);

public Form1()
{
    //Инициализация компонентов
    InitializeComponent();
    button2.Enabled = false;
}

//Функция подключения к серверу
private void connectServer(object sender, EventArgs e)
{
    //Проверка на нажатие кнопки
    if (isPressed)
    {
        isPressed = false;
        //Отключение от сервера, если нажата
        disconnectServer();
        button1.Text = "Подключиться";
        setText("Связь с сервером разорвана");
        return;
    }
    //Проверка корректности введенных данных
    if (textBox2.Text != "" && textBox1.Text != "")
    {
        try
        {
            {
                if (!IPAddress.TryParse(textBox1.Text, out iAddr))
                {
                    MessageBox.Show("Неверный формат ввода IP-адреса", "Ошибка",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
                    return;
                }
            }
            port = Convert.ToInt32(textBox2.Text);
        }
        catch (FormatException ex)
        {
            //Вывод сообщений об ошибке
            MessageBox.Show("Неверный формат ввода порта соединения", "Ошибка",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
        }
    }
    //Создание сокета
    sck = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
    ProtocolType.Tcp);
    IPEndPoint endPoint = new IPEndPoint(iAddr, port);
    try
    {
        {
            //Подключение к серверу
            sck.Connect(endPoint);
        }
    }
}

```

```

    }
catch
{
    //Вывод сообщений об ошибке
    MessageBox.Show("Невозможно подключиться к серверу", "Ошибка",
    MessageBoxButtons.OK, MessageBoxIcon.Error);
    return;
}
//Установка флага и соответствующих значений на кнопках
isPressed = true;
    button1.Text = "Отключиться";
    button2.Enabled = true;
    setText("Связь с сервером установлена");
}
else
{
    //Вывод сообщений недопустимых значениях, либо их отсутствии
    MessageBox.Show("Введите IP-адрес и порт соединения", "Внимание",
    MessageBoxButtons.OK, MessageBoxIcon.Warning);
    return;
}
}

//Функция отправки сообщения
private void sendMsg()
{
    //Проверка ввода SQL запроса
    if (textBox3.Text != "")
    {
        stringmsgString = textBox3.Text;

        //Перевод в байты и отправление в сокет
        byte[] msgBuffer = Encoding.Default.GetBytes(msgString);
        sck.Send(msgBuffer, 0, msgBuffer.Length, SocketFlags.None);
    }
    else
    {
        //Вывод сообщения об отсутствии значений
        MessageBox.Show("Введите SQL запрос", "Внимание", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        return;
    }
}

//Функция приема сообщения
private void recvMsg()
{
    byte[] msgBuffer = new byte[1024];
    intrecv = 0;
    try
    {

```

```

//Чтение из сокета
recv = sock.Receive(msgBuffer, 0, msgBuffer.Length, SocketFlags.None);
    }
    catch (SocketExceptionsEx)
    {
        //Вывод сообщения об ошибке в сокете и отключение от сервера
        MessageBox.Show("Невозможнопринятьрезультат, неполадкииссервером",
            "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
        isPressed = false;
        disconnectServer();
        button1.Text = "Подключиться";
        setText("Связь с сервером разорвана");
        return;
    }

//Проверка корректности соединения с сервером
if (recv == 0)
{
    setText("Ответ на запрос был некорректен или сервер отключился");
    return;
}

stringrecvData = Encoding.Default.GetString(msgBuffer);

if (recvData.StartsWith("Был отправлен неверный SQL запрос"))
{
    setText(recvData);
}
else
{
    {
        intcolumnCount = 0;

        //Декодирование сообщения
        for (int i = 0; i <recvData.Length; i++)
        {
            if (recvData[i] == '~')
                break;
            if (recvData[i] == '+')
                columnCount++;
        }

        string[] newString;

        newString = recvData.Split('+', '~');

        string[] wordInStr = new string[columnCount];

        //Добавление нужного количества колонок в таблицу
        dataGridView1.Invoke(new setColumnDataGridViewCallback(setColumnCount),
            new object[] { columnCount });
    }
}

```

```

int iter = 0;

//Форматирование результата запроса в вид,пригодный для записи в таблицу
while (newString.Length - 1 != iter)
{
    for (int i = 0; i < columnCount; i++)
    {
        if (newString[iter] == "")
            iter++;
        if (newString.Length - 1 == iter)
            break;
        wordInStr[i] = newString[iter];
        iter++;
    }
    //Добавление результатов запроса в таблицу
    if (newString.Length - 1 != iter)
        dataGridView1.Invoke(new addRowsDataGridViewCallback(addRows),
            new object[] { wordInStr });
    }
    setText("Результат запроса представлен ниже");
}
}

//Функция создания новых нитей для отправки и чтения запросов
и их результатов
private void sqlToServerFromServer(object sender, EventArgs e)
{
    dataGridView1.Rows.Clear();

    sendThread = new Thread(sendMsg);
    sendThread.Start();

    recvThread = new Thread(recvMsg);
    recvThread.Start();
}

//Функция отключения от сервера и закрытия сокета
private void disconnectServer()
{
    {
        sck.Shutdown(SocketShutdown.Both);
        sck.Close();
    }
}

//Функция добавления элементов в ListBox
private void setText(string text1)
{
    {
        if (this.listBox1.InvokeRequired)
        {
            setTextCallback d = new setTextCallback(setText);
            //Вызов делегата на добавление в listBox

```

```

this.Invoke(d, new object[] { text1 });
    }
else
    {
this.listBox1.Items.Add(text1);
    }
}

//Функция установки количества колонок в таблице
private void setColumnCount(intcolumnCount)
{
    dataGridView1.ColumnCount = columnCount;
}

//Функция добавления определенного количества строк
private void addRows(string[] str)
{
    dataGridView1.Rows.Add(str);
}
}
}
}

```

Т е м а 4.3. Разработка распределенного серверного приложения с доступом к нескольким базам данных с использованием семейства протоколов TCP/IP

Цель работы над темой – необходимо разработать серверное приложение, работающее с несколькими базами данных в сетевой вычислительной среде.

Задание: разработать программу сервера, которая должна:

- подключаться к нескольким базам данных;
- ожидать подключения клиентов;
- принимать запрос от клиента;
- отправлять ответ клиенту;
- закрывать соединение (отключаться от сервера).

Основные сведения

Взаимодействие клиента и сервера происходит на основе механизма сокетов. После запуска программы «сервер» создается сокет, и пользователь привязывает его к порту, который ждет подключения клиентских сокетов. Клиент со своей стороны указывает IP-адрес и порт сервера и подключается к нему. После успешного подключения клиент может посылать запросы серверу и ждать

ответа, и, как только сервер принял запрос, он ищет нужную информацию в подключенных к нему баз данных и отправляет ее клиенту. После отключения клиента соединение закрывается.

На рис. 32 показана архитектура клиент-серверного приложения.

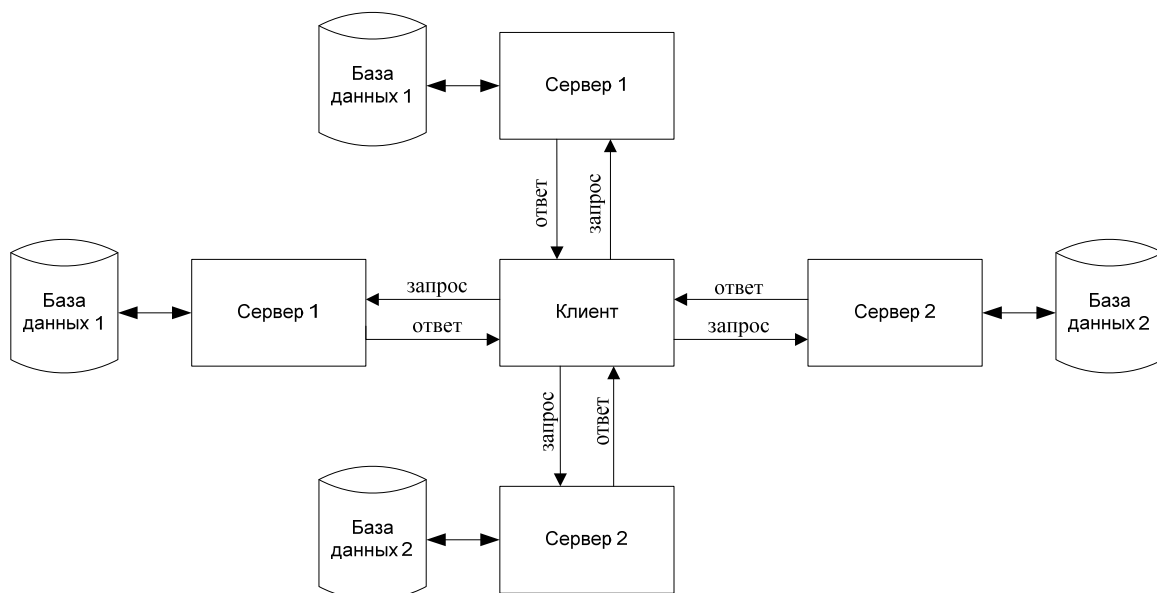


Рис. 32. Архитектура мультибазового приложения «клиент-сервер»

При реализации TCP-сервера были использованы функции класса `System.Net.Sockets.Socket`:

- 1) `Bind()` – связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение;
- 2) `Listen()` – помещает сокет в режим прослушивания (ожидания). Этот метод предназначен только для серверных приложений;
- 3) `Accept()` – создает новый сокет для обработки входящего запроса на соединение. Блокирует поток вызывающей программы до поступления соединения;
- 4) `Receive()` – получает данные от соединенного сокета;
- 5) `Send()` – отправляет данные соединенному сокету;
- 6) `Shutdown()` – запрещает операции отправки и получения данных на сокете;
- 7) `Close()` – закрывает сокет.

Сервер работает следующим образом:

- 1) после того, как пользователь нажимает на кнопку «Запуск сервера», программа устанавливает для сокета локальную конечную точку, создает сокет и связывает сокет с локальной конечной точкой;

2) сокет помещается в режим прослушивания. В этом режиме он будет ожидать входящие попытки соединения;

3) при поступлении входящего запроса на соединение сервер дает согласие на соединение;

4) установив соединение, сервер получает данные от клиента, обращается к базе данных и читает из нее требуемую клиентом информацию;

5) сервер отправляет ответ клиенту и завершает соединение;

6) если не дана команда «завершить работу», сервер возвращается к прослушиванию.

Ниже представлен код программы сервера:

```
namespace Server
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        ~Form1()
        {
            button3_Click("", null);
        }
        Thread rec;
        bool stopReceive = false;
        Socket handler = null;
        Socket sListener = null;
        private delegate void SetTextBoxDelegate(string message);
        private void SetTextBox(string message)
        {
            if (textBox1.InvokeRequired)
            {
                textBox1.Invoke(new SetTextBoxDelegate(AddText), message);
            }
            else
            {
                textBox1.Text = message;
            }
        }
        void msg_OnSendChatMessage(string message)
        {
            AddText(message);
        }
    }
}
```

```

}
void AddText(string message)
{
    textBox1.Text = textBox1.Text + message;
}

private string GetFromDB(string sqlcom)
{
    string result = "";
    string tmp="";
    string dname = "";
    bool find = false;
    List<Thread> myAL = new List<Thread>();
    List<Counter> myCoun = new List<Counter>();
    ListViewItem b;
    for (int i = 0; i < checkedListBox1.CheckedItems.Count;i++ )
    {
        b = (ListViewItem)checkedListBox1.CheckedItems[i];
        Counter counter = new Counter(sqlcom, b.Name);
        Thread myThread = new Thread(new ThreadStart(counter.someDB));
        myThread.Start();
        myAL.Add(myThread);
        myCoun.Add(counter);
    }

    for (int i = 0; i < myAL.Count; i++)
    {
        myAL[i].Join();
    }
    for (int i = 0; i < myAL.Count; i++)
    {
        tmp = myCoun[i].getRes();
        if (tmp != "Ничего не найдено" && tmp != "Ошибка базы данных")
        {
            result += tmp;
            find = true;
        }
    }
    if (find)
        return result;
    else
        return "Ничего не найдено";
}

private void TcpServer()
{

```

```

//Устанавливает для сокета локальную конечную точку
IPHostEntry ipHost = Dns.Resolve("localhost");
IPAddress ipAddr = ipHost.AddressList[0];
IPEndPoint ipEndPoint=null;
try
{
    ipEndPoint = new IPEndPoint(ipAddr, Convert.ToInt32(textBox3.Text));
}
catch (Exception ex)
{
    textBox1.Text = ex.ToString();
}

//Создаем сокет TCP/IP

sListener = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
sListener.Bind(ipEndPoint);
sListener.Listen(10);
SetTextBox("Сервер работает." + Environment.NewLine);

if (checkedListBox1.Items.Count == 0)
{
    SetTextBox("Добавьте базу данных." + Environment.NewLine);
}

//Назначаем сокет локальной конечной точке и "слушаем" входящие
сокеты
while (true)
{
    try
    {

        //Программа приостанавливается, ожидая входящее соединение
        handler = sListener.Accept();

        SetTextBox("Клиент " + handler.LocalEndPoint + " Подключен." + Envi-
ronment.NewLine);
        //Начинаем "слушать" соединение
        while (true)

```

```

{

//Мы дождались клиента, пытающегося с нами соединиться
string data = null;
int bytesRec;

byte[] bytes = new byte[4096];
bytesRec = handler.Receive(bytes);
if (textBox2.Text != "")
{
    byte[] psw = Encoding.UTF8.GetBytes(textBox2.Text);
    for (int i = 0; i < (bytes.Length); i++)
    {
        bytes[i] = (byte)(bytes[i] - psw[i % psw.Length]);
    }
}
data = Encoding.UTF8.GetString(bytes, 0, bytesRec);

//Выводим данные на экран
SetTextBox("Получен запрос от клиента!" + Environment.NewLine);

if (data == "")
{
    SetTextBox("Клиент " + handler.LocalEndPoint + " Отключен." + Environment.NewLine);
    handler.Shutdown(SocketShutdown.Both);
    handler.Close();
    break;
}
if (data != "clientout")
{
    string theReply = GetFromDB(data);
    byte[] msg = Encoding.UTF8.GetBytes(theReply);
    if (textBox2.Text != "")
    {
        byte[] psw = Encoding.UTF8.GetBytes(textBox2.Text);
        for (int i = 0; i < (msg.Length); i++)
        {
            msg[i] = (byte)(msg[i] + psw[i % psw.Length]);
        }
    }
    handler.Send(msg);
}
//handler.Shutdown(SocketShutdown.Both);
//handler.Close();
if (stopReceive == true) break;

```

```

    }
}
catch (Exception ex)
{
    break;
    //SetTextBox(ex.ToString() + Environment.NewLine);
}
}
sListener.Close();
}

private void button2_Click(object sender, EventArgs e)
{
    stopReceive = false;

    rec = new Thread(TcpServer);
    rec.IsBackground = true;
    rec.Start();
    button2.Enabled = false;
    button3.Enabled = true;
    button4.Enabled = true;
}

private void button3_Click(object sender, EventArgs e)
{
    // Останавливаем цикл в дополнительном потоке
    try
    {
        stopReceive = true;
        rec.DisableComObjectEagerCleanup();
        if (handler != null)
        {
            if (handler.Connected)
            {
                handler.Shutdown(SocketShutdown.Both);
                handler.Close();
            }
        }
        if (sListener != null)
        {
            sListener.Close();
        }
        button2.Enabled = true;
        button3.Enabled = false;
        button4.Enabled = false;
        SetTextBox("Сервер остановлен." + Environment.NewLine);
    }
}

```

```

        catch (Exception ex)
        {
            SetTextBox(ex.ToString() + Environment.NewLine);
        }
    }

    private void button4_Click(object sender, EventArgs e)
    {
        OpenFileDialog openFileDialog1 = new OpenFileDialog();
        openFileDialog1.Filter = "mdb files (*.mdb)|*.mdb";
        if (checkedListBox1.Items.Count > 0)
            checkedListBox1.Items.Clear();
        openFileDialog1.Multiselect = true;
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
            for (int i = 0; i < openFileDialog1.FileNames.LongLength; i++)
            {
                ListViewItem lvi = new ListViewItem(openFileDialog1.SafeFileNames[i]);
                lvi.Name = openFileDialog1.FileNames[i];
                lvi.Text = "База данных " + i + " : "+lvi.Text;
                checkedListBox1.DisplayMember = "Text";
                checkedListBox1.Items.Add(lvi, CheckState.Checked);
                SetTextBox("База данных успешно добавлена." + Environment.NewLine
+ "Сервер работает." + Environment.NewLine);
            }
        }
    }

    private void Form1_FormClosed(object sender, FormClosedEventArgs e)
    {
        button3_Click(" ", null);
    }
}

public class Counter
{
    private string sqlcom;
    private string name;
    private string result;
    public Counter(string _sqlcom, string _name)
    {
        this.sqlcom = _sqlcom;
        this.name = _name;
    }
    public string getRes()
    {
        return result;
    }
}

```

```

public void someDB()
{
    string table = "";

    string strConnect = "Driver={Microsoft Access Driver (*.mdb)};Dbq=" +
this.name;

    OdbcConnection dbMySQL = new OdbcConnection(strConnect);

    try
    {
        dbMySQL.Open();
        OdbcCommand sqlCommand1 = dbMySQL.CreateCommand();
        sqlCommand1.CommandText = sqlcom;
        OdbcDataReader sqlReader1 = sqlCommand1.ExecuteReader();

        while (sqlReader1.Read())
        {
            table = table + sqlReader1.GetString(0) + " " + sqlReader1.GetString(1) + "
" + sqlReader1.GetString(2) + " " + sqlReader1.GetString(3) + Environ-
ment.NewLine;
        }

        sqlReader1.Close();
        dbMySQL.Close();

        if (table != "")
        {
            this.result = table;
            //return table;
        }
        else
        {
            this.result= "Ничего не найдено";
        }
    }
    catch (OdbcException ex)
    {
        this.result= "Ошибка базы данных";
    }
    finally
    {
        if (dbMySQL != null) dbMySQL.Close();
    }
}
}
}

```

Т е м а 4.4. Разработка распределенного клиентского приложения с доступом к нескольким базам данных с использованием семейства протоколов TCP/IP

Цель работы над темой – разработка клиентского приложения, взаимодействующего с сервером в сетевой вычислительной среде.

Задание: разработать программу клиента, которая должна:

- подключаться к серверу;
- отправлять запрос серверу;
- принимать ответ от сервера;
- выводить результат на экран;
- закрывать соединение (отключаться от сервера).

Основные сведения

Разработка приложения началась с проектирования пользовательского интерфейса. Клиентское приложение реализовано с помощью интерфейса Windows Forms, настройки которого хранятся в файле Form1.Designer.cs.

При реализации TCP-клиента были использованы функции класса System.Net.Sockets.Socket:

- 1) Connect() – устанавливает соединение с удаленным хостом;
- 2) Send() – отправляет данные соединенному сокету;
- 3) Receive() – получает данные от соединенного сокета;
- 4) Shutdown() – запрещает операции отправки и получения данных на сокете;
- 5) Close() – закрывает сокет.

TCP-клиент работает следующим образом:

- 1) после того, как пользователь нажимает на кнопку «Подключиться», программа создает сокет и устанавливает соединение с удаленной конечной точкой;
- 2) после нажатия кнопки «Отправить запрос» программа обрабатывает данные, введенные в текстовые поля, и формирует sql-запрос;
- 3) клиент отправляет сообщение серверу по установленному ранее соединению;
- 4) получив данные от сервера, клиент закрывает соединение;
- 5) программа обрабатывает полученные данные и выводит их в текстовое поле.

Ниже, на рис. 33, представлена схема программы клиента.

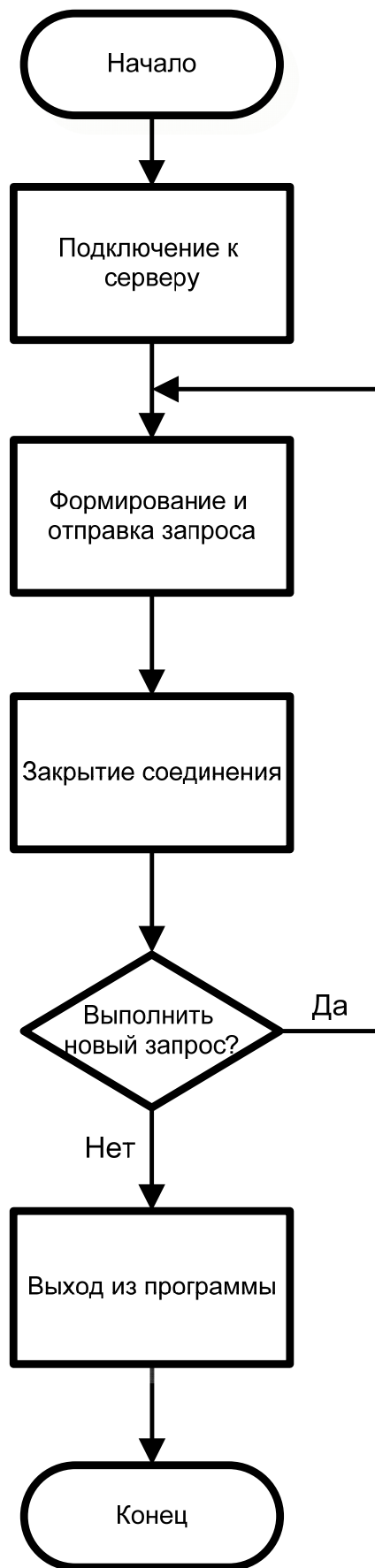


Рис. 33. Схема программы клиента

Ниже представлен код программы клиента:

```
namespace Client
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

        }
        private IPEndPoint ipHost;
        private IPAddress ipAddr;
        private IPEndPoint ipEndPoint;
        private Socket sSender;
        bool start = true;
        bool beg = false;
        private void SendMsg(string sqlcom)
        {
            byte[] msg = Encoding.UTF8.GetBytes(sqlcom);
            if (textBox9.Text != "")
            {
                byte[] psw = Encoding.UTF8.GetBytes(textBox9.Text);
                for (int i = 0; i < (msg.Length); i++)
                {
                    msg[i] = (byte)(msg[i] + psw[i % psw.Length]);
                }
            }

            try
            {
                //Отправляем данные через сокет
                int bytesSent = sSender.Send(msg);
            }
            catch (Exception ex)
            {
                textBox1.Text = ex.ToString();
            }
        }
        int count = 0;
        private void ReseiveData()
        {
            //Буфер для входящих данных
            byte[] bytes = new byte[65500];
            try
            {
                //Получаем ответ от удаленного устройства
```

```

int bytesRec = sSender.Receive(bytes);
if (textBox9.Text != "")
{
    byte[] psw = Encoding.UTF8.GetBytes(textBox9.Text);
    for (int i = 0; i < (bytes.Length); i++)
    {
        bytes[i] = (byte)(bytes[i] - psw[i % psw.Length]);
    }
}
textBox6.Text += sSender.RemoteEndPoint.ToString() + Environ-
ment.NewLine;

    textBox6.Text += Encoding.UTF8.GetString(bytes, 0, bytesRec) + Environ-
ment.NewLine;
}
catch (Exception ex)
{
    textBox1.Text = ex.ToString();
}
}
private void CloseServ(object sender, FormClosedEventArgs e)
{
    if (sSender != null)
    {
        if(sSender.Connected)
            sSender.Shutdown(SocketShutdown.Both);
        sSender.Close();
    }
    start = true;
    button1.Text = "Подключиться";
    textBox1.Text = "Соединение закрыто" + Environment.NewLine;
    beg = true;
}
private void button1_Click(object sender, EventArgs e)
{
    if (dataGridView1.RowCount == 1)
        return;

    if (start)
    {
        //Буфер для входящих данных
        byte[] bytes = new byte[4096];

        //Соединяемся с удаленным устройством
        try
        {

```

```

        //ipHost = Dns.Resolve(textBox7.Text);
        ipHost =
Dns.Resolve(dataGridView1.Rows[count].Cells[0].Value.ToString());
        ipAddr = ipHost.AddressList[0];
        ipEndPoint = new IPEndPoint(ipAddr, Con-
vert.ToInt32(dataGridView1.Rows[count].Cells[1].Value.ToString()));

        sSender = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
        beg = false;
        //Соединяем сокет с удаленной конечной точкой
        sSender.Connect(ipEndPoint);

        textBox1.Text = "Соединение установлено" + Environment.NewLine;
        button1.Text = "Отключиться";
        start = false;
        button2.Enabled = true;
    }
    catch (Exception ex)
    {
        textBox1.Text = ex.ToString();
        if (!sSender.Connected)
        {
            textBox1.Text = "Сервер не найден";
            button2.Enabled = false;
        }
    }
}
else
{
    button2.Enabled = false;
    CloseServ("", null);
}
}
private void GetData()
{
    string sqlcom = "select book_name, aname, pub_name, price from book b,
publisher p, author a where b.pub_id=p.pub_id and b.author_id=a.author_id";
    if (textBox2.Text != "")
    {
        sqlcom += " and book_name= " + textBox2.Text + "";
    }
    if (textBox3.Text != "")
    {
        sqlcom += " and aname= " + textBox3.Text + "";
    }
}

```

```

        if (textBox4.Text != "")
        {
            sqlcom += " and price>" + textBox4.Text;
        }
        if (textBox5.Text != "")
        {
            sqlcom += " and price<" + textBox5.Text;
        }

        SendMsg(sqlcom);
        ReseiveData();
    }
    private void button2_Click(object sender, EventArgs e)
    {
        if (textBox2.Text == "" && textBox3.Text == "" && textBox4.Text == "" && text-
Box5.Text == "")
        {
            textBox1.Text = "Заполните любое поле";
            return;
        }
        bool nconnect = false;
        textBox6.Text = "";
        if (beg)
            button1_Click("", null);
        GetData();
        if(1 != dataGridView1.RowCount - 1 && beg == false)
            CloseServ("", null);
        count++;
        while (count != dataGridView1.RowCount - 1)
        {
            button1_Click("", null);
            if (sSender.Connected == false)
            {
                dataGridView1.Rows.RemoveAt(count);
                count--;
            }
            GetData();
            count++;
            CloseServ("", null);
        }
        count = 0;
    }
}
}
}

```

Т е м а 4.5. Разработка программы-сервера с параллельным доступом к базе данных в архитектуре взаимодействия «клиент-сервер» на языке Java с использованием семейства протоколов TCP/IP

Программа «сервер» должна быть в любой момент готова к отправке результата запроса всем клиентам, которые прислали запрос. Для этого необходимо:

- вызвать функцию `int socket (int domain, int type, int protocol)`, которая создает гнездо заданного типа и с указанным протоколом. Domain определяет формат адреса. Аргумент type задает тип гнезда. Аргумент protocol указывает конкретный протокол, который следует использовать с данным гнездом;

- вызвать функцию `int bind (int sid, struct sockaddr* addr_p, int len)`, которая задает номер своего порта и адреса клиентов, которым разрешено к нему подключиться. Гнездо обозначается аргументом sid, значение которого, возвращенное функцией socket, представляет собой дескриптор гнезда. Аргумент addr_p указывает на структуру, содержащую имя (адрес), которое должно быть присвоено гнезду. Аргумент len задает размер структуры, на которую указывает аргумент addr_p;

- используя функции `int send (int sid, const char* buf, int len, int flag)` и `int accept (int sid, char* buf, int len, int flag)` организовать передачу данных клиентам;

- Закрывать сокет с помощью функции `int closesocket (int sid)`, которая уничтожает гнездо с дескриптором sid.

При запуске программы «сервер» (вызов метода `jButton1ActionPerformed()` – главного класса окна) производится попытка инициализировать класс `ServerSocket`. При создании сервера запускается поток прослушивания порта указанного в приложении. При подключении клиента запускается новый поток, в котором происходит обработка строки и отправка ее клиенту.

Ниже, на рис. 34, представлена схема программы сервера.

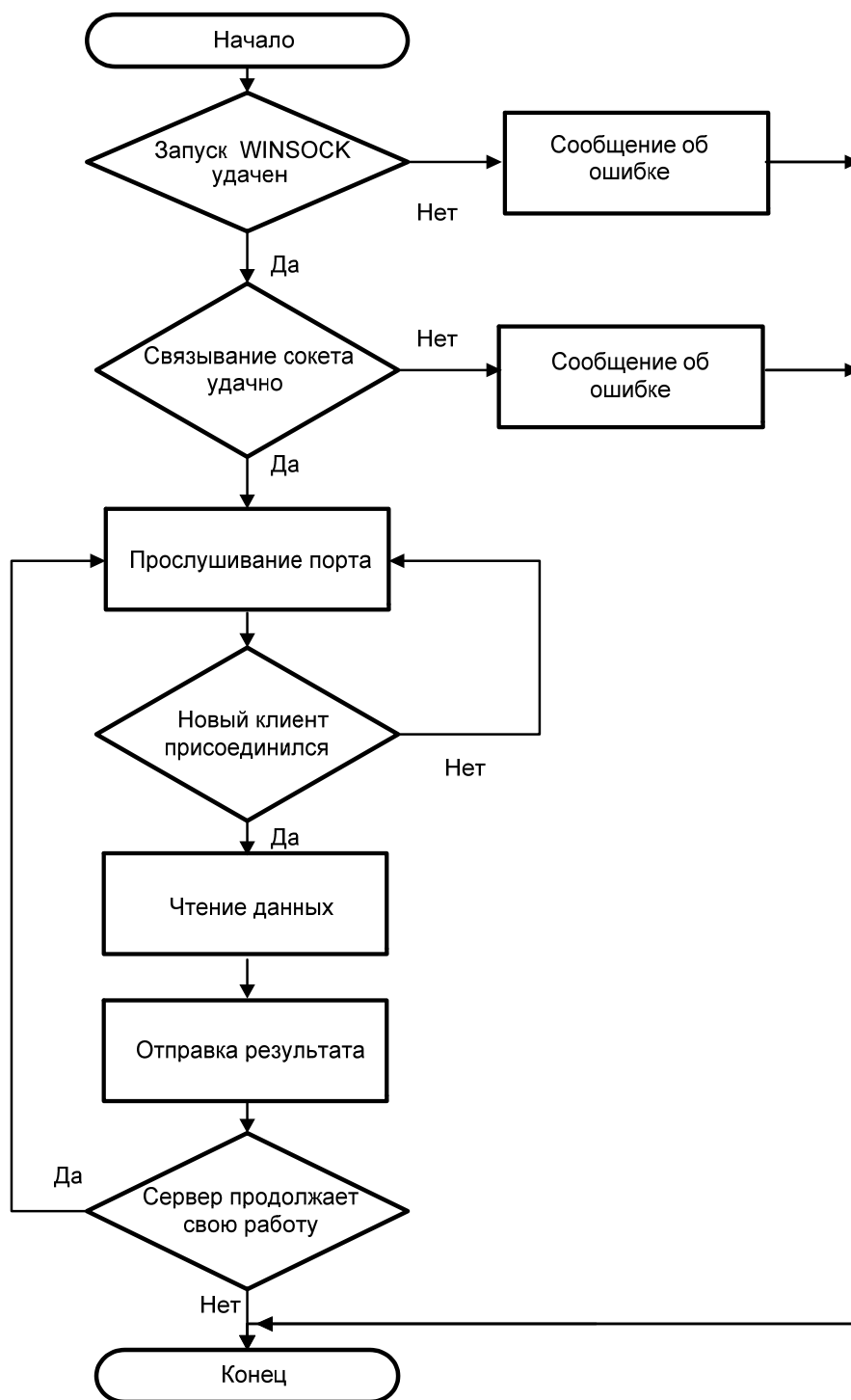


Рис. 34. Схема программы-сервера

Пример кода приложения-сервера на языке Java приведен ниже.

```

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    dateBase = jComboBox1.getSelectedItem().toString();
    password = jTextField2.getText();

```

```

try {
    ss = new ServerSocket(Integer.parseInt(jTextField1.getText()));
} catch (IOException ex) {
    Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
}
outputText("Server started \nWaiting client....");
jTextField2.setEnabled(false);
jButton1.setVisible(false);
jTextField1.setEnabled(false);
jComboBox1.setEnabled(false);
main = new Main();
mymain = new Thread(main);
mymain.start();

}

Socket WaitCl(Socket client, int i)
{
    try {
        client=ss.accept();
        is = client.getInputStream();
        os = client.getOutputStream();
        String s = Input(i,is);
        if (s.equals(password)){
            outputText("Client "+i+" connection");
            s = "true";
            os.write(s.getBytes("UTF-8"));
            clientbool[i] = true;
            return client;
        }
        if (s.equals("guest"))
        {
            outputText("Client "+i+" connection");
            s = "guest";
            os.write(s.getBytes("UTF-8"));
            clientbool[i] = true;
            return client;
        }
        s = "false";
        os.write(s.getBytes("UTF-8"));
        outputText("Error Connection");
        os.write(0);
        is.close();
        os.close();
        client.close();
        return null;
    } catch (IOException ex) {

```

```

        Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
    }
    return null;
}

String Input(int i,InputStream ist )
{
    String str;

    int lenght;
    try {
        lenght = ist.read(bKbdInput);
        str = new String(bKbdInput,"UTF-8");
        String s = str.substring(0, lenght);
        return s;
    } catch (IOException ex) {

        client[i]=null;
        kol--;
        outputText("Client "+i +" disconnect");
        return null;
    }

}

}

void outputText(String str)
{
    jTextArea1.append(str+"\n");
}
/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
    //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
    /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
     * For details see
     * http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
     */
    try {
        for (javax.swing.UIManager.LookAndFeelInfo info :
            javax.swing.UIManager.getInstalledLookAndFeels()) {

```

```

        if ("Nimbus".equals(info.getName())) {
            javax.swing.UIManager.setLookAndFeel(info.getClassName());
            break;
        }
    }
} catch (ClassNotFoundException ex) {
    ja-
va.util.logging.Logger.getLogger(Server.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    } catch (InstantiationException ex) {
        ja-
va.util.logging.Logger.getLogger(Server.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    } catch (IllegalAccessException ex) {
        ja-
va.util.logging.Logger.getLogger(Server.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    } catch (javax.swing.UnsupportedLookAndFeelException ex) {
        ja-
va.util.logging.Logger.getLogger(Server.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }
}
//</editor-fold>

```

```

/* Create and display the form */
java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        new Server().setVisible(true);
    }
});
}

```

```

// Variables declaration - do not modify
private javax.swing.JButton jButton1;
private javax.swing.JComboBox jComboBox1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JLabel jLabel5;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JTextArea jTextArea1;
private javax.swing.JTextField jTextField1;
private javax.swing.JTextField jTextField2;
// End of variables declaration

```

```

class Main
implements Runnable

```

```

{
    public void run()
    {
        int i=0;

        while(true)
        {

            if (client[i]==null)
            {
                client[i] = WaitCl(client[i], i);
                if (clientbool[i]==true) {
                    try {
                        mess = new Messeger(client[i],i);
                    } catch (IOException ex) {
                        Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
                    }
                    mes = new Thread(mess);
                    mes.start();
                    kol++;

                }
            }
            else
            {
                i++;
                if (i==5)
                {
                    i=0;
                }
            }
        }
    }
}

class Messeger
implements Runnable           //(содержащее метод run())
{Socket s;
Connection con;
InputStream ist;
OutputStream ost;
int i;

private Messeger(Socket client, int k) throws IOException {
    s = client;
    i=k;
}

```

```

    ist = s.getInputStream();
    ost = s.getOutputStream();
}

public void run()          //Этот метод будет выполняться в побочном потоке
{

    while (true){

        try {
            Class.forName("org.sqlite.JDBC");
            con = DriverManager.getConnection("jdbc:sqlite:"+dateBase+".db");

            java.sql.Statement st = con.createStatement();
            ist = s.getInputStream();
            ost = s.getOutputStream();
            String str = Input(i,is);
            if (str==null)
            {
                client[i]=null;
                return;
            }
            outputText("Client " + i + " "+str.replaceAll("parfume", dateBase));
            if(str.matches("Update .*"))
            {
                st.execute(str);
                String strk = "good";
                ost.write( strk.getBytes("UTF-8"));
            }
            else{

                ResultSet res = st.executeQuery(str.replaceAll("parfume", dateBase));
                if (str.matches(".*where id.*"))
                {
                    String strk = "";
                    while (res.next())
                    {
                        strk = strk + res.getString(1) + ">" + res.getString(2) + ">" +
res.getString(3) + ">" + res.getString(4) + ">" + res.getString(5) + ">";

                    }
                    strk = strk+ "";
                    ost.write( strk.getBytes("UTF-8"));
                }
                else
                {

```

```

        String strk = "";
        while (res.next())
        {
            strk = strk + res.getString(1) + ">" + res.getString(2) + ">" +
res.getString(3) + ">" + res.getString(4) + ">" + res.getString(5) + ">";

        }
        strk = strk+ "";
        ost.write( strk.getBytes("UTF-8"));

    }
}
} catch (SQLException ex) {
    Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
} catch (ClassNotFoundException ex) {
    Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
}
}
}
}
}

```

Т е м а 4.6. Разработка программы-клиента в архитектуре взаимодействия «клиент-сервер» на языке Java с использованием семейства протоколов TCP/IP

Программа «клиент» отводит для сетевого взаимодействия всего одну нить, тогда как серверу в общем случае требуется создавать новую нить для каждого нового соединения. Клиентское приложение, как и серверное, вызывает функции socket, send, accept и closesocket.

При запуске приложения клиента производится попытка инициализировать класс Socket. При нажатии кнопки «Соединиться» производится попытка подключиться к сокету сервера с указанным IP адресом и портом. При удачном соединении запускается поток, в котором серверу отправляется имя клиента, а затем начинается прослушивание сокета.

Ниже, на рис. 35, представлена схема программы клиента.

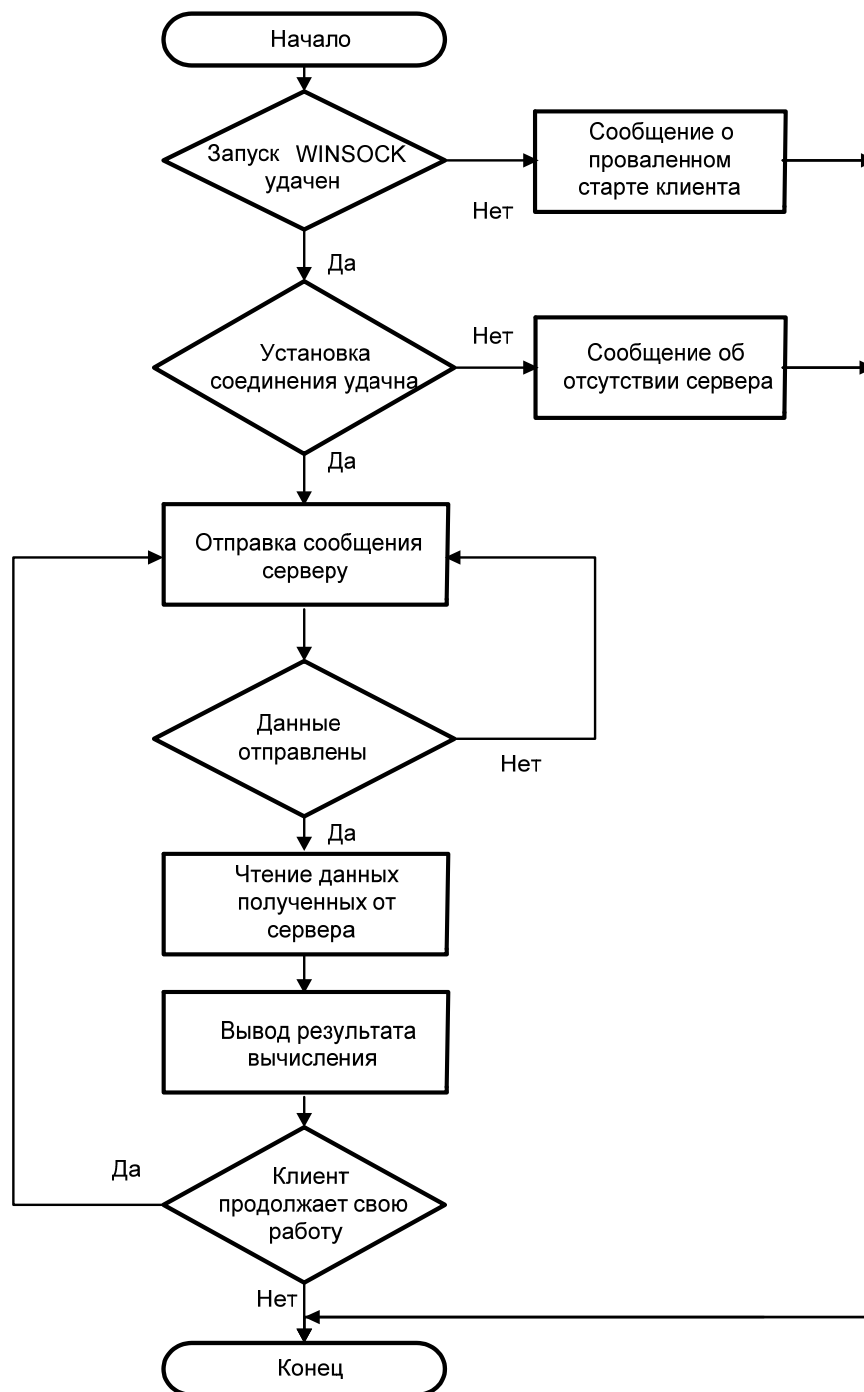


Рис. 35. Схема программы клиента

Пример кода приложения-клиента на языке Java приведен ниже.

```

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        String str;
        jTable1= jTable2;
        s = new Socket(jTextField1.getText(),Integer.parseInt(jTextField2.getText()));
    }
}
  
```

```

is = s.getInputStream();
os = s.getOutputStream();
str = jTextField3.getText();
if (str.equals(""))
{
    str="guest";
}
os.write(str.getBytes("UTF-8"));
int lenght = is.read(bKbdInput);
str = new String(bKbdInput,"UTF-8");
String st = str.substring(0, lenght);
if (!st.equals("true"))
{
    if (st.equals("guest"))
    {
        jLabel5.setText("Подключено");
        jButton1.setVisible(false);
        jTextField1.setVisible(false);
        jTextField2.setVisible(false);
        jTextField3.setVisible(false);
        jLabel1.setVisible(false);
        jLabel2.setVisible(false);
        jLabel3.setVisible(false);
        jLabel4.setVisible(false);
        jButton2.setVisible(false);
        jButton2ActionPerformed(evt);

        return;
    }
    jLabel5.setText("Ошибка подключения");
    is.close();
    os.close();
    s.close();
    return;
} else {
}
jLabel5.setText("Подключено");
jButton1.setVisible(false);
jTextField1.setVisible(false);
jTextField2.setVisible(false);
jTextField3.setVisible(false);
jLabel1.setVisible(false);
jLabel2.setVisible(false);
jLabel3.setVisible(false);
jLabel4.setVisible(false);
jButton2.setVisible(false);
jLabel12.setVisible(true);
jButton3.setVisible(true);
jTextField8.setVisible(true);

jButton2ActionPerformed(evt);

```

```

    } catch (UnknownHostException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);

    } catch (IOException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    }
}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        String n;
        String str = null;

        str = "Select * from perfume";

        os.write(str.getBytes("UTF-8"));
        str = Input();
        Matcher matcher = pattern.matcher(str);
        int i=0;
        int j=0;

        jTable2 = new JTable(myModel);
        Vector newRow = new Vector();
        myModel.addRow(newRow);
        while(matcher.find())
        {

            if(i==5)
            {
                newRow = new Vector();
                myModel.addRow(newRow);
                i=0;
                j++;
            }

            str = matcher.group();
            jTable2.setValueAt(str, j, i);
            i++;

        }
        myModel=myModel1;
    } catch (UnsupportedEncodingException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

```

private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    if (jTextField8.getText().isEmpty() && jTextField8.getText().isEmpty())
    {
        jButton2ActionPerformed(evt);
    }
    else
    {
        String str = jTextField8.getText();
        try {
            os.write(str.getBytes("UTF-8"));
            str = Input();
            if (str.equals("good"))
            {
                jButton2ActionPerformed(evt);
            }
            else
            {
                Matcher matcher = pattern.matcher(str);
                int i=0;
                int j=0;

                jTable2 = new JTable(myModel);
                int l = myModel.getRowCount()-1;
                while (l!=0)
                {
                    myModel.removeRow(l);
                    l--;
                }
                Vector newRow = new Vector();
                myModel.addRow(newRow);
                while(matcher.find())
                {

                    if(i==5)
                    {
                        newRow = new Vector();
                        myModel.addRow(newRow);
                        i=0;
                        j++;
                    }

                    str = matcher.group();
                    jTable2.setValueAt(str, j, i);
                    i++;
                }
            }
        } catch (UnsupportedEncodingException ex) {
            Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

```

    } catch (IOException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    }

}

}

private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    if (jTextField9.getText().isEmpty() && jTextField9.getText().isEmpty())
    {
        jButton5ActionPerformed(evt);
    }
    else
    {
        String str = jTextField9.getText();
        try {
            os2.write(str.getBytes("UTF-8"));
            str = Input2();
            if (str.equals("good"))
            {
                jButton5ActionPerformed(evt);
            }
            else
            {
                Matcher matcher = pattern.matcher(str);
                int i=0;
                int j=0;

                jTable4 = new JTable(myModel2);
                int l = myModel2.getRowCount()-1;
                while (l!=0)
                {
                    myModel2.removeRow(l);
                    l--;
                }
                Vector newRow = new Vector();
                myModel2.addRow(newRow);
                while(matcher.find())
                {

                    if(i==5)
                    {
                        newRow = new Vector();
                        myModel2.addRow(newRow);
                        i=0;
                        j++;
                    }

                }
                str = matcher.group();
                jTable4.setValueAt(str, j, i);
                i++;
            }
        }
    }
}

```

```

    }
    myModel2=myModel3;
    }
} catch (UnsupportedEncodingException ex) {
    Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
} catch (IOException ex) {
    Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
}
}
}

```

```

private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        String n;
        String str = null;

        str = "Select * from perfume";

        os2.write(str.getBytes("UTF-8"));
        str = Input2();
        Matcher matcher = pattern.matcher(str);
        int i=0;
        int j=0;

        jTable4 = new JTable(myModel2);
        Vector newRow = new Vector();
        myModel2.addRow(newRow);
        while(matcher.find())
        {

            if(i==5)
            {
                newRow = new Vector();
                myModel2.addRow(newRow);
                i=0;
                j++;

            }
            str = matcher.group();
            jTable4.setValueAt(str, j, i);
            i++;

        }
        myModel2=myModel3;
    } catch (UnsupportedEncodingException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

```

    }
}

private void jButton6ActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        String str;
        s2 = new Socket(jTextField4.getText(), Integer.parseInt(jTextField5.getText()));
        is2 = s2.getInputStream();
        os2 = s2.getOutputStream();
        str = jTextField6.getText();
        if (str.equals(""))
        {
            str="guest";
        }
        os2.write(str.getBytes("UTF-8"));
        int lenght = is2.read(bKbdInput);
        str = new String(bKbdInput,"UTF-8");
        String st = str.substring(0, lenght);
        if (!st.equals("true"))
        {
            if (st.equals("guest"))
            {
                jLabel7.setText("Подключено");
                jButton6.setVisible(false);
                jTextField4.setVisible(false);
                jTextField5.setVisible(false);
                jTextField6.setVisible(false);
                jLabel6.setVisible(false);
                jLabel8.setVisible(false);
                jLabel9.setVisible(false);
                jLabel10.setVisible(false);
                jButton5.setVisible(false);
                jButton5ActionPerformed(evt);
                return;
            }
            jLabel7.setText("Ошибка подключения");
            is2.close();
            os2.close();
            s2.close();
            return;
        } else {
        }
        jLabel7.setText("Подключено");
        jButton6.setVisible(false);
        jTextField4.setVisible(false);
        jTextField5.setVisible(false);
        jTextField6.setVisible(false);
        jLabel6.setVisible(false);
        jLabel8.setVisible(false);
        jLabel9.setVisible(false);
        jLabel10.setVisible(false);
    }
}

```

```

        jButton5.setVisible(false);
        jLabel13.setVisible(true);
        jButton4.setVisible(true);
        jTextField9.setVisible(true);

        jButton5ActionPerformed(evt);

    } catch (UnknownHostException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);

    } catch (IOException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
    }
}

String Input() throws IOException
{
    String str;
    int lenght = is.read(bKbdInput);
    str = new String(bKbdInput,"UTF-8");
    String s = str.substring(0, lenght);
    return s;
}
String Input2() throws IOException
{
    String str;
    int lenght = is2.read(bKbdInput);
    str = new String(bKbdInput,"UTF-8");
    String s = str.substring(0, lenght);
    return s;
}
/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
    //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
    /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look
    and feel.
    * For details see
    http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
    */
    try {
        for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.getInstalledLookAndFeels()) {
            if ("Nimbus".equals(info.getName())) {
                javax.swing.UIManager.setLookAndFeel(info.getClassName());
                break;
            }
        }
    }
    catch (ClassNotFoundException ex) {
        java.util.logging.Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
    catch (InstantiationException ex) {
        java.util.logging.Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
    catch (IllegalAccessException ex) {
        java.util.logging.Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

```

    }
    }
    } catch (ClassNotFoundException ex) {
        ja-
        va.util.logging.Logger.getLogger(Client.class.getName()).log(java.util.logging.Level
        I.SEVERE, null, ex);
    } catch (InstantiationException ex) {
        ja-
        va.util.logging.Logger.getLogger(Client.class.getName()).log(java.util.logging.Level
        I.SEVERE, null, ex);
    } catch (IllegalAccessException ex) {
        ja-
        va.util.logging.Logger.getLogger(Client.class.getName()).log(java.util.logging.Level
        I.SEVERE, null, ex);
    } catch (javax.swing.UnsupportedLookAndFeelException ex) {
        ja-
        va.util.logging.Logger.getLogger(Client.class.getName()).log(java.util.logging.Level
        I.SEVERE, null, ex);
    }
    //</editor-fold>

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new Client().setVisible(true);
        }
    });
}

```

В результате изучения тем части 4 должен быть получен практический опыт проектирования приложений на языке C# и Java, которые могут работать с несколькими базами данных, а также навыки в проектировании параллельных приложений доступа к базам данных, нескольких клиентов с использованием сокетов TCP/IP.

Часть 5

ЗАДАНИЯ НА САМОСТОЯТЕЛЬНУЮ РАБОТУ И КОНТРОЛЬНЫЕ ВОПРОСЫ

5.1. Общие требования

Сетевое приложение должно быть выполнено в архитектуре «клиент-сервер» или ее модификации с многопоточным сервером с организацией взаимодействия с базой данных на объектно ориентированном языке (по согласованию с преподавателем).

Уровни архитектуры: серверное приложение может быть реализовано в виде консольного приложения или GUI-приложения. Настройки сервера должны меняться без изменения исходного кода (аргументы командной строки, config-файлы и т.д.). Клиентское приложение: оконное приложение с использованием стандартных библиотек пользовательского интерфейса.

Должны быть использованы следующие элементы технологии:

- 1) разработка и использование собственной иерархии, расширение базовых классов;
- 2) реализация не менее двух паттернов проектирования на свой выбор (по желанию);
- 3) использование сокрытия данных (инкапсуляция), перегрузка методов, переопределение методов, сериализация, абстрактные типы данных (интерфейсы, абстрактные классы), статические методы, обработка исключительных ситуаций;
- 4) бизнес-логика системы должна быть реализована только на серверной части приложения;
- 5) на сервере должна быть предусмотрена возможность параллельной обработки запросов;
- 6) доступ к данным в СУБД должен осуществляться через драйвер, предоставляемый производителем СУБД или через использование специальных технологий;
- 7) обязательные требования к функционалу системы:
 - а) в разрабатываемом приложении необходимо обеспечить добавление, редактирование и удаление записей из базы данных, сохранение табличных результатов в файле (создание текстового отчета);

- b) предусмотреть возможность сохранять информацию в любой момент на сервере и загружать ранее сохраненные данные;
- c) предусмотреть механизм авторизации пользователей (роли Администратор и Пользователь);
- d) база данных (не менее трех связанных таблиц) должна быть приведена к третьей нормальной форме.

5.2. Примерный перечень заданий на самостоятельную работу

1. Журнал учета рабочего времени сотрудников формы.
2. Учет движения товаров на складе.
3. Учет компьютерной техники на предприятии.
4. Ежедневник. Учет мероприятий сотрудников фирмы.
5. Ведение картотеки ресурсов домашней библиотеки.
6. Каталог заменяемых деталей автомобиля.
7. Учет транспортных расходов предприятия.
8. Учет договоров на оказание услуг.
9. Расчет дополнительной заработной платы работников предприятия.
10. Контроль движения входящей корреспонденции.
11. Автоматизированное рабочее место менеджера торговой фирмы.
12. Учет фондов домашней дискотеки.
13. Учет оргтехники на предприятии.
14. Разработка системы экстраполяции показателей объемов продаж на предприятии.
15. Разработка автоматизированной системы оценки финансовых рисков компании.
16. Разработка системы расчета коэффициентов финансовой устойчивости (показателей структуры капитала).
17. Разработка системы учета предоставления правовой информации на предприятии.
18. Разработка системы прогнозирования объемов продаж на основе трендовых моделей.
19. Разработка системы учета затрат и калькуляции себестоимости с использованием нормативных затрат.
20. Разработка системы для расчета эффективных процентных ставок по кредитам.

21. Разработка системы оценки деятельности субъекта хозяйствования на основе показателей рентабельности.
22. Разработка системы оценки уровня качества продукции.
23. Разработка системы прогнозирования продаж на основе мультипликативной модели.
24. Разработка системы оценки стоимости компании.
25. Разработка системы моделирования вероятности банкротства организаций.
26. Разработка системы оценки финансовых рисков активов и пассивов бухгалтерского баланса предприятия.
27. Разработка системы оценки эффективности инвестиционного проекта.
28. Разработка системы анализа структуры капитала и долгосрочной платежеспособности.
29. Разработка системы автоматизации оценки объектов недвижимости.
30. Система анализа инвестиционной привлекательности организации-эмитента.
31. Разработка автоматизированной системы анализа затрат.
32. Разработка системы прогнозирования финансовой устойчивости предприятия.
33. Разработка системы прогнозирования платежеспособности предприятия.

5.3. Контрольные вопросы по сетевому программированию

1. Раскройте понятие «процесс», приведите его основные свойства, структуру и описание функции запуска.
2. Дайте понятие «поток», опишите содержимое контекста потока, функции создания потока и ее параметров.
3. Опишите предназначение и использование переменных окружения.
4. Как определить дескриптор процесса?
5. Как определить дескриптор первичного потока?
6. Как определить текущий каталог?
7. Дайте определение каналам (pipes).
8. В чем отличие именованных каналов от анонимных?
9. Перечислите основные функции для работы с каналами.
10. Дайте понятие «мапирование» файлов.

11. Опишите общую схему использования мапирования файлов для организации обмена данными между процессами.
12. Перечислите основные функции мапирования файлов.
13. Для чего нужна синхронизация потоков?
14. Укажите достоинства и недостатки синхронизации потоков в пользовательском режиме?
15. Для чего нужны функции WaitForSingleObject, WaitForMultipleObject?
16. Укажите особенности использования объектов «событие» для организации синхронизации потоков.
17. Проведите сравнительный анализ критических секций и мьютексов.
18. Опишите алгоритм работы семафоров для учета ресурсов.
19. Раскройте понятие «протокол», укажите основное назначение протоколов IP, TCP, UDP.
20. Дайте определение понятию «сокет».
21. Перечислите основные функции для работы с сокетами.
22. Укажите последовательность вызовов функций для организации сокета на стороне клиента.
23. Укажите последовательность вызовов функций для организации сокета на стороне сервера.
24. Опишите основную схему организации сервера, основанного на сокетах.
25. Опишите основную схему организации клиента, основанного на сокетах.
26. Раскройте протокол HTTP.
27. Опишите структуру WWW сервера.
28. Опишите функции и методы модуля socket.
29. Опишите функции и методы модуля threading.
30. Запишите общую структуру сервера, реализованного на сокетах.
31. Запишите организацию основного цикла сервера.
32. Запишите основные механизмы обмена данными.
33. Опишите способы организации мультипоточности.
34. Раскройте механизмы синхронизации потоков с помощью модуля threading.
35. Раскройте отличия объектов семафор от объекта события.
36. Что такое сервлет?
37. Какие еще существуют технологии, похожие на сервлеты?

- 38. Какова структура каталогов web-приложения?
- 39. Какой класс является базовым для сервлетов?
- 40. Каков жизненный цикл у сервлета?
- 41. Каким образом послать ответ клиенту?
- 42. Каковы основные цели мониторинга сетевого трафика?
- 43. Чем отличается мониторинг трафика от фильтрации?
- 44. Каковы основные функции и возможности Zabbix-агента?

5.4. Примерный перечень вопросов для тестирования

1. Протокол IP относится к уровню модели OSI...

- а) канальный;
- б) сетевой;
- в) транспортный;
- г) сеансовый.

2. Протокол TCP относится к уровню модели OSI...

- а) канальный;
- б) сетевой;
- в) транспортный;
- г) сеансовый.

3. Протокол FTP относится к уровню модели OSI...

- а) канальный;
- б) сетевой;
- в) транспортный;
- г) сеансовый.

4. Протокол Ethernet относится к уровню модели OSI...

- а) канальный;
- б) сетевой;
- в) транспортный;
- г) сеансовый.

5. Протокол HTTP относится к уровню модели OSI...

- а) канальный;
- б) сетевой;
- в) транспортный;
- г) сеансовый.

6. Дана запись чисел для компьютерной сети в виде 191.200.182.101. Это...

- а) IP-адрес;
- б) порт;
- в) сокет;
- г) доменное имя.

7. Дана запись `www.pnzgu.ru` для компьютерной сети. Это записано...

- а) IP-адрес;
- б) порт;
- в) сокет;
- г) доменное имя.

8. Дана следующая запись для компьютерной сети в виде пяти чисел, например, `[191.200.182.101.],80`. Это записано...

- а) IP-адрес;
- б) порт;
- в) сокет;
- г) доменное имя.

9. Сокет с номером порта и IP-адресом связывает функция API...

- а) `socket()`;
- б) `bind()`;
- в) `accept()`;
- г) `listen()`.

10. Функция устанавливающая очередь для запросов на соединение, – это...

- а) `socket()`;
- б) `bind()`;
- в) `accept()`;
- г) `listen()`.

11. Функция, принимающая запрос на соединение, – это...

- а) `socket()`;
- б) `bind()`;
- в) `accept()`;
- г) `listen()`.

12. Объектом синхронизации, не используемым в системном режиме, является...

- а) Критическая секция;
- б) Mutex;
- в) Event;
- г) Semaphore.

13. Интерфейс, не относящийся к программированию WWW сервера, – это...

- а) CGI;
- б) PHP;
- в) ISAPI;
- г) WINSOCK.

14. В языке Java используется интерфейс...

- а) Servlet;
- б) CGI;
- в) PHP;
- г) ISAPI.

15. Модулем для создания клиент-серверных приложений на языке Java является...

- а) Servlet;
- б) Socket;
- в) PHP;
- г) ISAPI.

СПИСОК ЛИТЕРАТУРЫ

1. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы. СПб. : Питер, 2021. 1008 с.
2. Таненбаум, Э., Уэзеролл, Д. Компьютерные сети. 5-е изд. М. : ЛитРес, 2019. 961 с.
3. Антонов А. С. Технологии параллельного программирования MPI и OpenMP. М. : Изд-во МГУ, 2012. 339 с.
4. Зинкин С. А., Джафар М. С. Развитие информационно-коммуникационных инфраструктур распределенных вычислительных систем на основе концепции «Сеть – это компьютер» // Известия Юго-Западного государственного университета. 2018. Т. 22, № 4 (79). С. 75–93.
5. Коннолли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. М. : Вильямс, 2017. 1440 с.
6. Хортон А. Visual C++ 2010. Полный курс. М. : Вильямс, 2011. 1216 с.
7. Кумар В., Кровчик Э., Лагари Н. [и др.]. .NET. Сетевое программирование. М. : Лори, 2020. 400 с.
8. Работа с базами данных на языке C#. Технология ADO .NET : учеб. пособие / сост. О. Н. Евсеева, А. Б. Шамшев. Ульяновск : Изд-во УЛГТУ, 2009. 170 с.
9. MySQL Connector/ODBC Developer Guide. URL: <https://dev.mysql.com/doc/connector-odbc/en/>
10. Шилдт Г. Java 8. Полное руководство. М. : Диалектика, 2018. 1488 с.
11. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C#. СПб. : Питер, 2014. 896 с.
12. Шмидт, Д., Хьюстон, С. Программирование сетевых приложений на C++. М. : Бином, 2009. Т. 1. 576 с.
13. Иванов В. Б. Прикладное программирование на C/C++. С нуля до мультимедийных и сетевых приложений. М. : Литрес, 2016. 241 с.
14. Прибыльская Н. М. Программирование сетевых приложений. Минск : Изд-во БНТУ, 2013. 69 с.
15. Стивенс У. Р. Феннер Б., Рудофф Э. М. UNIX. Разработка сетевых приложений. СПб. : Питер, 2007. 1040 с.
16. Шлее, М. Qt 5.10. Профессиональное программирование на C++. СПб. : БХВ-Петербург, 2018. 1074 с.

17. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5. М. : Вильямс, 2013. 1312 с.
18. Назарр, К., Рихтер, Дж. Windows via C/C++. Программирование на языке Visual C++. СПб. : Питер, 2008. 868 с.
19. Жемеров Д., Исакова С. Kotlin в действии. М. : ДМК Пресс, 2018. 402 с.
20. Хорстманн К. Java. Библиотека профессионала. Т. 1. Основы. М. : Диалектика, 2019. 866 с.
21. Хорстманн К., Корнелл Г. Java. Библиотека профессионала. Т. 2. Расширенные средства программирования. М. : Вильямс, 2014. 994 с.
22. Глейзер Д., Мадхав С. Многопользовательские игры. Разработка сетевых приложений. СПб. : Питер, 2017. 368 с.
23. Васильчиков В. В. Основы разработки сетевых Windows-приложений. Ярославль : Изд-во ЯрГУ, 2007. 212 с.
24. Васильчиков, В. В. Разработка сетевых приложений для ОС Windows (практические примеры). Изд-во ЯрГУ, 2009. 216 с.
25. Агальцов В. Базы данных : в 2 кн. Кн. 1. Локальные базы данных. М. : Форум, 2013. 352 с.
26. Агальцов В. Базы данных : в 2 кн. Кн. 2. Распределенные и удаленные базы данных. М. : Форум, 2018. 272 с.
27. Беляков А. Ю. Клиент-серверное приложение с базой данных MySQL. URL: <https://pcoding.ru/pdf/CSharpMySQL.pdf>
28. Рудалев В. Г., Пронин С. С. Клиент-серверные приложения баз данных : учеб. пособие для вузов. Воронеж : Издательско-полиграфический центр Воронежского государственного университета, 2007. 82 с.
29. Гудсон Дж., Стюард Р. Практическое руководство по доступу к данным. СПб. : БХВ-Петербург, 2013. 304 с.
30. Архитектура клиент-сервер. URL: <http://www.lib.knigi-x.ru>; <https://portal.tpu.ru/SHARED/f/FAS/study/avis/lectures/cli-se.pdf>
31. Свистунов А. Построение распределенных систем на Java. URL: <https://intuit.ru/studies/courses/633/489/info>
32. Кручинин В. В. Разработка сетевых приложений : учеб. пособие. Томск : Изд-во ТУСУР, 2013. 120 с.
33. Морозова Ю. В., Кручинин В. В. Методы и технологии разработки клиент-серверных приложений : учеб. пособие. Томск : Изд-во ТУСУР, 2018. 106 с.

Учебное издание

**Попов Константин Владимирович,
Карамышева Надежда Сергеевна,
Зинкин Сергей Александрович**

**Методы и технологии разработки
сетевых приложений для распределенных
вычислительных систем**

Редактор *В. В. Чувашова*
Технический редактор *Н. В. Иванова*
Компьютерная верстка *Н. В. Ивановой*

Подписано в печать 17.02.2022.
Формат 60×84¹/₁₆. Усл. печ. л. 10,58.
Тираж 25. Заказ № 96.

Издательство ПГУ
440026, Пенза, Красная, 40.
Тел.: (8412) 66-60-49, 66-67-77; e-mail: iic@pnzgu.ru

