Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Локтионова Оксана Геннадьевна
Должность: проректор по учебной работе
Дата подписания: 26.07.2022 10:13:58
Уникальный программный ключ:
0b817ca911e6668abb13a5d426d39e5f1c11eabbf73e943df4a4851fda56d089

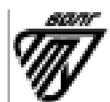
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

На правах рукописи

М. А. Кузнецов, А. Е. Андреев

Разработка распределенных кроссплатформенных программных систем

Учебно-методическое пособие



Волгоград
2021

УДК 004.4’2, 004.41, 004.42

Р е ц е н з е н т ы :

кафедра информационных систем и математического моделирования Волгоградского института управления – филиала Российской академии народного хозяйства и государственной службы при Президенте РФ, зав. кафедрой, канд. техн. наук, доцент *О. А. Астафурова*;

доцент кафедры математического анализа и теории функций, Института математики и информационных технологий ВолГУ, к.ф.-м.н. *П.Б. Жданович*

Печатается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Кузнецов М. А.

Разработка распределенных кроссплатформенных программных систем: учебно-методическое пособие / М. А. Кузнецов, А. Е. Андреев; ВолгГТУ. – Волгоград, 2021. – 80 с.

ISBN 978-5-9948-0000-0

В учебном пособии представлены основные принципы разработки программ в сетевых архитектурах на базе локальных и глобальных компьютерных систем, с использованием мобильных устройств в качестве тонких клиентов. Рассмотрены вопросы, связанные с особенностями реализации распределенных систем и облачных технологий.

Учебное пособие предназначено для магистров, обучающихся по программам магистратуры по профилю «искусственный интеллект» по направлениям 09.04.01 «Информатика и вычислительная техника», 09.04.03 «Прикладная информатика», 09.04.02 «Информационные системы и технологии». Учебное пособие выполнено в рамках реализации гранта на разработку программ бакалавриата и программ магистратуры по профилю «Искусственный интеллект», а также на повышение квалификации педагогических работников образовательных организаций высшего образования в сфере искусственного интеллекта (конкурс 2021-ИИ-01 от 10.06.2021).

Ил. 31. Табл. 3. Библиогр.: 8 назв.

ISBN 978-5-9948-0000-0

© Волгоградский государственный технический университет, 2021
© М.А. Кузнецов, А. Е. Андреев, 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
1 ТЕОРЕТИЧЕСКИЙ МАТЕРИАЛ 6
1.1 Программные архитектуры 6
1.1.1 Возникновение программных архитектур 6
1.1.2 Персональная архитектура 7
1.1.3 Терминальная архитектура 8
1.1.4 Файл-серверная архитектура 10
1.1.5 Достоинства и недостатки файл-серверных архитектур 13
1.1.6 Двухуровневые архитектуры и СУБД 14
1.1.7 Технологии клиент-серверного взаимодействия 15
1.1.8 Принципы абстрагирования запросов 17
1.1.9 Способы обработки запросов 20
1.1.10 Принципы создания двухуровневых архитектур 22
1.1.11 Преимущества и недостатки двухуровневых архитектур 23
1.1.12 Совместимость СУБД разных производителей 24
1.1.13 Функции уровней двухзвенной архитектуры 26
1.1.14 Трехуровневые и многоуровневые архитектуры 26
1.1.15 Масштабирование трехуровневых архитектур 28
1.1.16 Многоуровневые архитектуры 30
1.1.17 Пример многоуровневой архитектуры 31
1.1.18 Принципы создания распределенных архитектур 33
1.1.19 Физическая и логическая архитектура 34
1.1.20 Remote Procedure Call (RPC) 36
1.1.21 Технология ORB и прозрачность расположения сервера 40
1.1.22 Возможности многоуровневых архитектур 42
1.1.23 Скриптовые языки для управления нагрузкой уровней 45
1.2. Сетевые протоколы и сервисы 46
1.2.1 Модель OSI 46

1.2.2 Протоколы HTTP и MQTT 47
1.2.3 Вопросы и задачи для самостоятельной подготовки 50
1.3. Облачные вычисления 51
1.3.1 Классификация облачных сервисов 51
1.3.2 Облачные серсисы PaaS 55
2. МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ... 55
2.1 Разработка RESTful интерфейса 55
2.2 Реализация тонкого клиента на основе web-интерфейса 57
2.3 Реализация тонкого клиента на основе мобильной платформы 59
2.4 Требования и состав отчёта 59
3. ВЫПОЛНЕНИЕ КОНТРОЛЬНОЙ РАБОТЫ 59
3.1 Общая характеристика контрольной работы 59
3.2 Создание веб-функций в PaaS 62
3.2.1 Реализация web функции в AWS 62
3.2.2 Реализация веб-функции на Python для Yandex.Cloud 70
3.2.3 Реализация веб-функции в Azure 71
ЗАКЛЮЧЕНИЕ 77
Рекомендуемая литература 78

ВВЕДЕНИЕ

Архитектурные вопросы при создании любого программного обеспечения имеют важнейшее значение, так как разработка программной структуры будущей системы закладывает стратегические направления ее развития. Выбор программной архитектуры — это один из первых вопросов, возникающий в самом начале разработки сложного программного обеспечения. Распределенные системы имеют важную особенность, заключающуюся в реализации возможности масштабирования. Эта особенность позволяет легко адаптировать систему к разной вычислительной нагрузке. А возможность кроссплатформенности делает независимым программный код от аппаратного обеспечения. Совершенствование инструментов разработки подобных систем, а также их поддержка облачными платформами, приводят к необходимости использования новых подходов в рамках практической деятельности в разных предметных областях, уже имеющих свои стандарты создания и эксплуатации программного обеспечения. Вследствие этого возникает необходимость сочетать программные компоненты из разных предметных областей в рамках одной комплексной программной системы. Такая задача решается только с помощью разработки или же глубокой модификации программных архитектур. С учетом тенденции к миниатюризации аппаратного обеспечения вопросы включения в комплексные системы устройств на мобильных платформах и встраиваемых систем не вызывает сомнения.

1 ТЕОРЕТИЧЕСКИЙ МАТЕРИАЛ

1.1 Программные архитектуры

1.1.1 Возникновение программных архитектур

С самого зарождения программирования, а точнее с появления повторно используемого кода, существующие программные компоненты можно классифицировать на два класса: верхний и нижний. К нижнему классу относят компоненты, которые выполняют рутинную работу, напрямую не связанную со спецификой приложения. Наиболее наглядно это можно представить на примере разработки баз данных (см. рис. 1).

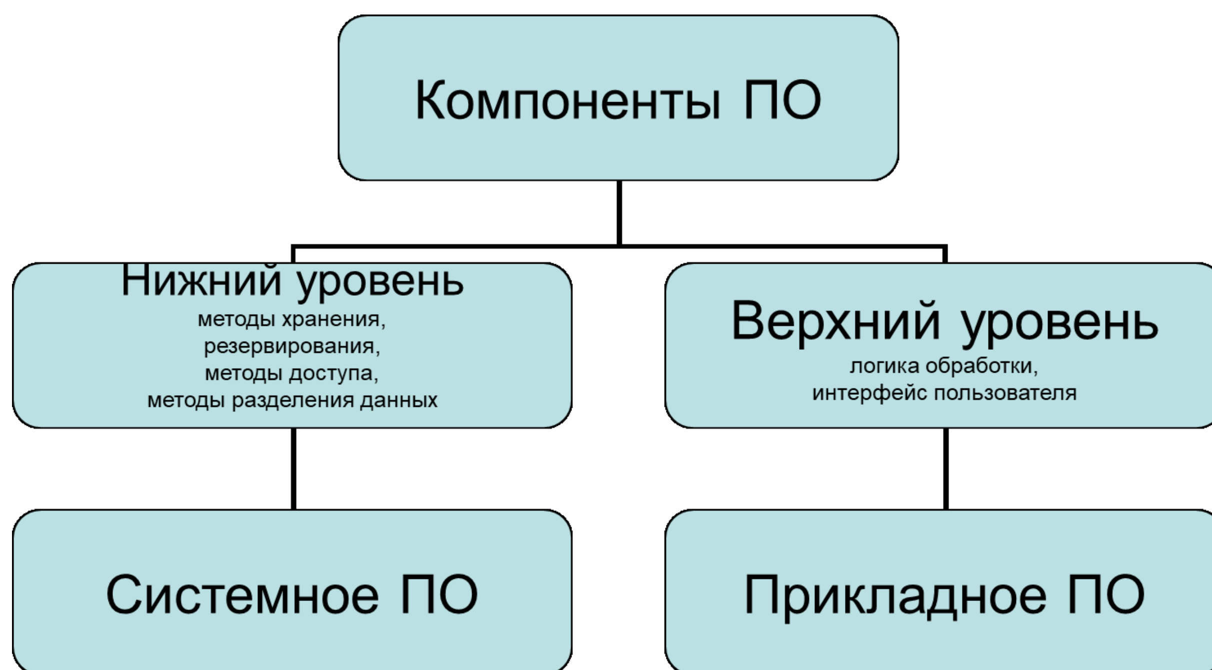


Рис. 1. Классификация компонентов программного обеспечения

К нижнему уровню при разработке баз данных относят программный код, обеспечивающий методы хранения информации, методы доступа к данным, методы разделения доступа и синхронизации работы нескольких клиентов. К нижнему уровню также относят множество алгоритмов, обеспечивающих шифрование и электронную подпись, резервирование и ре-

пликацию данных. Сюда же относят низкоуровневую обработку, связанную с фильтрацией данных и ее сортировкой. Все эти методы достаточно универсальны и могут быть использованы в разных прикладных задачах. К верхнему уровню относят код, который специфицирует программное приложение. Это интерфейс с пользователем и логика обработки информации. Фактически методы верхнего уровня для выполнения своих специфических прикладных функций вызывают методы низкого уровня.

Таким образом методы верхнего уровня являются активными, реализующими прикладные алгоритмы всей программной системы целиком. Методы же нижнего уровня пассивны. Их задача выполнить запросы методов верхнего уровня. Такое разделение концептуально формирует разбиение программного кода по принципу клиент-серверной технологии. Задача клиента реализовать верхний уровень, а сервера – нижний.

Верхний уровень решает прикладные специфические задачи в приложении, а нижний реализует универсальные подзадачи. Код нижнего уровня вызывается по запросу верхнего. Нижний уровень считается системным программным обеспечением и достаточно консервативен с точки зрения модификации при совершенствовании программного обеспечения. Верхний уровень является прикладным и постоянно модифицируется при выпуске новых версий программ. Рисунок показывает структурирование программного обеспечения на два компонента. Фактически он демонстрирует логику разделения кода и формирование понятия программной архитектуры. Рассмотрим разновидности архитектур и их совершенствование с исторической точки зрения.

1.1.2 Персональная архитектура

Структура приложения, функционирующего на одном компьютере без взаимодействия с другими, называется персональной программной архи-

тектурой. С точки зрения компьютерной сети это вырожденная архитектура. В ней невозможно четко разделить код на клиентский и серверный, так как оба компонента функционируют на одном узле, в одной задаче, разделяя оперативную память. Работающий за конкретным компьютерным узлом пользователь при такой архитектуре не может программным способом взаимодействовать с другими пользователями за другими узлами. Для синхронизации своей работы он вынужден обмениваться данными по сети вручную, используя механизмы, подобные электронной почте.

1.1.3 Терминальная архитектура

Терминальная архитектура является альтернативной программно-аппаратной архитектурой, позволяющей работать нескольким пользователям за одним компьютером. Терминалом называется совокупность устройств ввода/вывода, с которыми работает один пользователь. Один компьютер в этой архитектуре имеет несколько терминалов. Так как поддержка терминалов осуществляется одним компьютером, то разделение и синхронизация данных между пользователями решаются средствами одной операционной системы. Все процессы чтения и записи осуществляются в рамках одного компьютера, синхронизация может выполняться без длительных сетевых операций. Однако для обслуживания нескольких пользователей программно-аппаратная платформа терминальной архитектуры должна быть многопоточной. В современном мире терминальная архитектура реализуется программным способом, при этом система работает в составе компьютерной сети. Предназначенные для работы с пользователем узлы сети называют терминальными рабочими станциями, а выделенный для обработки данных компьютер называют терминальным сервером.

Задача рабочих станций заключается в реализации терминала, т.е. в построении внешнего вида приложения и функционировании графического

пользовательского интерфейса (GUI). Терминальный клиент, взаимодействуя с пользователем, осуществляет ввод/вывод. Вводимые данные передаются на терминальный сервер для обработки, а результаты обработки отсылаются обратно терминальному клиенту для отображения. Задача терминального сервера как узла для обработки, разделения данных и синхронизации работы нескольких пользователей остается неизменным независимо от разновидностей реализации терминальной архитектуры.

По сравнению с персональной архитектурой терминальная требует сложных механизмов для обеспечения взаимодействия клиентов. Высокая централизация всех процессов на терминальном сервере делает эту архитектуру слабой с точки зрения гибкости и масштабируемости серверных компонентов, а сам терминальный сервер является высоконагруженным узлом компьютерной сети.

Достоинства и недостатки терминальной архитектуры близки для аппаратного и программного способа реализации. Терминальная архитектура делает возможным синхронную совместную работу для нескольких пользователей, но не позволяет реализовывать высоко масштабируемые системы. Аппаратный способ существенно ограничен расстоянием от терминалов до компьютера. Фактически роль терминального клиента при программной реализации близка к роли тонкого клиента многоуровневой архитектуры. Но об этом позже, после изучения других видов архитектур.

Основные недостатки терминальных архитектур кроются в сосредоточении обработки и хранения данных в одном узле. Такое объединение затрудняет масштабирование системы, делает сложным синхронизацию работы пользователей. Центральное звено – сервер, становится критическим с точки зрения производительности всей системы целиком.

1.1.4 Файл-серверная архитектура

Развитие программных архитектур происходило одновременно с совершенствованием сетевого программного обеспечения. Разделение по узлам компьютерной сети программного кода, который работает в составе сложного распределенного приложения, требует четких границ между клиентским и серверным компонентами. Именно сетевая многокомпонентная реализация программного обеспечения четко сформировало понятие программной архитектуры. Однако жесткое разделение программного кода на компоненты даже без использования сети позволяет решить также многие технологические задачи программирования.

Необходимость реализовать разделение общей работы на несколько узлов, а также синхронизацию результатов обработки информации на различных узлах, приводит к выделению специального компьютера для таких задач. Этот узел сети называют серверным. Первые попытки реализации подобного сервера заключались в создании специфичной операционной системы. Она позволяла принимать запросы об операциях с файловой системой от клиентов через сеть. Таким образом множеству узлов компьютерной сети становилось возможным программным способом обратится за данными к файловой системе сервера. Удаленное обращение к файлам сервера позволяло также сохранять результаты обработки централизовано. Данная архитектура получила название файл-серверной архитектуры (см. рис. 2).

Фактически файл-серверной операционной системе делегируется роль хранителя данных. Сервер выполняет по запросам других узлов простейшие операции. К ним относятся:

открытие и закрытие файлов,

чтение и передача данных из файлового потока клиенту,

прием данных от клиента и запись потока в файл.

Разделение и синхронизация информации на сервере может организовываться только путем блокировки обращения к данным и выстраивания запросов от клиентов в очередь заданий на низкоуровневые серверные операции.

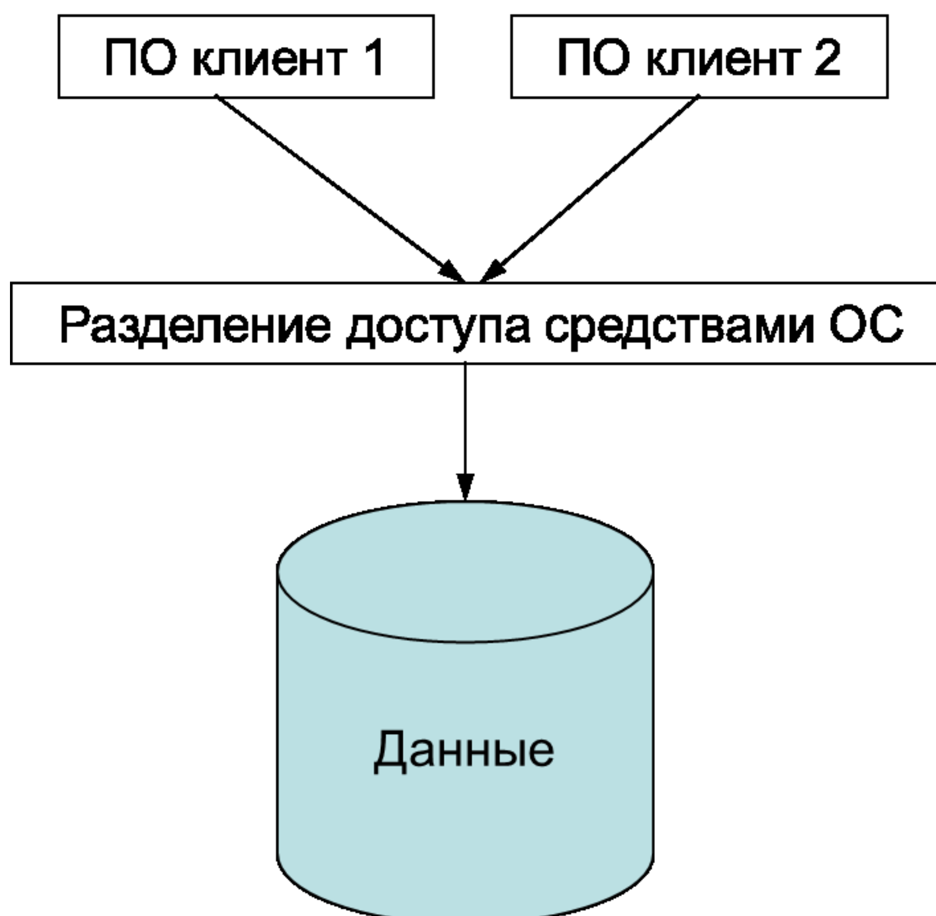


Рис.2. Файл-серверная архитектура

Клиенты делегировали файл-серверу функции чтения и записи. Клиент, требующий чтения, назывался читателем. Соответственно клиент, записывающий данные, – писатель. Читатели не модифицировали данные, следовательно, их операции не влияли на состояние файлов и потому такие операции легко разделялись между несколькими клиентами. В идеале они могли выполняться параллельно. В свою очередь, запросы писателей требовали жесткой регламентации порядка записи. Писатели также влияли на

результаты, получаемые читателями. Одновременно мог работать только один писатель. Работа писателей требовала запрашивания эксклюзивного доступа к файлам. Пока с данными работал один писатель, операции для других клиентов были запрещены. Запрет работы с файлом называется файловой блокировкой.

Позже файл-серверные операционные системы стали реализовывать потоковую блокировку. Писатель, прежде чем начинать работу с модифицированием файла, должен наложить запрет на операции для других клиентов лишь с изменяемой частью файла. Запрет на работу с файлом целиком не требовался. Это повысило производительность всей системы, но следствием являлось усложнение клиентских алгоритмов. Клиенты-писатели теперь были вынуждены информировать файл-сервер о своих операциях перед их выполнением.

Сначала писатель посылает запрос на блокировку части файла (или нескольких частей разных файлов для сложной операции). Если запрашиваемые части файлов не были заблокированы другими писателями, блокировка считается успешной. После этого писатель может выполнять модификацию заблокированных частей файла. Если блокировка невозможна, писатель должен ожидать освобождения блокировок от других писателей. Естественно, после завершения работы по модификации информации писатель обязан освободить блокировки.

Файл-серверная архитектура позволила осуществлять взаимодействие между несколькими клиентами на основе синхронизации по данным. Последовательность операций блокировка-запись-разблокировка является неделимой на логическом уровне и называется транзакцией. В настоящий момент файл-серверная архитектура аппаратно реализуется в устройствах NAS (Network Attached Storage). Другое название этих устройств – сетевые накопители данных. Нередко современные NAS устройства совмещены с некоторыми сервисами, расширяющими файл-серверные архитектуры: ме-

диацентр, фотогалерея, битторрент, почтовый сервер, станция видеонаблюдения и т. д.

1.1.5 Достоинства и недостатки файл-серверных архитектур

Основным достоинством файл-серверной архитектуры является возможность синхронизации работы нескольких пользователей на основе простейших принципов. Но такая простота реализации архитектуры не позволяет обойти несколько существенных проблем.

Во-первых, все данные хранятся на файл-сервере, а обрабатываются на клиенте. Требуется высокий траффик на двухстороннюю передачу данных, что приводит к высокой нагрузке как на сеть, так и на файловый сервер. Причем нагрузка будет увеличиваться прямо пропорционально количеству клиентов. Нагрузка на клиентские узлы также высоки, так как задача клиентов состоит в обработке запрашиваемых данных.

Во-вторых, контроль за разделением данных лежит на клиентском программном обеспечении. Именно оно вынужденно дополнительно блокировать и разблокировать обрабатываемые данные во время работы с файл-сервером. Ошибки приводят к искажению разделяемой информации, т.е. к потере корректных данных.

В-третьих, синхронизация работы писателей требует ожидания очереди при выполнении запросов. Это приводит к большим задержкам при обращении к данным. Задержки увеличиваются пропорционально росту количества клиентов.

В-четвертых, безопасность доступа к данным решается по принципу «все или ничего». Средствами файл-серверной операционной системы возможно только разрешить или запретить для клиента элементарные операции: открытие файла, чтение, запись и модификацию. Такие операции очень глобальны. Более тонкий контроль над данными невозможен.

1.1.6 Двухуровневые архитектуры и СУБД

Появление систем управления базами данных (СУБД) и на их основе языков запросов позволили клиентским компонентам делегировать более сложные рутинные операции на уровень сервера данных. Клиент файлового сервера запрашивал информацию для обработки на своей стороне. Клиент же СУБД запрашивает обработку на стороне сервера. Клиент получает готовые результаты обработки. Если рассмотреть, например, такую задачу как фильтрация данных, то в случае запроса к СУБД, клиент передает серверу правила фильтрации. СУБД выполняет фильтрацию и возвращает на сторону клиента только ту информацию, которая удовлетворяют переданным ему правилам. Если существенная часть данных не соответствует заявленным правилам фильтрации, то значительно снижается не только нагрузка на сеть, но и нагрузка на клиентскую сторону. Клиент лишь использует готовые результаты обработки. Такая реализация компонентов программного обеспечения, в которой клиент имеет возможность делегировать обработку на уровень сервера, называют двухуровневой или же двухзвенной архитектурой (см. рис. 3).

Структурно архитектура файл-сервер и двухзвенная архитектура похожи. Различие кроется в сложности интерфейса между уровнями и в величине абстракции запросов. Файл-серверная архитектура ориентирована на запросы к потокам данных, представляемым в виде файлов. Полноценная двухзвенная архитектура строится на запросах к абстрактным сущностям. Правила извлечения абстрактных сущностей ложатся на технологию, обеспечивающую работу СУБД. Например, для реляционных СУБД стандартом запросов стал язык SQL, а абстракциями являются элементы реляционной алгебры.

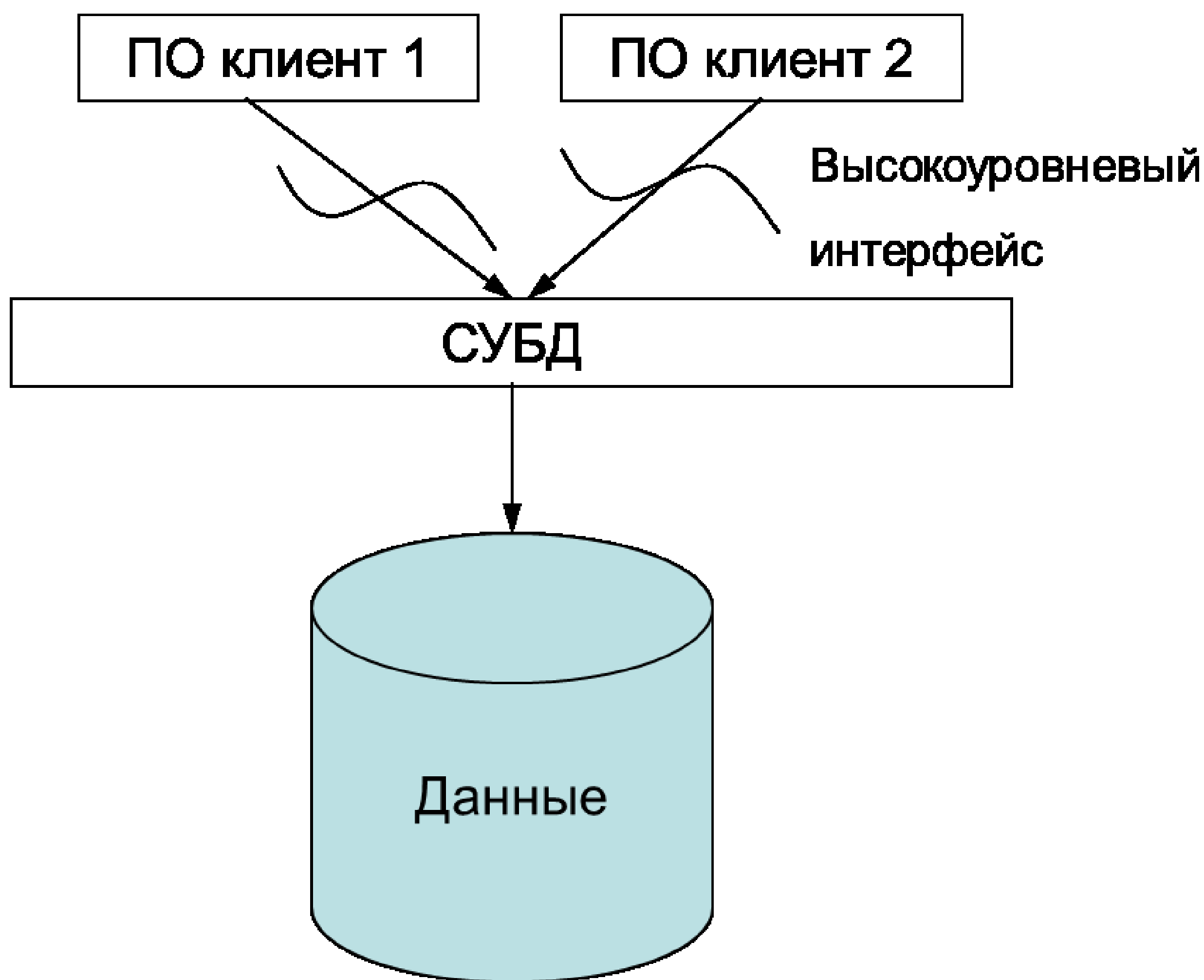


Рис. 3. Двухуровневая архитектура

1.1.7 Технологии клиент-серверного взаимодействия

Разделение кода программ на «Клиент» и «Сервер» потребовало создания механизма для их взаимодействия. Механизм будет отличаться в зависимости от условий, в которых клиент и сервер обменивается данными. Можно выделить три типа взаимодействия:

внутрипроцессное,

межпроцессное,

сетевое.

Внутрипроцессное взаимодействие наиболее простой способ обращения клиента к серверу. Оно возникает при вызове кодом клиента кода сервера в случае расположения их в одном процессе операционной системы.

Это означает, что адресное пространство сервера и клиента совпадают. При передаче информации клиент и сервер могут оперировать ссылками на расположенную в оперативной памяти информацию, так как имеют общее адресное пространство.

Внутрипроцессное взаимодействие возникает в случае технологического разбиения кода на пакеты. Частный случай – вызов кода, расположенного в динамической библиотеке. В однопоточной программе внутрипроцессное взаимодействие реализуется как вызов процедуры. Процедура реализует код сервера, а клиент осуществляет обращение к нему через вызов процедуры. Код клиента передает управление коду процедуры и в том же вычислительном потоке начинается выполнение серверного алгоритма. Завершение выполнения процедуры приводит к продолжению работы кода клиента.

Внутрипроцессное взаимодействие может быть осложнено мультипоточным режимом работы клиента и сервера. Такой случай требует потоковой синхронизации. Клиент формирует обращение к серверу в виде запроса и останавливает свою работу (засыпает). Сервер в другом потоке выполняет запрос, а после возобновляет работу клиентского потока (пробуждает его). К передаваемой между клиентом и сервером информации оба компонента обращаются путем запроса к общему адресному пространству. При мультипоточном внутрипроцессном взаимодействии возможен и асинхронный механизм обработки клиентских запросов. В этом случае клиентский и серверный потоки работают независимо друг от друга, осуществляя взаимодействие путем передачи сообщений друг другу.

Межпроцессное взаимодействие осуществляется в том случае, если клиент и сервер реализованы на одном компьютере, но в разных процессах операционной системы. Это означает что клиент и сервер работают в разных потоках и адресное пространство у них разное. Такая ситуация возни-

кает при необходимости управления действиями одной программы с помощью другой.

Управляющий код является клиентом, а управляемая программа сервером. Для передачи информации между процессами потребуется маршалинг - пересылка данных через границы процессов. Клиент должен осуществить сохранение параметров вызова не в личном адресном пространстве. Он использует какой-либо механизм операционной системы. Например, файл, отображение файла (map), разделяемая память (shared memory) и т. д. Затем клиент косвенно активирует работу алгоритма сервера в потоке другого процесса, а сам засыпает. Сервер осуществляет обработку параметров запроса через обращение к операционной системе за маршализованными данными. По окончании работы сервер осуществляет обратный маршалинг и пробуждает поток в процессе клиента. В межпроцессном взаимодействие также возможнf асинхронная обработка запросов.

Наиболее сложный случай клиент-серверного взаимодействия – это сетевое взаимодействие. Здесь поток клиента и поток сервера расположены не только в разных процессах, но и работают на различных узлах компьютерной сети. В технологически самом сложном случае узлы работают под управлением разных операционных систем. Это называют кроссплатформенным взаимодействием. К сложностям межпроцессного взаимодействия добавляется организация маршалинга через сеть и сетевая межпоточная синхронизация работы клиента и сервера.

1.1.8 Принципы абстрагирования запросов

Рассмотрим типы маршалинга, обеспечивающие сетевое взаимодействие клиента и сервера. В любой архитектуре, при размещении компонентов на разных сетевых узлах, взаимодействие реализуется одной из предлагаемых разработчиками технологий. Такой подход снижает время на

разработку системы и ее стоимость. Имеющиеся технологии для взаимодействия между узлами компьютерной сети можно разделить на четыре вида по принципам абстрагирования передаваемых данных. Модель передачи данных фактически представляет собой модель маршалинга. Это потоковая, процедурно-ориентированная, объектно-ориентированная абстракции и абстракция на основе языка запросов.

В потоковой абстракции клиент и сервер пересылают друг другу потоки данных для обмена информацией. Фактически можно считать, что клиент «пишет» запрос в поток данных, который затем «передается» серверу. Сервер «принимает» поток и считывает запрос в виде последовательности байт. Разбирая поток сервер распознает запрос и выполняет его. При ответе сервера все происходит в обратном порядке.

Для обмена информацией в потоковой абстракции разработчик клиент-серверной архитектуры должен создать интерфейс взаимодействия в виде протокола – определенной структуры в последовательности байт как для запроса, так и для ответа. Частный случай протокола — это текст с заданной грамматикой.

Более высокий уровень абстракции выглядит в виде концепции удаленной процедуры (RPC – Remote Procedure Call). На сервере существуют алгоритмы, представленные в виде процедур или функций. Клиент, обращаясь к серверу, вызывает одну из процедур. При этом аргументы процедуры передаются в качестве параметров клиентского запроса. Результат выполнения удаленной процедуры возвращается обратно клиенту в качестве сетевого ответа. Удаленная процедура фактически является статической функцией. Функция может сохранить промежуточные данные между вызовами. Однако, для того чтобы сохранить между вызовами уникальные для каждого клиента данные, или же организовать сложную транзакцию, потребуются дополнительные действия в алгоритме серверной процедуры. Это основной недостаток RPC технологии.

Существует еще более высокий уровень абстракции – объектно-ориентированный, решающий проблему сохранения промежуточных данных. Объектно-ориентированная абстракция клиент-серверного взаимодействия позволяет сохранять информацию в виде порции данных как существующий на сервере объект. К этому объекту можно обратиться дистанционно несколько раз, выполнив серию последовательных вызовов. Такой способ сохранения на сервере промежуточных результатов между удаленными вызовами позволяет реализовать сложные транзакции, уникальные для каждого клиента.

С помощью объектно-ориентированной абстракции легко можно организовывать сложные взаимодействия, с сохранением на сервере промежуточных данных. Объект может быть создан сервером самостоятельно, по запросу клиента или же ссылка на него передана от одного клиента другому. Объект существует удаленно на сервере, и технология позволяет обращаться к его методам дистанционно. Эта концепция является развитием технологии вызова удаленных процедур. В ней данные запроса делятся на маршализуемые (данные, передаваемые через границы процессов) и объектные - хранимые на сервере.

Маршализуемые данные имеют скрытый параметр, позволяющий связать запрос с каким-либо объектом на сервере. Таким образом серверная процедура работает как с информацией, полученной от клиента непосредственно через маршалинг (аналогично RPC), так и с хранящиеся на сервере информацией, с которой логически через скрытый параметр связан обрабатываемый запрос.

На практике наиболее часто потоковая абстракция реализуется через сокеты. Это связано с высокой переносимостью сокетов. Существует и альтернативные библиотеки с потоковой абстракцией. Например, именованные каналы от Microsoft.

Концепция вызова удаленных процедур реализуется несколькими различающимися стандартами RPC. Некоторые источники относят к RPC и объектно-ориентированные технологии: COM/DCOM, Java RMI, CORBA и т. д.

Особняком стоят процедурно-ориентированные и объектно-ориентированные абстракции на основе HTTP протокола, которые позволяют использовать web технологии для реализации клиент-серверных приложений. В них фактически любой HTTP запрос является вызовом удаленного метода. Клиентом может служить браузерное приложение или же независимая от браузера программа, которая использует HTTP протокол в качестве транспортного.

Наиболее высокоуровневой абстракцией при реализации клиент-серверных интерфейсов является организация интеракций с помощью языка запросов. Специализированный язык позволяет переслать описание запроса на сервер. Это может быть дескриптивное описание, такое как SQL запрос, или же императивное описание алгоритма, который должен выполнить сервер. Например, хранимая процедура.

1.1.9 Способы обработки запросов

Для понимания основных принципов выполнения запросов к серверу с помощью интерфейса на основе скриптового языка, рассмотрим виды обработки серверных SQL запросов. Классический способ обработки SQL запроса подразумевает интерпретацию. Получая SQL запрос СУБД интерпретирует его и выполняет.

Для ускорения обработки частых, незначительно отличающихся запросов, используется следующая технология. Запрос формируется в виде шаблона, с указанием изменяемых параметров. Эти параметры будут уникальны для каждого запроса, однако структура запроса при повторных обраще-

ниях меняться не будет. На сервере подобный шаблон запроса обрабатывается и формируется алгоритм для его выполнения. Алгоритм записывается или же в машинных кодах, либо на промежуточном языке. Оба варианта реализуются в виде процедуры. Такой процесс называют компиляцией запроса или preview запросом – предварительным просмотром шаблона запроса.

Аргументами скомпилированной процедуры являются изменяемые параметры шаблона. При необходимости выполнить запрос клиент указывает серверу на скомпилированный заранее шаблон запроса и уточняет изменяемые параметры. Аргументы передаются предварительно подготовленной процедуре для обработки. Работа скомпилированной процедуры с уточненными параметрами будет происходить быстрее, чем полная интерпретация запроса. Это происходит, потому что при повторных запросах не тратится время на их распознавание. В момент получения запроса у сервера уже существует готовый алгоритм для его выполнения.

Наиболее кардинальным способом ускорения обработки данных при работе с СУБД является способ делегирования алгоритмов стороны клиента на сервер. Частный пример — это хранимые процедуры SQL. Передавая хранимую процедуру на сторону сервера, клиент может затребовать неоднократное выполнение этого алгоритма простым удаленным вызовом. Скомпилированный алгоритм процедуры будет содержаться вместе с данными на стороне сервера. Таким образом база данных хранит не только сами данные, но и методы специфичной работы с этими данными, т. е. фактически используется объектно-ориентированный подход с динамически генерируемым во время выполнения кодом.

1.1.10 Принципы создания двухуровневых архитектур

При разработке приложений на основе двухуровневой архитектуры разработчик обычно использует следующие порядок действий.

I. Выбирается и приобретается одна из СУБД, существующих на рынке системного программного обеспечения. Выбранная СУБД должна удовлетворять заявленным требованиям разработчика по технико-экономическим показателям.

II. СУБД устанавливается на серверном узле компьютерной сети. В частном случае СУБД может быть установлена непосредственно в среде разработки и/или исполнения.

III. Разработчик СУБД вместе с сервером поставляет клиентскую часть СУБД и систему программирования. Эти модули ставятся в среде разработки. Вместо полноценной системы программирования разработчик СУБД может поставлять модули расширения широко известной системы программирования. В среде исполнения на клиенте инсталлируется клиентская часть СУБД.

IV. Разрабатывается логика работы и графический пользовательский интерфейс, которые имплементируется в клиентскую часть. Клиентская часть приложения состоит из разрабатываемого прикладного кода и клиентской части СУБД. Вызовы к СУБД осуществляются через клиентскую часть СУБД на основе открытого API.

Архитектура двухуровневой системы выглядит в виде двух уровней с сетевой точки зрения и трех уровней с точки зрения разделения на программные пакеты. Клиентская часть делится на разрабатываемый прикладной код и клиентскую часть СУБД в виде динамической библиотеки, представляющую системный многократно используемый код от разработчиков СУБД (см. рис. 4).

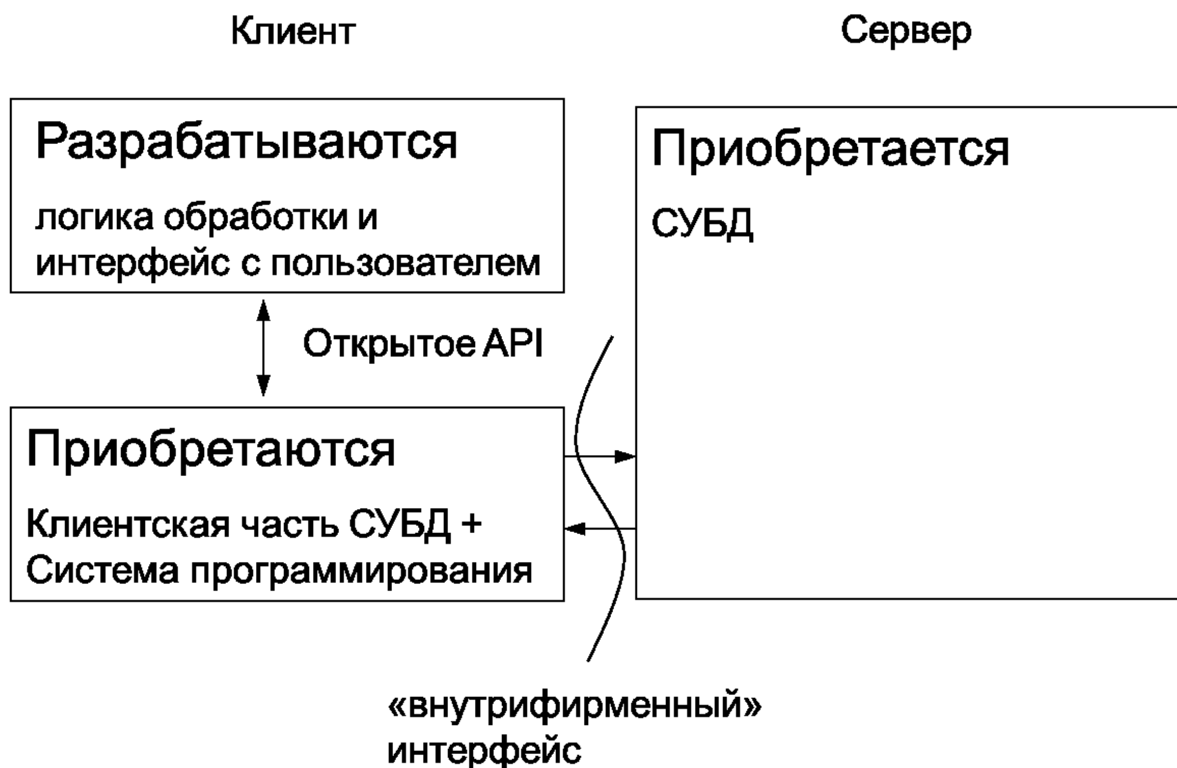


Рис. 4. Принципы разработки двухуровневых архитектур

1.1.11 Преимущества и недостатки двухуровневых архитектур

Рассмотрев технологии организации клиент-серверного взаимодействия, уделим внимание основным достоинствам и недостаткам двухуровневой архитектуры.

Сначала достоинства.

I. Разработчики не проектируют средства для хранения данных, разделения доступа к ним, их защиты, резервирования и т. д. Разработчикам не требуется заботится о многих рутинных низкоуровневых задачах, таких как сортировка, фильтрация, группировка данных и т. д.

II. Приобретая сервер СУБД, разработчики получают под ключ готовые элементы системы, соответствующие международным стандартам. Например, стандарты шифрования, электронной подписи, репликации и т. д.

III. Делегируя многие операции на сервер программное обеспечение клиента снижает аппаратные требования к своему уровню.

IV. Значительно снижается нагрузка на сеть по сравнению с файл-серверной архитектурой, так как сервер передает клиенту лишь результаты низкоуровневой обработки, а не исходные данные. Фильтрация данных, разные виды выборок, передача алгоритмов обработки данных — это наиболее эффективные способы сокращения трафика.

V. Система легко масштабируется на стороне клиента, так как задачи синхронизации работы множества клиентов уже решены на стороне СУБД.

Недостатки двухуровневых архитектур вытекают из их достоинств.

I. Управление данными и низкоуровневую обработку осуществляет сервер, а высокоуровневую специфичную обработку – клиент. В случае невозможности делегировать сложную обработку на сторону сервера получаем недостаточное снижение нагрузки на сеть. В случае возможности переноса сложной обработки на сервер – получаем перегруженный сложными запросами сервер.

II. Изменение логики обработки данных на стороне клиента, а также изменение требований к интерфейсу с пользователем приводит к затратным задачам реинсталляции новых версий клиентского программного обеспечения. При большом количестве клиентских узлов в составе сети высокая трудоемкость клиентского администрирования обеспечена.

1.1.12 Совместимость СУБД разных производителей

Рассмотрим еще одно существенное достоинство двухуровневых архитектур в случае соблюдения при разработке международных стандартов. Если двухуровневая система строится на основе взаимодействия по внутрифирменному (проприетарному) стандарту от поставщика СУБД, то пе-

реход на другого поставщика СУБД затруднен. А такой переход требуется при изменении первоначальных требований к СУБД. Например, в будущем может стать необходимым повышение производительности сервера.

Использование международного стандарта подключения клиента к СУБД через провайдеров данных снижает трудоемкость такого перехода. В качестве примера одного из первых стандартов подключения рассмотрим Open Data Bases Connectivity (ODBC) - открытый стандарт подключения к базам данных (см. рис. 5).

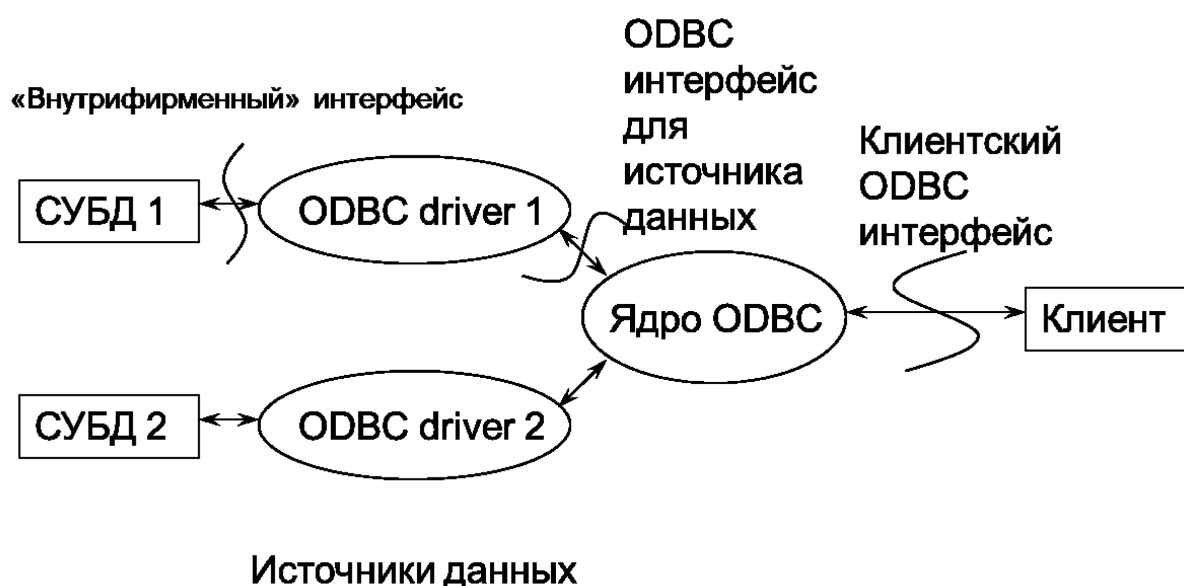


Рис. 5. Архитектура ODBC

Использование международных стандартов подключения к СУБД позволяет добиться высокой независимости клиентского кода от используемой СУБД. Побочным эффектом может быть снижение производительности приложения за счет использования дополнительной инфраструктуры и уменьшение технологичности разработки из-за отказа от поддержки специфических реализаций СУБД.

1.1.13 Функции уровней двухзвенной архитектуры

Разделение задач в двухзвенной архитектуре между уровнями заключается в следующем.

Клиент выполняет две важные функции:

организация пользовательского интерфейса,

прикладная специфика обработки данных.

Сервер отвечает за

хранение данных,

низкоуровневую обработку,

синхронизацию работы клиентов,

а также множество вспомогательных функций, начиная от резервирования и репликации данных, кончая криптозащитой хранимой информации.

Выделение прикладной обработки данных в отдельный специфический компонент позволяет перейти на новую архитектуру.

1.1.14 Трехуровневые и многоуровневые архитектуры

Клиент в трехзвенной архитектуре выполняет лишь функции, связанные с организацией пользовательского интерфейса. Из-за снижения требований к клиенту первый уровень трехзвенной архитектуры называют «тонким клиентом» («thin client»). Получив от пользователя входные данные, тонкий клиент отправляет их на следующий уровень, отвечающий за организацию сложной обработки. Второй уровень называют «application server» («сервер приложений») или же «middleware». Название middleware (среднее программное обеспечение) закрепилось за этим уровнем из-за центрального места расположения в структуре взаимодействия компонентов программного обеспечения. Сервер приложений для получения дополнительной информации из базы данных может отправить запрос к СУБД (см. рис. 6).

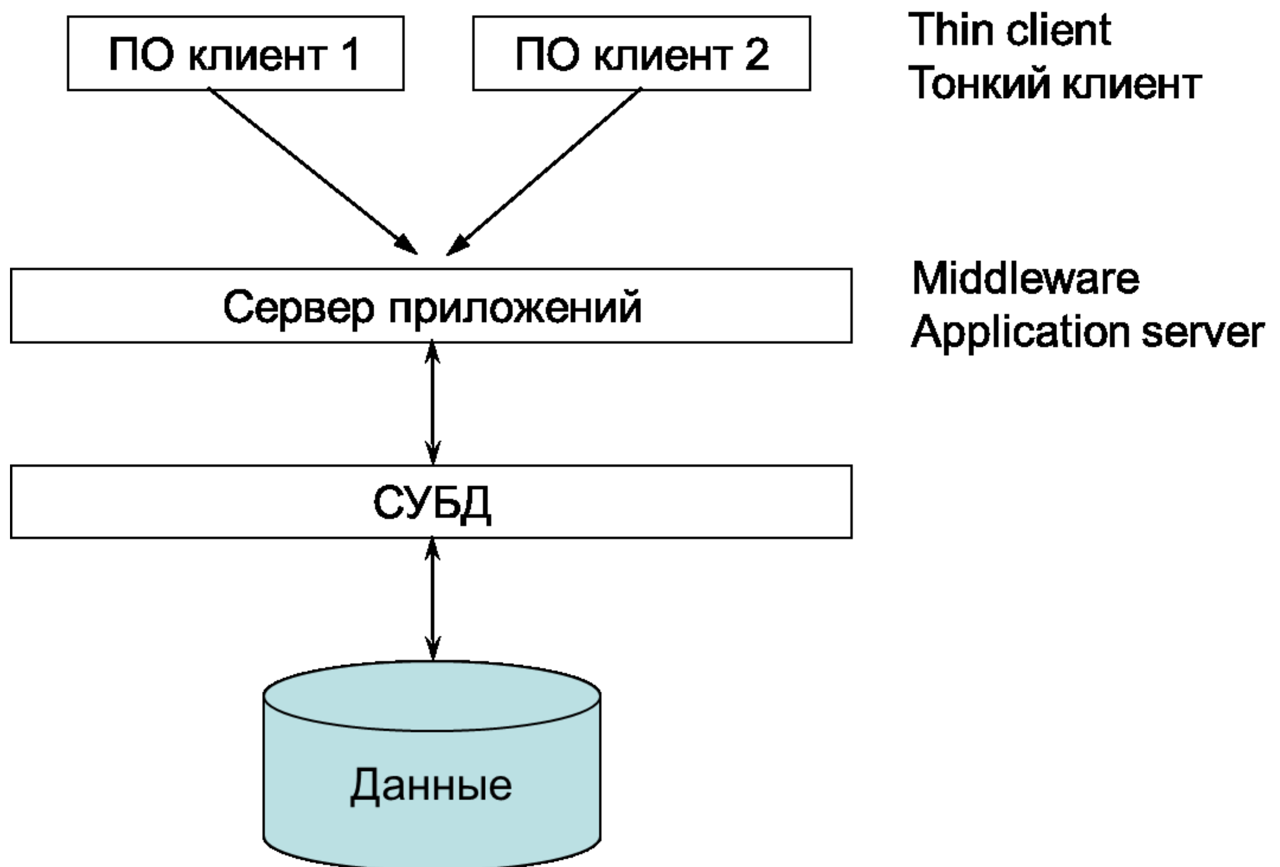


Рис. 6. Трехуровневая архитектура

Третий уровень трехзвенной архитектуры называют сервером баз данных. Функции сервера баз данных остаются такими же, как и у сервера в двухуровневой архитектуре. В отличии от сервера двухзвенной архитектуры, в трехзвенной архитектуре принято уточнять вид сервера. Ведь в трехуровневой архитектуре существует два вида серверов: сервер приложений и сервер баз данных.

Сервер приложений может отправить результаты своей работы для сохранения в базе данных. Его задачей является также передача результатов обработки тонкому клиенту для отображения пользователю. Сервер приложений при работе с СУБД может организовывать сложные транзакции.

При сравнении двух и трехзвенной архитектур для наименования клиента двухуровневой архитектуры иногда используют термин «толстый

клиент». Это позволяет точно определить о каком клиенте идет речь. Толстый клиент реализует как интерфейс с пользователем, так и сложную обработку, а тонкий клиент - только интерфейс с пользователем. Наличие в архитектуре тонкого клиента означает обязательное присутствие в ней и сервера приложений. Именно он определяет логику решения прикладных задач для программной системы в целом.

Функции тонкого клиента прослеживаются и в пользовательском уровне терминальной архитектуры. В этом случае взаимодействующие с пользователями системы узлы компьютерной сети называют терминальными клиентами. Задача терминального клиента аналогична задачам тонкого клиента трехуровневой архитектуры – взаимодействие с пользователем системы. Однако в терминальной архитектуре нет четкого разделения серверной части системы на функции сервера приложений и сервера базы данных. Обработка и хранение информации, а также многопоточная синхронизация, осуществляется централизовано на одном и том же серверном узле. Масштабирование терминального сервера затруднено так как он играет ключевую роль в реализации всех основных задач, а в функции терминального клиента не входят задачи маршрутизации запросов при условии существования нескольких терминальных серверов. Не способствует масштабированию и привязка к терминальному серверу функций хранения данных.

1.1.15 Масштабирование трехуровневых архитектур

Двухзвенная архитектура позволяет относительно легко масштабировать клиентский уровень. Трехзвенная архитектура допускает легкое масштабирование как тонкого клиента, так и сервера приложений. Кроме того, некоторые реализации могут масштабировать и уровень сервера баз данных.

Масштабирование в трехзвенной архитектуре позволяет независимо увеличивать производительность каждого из уровней. Если системе требуется подключить дополнительное количество пользователей – легко масштабируются уровень тонких клиентов. Если требуется увеличение производительности обработки данных, масштабируется уровень сервера приложений (см. рис. 7). Запросы тонких клиентов маршрутизируются с учетом текущей нагрузки на узлы серверов приложений. А если требуется масштабирование уровня сервера баз данных, то сервер приложений может обеспечить с помощью алгоритмов хэширования вычисление сетевого месторасположения конкретного узла хранилища данных на основе информации о запросе. Таким образом происходит распределение нагрузки на уровне сервера баз данных на несколько параллельно работающих узлов.

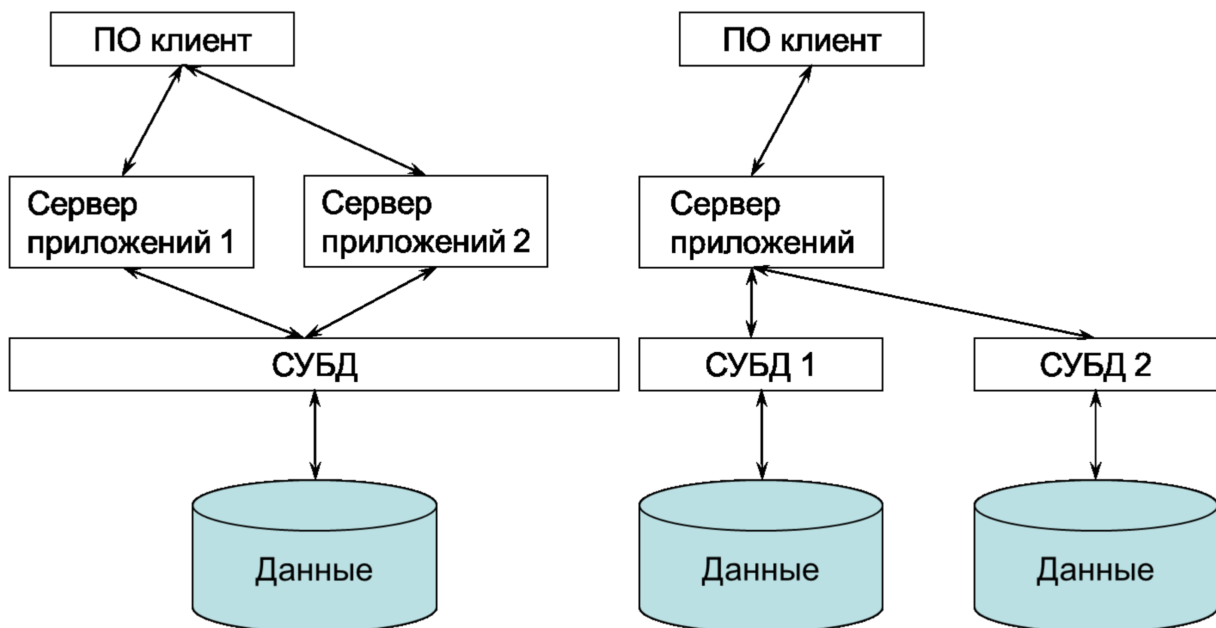


Рис. 7. Масштабирование уровней в трехзвенной архитектуре

1.1.16 Многоуровневые архитектуры

Подобно преобразованию двухзвенной архитектуры в трехзвенную путем разделения клиентского уровня на две части, трехзвенные архитектуры тоже могут совершенствоваться путем дополнительного разделения уровней. Их развитие происходит по принципу «размножения» уровня сервера приложений.

«Почкование» уровня сервера приложений обуславливается двумя причинами. Первая причина заключается в возможности масштабирования вновь выделяемого уровня. Если в требования разрабатываемой системы закладывается увеличение в ближайшем будущем нагрузки на какие-либо алгоритмы, то эти алгоритмы выносятся в отдельный компонент. Изоляция компонента за счет формирования определенного сетевого интерфейса позволяет создать новый уровень в архитектуре. Развёртывание изолированного компонента на отдельном сетевом узле позволяет легко осуществить масштабирование, а это снижает затраты на потенциальное увеличение производительности системы в будущем. Фактически задача повышения производительности сводится к задачам покупки и развертывания новых сетевых узлов, а также к задачам администрирования программного обеспечения масштабируемого уровня.

Второй причиной совершенствования программной архитектуры за счет увеличения уровней является технологичность системы в целом. Компоненты выделенного уровня становится легче модифицировать в функционирующей системе. Возможность взаимодействия с ним по заранее документированному интерфейсу позволяет выполнять замену одной версии компонента на другую без затрагивания узлов, на которых функционируют остальные уровни программной архитектуры. Ответственность за работу системы также можно разделить по компонентам на их разработчиков, что делает сопровождение работы сложной системы более простым.

1.1.17 Пример многоуровневой архитектуры

В качестве примера рассмотрим вариант пятиуровневой архитектуры поисковой системы Google. Сразу стоит обратить внимание на то, что представленная структура несколько упрощена по сравнению с архитектурой реальной поисковой системы. В частности, в ней не представлены уровни, отвечающие за сбор и индексирование информации об интернет ресурсах. В представленной архитектуре приведены только уровни, ответственные за обработку поисковых запросов (см. рис. 8).

Первый уровень архитектуры (браузерный) обусловлен самой структурой сети интернет. Это тонкий клиент на базе интернет браузера. Его задача осуществить получение поискового запроса от пользователя и показать найденные результаты. Масштабирование данного уровня происходит «автоматически» за счет подключения к поисковой системе новых пользователей. Поддержку работы этого уровня осуществляют сами пользователи поисковой системы.

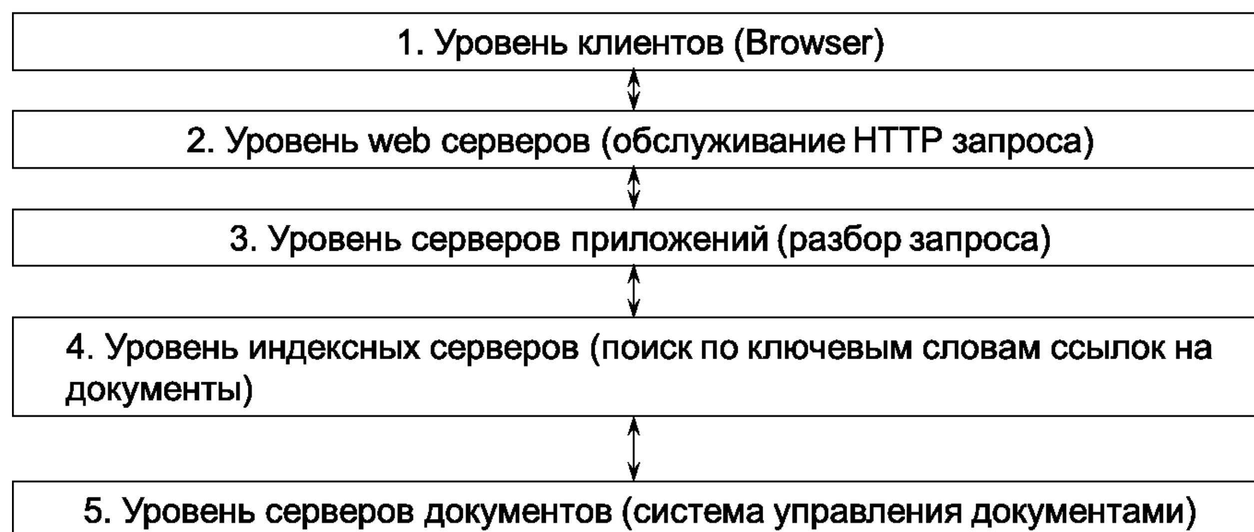


Рис. 8. Многоуровневая архитектура поисковой системы Google

Второй уровень (web-серверный) опять обусловлен web архитектурой, на базе которой работает поисковая система. Задача данного уровня получить запрос от клиентской стороны, правильно его классифицировать

(например, по языку запроса), передать для обработки нижнему уровню и корректно сформировать страницу с найденными результатами для ответа пользователю системы. Масштабирование уровня web-сервера требуется в случае увеличения количества запросов от пользователей. Это происходит как при увеличении числа самих пользователей системы, так и при росте интенсивности использования системы существующими пользователями.

Третий уровень (обработка запроса) выполняет распознавание поисковых запросов. Обработка запроса заключается в лексическом, синтаксическом, семантическом анализе запроса и приводит к формированию ключевых слов и фраз, по которым необходимо производить поиск документов. Масштабирование этого уровня происходит с ростом как числа запросов, так и их сложности, а также разнородности.

Четвертый уровень (поиск индексов) выполняет извлечение из индексированной базы данных ссылок на документы, содержащих искомые сочетания ключевых слов и фраз. Эта операция похожа на чтение оглавления большой книги и выборку номеров страниц, на которых есть информация с заданными ключевыми словами и фразами. Масштабирование уровня происходит с ростом числа уникальных ключевых слов и фраз. Чем их больше, тем выше число индексов, которые требуется проанализировать.

Пятый уровень (поиск документов) выполняет анализ самих документов, ссылки на которые получаются уровнем выше. Этот уровень производит непосредственный разбор потенциально подходящих к поисковому запросу документов, извлекает релевантные фрагменты текста и формирует содержательную часть для результатов поиска. Масштабирование уровня происходит с ростом числа страниц интернета. Чем больше интернет-ресурсов, тем большее количество документов требуется обрабатывать.

1.1.18 Принципы создания распределенных архитектур

При разработке приложений на основе трехуровневой архитектуры разработчик использует следующие принципы.

I. Выбирается и приобретается одна из СУБД, существующих на рынке системного программного обеспечения. Выбранная СУБД должна удовлетворять заявленным требованиям разработчика по технико-экономическим показателям.

II. СУБД устанавливается на серверном узле компьютерной сети. В частном случае СУБД может быть установлена непосредственно в среде разработки и/или исполнения.

III. Разработчик СУБД вместе с сервером поставляет клиентскую часть СУБД и систему программирования. Эти модули ставятся в среде разработки. Вместо полноценной системы программирования разработчик СУБД может поставлять модули расширения известной системы программирования. В среде исполнения на сетевом узле сервера приложений инсталлируется клиентская часть СУБД.

IV. Разрабатывается интерфейс для взаимодействия между тонким клиентом и сервером приложения. Интерфейс документируется.

V. Разрабатывается графический пользовательский интерфейс, который реализуется в тонком клиенте. Тонкий клиент формирует запросы на основе правил, задокументированных в пункте IV.

VI. На основе интерфейса из пункта IV проектируется сервер приложений. Сервер приложений состоит из разрабатываемого прикладного кода для обеспечения логики работы приложения и клиентской части СУБД. Частично принципы разработки трехуровневой архитектуры пересекаются с принципами разработки двухуровневых архитектур. Для многоуровневых архитектур (с числом уровней выше трех) проектируется больше межуровневых интерфейсов и кодируется на прикладном уровне большее число звеньев архитектуры.

1.1.19 Физическая и логическая архитектура

Деление программной системы на компоненты может осуществляться по разным правилам. Если компоненты делить по принципу развертывания (размещения на разных узлах компьютерной сети), то полученная архитектура носит название физической архитектуры. Если в правила деления заложен технологический аспект, то архитектуру называют логической архитектурой. Приложение на основе многоуровневых архитектур логически состоит из совокупности прикладного и системного кода. На примере трехзвенной архитектуры — это можно представить в виде слоистой структуры похожей на торт «наполеон» с попеременно меняющимися слоями из системного и прикладного кода (см. рис. 9).

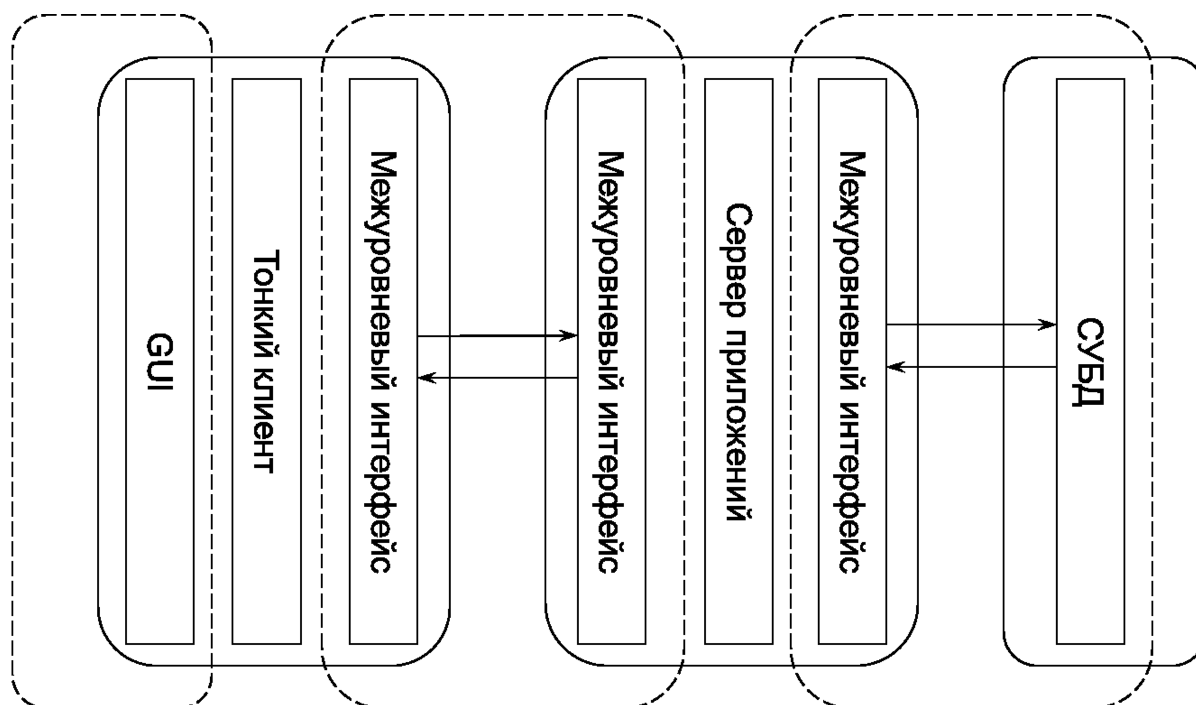


Рис. 9. Системный и прикладной код в трехуровневой архитектуре

На рисунке можно видеть семь компонентов логической архитектуры.

I. GUI - библиотека для организации взаимодействия с пользователем.

II. Тонкий клиент – прикладной код для первичной обработки вводимых данных и финальной подготовки к показу результатов обработки.

III. Proxy сервер приложений - реализация межуровневого интерфейса, обеспечивающего передачу запросов тонкого клиента на сервер приложений (или же представитель сервера приложений на стороне тонкого клиента).

IV. Stub сервер приложений - реализация межуровневого интерфейса, обеспечивающего прием запросов тонкого клиента на сервере приложений (или же представитель тонкого клиента на стороне сервера приложений).

V. Сервер приложений – прикладной код для реализации логики обработки данных при работе системы.

VI. Proxy сервер баз данных - реализация межуровневого интерфейса, обеспечивающего передачу запросов сервера приложений на сервер баз данных (или же представитель СУБД на стороне сервера приложений).

VII. СУБД (система управления базой данных) – системный компонент работы с моделью хранения данных. Разработчики СУБД могут разделить данный компонент на большее количество логических модулей. Но для прикладного программирования такое разделение не имеет смысла, так как СУБД с точки зрения прикладного разработчика представляется сложным, но единым (целостным) компонентом разрабатываемой системы.

Рисунок содержит также три компонента физической архитектуры.

I. Код узла тонкого клиента (компонент, состоящий из системной библиотеки GUI, прикладного кода тонкого клиента и системного кода proxy сервера приложений).

II. Код сервера приложений (компонент, состоящий из системного кода stub сервера приложений, прикладного кода сервера приложений и системного кода proxy СУБД).

III. Код СУБД.

1.1.20 Remote Procedure Call (RPC)

Современные технологии для сетевого взаимодействия на процедурно-ориентированных и объектно-ориентированных абстракциях используют схему обработки запросов RPC (Remote Procedure Call – вызов удаленной процедуры). RPC взаимодействие реализуется с помощью четырех компонентов. Два компонента - клиентское приложение и серверная программа реализуют соответственно клиентскую и серверную часть кода. В случае взаимодействия между тонким клиентом и сервером приложения – обе части являются прикладным кодом. Заглушка proxy и заглушка stub являются инфраструктурой RPC. Программный код этих заглушек автоматически генерируется на основе специальной технологии (см. рис. 10).

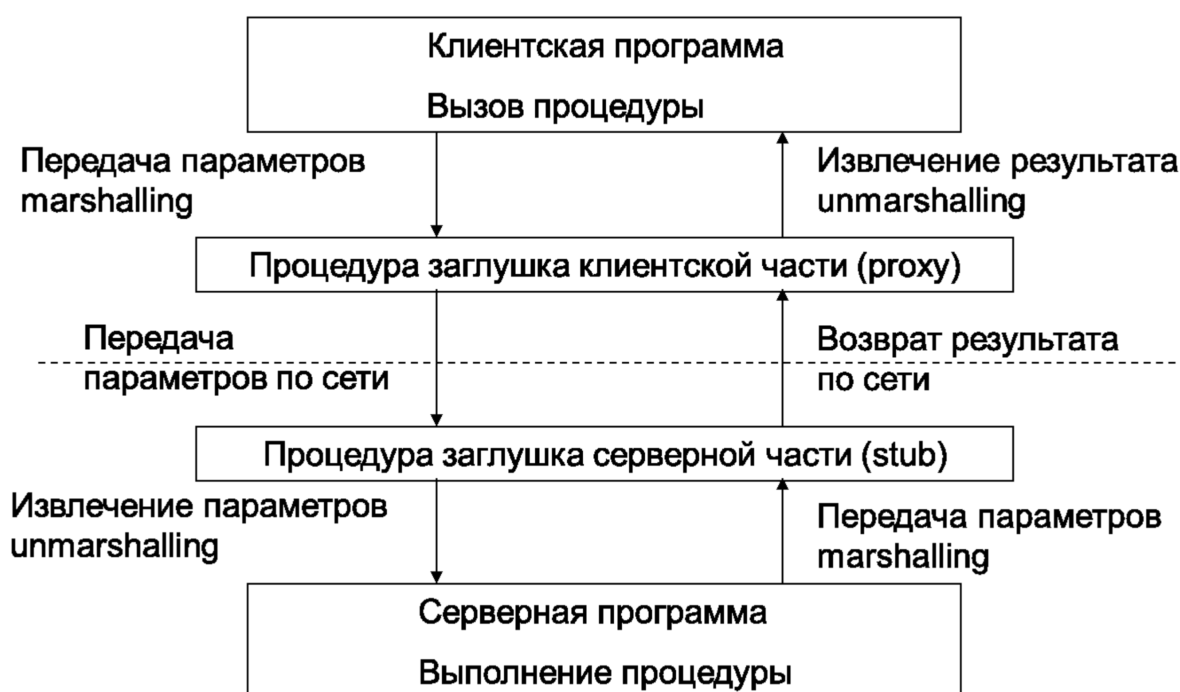


Рис. 10. Схема взаимодействия RPC

Для получения кода proxy и stub необходимо описать интерфейс клиент-серверного взаимодействия. Для RPC это выглядит как декларирование прототипов удаленных процедур. Если язык программирования напрямую не поддерживает разработку клиент-серверных приложений, то

прототипы формулируются на языке IDL (interface definition language – язык описания интерфейса). В современные языки программирования уже интегрированы конструкции, характерные для языка IDL. Обычно это реализуется с помощью атрибутивного программирования. Т.е. удаленные процедуры определенным образом помечаются значением некоторых атрибутов. Также отмечаются атрибутами аргументы процедур для декларирования принципов маршалинга параметров.

Описание механизма вызова удаленной процедуры на языке IDL похоже на формулировку прототипа библиотечной функции языка С в подключаемых заголовочных файлах. Оно не содержит алгоритма, но указывает правила вызова процедуры: сколько параметров нужно передать при вызове и каковы их типы. Язык IDL позволяет также уточнить с помощью поддержки атрибутивной грамматики некоторые дополнительные свойства аргументов. Например, атрибут [in] перед любым параметром-указателем означает передачу данных от клиента на сервер. Содержимое области памяти по переданному указателю отсылается на сервер приложений. Серверная удаленная процедура работает уже с копией содержимого по указателю [in]. Атрибут [out] – означает возврат информации. То есть результат выполнения алгоритма удаленной процедуры копируется на клиентский узел по указателю [out]. Помеченный [out] аргумент-указатель используется как приемник для результата работы серверной процедуры. Перед работой процедуры этот участок памяти не требует инициализации, т.е. может не содержать осмысленного значения. Атрибут [in, out] позволяет сочетать оба действия.

Приведенные примеры атрибутов используются для указания правил генерации кода копирования данных в proxy и stub заглушках. Они необходимы для формирования двунаправленной передачи данных при вызове удаленной процедуры. Для объектно-ориентированной абстракции наборы удаленных процедур группируются внутри объекта (класса). Все они отно-

сятся к обработке одного и того же унифицированного набора данных. Для генератора кода это означает, что перечисленные внутри объекта процедуры работают с одной и той же структурой данных, хранящейся на стороне сервера.

В объектно-ориентированном программировании удаленные процедуры реализуются как публичные методы класса, к которым можно обратится дистанционно. Язык IDL с помощью атрибутов для параметров позволяет детализировать способ передачи информации между клиентом и сервером.

Компилятор языка IDL формирует компоненты proxy и stub автоматически. Фактически компилятор IDL это генератор кода proxy и stub компонентов. Алгоритмы клиентского и серверного кода создаются и кодируются прикладным разработчиком. Редактор связей системы программирования компонует клиентский код с proxy, а серверный код со stub компонентами. В результате получаются два физических модуля, которые разворачиваются на разных узлах сети, но при этом работают согласовано.

Порядок действий при выполнении запроса следующий.

I. Клиентская часть выполняет вызов процедуры-заглушки proxy с передачей параметров запроса.

II. Задача заглушки-proxy выполнить упаковку в поток данных переданных [in] и [in, out] параметров запроса и сформировать сетевые пакеты для передачи полученного потока на сторону сервера. Эта операция носит название маршалинг (marshaling). Сериализация параметров запроса является частью маршалинга. После получения потока данных заглушка-proxy формирует из него последовательность сетевых пакетов на заглушку-stub сервера.

III. Сетевой запрос ожидается на серверном узле заглушкой-stub. Она извлекает [in] и [in, out] параметры из сетевого запроса в оперативную память сервера. Данная операция называется анмаршалинг (unmarshling). За-

тем заглушка-stub формирует выделение памяти для [out] параметров и вызывает реализованную в компоненте сервера процедуру. Процедуре передаются извлеченные из сетевого запроса [in] и [in, out] параметры, а также указатели на выделенную память для [out] параметров. Сигнатура вызова серверной процедуры полностью совпадает с сигнатурой вызова клиентом процедуры заглушки-proxy в пункте I.

IV. Серверная процедура выполняет содержательную часть действий, определяемую прикладными алгоритмами. Результаты работы серверной процедуры передаются заглушке stub в виде изменений в [out] и [in, out] параметрах.

V. Заглушка-stub выполняет маршалинг результатов (параметров [out] и [in, out]) для передачи ответа сервера по сети на заглушку proxy. После освобождает оперативную память от отработанных параметров.

VI. Заглушка proxy выполняет анмаршалинг для извлечения результатов работы сервера (т.е. из сетевого ответа извлекаются [out] и [in, out] параметры). Заглушка proxy выполняет возврат результата работы сервера путем изменения на клиенте содержимого по [out] и [in, out] указателям. Задача заглушки-proxy после передачи сетевого запроса в пункте II и до получения результатов работы сервера в пункте VI заключается только лишь ожидании. Клиентский поток между II и VI шагом «засыпает».

В свою очередь сервер «спит» между запросами от клиентского программного обеспечения. Клиент и сервер работают попеременно. Поэтому данный режим клиент-серверного взаимодействия называют синхронным. Если изъять из структуры RPC заглушки stub и proxy, то компоновка клиентского и серверного модуля формирует простой прямой вызов серверной процедуры внутри одного модуля. Фактически получается персональная физическая архитектура.

Proxy и stub реализует физическое разделение модулей по узлам компьютерной сети. Их задача в маршалинге/анмаршалинге данных, а также в правильной маршрутизации запросов/ответов.

1.1.21 Технология ORB и прозрачность расположения сервера

Термин прозрачность месторасположения сервера определяет возможность разработчика клиентского приложения не задумываться о локализации сервера при посылке запроса. Когда происходит обращение, инфраструктура удаленного взаимодействия сама должна определить адрес назначения запроса в компьютерной сети. Разные технологии сетевого взаимодействия предоставляют различные пути достижения поставленной задачи.

Во-первых, адрес сервера может быть инкапсулирован внутри программного кода в отдельном модуле. В целях быстрого сетевого администрирования конечно же данную информацию все-же лучше хранить отдельно от программного кода. Это не потребует перекомпиляции программ только лишь для смены сетевого месторасположения сервера.

Во-вторых, адрес сервера и правила отсылки запросов могут хранится в отдельном конфигурационном файле. Исправляя в нем параметры, администратор может быстро настроить правильную маршрутизацию запросов. Частным случаем отдельного хранилища информации о сетевой инфраструктуре в операционной системе Windows является реестр. Технология COM/DCOM от Microsoft извлекает адреса серверов и правила взаимодействия с ними как раз из этого хранилища.

В-третьих, информацию о серверах можно хранить централизовано на отдельном сетевом узле в базе данных. Инфраструктура, организующая удаленное взаимодействие, обращается к этой базе для получения адреса сервера, т.е. адресата запроса. Вариант, при котором выделяется отдельный сетевой узел, отвечающий за правильную маршрутизацию всех сер-

верных запросов, называют CORBA – Common Object Requested Broker Architecture – обобщенную архитектуру запросов к объектам через брокера системы. Сетевой узел, на который первоначально отправляются все клиентские запросы, как раз и называется брокером.

Получив запрос от клиента, брокер по его свойствам определяет объект-получатель запроса и адрес сервера, на котором он расположен. При этом, используется база данных брокера, в которой содержится описание всех удаленных объектов и их физическое расположение на узлах компьютерной сети.

База данных хранит правила маршрутизации любых запросов CORBA. CORBA предоставляет один из самых удобных механизмов прозрачности месторасположения сервера с точки зрения легкости администрирования. Достаточно централизовано изменить базу данных брокера для того, чтобы все запросы от множества клиентов направлялись к другому узлу (см. рис. 11).

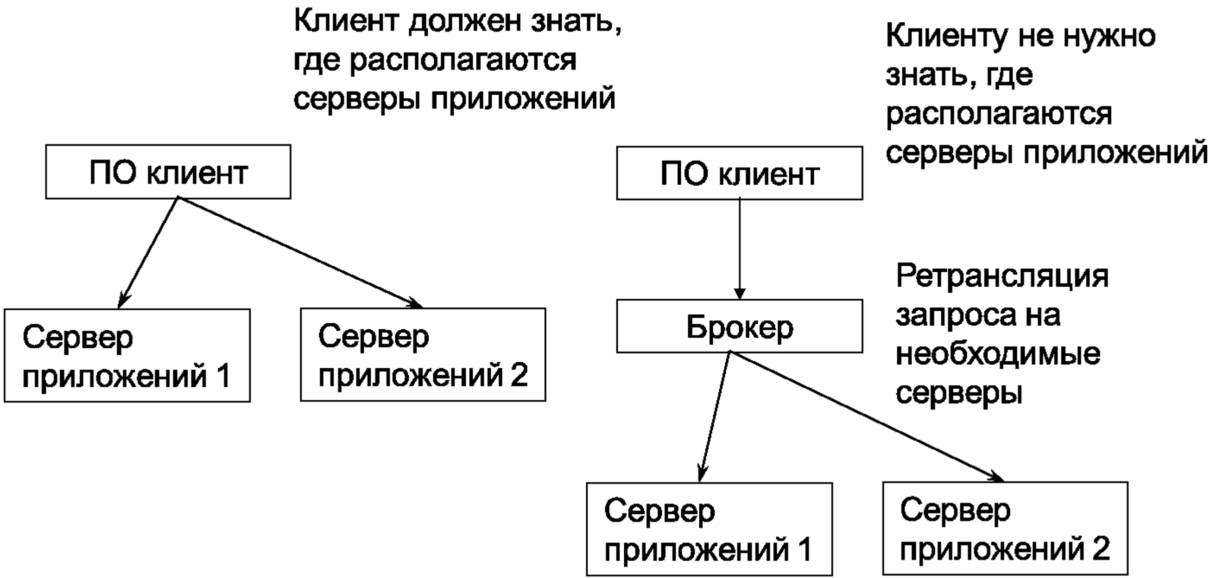


Рис. 11. Принцип прозрачности месторасположения серверов в ORB

1.1.22 Возможности многоуровневых архитектур

Широкое распространение многопоточных систем позволило отойти от принципов синхронного клиент-серверного взаимодействия. Рассмотренные выше технологии сетевого взаимодействия подразумевали ожидание клиентом результата работы сервера. Однако если клиент не будет ждать результатов обработки запроса, система может стать более производительной.

Клиент не простаивает, а приступает к решению следующей задачи пока предыдущий его запрос обрабатывается сервером. Данный принцип взаимодействия называют асинхронным. С технологической точки зрения такое взаимодействие отходит от принципа синхронного вызова на сервере удаленной процедуры. Серверу просто передается информация о необходимости выполнить какое-то действие.

Фактически серверу «сообщается» о некотором событии на стороне клиента, а он уже «реагирует» на это событие. Чтобы предать результаты обработки клиентского запроса-события сервер должен «сообщить» клиенту о завершении обработки так же с помощью посылки сообщения о соответствующем событии.

При такой реализации клиент-серверного взаимодействия исчезает различие между активным клиентом и пассивным сервером. При изначальном событии - клиент сообщает серверу о необходимости выполнения каких-либо действий, - клиент активен, а сервер пассивен. При событии, символизирующем окончание серверной обработки, - сервер сообщает клиенту о завершении выполнения его предшествующего запроса, – сервер активен, а клиент пассивен.

Клиент и сервер становятся равнозначными с точки зрения реализации взаимодействия. С функциональной же точки зрения клиент и сервер различны. Например, тонкий клиент организует интерфейс с пользователем, а сервер приложений – логику работы с данными. Тонкий клиент передает

серверу события о необходимости выполнять какие-то действия, а сервер пересылает клиенту события о результатах запрошенной ранее обработки. Такое взаимодействие является асинхронным. Асинхронное клиент-серверное взаимодействие иногда называют Server-To-Server дабы подчеркнуть технологическое единство клиента и сервера.

На асинхронном взаимодействии базируются архитектуры MOM – Message Oriented Middleware (сервер приложения, ориентированный на сообщения). Часто реализация MOM позволяет маршрутизировать сообщения подобно принципам, заложенным в CORBA. В сети существует централизованный узел, который перенаправляет сообщения адресатам и отслеживает корректную доставку.

Кроме того, технология MOM может обеспечить гарантированную доставку сообщений путем повторной его посылки в случае потери при возможных сбоях. Ориентированные на MOM архитектуры по принципам доставки сообщений похожи на системы электронной почты. Узлы разных уровней общаются между собой с помощью пересылок «писем-сообщений».

Достоинствами MOM являются.

I. Эффективная параллельная работа клиента и сервера.

II. Элементарный принцип маршрутизации сообщений, подобный почтовым рассылкам.

III. Простой механизм масштабирования и балансировки нагрузки уровня сервера приложения на основе распределения событий по наименее загруженным узлам.

IV. Возможность примитивной повторной пересылки запроса как средства восстановления соединения при сетевых сбоях.

В то же время MOM имеет и существенные недостатки.

I. Сложность синхронизации параллельной работы клиента и сервера, в некоторых случаях приводящий к взаимным блокировкам.

II. Повышенные требования к ресурсам тонкого клиента для обработки обратных сообщений от сервера.

III. Обязательная поддержка многопоточности как на сервере, так и на клиенте для реализации очередей обработки приходящих сообщений.

Любой узел, являющийся получателем событий, обязан иметь минимум два вычислительных потока. Один поток (рабочий) отвечает непосредственно за логику обработки запросов, а второй (коммуникационный) обеспечивает сетевое взаимодействие со смежными уровнями архитектуры.

К коммуникационным операциям относят: сетевой прием сообщений, анмаршалинг и ведение очереди полученных необработанных запросов. Рабочий поток извлекает необработанные сообщения из очереди по мере выполнения текущих запросов. Все это ведет к увеличению трудозатрат на разработку систем на базе МОМ архитектуры и требует большего времени для проектирования, кодирования и отладки.

Независимо от способа реализации многоуровневой архитектуры как синхронной, так и асинхронной, она представляет следующие преимущества.

I. Невысокие аппаратно-программные требования к уровню тонкого клиента (по сравнению, например, с двухуровневой архитектурой), что приводит к низкой совокупной стоимости многопользовательской системы с большим количеством клиентских узлов.

II. Простота администрирования.

III. Легкость масштабирования уровней.

IV. Низкие трудозатраты на сопровождение и модификацию отдельных уровней без существенного влияния на остальные

1.1.23 Скриптовые языки для управления нагрузкой уровней

Поддержка межуровневых интерфейсов на основе сложных языков позволяет разработчику многозвенной архитектуры перераспределять некоторые задачи между уровнями (см. рис. 12).

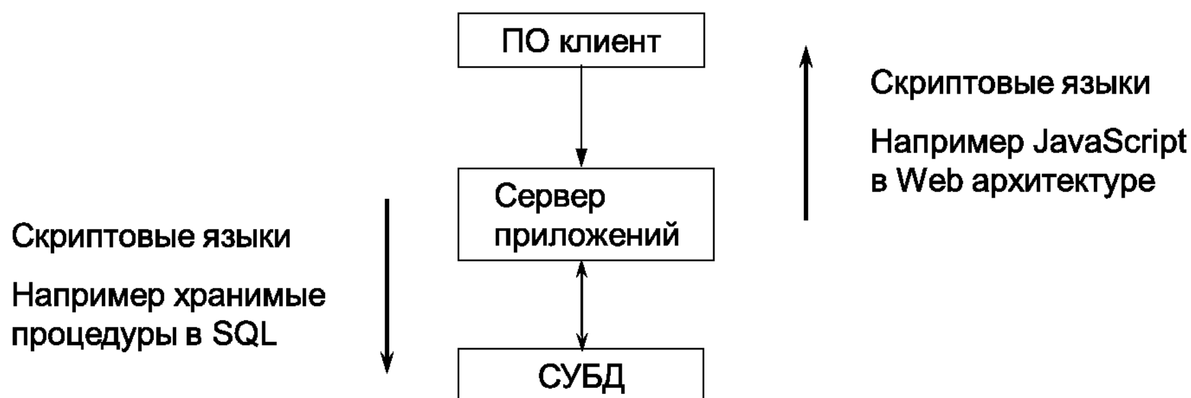


Рис. 12. Делегирование операций между уровнями

Приведенные ниже примеры показывают сложившиеся стандарты в некоторых областях проектирования.

I. В web архитектуре сервер приложений (серверный код) с помощью языка JavaScript может поручить выполнение отдельных алгоритмов тонкому клиенту. При большом количестве клиентов использование такой возможности позволяет значительно снизить нагрузку на уровень сервера приложений, а также уменьшить нагрузку на сеть.

II. При разработке информационных систем сервер приложений с помощью хранимых процедур SQL может поручить уровню сервера баз данных выполнить некоторые сложные алгоритмы относящиеся к логике работы приложения. Это снижает нагрузку на сеть и на сервер приложений.

Подобное перераспределение задач называют балансировкой нагрузки между уровнями системы. Разработчики активно используют возможности межуровневой балансировки нагрузки для увеличения производительности работы всей системы в целом.

1.2. Сетевые протоколы и сервисы

1.2.1 Модель OSI

Основным термином, используемым при формирования взаимодействия между программными компонентами, разделенными на основе какого-то принципа, является понятие протокола. Например, это может быть взаимодействие программных компонентов, расположенных на разных узлах компьютерной сети, как локальной, так и глобальной.

Протокол – это набор правил и соглашений, описывающих процедуру взаимодействия между компонентами системы. Встречается близкое понятие – интерфейс. Исторически сложилось что под протоколом обычно понимают порядок передачи данных в виде байтов, октетов, структур данных или же грамматических конструкций. Термин интерфейс чаще всего используют в более сложных способах взаимодействия со сложными транзакциями. Например, на основе запросов к удаленным объектам. Таким образом программный интерфейс — это способ обеспечения сложного взаимодействия между компонентами программного обеспечения.

Способы реализации программных интерфейсов могут быть разными. Например, широко используется термин API – Application Programming Interface (интерфейс прикладного программирования). Фактически понятие API реализует подход программирования путем вызова процедур, реализованных в другом программном компоненте. Такой способ ориентирован на решение технологических задачи разработки программного обеспечения.

Особенное значение при разработке распределенных систем приобретает возможность размещения разных компонентов на различных узлах компьютерной сети. При этом каждый программный компонент в этом случае занимает отдельный узел и опирается на аппаратно-программную платформу, наиболее оптимизированную для решения его конкретных задач. Масштабирование и администрирование таких систем упрощается.

Для реализации сетевого взаимодействия используется модель open system interconnection - OSI (см. табл. 1)

Таблица 1

Модель OSI

Модель					
Уровень (layer)		**Тип данных (PDU)**	**Функции**	**Примеры**	**Оборудование**
Host layers	7. Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, POP3, WebSocket	Хосты (клиенты сети)
	6. Представлен ия (presentation)		Представление и шифрование данных	ASCII, EBCDIC	
	5. Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP	
	4. Транспортн ый (transport)	Сегменты (segment) /Датаграммы (da tagram)	Прямая связь между конеч-ными пунктами и надёжность	TCP, UDP, SCTP, PORTS	
Media layers	3. Сетевой (network)	Пакеты (packet)	Определение маршрута и ло-гическая адресация	IPv4, IPv6, IPsec, AppleTalk, ICMP	Маршрутизатор
	2. Канальный (data link)	Биты (bit)/ Кадры (frame)	Физическая адресация	PPP, IEEE 802.22, Ethernet, D SL, ARP, сетевая карта.	Коммутатор, точка доступа
	1. Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными дан-ными	USB, кабель («ви-тая пара», коакси-альный, оптоволо-конный), радиока-нал	Концентратор, Повторитель (сете-вое оборудование)

1.2.2 Протоколы HTTP и MQTT

HTTP (Hyper Text Transfer Protocol) – протокол передачи гипертекста.

Каждый серверный ресурс имеет уникальный web адрес, называемый универсальным идентификатором ресурса (URI – Universal Resource Identi-fier). В самом общем случае URI выглядит следующим образом:

protocol://user:password@host:port/path/file?parameters#fragment,

где protocol - прикладной протокол, посредством которого получают

доступ к ресурсу;

user - пользователь, от имени которого получают доступ к ресурсу либо сам пользователь в качестве ресурса;

password - пароль пользователя для аутентификации при доступе к ресурсу;

host - IP-адрес или имя сервера, на котором расположен ресурс;

port - номер порта, на котором работает сервер, предоставляющий доступ к ресурсу;

path - путь к файлу, содержащему ресурс;

file - файл, содержащий ресурс;

parameters - параметры для обработки ресурсом-программой;

fragment - точка в файле, начиная с которой следует отображать ресурс.

Сообщение запроса от клиента к серверу состоит из строки запроса (request-line), заголовков (общих, запросов, объекта) и, возможно, тела сообщения. Строка запроса начинается с метода, затем следует идентификатор запрашиваемого ресурса, версия протокола и завершающие символы конца строки:

<Метод> <Идентификатор> <Версия HTTP> <CR><LF>

Примеры методов HTTP запросов приведены ниже.

GET - позволяет получать любую информацию, связанную с запрашиваемым ресурсом. В большинстве случаев, если идентификатор запрашиваемого ресурса указывает на документ, то сервер возвращает содержимое этого документа. POST используется для запроса, при котором адресуемый сервер принимает данные, включенные в тело сообщения запроса, и отправляет их на обработку приложению, указанному как запрашиваемый ресур.

PUT - тело сообщения сохраняется на сервере, причем идентификатор запрашиваемого ресурса будет идентификатором сохраненного документа.

Если ресурс уже существует, включенный в тело сообщения объект заменяет существующий.

DELETE - запрашивает сервер об удалении ресурса, имеющего запрашиваемый идентификатор.

OPTIONS - выполняет запрос информации об опциях соединения (например, методах, типах документов, кодировках), которые поддерживает сервер для запрашиваемого ресурса.

HEAD - идентичен GET, за исключением того, что сервер не возвращает в ответе тело сообщения. Этот метод может использоваться для получения информации об объекте запроса без непосредственной пересылки тела объекта, например с целью тестирования связей и пр.

TRACE - используется для возврата переданного запроса на уровне протокола HTTP

После получения и интерпретации сообщения запроса сервер отвечает сообщением HTTP ответа.

Первая строка ответа – это строка состояния (Status-Line). Она состоит из версии протокола, числового кода состояния, поясняющей фразы, разделенных пробелами и завершающих символов конца строки: <Версия HTTP> <Код состояния> <Поясняющая фраза> <CR><LF>

MQTT (Message Queue Telemetry Transport) - лёгкий сетевой протокол, работающий поверх TCP/IP. Используется для обмена сообщениями между устройствами по принципу издатель-подписчик (publish–subscribe).

Первая версия протокола была разработан доктором Энди Станфорд-Кларком (IBM) и Арлен Ниппер (Arcom) в 1999 и опубликована под роялти-фри лицензией. Спецификация MQTT 3.1.1 была стандартизирована консорциумом OASIS в 2014 году.

Достоинства MQTT:

Прост в использовании. Программная реализация без лишней функциональности может быть легко встроена в любую систему

Паттерн проектирования издатель-подписчик удобен для большинства решений с датчиками. Дает возможность устройствам выходить на связь и публиковать сообщения, которые не были бы заранее известны или предопределены

Легок в администрировании

Снижена нагрузка на канал связи. Сообщения, насколько это возможно, несут в себе только полезную нагрузку

Работа в условиях постоянной потери связи или других проблем на линии

Нет ограничений на формат передаваемого контента.

1.2.3 Вопросы и задачи для самостоятельной подготовки

1 Понятие сетевого протокола.

2 Уровни модели OSI и TCP/IP. Примеры протоколов разных уровней.

3 Протокол HTTP. Назначение, команды. Серверы и клиенты HTTP.

4 Протокол MQTT.

5 Клиент-серверное взаимодействие.

6 Сериализация. XML и JSON.

7 Понятие о веб-сервисах / службах

8 Веб-сервисы SOAP. UDDI, WSDL.

9 REST архитектуры. RESTful веб-сервисы.

10 Создание клиентов веб-служб

11 SOA.

12 Микросервисная архитектура

1.3. Облачные вычисления

1.3.1 Классификация облачных сервисов

Технологии разработки и использования программного обеспечения, в которых аппаратно-программная инфраструктура реализуется на основе аренды, формирует понятие облачных вычислений. С точки зрения акцента аренды облачные вычисления могут базироваться на инфраструктуре как услуге (IaaS – Infrastructure as a Service), платформе как услуге (PaaS – Platforma as a Service) и программном обеспечении как услуге (SaaS – Software as a Service).

Принципы организации облачных вычислений в модели IaaS базируются на аренде вычислительных мощностей и объемов памяти для хранилищ данных, а также предоставление системного программного обеспечения. При этом разработка прикладного программного обеспечения лежит полностью на потребителе услуги IaaS.

В IaaS сервис не может функционировать без поддержки прикладных разработчиков на стороне потребителя сервиса. Их задача заключается в разработке и размещении прикладного программного обеспечения в удаленном «облаке».

PaaS сервис направлен на предоставлении удаленных технологических платформ для создания и функционирования прикладного программного обеспечения. При этом значительно снижаются вложения в разработку программного обеспечения за счет использования инструментального программных средств в составе арендуемой платформы.

SaaS сервис направлен на конечного пользователя. Вместе с аппаратным и системным программным обеспечением заказчик арендует и прикладное программное обеспечение, а также данные, необходимые для решения задач потребителя услуг. Такая модель в идеале не требует вообще вложений в разработку программного обеспечения. Вопросы адаптирова-

ния арендуемого комплекса вместе с предоставляемыми программными, аппаратными и информационными ресурсами к нуждам потребителя решаются средствами администрирования.

Особенности облачных моделей представлены в таблице 2. Арендуемые элементы инфраструктуры показаны символом ‘X’.

Таблица 2

Аренда элементов IT инфраструктуры для разных облачных моделей

	Традиционное IT	IaaS	PaaS	SaaS
Прикладные приложения				X
Данные				X
Фреймворк			X	X
Система программирования			X	X
Системное администрирование			X	X
Серверы приложений			X	X
Серверы баз данных			X	X
Операционная система		X	X	X
Виртуализация		X	X	X
Аппаратные серверы		X	X	X
Аппаратные хранилища		X	X	X
Сетевая инфраструктура		X	X	X

Рассмотрим достоинства и недостатки различных облачных моделей.

IaaS – полномасштабная модель, которая обеспечивает почти полный спектр возможностей в сравнении с традиционными IT технологиями. Среди преимуществ инфраструктуры как услуги можно выделить следующие:

- экономия на расходах на покупку аппаратного обеспечения;
- сторонняя поддержка аппаратного обеспечения (серверов, хранилищ, сетевой инфраструктуры);
- возможность быстрого масштабирования;

- возможность «безболезненного» уменьшения потребляемых ресурсов, в случае снижения требований;

- приемлемая надежность и безопасность (при доверии к поставщику);

- полный контроль над настройками инфраструктуры.

У IaaS существуют и недостатки. Эта облачная модель наиболее дорогая, поскольку арендует в виде бандла (пакета услуг) аппаратную инфраструктуру с системным программным обеспечением в независимости от ее практического использования**.** IaaS экономит на покупке аппаратно-программного комплекса, но не на его содержании. Фактически очень высокие первоначальные вложения в инфраструктуру заменяются на более высокие по сравнению с традиционным IT постоянные расходы на сопровождение. IaaS - это экономия на начальных вложениях. Траты откладываются на более поздний период. Все вопросы, связанные с настройкой и управлением аппаратно-программной инфраструктурой**,** остаются полностью на арендаторе. Соответственно, помимо арендной платы, необходимо выделять финансы и на эти задачи. Также нужно обладать профессиональными кадрами, для организации наиболее эффективного использования арендуемых ресурсов.

PaaS – концепция облачных вычислений, ориентированная на прикладных разработчиков. PaaS имеет следующие преимущества:

- сокращает время разработки и развертывания программно-аппаратного комплекса, так как системные серверные компоненты и программное обеспечение как для функционирования, так и для разработки прикладного кода настраиваются поставщиком;

- позволяет развивать разные проекты, с поддержкой нескольких языков программирования и фреймворков;

- обеспечивает поддержку работы удаленных команд. Эта особенность актуальна для аутсорсинга.

Однако у PaaS существует два больших недостатка:

- нет возможности контролировать низкоуровневые задачи, например, нет доступа к управлению виртуальной машиной. Соответственно теряется возможность низкоуровневой оптимизации;

- PaaS имеет меньшую гибкость и масштабируемость по сравнению с IaaS.

SaaS предлагает множество преимуществ для частных лиц и малых компаний:

- доступ к прикладным программам из любой точки мира;
- использование на многих устройствах (включая смартфоны, поскольку большинство SaaS - решений имеют мобильные версии);
- автоматическое обновление программного обеспечения;
- полное сопровождение работы аппаратно-программного комплекса поставщиком (в том числе устранение неполадок и модернизация);
- минимальные дополнительные затраты;
- начало использования «из коробки», поставка услуг «под ключ», т.к. для запуска такого сервиса нужно только зарегистрироваться (и, соответственно, вовремя оплачивать услуги).

Облачная модель SaaS тоже не идеальна. Прежде всего, полностью отсутствует контроль за работой аппаратного оборудования, а также системного и прикладного программного обеспечения. Поставщик имеет право менять конфигурацию аппаратного и программного обеспечения в любой момент, что может повлечь за собой неожиданные ошибки в работе системы. А их исправление будет более медленным, по сравнению с другими облачными моделями, т. к. требуют согласования работ по устранению проблем между арендатором и поставщиком услуг.

С точки зрения «золотой середины» облачная модель PaaS имеет приемлемый компромисс при сравнении достоинств и недостатков представленных моделей. С одной стороны PaaS дает приемлемую гибкость модификации программного кода и полный контроль за прикладными програм-

мами, с другой стороны - ответственность за поддержание аппаратного и системного программного обеспечения в рабочем состоянии лежит на поставщике услуг, что позволяет пользователю сервиса сосредоточиться на решении прикладных проблем.

1.3.2 Облачные серсисы PaaS

Из существующих облачных сервисов модели PaaS в качестве примера можно представить следующие: Heroku, AWS Elastic Beanstalk, Dokku, DigitalOcean App Platform, Google App Engine, Red Hat OpenShift, Engine Yard, AWS Lambda, Salesforce Lightning Platform, Pivotal Cloud Foundry, Windows Azure. Примеры отечественных сервисов: Yandex.Cloud от Яндекс, SberCloud от Sber, Mail.ru Cloud Solutions (VK Cloud Solutions) от Mail.ru, BeeCloud от Beeline.

В качестве основных популярных используемых облачных систем PaaS сейчас можно указать MS Azure и AWS. Для проекта на .NET легче всего использовать Azure.

В разделе 3 далее рассмотрены примеры использования AWS, Yandex.Cloud и MS Azure, приводится самый простой вариант – реализация безсерверных веб-функций. Такой вариант в некотором смысле и самый дешевый при небольшом числе обращений и является своего рода страховкой от незапланированных затрат при случайном использовании платных опций платформ.

2. МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

2.1 Разработка RESTful интерфейса

Необходимо разработать RESTful интерфейс для выданного преподавателем варианта задания. Выделить сущности (объекты данных) и операции (действия над объектами).

Результат разработки представить в виде таблицы, где столбцы представляют собой сущности, а строки - операции. На пересечении операции описывается действие сервера приложений.

Вариант определяется последней цифрой зачетки. Число - 0 означает 10 вариант.

1. Форум
2. Система голосования
3. Каталог библиотеки
4. Заказ пиццы
5. Электронная очередь МФЦ
6. Служба знакомств
7. Школьный дневник
8. Запись к врачу
9. Внесение показаний квартирных счетчиков
10. Бронирование билетов в театр

Пример описания интерфейса приведен далее. Необходимо разработать простейший интернет-магазин. Магазин должен обеспечивать регистрацию новых пользователей, вход зарегистрированных пользователей. Публикацию информации о товарах, а также работу с корзиной и заказами. Объекты (сущности) интерфейса.

USER - пользователь (обеспечивает работу с аккаунтом пользователя)

GOODS - товары (реализует работу с витриной магазина)

BUSKET - корзина (выполняет работу с корзиной зарегистрированного пользователя)

ORDERS - заказы (предоставляет работу с оформленными заказами пользователя)

При работе зарегистрированных пользователей в атрибутах любого запроса передается код авторизации. Отсутствие кода авторизации означает работу анонимного незарегистрированного пользователя.

Детализация параметров работы конкретных запросов выполняется после согласования схемы программного интерфейса с преподавателем. В учебном режиме согласуете с преподавателем разработанную вами таблицу. Результаты разработки интерфейса будут основой для выполнения следующих лабораторных работ. На них будет необходимо реализовать этот интерфейс для разных платформ.

Таблица 3

ReST интерфейс интернет-магазина

Сущность \Операция (Предмет \Действие)	GET	POST	PUT	DELETE
USER	Вход	Корректировка свойств пользователя	Регистрация	Удаление собственного аккаунта
GOODS	Получить список товаров	-	-	-
GOODS /{ID товара}	Получить детальную информацию о товаре с номером {ID товара}	-	-	-
BUSKET	Получить список товаров в корзине	Добавить товар в корзину	Оформить заказ товаров в корзине	Удалить товар из корзины
ORDERS	Получить список заказов	-	-	-
ORDERS /{ID заказа}	Получить информацию о заказе с номером {ID заказа}	-	-	Отменить заказ с номером {ID заказа}

2.2 Реализация тонкого клиента на основе web-интерфейса

Необходимо реализовать на основе JavaScript браузерный тонкий клиент, посылающий запросы на сервер по разработанному вами ранее ReST

интерфейсу. Также нужно сделать заглушку сервера, выполняющую обработку ReST запросов для тестирования работы тонкого клиента.

Описание реализации примера ReST сервера на "чистом" PHP приведено в файле ReST_PHP.docx расположенном в электронной информационной образовательной системе (ЭИОС) на странице дисциплины [1].

Пример реализации фрагмента тонкого клиента показывает отсылку ReST запросов к товарам (GOODS) на "чистом" JavaScript с демонстрацией формы запроса и ответа в отладочном режиме. Исходный код расположен в файле Lab2.zip также в ЭИОС.

Ссылка на один из вариантов пакета для web программирования: https://ospanel.io/download/

Lab2.zip нужно распаковать в одну из папок публикации web сервера. Пример требует web сервер Apach из-за особенностей реализации перенаправления запросов. Первый HTTP запрос от броузера можно отправить на '/get.html' или '/post.html'. Это страницы для демонстрации соответственно GET и POST запросов.

Реализация тонкого клиента обязательна на JavaScript. При реализации серверной заглушки возможно применять различные варианты инструментальных средств.

Основная задача второй лабораторной работы – реализация мобильного тонкого клиента. Во главу угла ставится принцип обеспечения переносимости ПО на стороне клиента на основе браузерных технологий. Также важно обслуживание сервером запросов от клиентов, работающих на разных платформах. В дальнейшем вы будете реализовывать тонкого клиента на мобильных платформах (Android, iOS). Сервер для поддержки этих платформ должен быть один и тот же: одна реализация сервера, много реализаций клиента.

2.3 Реализация тонкого клиента на основе мобильной платформы

Лабораторная работа посвящена разработке тонкого клиента для OS Android с помощью Android Studio. В ЭИОС [1] приведены:

1. Прямая ссылка на загрузку Android Studio.
2. Пример реализации запроса в архиве "AndrRest.rar",
3. Используется сервер из предыдущей лабораторной работы.

Запрос отправляется на корневую точку соединения, указанную в поле url класса MainActivity. Переменная proxy_ip того же класса позволяет перенаправить запрос не на указанный в url домен, а по соответствующему IP адресу. Таким образом, возможно в локальной сети отправлять запросы на интранет web-сервер. В примере подразумевается использование запущенного сервера Apach+PHP, на котором располагается PHP заглушка из предыдущей лабораторной работы. Можно использовать пакет Open Server Panel.

2.4 Требования и состав отчёта

1. Отчёт должен быть выполнен на листе размером А4.
2. Отчёт должен начинаться с титульного листа с названием вуза и факультета, номером и названием лабораторной работы, вариантом, ФИО студента, № группы, ФИО преподавателя, городом и годом.
3. В отчёте нужно кратко описать задание, показать основные этапы вычисления при выполнении всех операций, сформулировать выводы.
4. Отчёт предоставить в бумажном или электронном виде (записать на флэш-накопитель и продублировать себе на электронную почту).

3. ВЫПОЛНЕНИЕ КОНТРОЛЬНОЙ РАБОТЫ

3.1 Общая характеристика контрольной работы

На контрольную работу студенту выдается индивидуальное задание (по вариантам), заключающееся в разработке несложного клиент-

серверного приложения с мобильным и/или веб-клиентом и RESTful веб-сервисом (или несколькими сервисами) в качестве серверной части. Сервисы могут выполнять какие-то задачи из области бизнес-аналитики или искусственного интеллекта, в частности, задачи, связанные с машинным обучением и применением обученных моделей.

Работа выполняется в письменной форме в течение 10 недель с момента выдачи задания. Контрольный срок сдачи – последний месяц семестра.

Примерное содержание контрольной работы :

1. Титульный лист.
2. Формулировка варианта задания.
3. Основная часть, включающая:

1) описание требований к приложению (состав функций),

2) описание используемых средств разработки, технологий, библиотечных функций и классов,

6) архитектура приложения и используемые архитектурные шаблоны

7) описание REST API

8) диаграммы классов программы, диаграммы взаимодействия (если есть),

9) экранные формы работы приложения,

10) коды программы (в приложении).

11). Список использованных источников (включая источники Интернет).

Правила оформления контрольной работы

- контрольная работа оформляется в редакторе MS Word / OpenOffice (*.doc, *.docx, *.odt);
- листы формата А4, ориентация книжная;

- поля: левое – 2 см, остальные – по 1 см;
- шрифт – Times New Roman;
- размер шрифта 14 pt;
- междустрочный интервал – 1,5;
- абзацный отступ – 1,25 см;
- нумерация страниц сквозная, номер на первой странице не ставится;
- в конце работы необходим список использованной литературы согласно ГОСТ Р 7.0.5 – 2008;
- объем работы зависит от степени раскрытия основных пунктов контрольной работы.

Конкретный вариант работы - задача из области искусственного интеллекта (машинное обучение, применение обученных моделей в разных областях, задача обработки естественного языка, система, основанная на знаниях), либо простая система бизнес-аналитики), также задаются различные конкретные варианты реализации каркасов для веб-сервисов, технологий клиентских веб- и мобильных приложений, используемые облачные платформы.

При реализации контрольной работы студент разрабатывает основную функциональную часть (например, небольшую иерархию классов для обработки списка геометрических фигур, как в [2]), описывает ее, а затем реализует ее в виде веб-службы, разрабатывая REST API в соответствии с материалами лабораторной работы 1. Затем реализуется тонкий веб-клиент или мобильный клиент в соответствии с материалами лабораторной работы 2 или 3 (возможна реализация обоих типов клиентов) и в заключении рассматривается размещение всего клиент-серверного приложения, либо только серверной части на облачной платформе, например, в виде веб-функции. Рассмотрим последний вопрос более подробно.

3.2 Создание веб-функций в PaaS

3.2.1 Реализация web функции в AWS

Создадим AWS Lambda с REST-интерфейсом, реализуемым через AWS API Gateway. Для создания и публикации используется AWS Toolkit for Visual Studio с типом проекта AWS Lambda Project.

Для реализации и размещения веб-функций и более сложных примеров в облачных системах можно воспользоваться документацией самих облачных сервисов. Необходимо сначала зарегистрировать аккаунты. Для этого потребуется номер платежной карты (при использовании бесплатных сервисов деньги сниматься не будут, но карта нужна для регистрации) и номер телефона. Нужно внимательно следить за тем, какие сервисы вы используете, чтобы не тратить деньги на учебные задачи! Для создания веб-функций и интерфейса к ним обычно тарификация не требуется. В AWS предоставляется доступ к основным функциям на год, в Yandex – на 2 месяца. При использовании instances (виртуальных машин), если они нужны, нужно выбирать мнимальные конфигурации. В AWS также необходимо настроить права доступа (роли). При настройке API Gateway для Lambda лучше использовать proxy. Для использования типа `APIGatewayProxyRequest` необходимо скачать и установить библиотеку Amazon.Lambda.APIGatewayEvents.

Ниже приведен код примера лямбда-функции. Ей передаются парметры id, а также shape, a, b и c (строки с параетрами неких геометрических фигур). По id определяется, с какими параметрами вызвать метод объекта класса Report, генерирующего отчет. Результат отчета – строка в поле ReportText.

```
// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly: LambdaSerializ-
er(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
```

```
namespace AWS_ShapesLambda
{
    public class Responce
    {
        public int statusCode { get; set; }
        public String body { get; set; }
    }

    public class Function
    {
        /// <summary>
        /// A simple function that takes a string and does a ToUpper
        /// </summary>
        /// <param name="request"></param>
        /// <param name="context"></param>
        /// <returns></returns>
        public Responce FunctionHandler(APIGatewayProxyRequest request,
                  ILambdaContext context)
        {
            string input = request.QueryStringParameters["ID"];

            Report rep = new Report();

            int id = Int32.Parse(input);
             if (id != 3)
            {
                rep.ShowReport(id);
            }
            else
            {
                List<string> lst = new List<string>();
                lst.Add(request.QueryStringParameters["shape"] + ";" +
                    request.QueryStringParameters["a"] + ";" +
                    request.QueryStringParameters["b"] + ";" +
                    request.QueryStringParameters["c"]);
                rep.ShowReport(lst);
            }

            Responce res = new Responce();
            res.statusCode = 200;
            res.body = rep.ReportText;
```

```
            return res;
        }
    }
```

После успешной компиляции необходимо указать проект с лямбда-функцией и выбрать пункт контекстного меню Publish to AWS Lambda… (еще раз – требуется установка AWS Toolkit for Visual Studio). Откроется окно, показанное на рисунке 13.

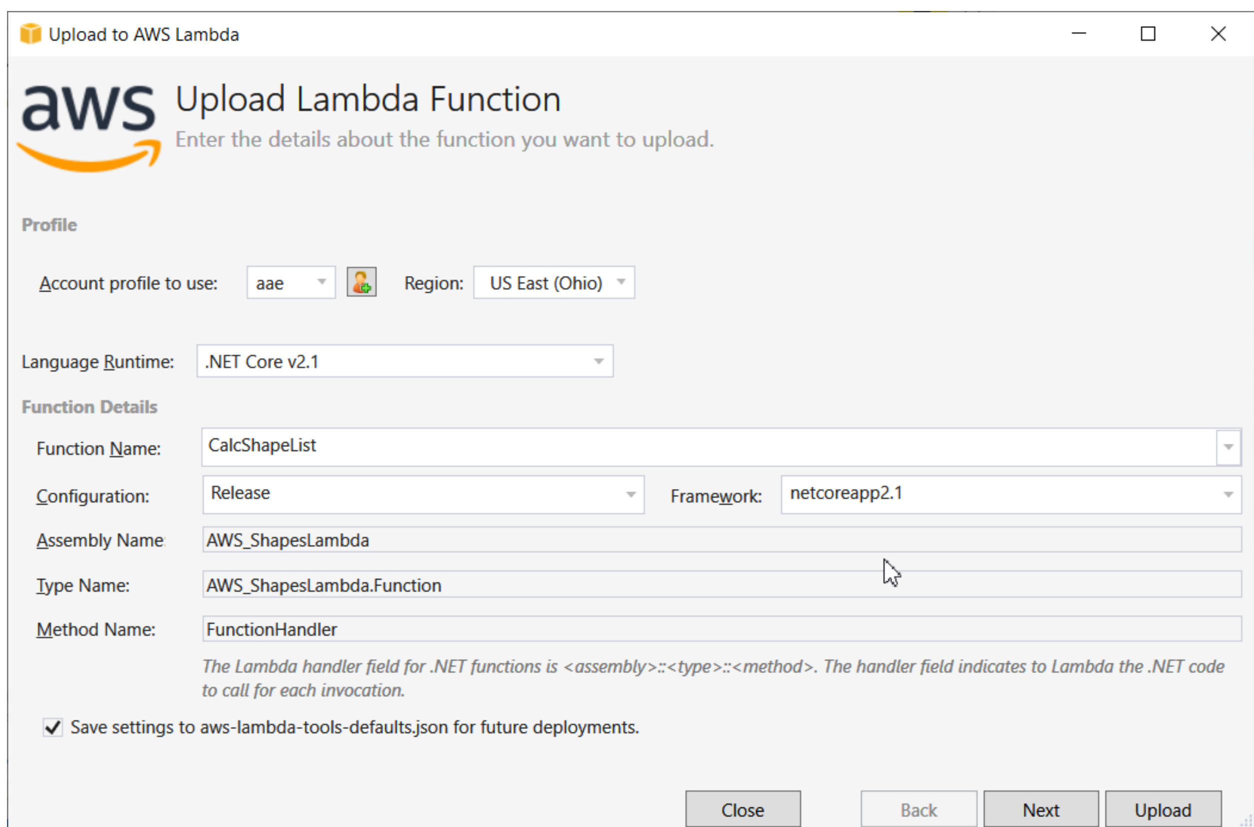


Рис. 13. Окно публикации Lambda функции в AWS из Visual Studio

Необходимо отметить, что сначала нужно настроить свой аккаунт, указать там предпочитаемый регион, настроить роли, также необходимо будет настроить роли для самой функции и API Gateway, для чего, возможно, придется повозиться с документацией к AWS. После публикации можно протестировать функцию непосредственно в Visual Studio, либо в консоли AWS (см. рис. 14-15).

Тестировать функцию несложно, но в данном случае из-за применения в функции параметра типа APIGatewayProxyRequest сама строка с запросом будет несколько громоздкой (но при этом легче создать REST-сервис), поэтому тут ее приводить не будем. Сам REST веб-сервис настраивается через AWS Api Gateway визуально.

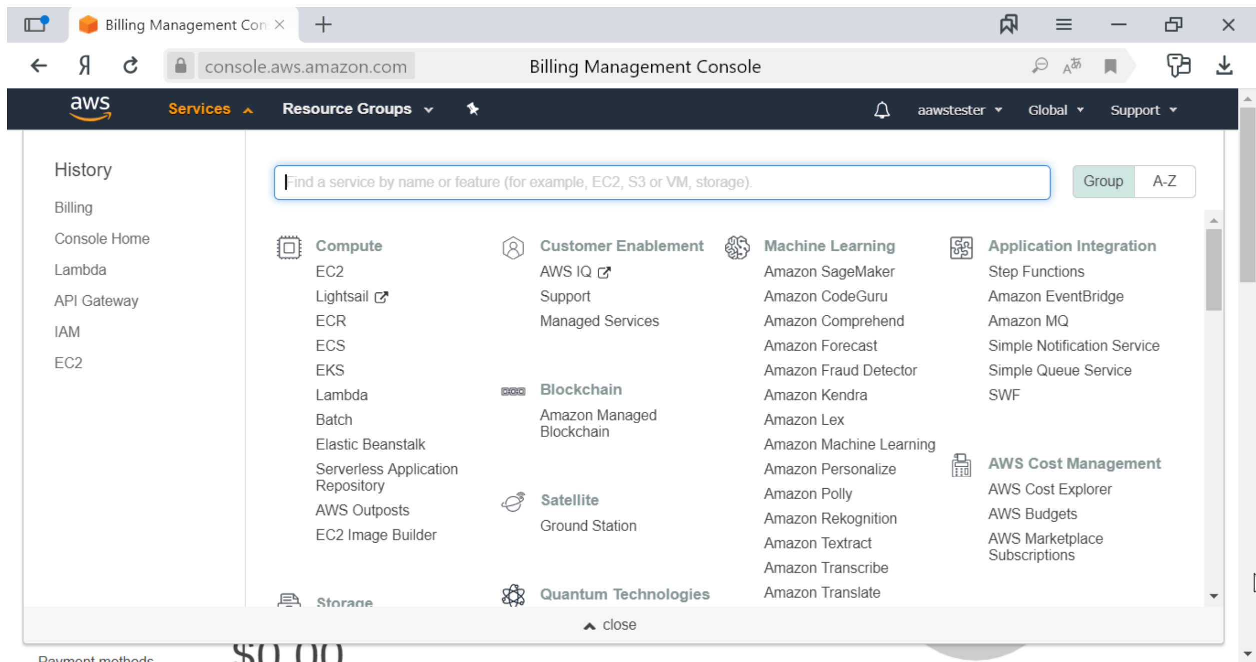


Рис. 14. Список сервисов в панели AWS

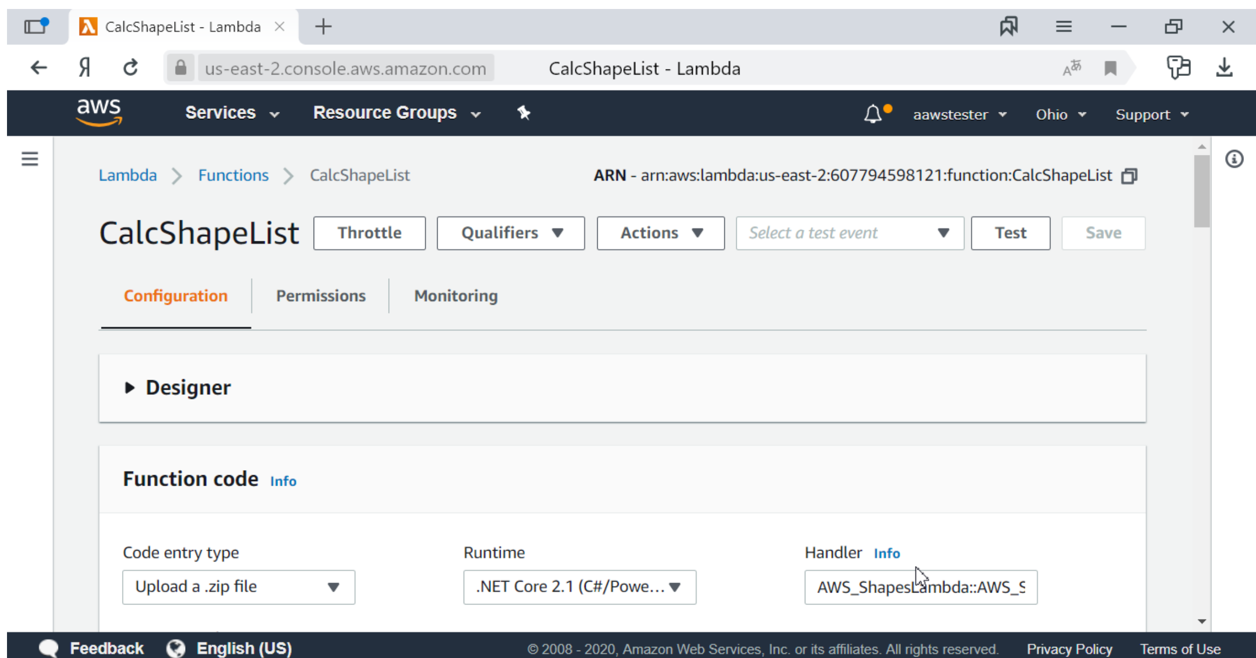


Рис. 15. Lambda функция в панели AWS

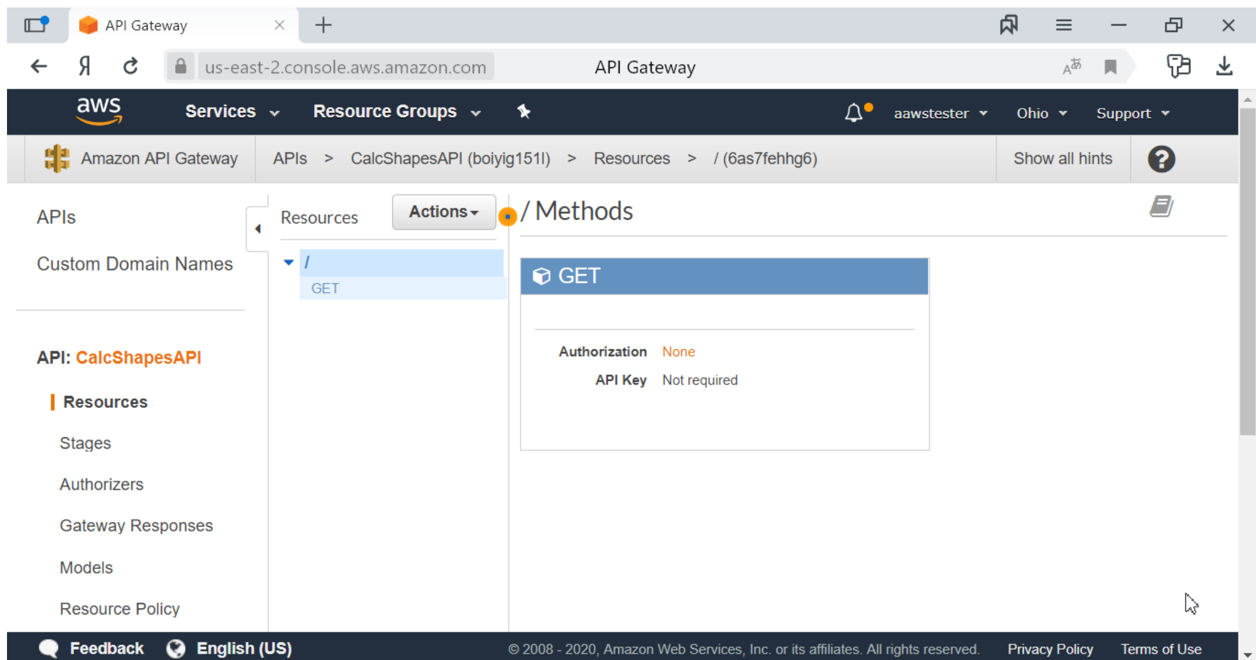


Рис. 16. Новый REST API для лямбда-функции

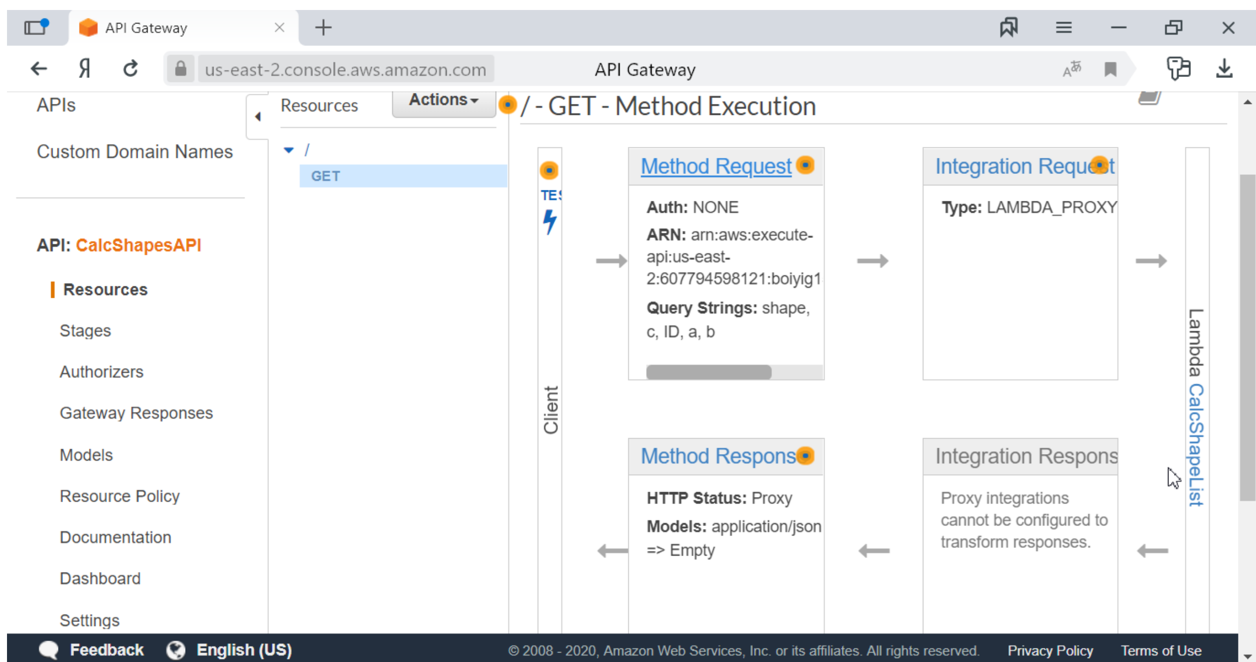


Рис. 17. Настройка выполнения метода GET

Создадим API Gateway для доступа к лямбде с помощью запроса GET (см. рис. 16). На рисунке 17 показаны компоненты API с Integration request типа LAMBDA_PROXY (более простой вариант для создания).

Фактически нам потребуется только создать список специфических параметров нашего запроса с именами a, b, c, ID и shape (см. код самой лямбды выше).

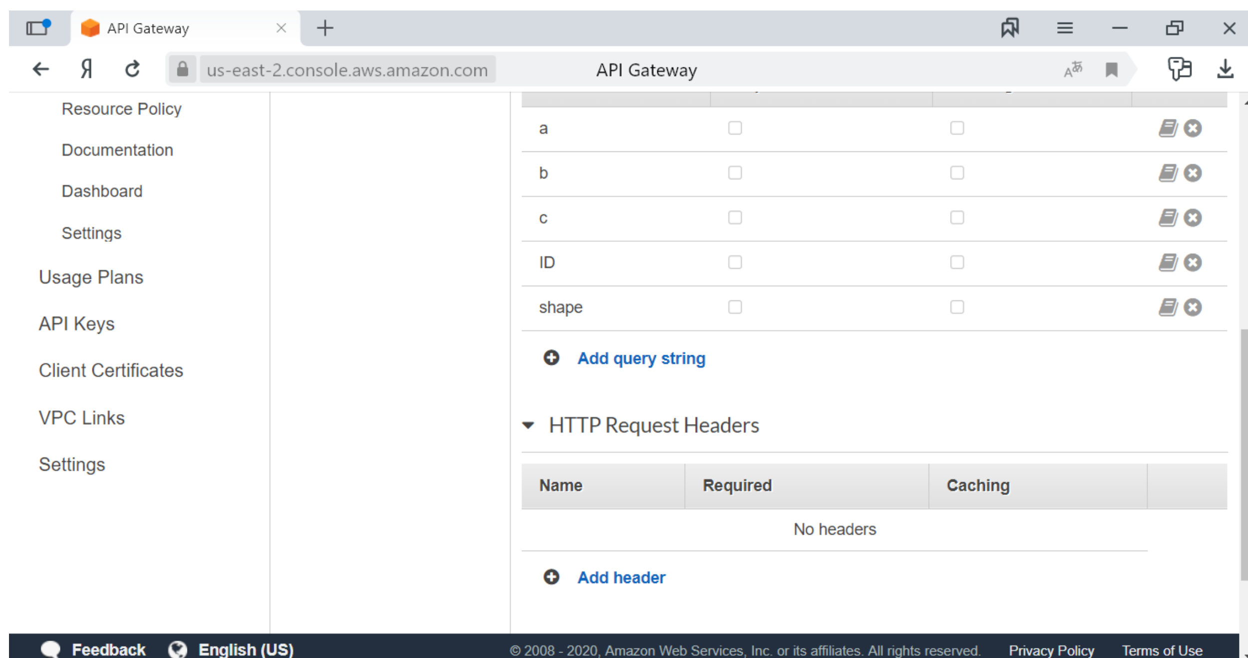


Рис. 18. Настройка списка параметров GET

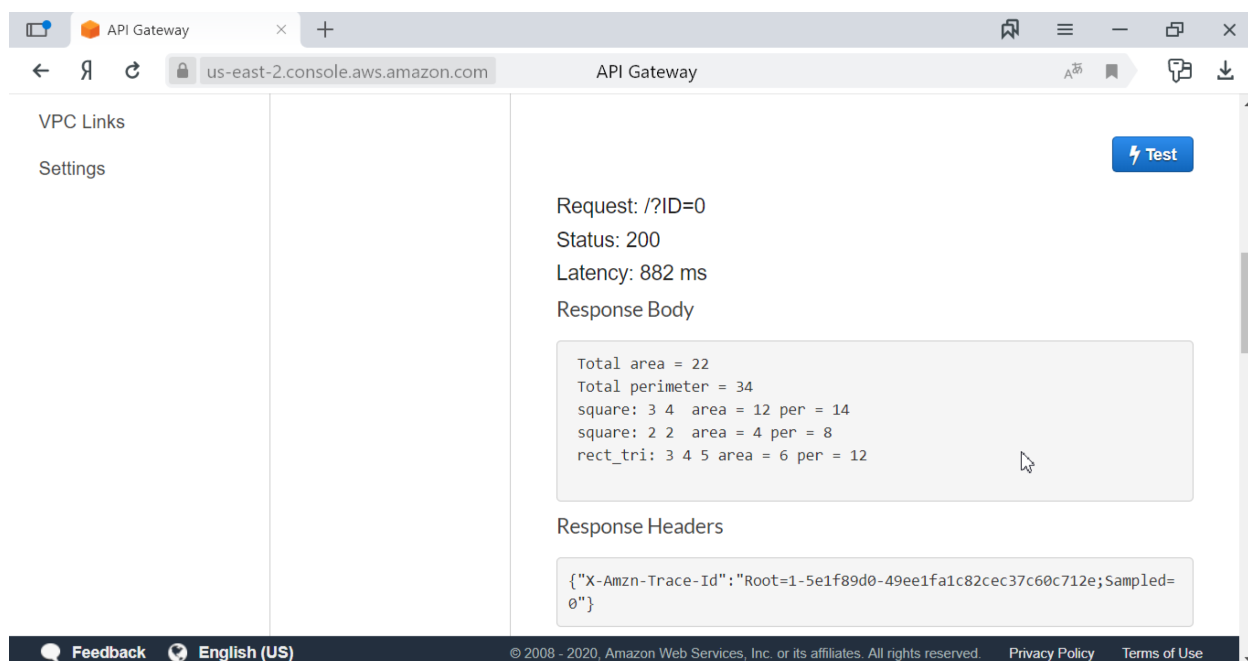


Рис. 19. Тестирование API

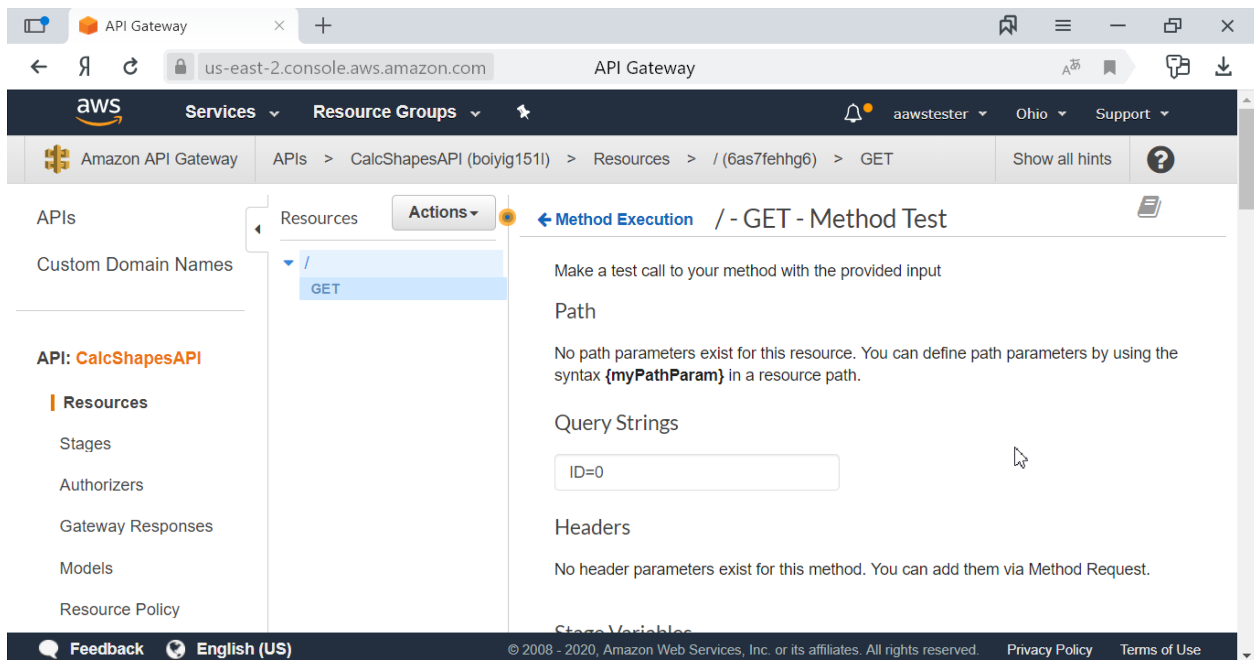


Рис. 20. Параметры запроса при тестировании в панели

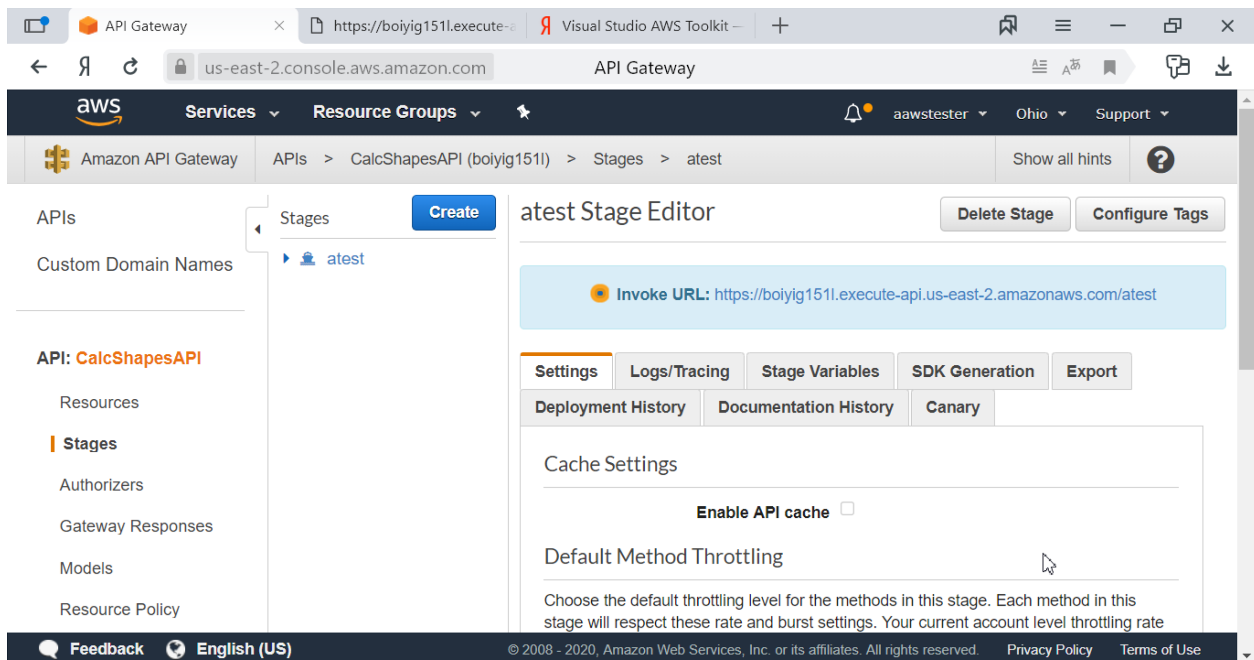


Рис. 21. Результаты развертывания сервиса с REST API

После создания REST API (опять же может потребоваться настройка ролей для запуска функции через API) его можно протестировать и по-

смотреть ответ сервиса вместе с заголовками (см. рис. 19), нажав кнопку Test и указав параметр ID=0 в окне Query String (см. рис. 20).

Далее необходимо выполнить развернуть сам сервис API, выбрав действие Deploy API, доступное по кнопке Actions. Там нужно будет указать этап поставки (тест, product и другие), комментарий и выполнить развертывание. В результате станет доступна строка для вызова сервиса из браузера (см. рис. 21).

Вызов функции через веб-сервис из браузера (для параметра ID=0):

https://boiyig151l.execute-api.us-east-2.amazonaws.com/atest?ID=0 (ссылка для конкретной функции приведена для примера, использовать ее не нужно).

Вид ответа в браузере приведен на рисунке 22.

Аналогично можно вызвать функцию с другими параметрами, например, с запросом на расчет конкретной фигуры :

```
https://boiyig151l.execute-api.us-east-
2.amazonaws.com/atest?ID=3&shape=1&a=3&b=4&c=0
```

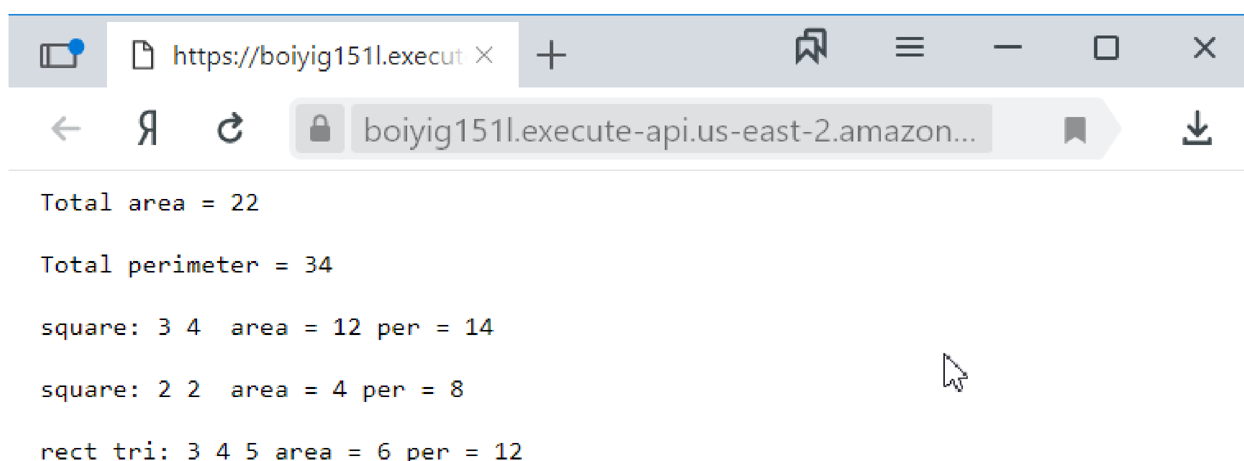


Рис. 22. Ответ AWS Lambda на запрос через REST

3.2.2 Реализация веб-функции на Python для Yandex.Cloud

Также была создана аналогичная функция для примера на Pyhon для новой облачной PaaS платформы от Yandex – Yandex Cloud. В ней также поддерживается создание облачных функций, причем они сразу имеют веб-интерфейс в виде REST API. За последние годы функциональность облака Яндекс (запущенного в 2019 году) сильно расширилась, в частности, там добавлена поддержка различных языков (ранее это был только Python и JS, теперь также C#, Java, появилось много новых опций в самом облаке).

Был создан код для Cloud Function:

```
from controller import *

def shapes(event, context):
    param = ''
    if 'queryStringParameters' in event and 'param' in event['queryStringParameters']:
        param = event['queryStringParameters']['param']
    sh_handler = ShapeHandler()
    sh_handler.do_calc(param)
    return {
        'statusCode': 200,
        'headers': {
            'Content-Type': 'text/plain'
        },
        'isBase64Encoded': False,
        'body': 'Results:' + sh_handler.get_text()
    }

class ShapeHandler:
    def __init__(self):
        self.ctrl = Controller()
        self.text = 'Hello from ShapeHandler !'

    def do_calc(self, src):
        self.ctrl.model.addCallback(self.log_totals)
```

```
        self.ctrl.get_total_results_by_id(2, src)

    def log_totals(self, area, perimeter):
        self.text=f' area = {area}; perimeter = {perimeter}'
        return self.text

    def get_text(self):
        return self.text
```

К этой функции можно непосредственно обратиться через веб. Тогда в браузере увидим ответ:

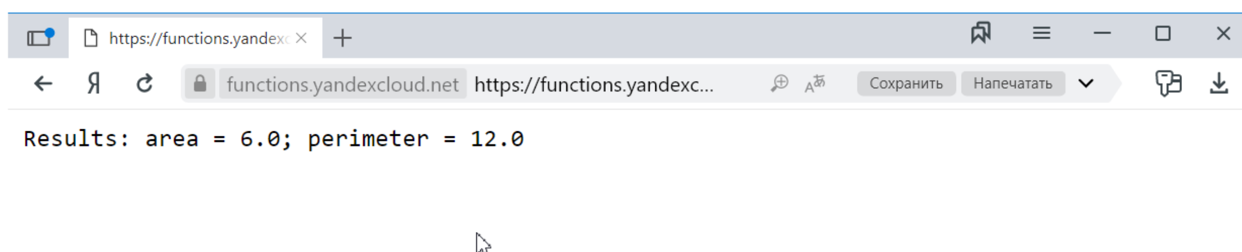


Рис. 23. Ответ Yandex Cloud функции на REST запрос

3.2.3 Реализация веб-функции в Azure

Рассмотрим создание веб-функции с REST API в Azure из Visual Studio 2017.

В данном случае не требуется дополнительно ничего устанавливать, если при установке Visual Studio Вы указали инструменты Azure.

Создаем проект функции Azure (рисунок 24) и далее выбираем Http Trigger.

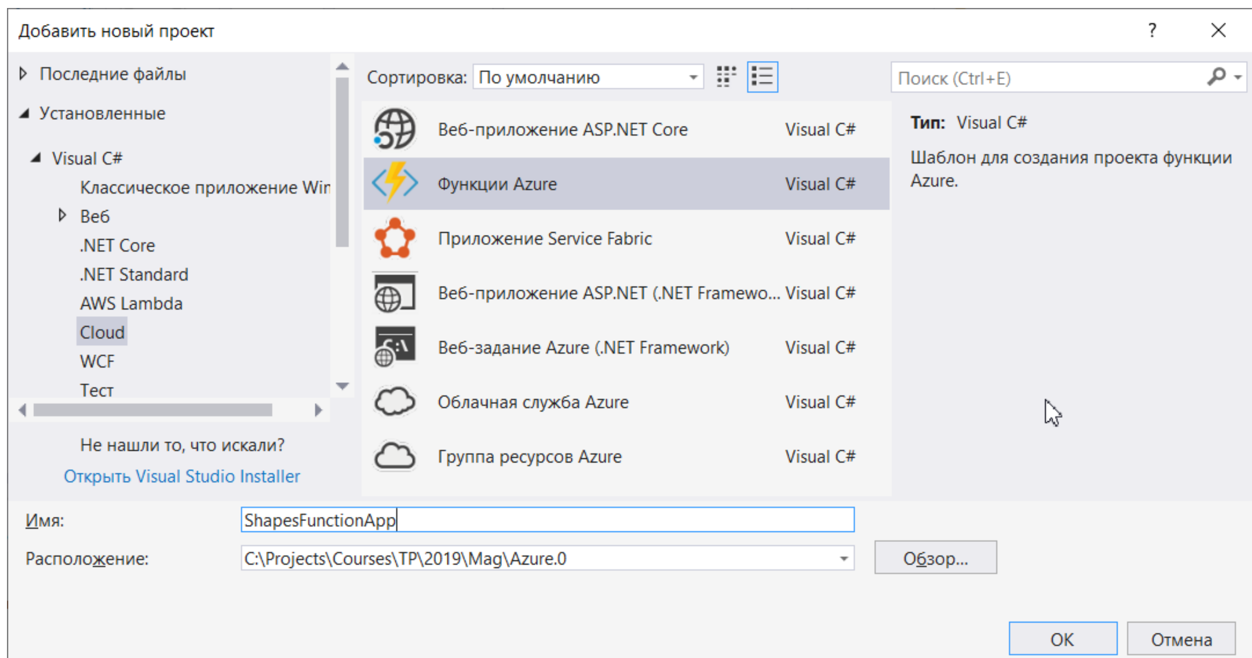


Рис. 24 Создание проекта функции Azure

Затем создаем код самой функции, в целом аналогичный рассмотренному выше для AWS (используется запрос GET):

```
public static class ShapesFunction
{
    [FunctionName("ShapesFunction")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post",
    Route = null)] HttpRequest req,
        ILogger log)
    {
        string input = req.Query["ID"];
        Report rep = new Report();
        int id = Int32.Parse(input);
        if (id != 3)
        {
            rep.ShowReport(id);
        }
        else
        {
            List<string> lst = new List<string>();
            lst.Add(req.Query["shape"] + ";" +
                req.Query["a"] + ";" +
```

```
                req.Query["b"] + ";" +
                req.Query["c"]);
            rep.ShowReport(lst);
        }

        string requestBody = await new
             StreamReader(req.Body).ReadToEndAsync();
        dynamic data = JsonConvert.DeserializeObject(requestBody);

        return (ActionResult)new OkObjectResult(rep.ReportText);
    }
}
```

После компиляции и сборки проекта, если веб-функция установлена как проект по умолчанию, ее можно запустить на локальном компьютере во встроенном веб-сервере и протестировать через браузер, о чем сообщает после запуска текст в консоли (рис. 25), отладка поддерживается:

```
C:\Users\Andrei\AppData\Local\AzureFunctionsTools\Releases\2.43.0\cli\func.exe
[15.01.2020 19:35:26] Starting JobHost
[15.01.2020 19:35:26] Starting Host (HostId=desktope77tglc-2114454069, InstanceId=aad9a3d0-50a4-4121-87f7-38971d49fed2, Version=2.0.12888.0, ProcessId=6540, AppDomainId=1, InDebugMode=False, InDiagnosticMode=False, FunctionsExtensionVersion=(null))
[15.01.2020 19:35:26] Loading functions metadata
[15.01.2020 19:35:26] 1 functions loaded
[15.01.2020 19:35:26] Generating 1 job function(s)
[15.01.2020 19:35:26] Found the following functions:
[15.01.2020 19:35:26] ShapesFunctionApp.Function1.Run
[15.01.2020 19:35:26]
[15.01.2020 19:35:26] Initializing function HTTP routes
[15.01.2020 19:35:26] Mapped function route 'api/Function1' [get,post] to 'Function1'
[15.01.2020 19:35:26]
[15.01.2020 19:35:26] Host initialized (548ms)
[15.01.2020 19:35:26] Host started (560ms)
[15.01.2020 19:35:26] Job host started
Hosting environment: Production
Content root path: C:\Projects\Courses\TP\2019\Mag\Azure.0\ShapesFunctionApp\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

        Function1: [GET,POST] http://localhost:7071/api/Function1

[15.01.2020 19:35:32] Host lock lease acquired by instance ID '000000000000000000000000117404C7'.
```

Рис. 25 Запуск веб-функции для тестирования на локальной машине

Протестировать функцию можно в браузере, указав выведеный выше в окне путь (рис. 26)

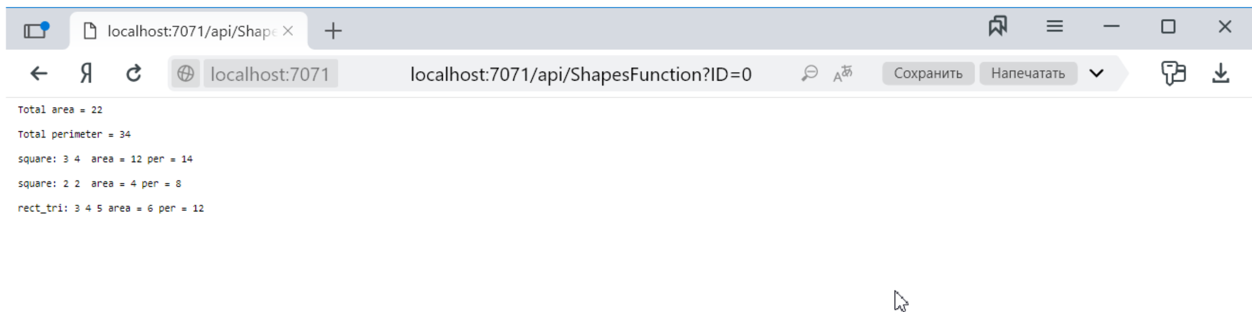


Рис. 26. Запуск веб-функции в браузере на локальном сервере

Далее можно создать веб-функцию в Azure, выбрав в контекстном меню проекта пункт Опубликовать (вот тут как раз потребуется аккаунт Azure !) – рисунок 27 и далее – рисунок 28.

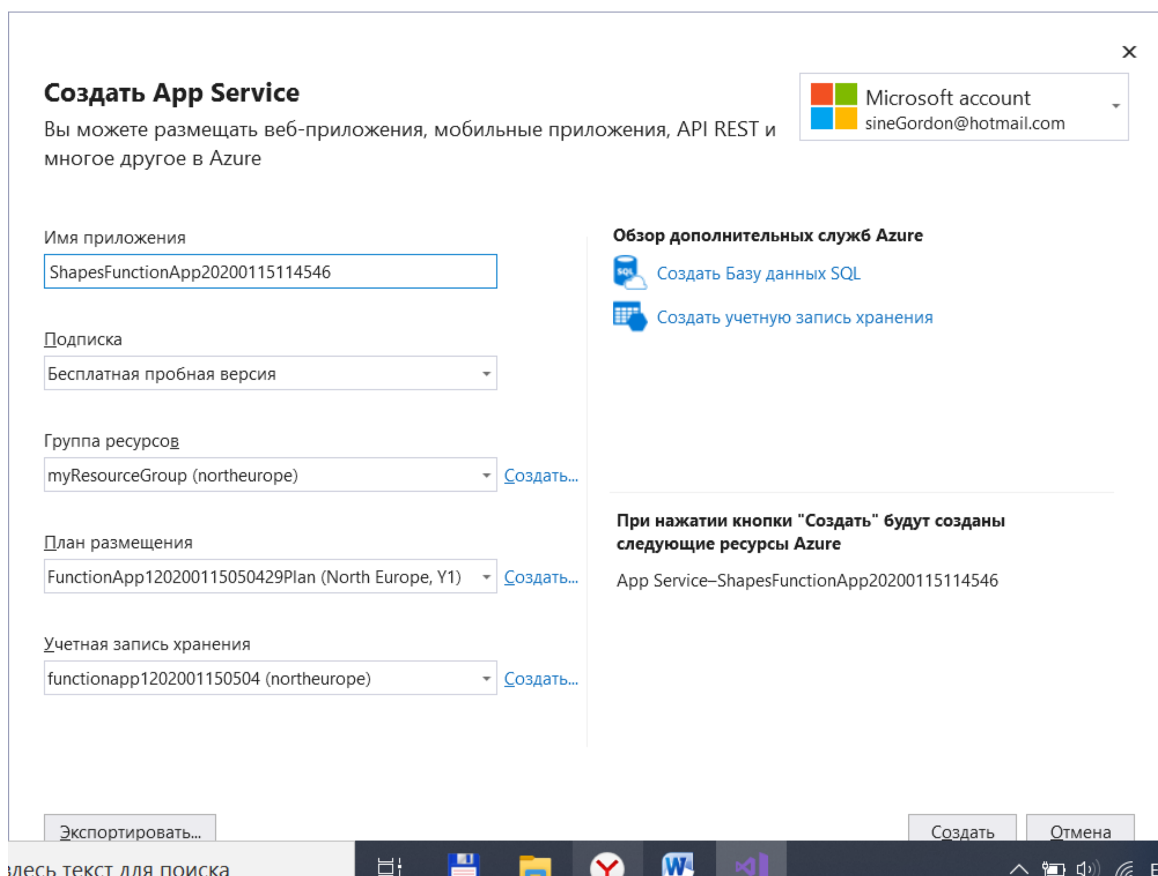


Рис. 27 Публикация функции в Azure

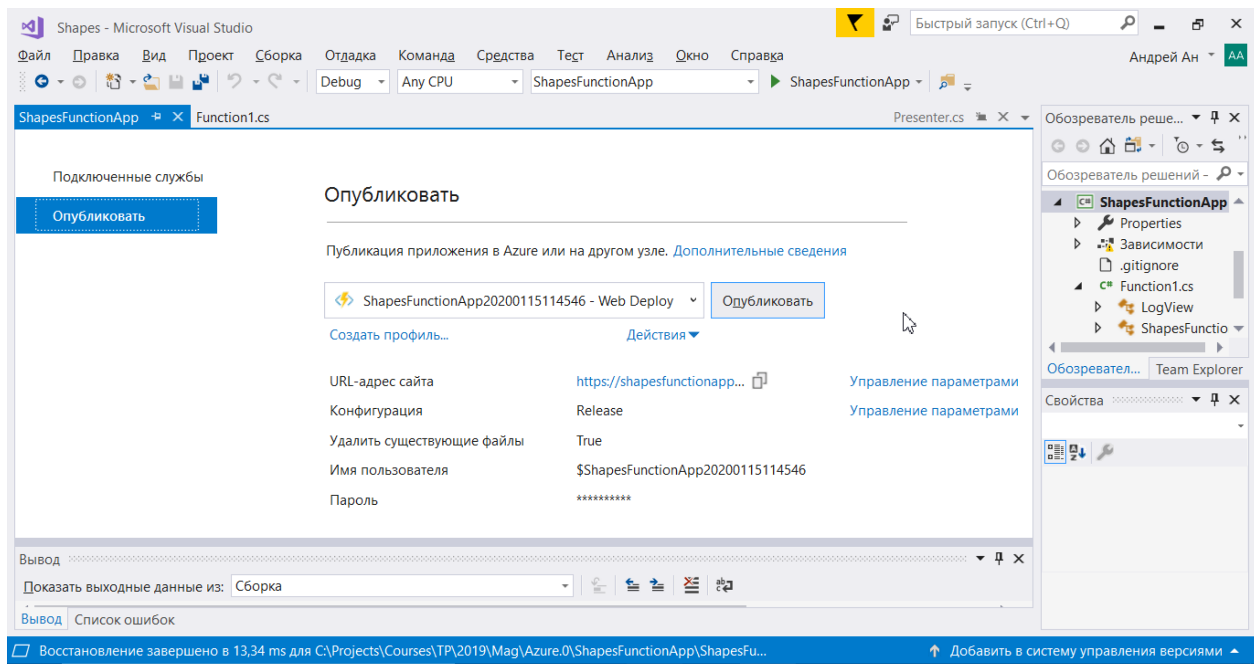


Рис. 28. Публикация функции в Azure (завершение)

Далее выбрав указанный на странице URL-адрес сайта и указав параи-етры (/api/ShapesFunction?ID=0) по идее мы должны вызвать функцию, однако при указании такого URL и параметров мы полчаем ошибку, а при выборе просто указанного URL – информационную страницу о работе функции (рис. 29):

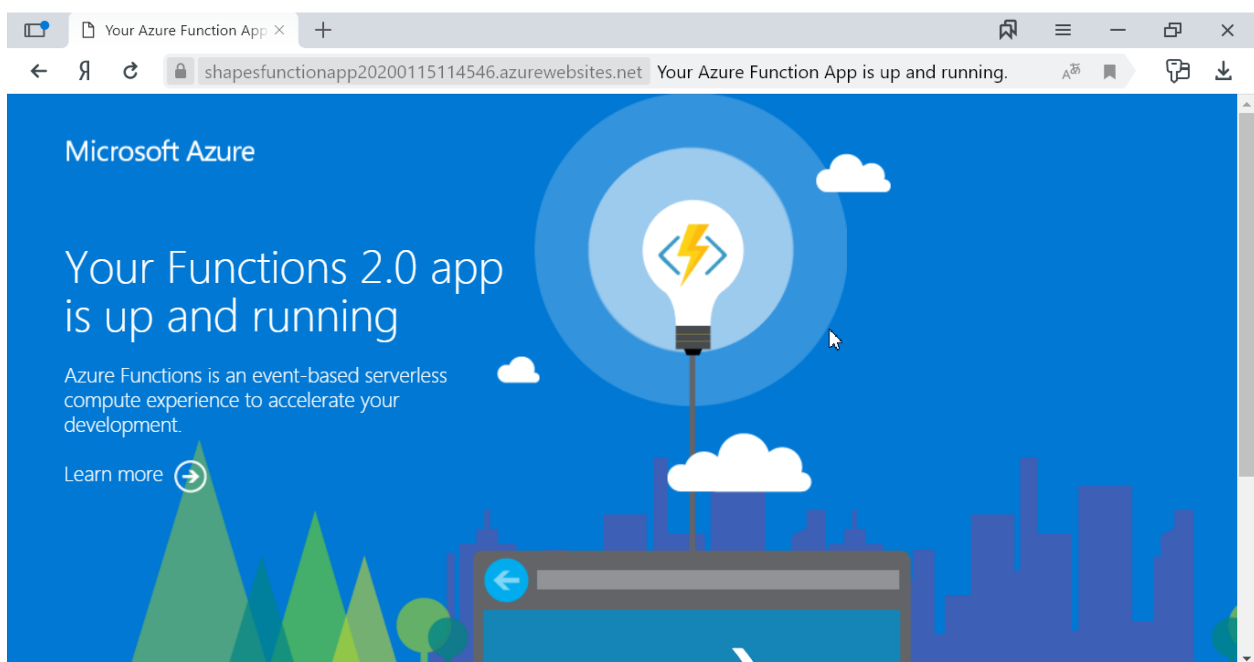


Рис. 29. Информация о работе приложения при выборе его URL

Подобное поведение функции не соответствует документации, возможно, это связано с использованием VS2017, а не VS2019, либо с конфигурацией VS / ПК. Для получения реальной ссылки с дополнительными параметрами мы можем зайти в консоль (портал) Azure в свой кабинет, найти функцию и там нажать на Получить URL-адрес функции (рисунок 30), скопировать и сохранить строку запроса.

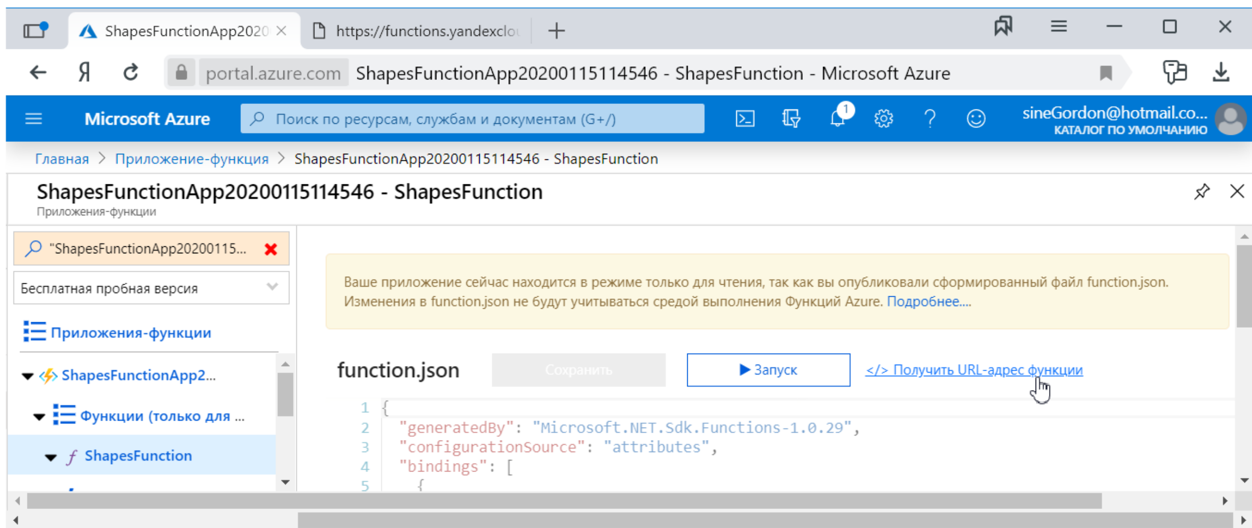


Рис. 30. Получение URL веб-функции на портале Azure

Для вызова нашей веб-функции нужно указать строку :

https://shapesfunctionapp20200115114546.azurewebsites.net/api/ShapesFunction?code=8SPvE4JqNI1K9LwLgTWZsOLK51GOwBvpzPExeHNrAkHD55qEr/VWAg==&ID=0 (как и ранее эта строка приводится для примера).

Ответ функции в браузере показан на рисунке 31

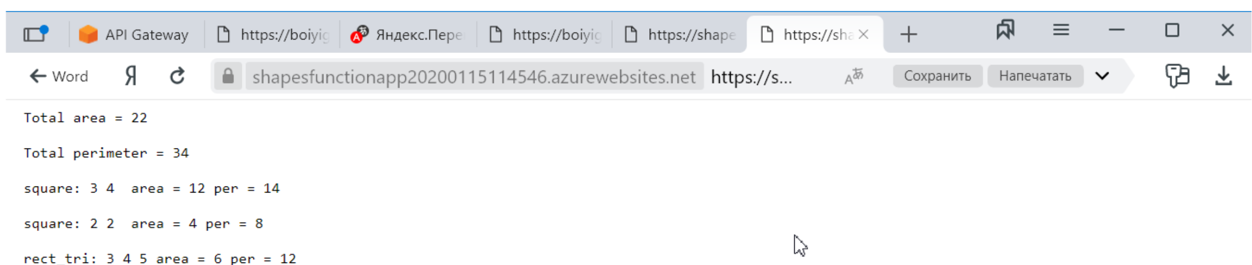


Рис. 31. Ответ функции в браузере (ID=0)

ЗАКЛЮЧЕНИЕ

Решение сложных задач из различных предметных областей ведет к привлечению разных инструментов для создания программного кода. Основу сложных приложений, как правило, составляет многоуровневая программная архитектура. Разработка такой архитектуры требует глубоких знаний существующих архитектур, их анализа и модификации для достижения наиболее оптимальной структуры кода. Это связано как с реализацией гибкой легко масштабируемой кроссплатформенной сетевой инфраструктуры, в том числе на основе облачных технологий, так и необходимостью сочетать уже имеющиеся наработки в виде готовых компонентов. В любом случае архитектурные вопросы являются фундаментальными при разработки сложных программных систем.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Курс «Мобильные и сетевые технологии» [Элктронный ресурс] . — Режим доступа https://eos2.vstu.ru/course/view.php?id=130 (для авторизованных пользователей).
2. GitHub - aae-at-vstu [Электронный ресурс]. Режим доступа : https://github.com/aae-at-vstu/tp_shapes_sample
3. Заяц, А. М. Проектирование и разработка WEB-приложений. Введение в frontend и backend разработку на JavaScript и node.js : учебное пособие / А. М. Заяц, Н. П. Васильев. — 2-е изд., стер. — Санкт-Петербург : Лань, 2020. — 120 с. — ISBN 978-5-8114-5278-1. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/139286 (дата обращения: 27.10.2021). — Режим доступа: для авториз. пользователей.
4. Джош, Л. Современный PHP. Новые возможности и передовой опыт / Л. Джош ; перевод с английского Р. Н. Рагимов. — Москва : ДМК Пресс, 2016. — 304 с. — ISBN 978-5-97060-184-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/93269 (дата обращения: 27.10.2021). — Режим доступа: для авториз. пользователей.
5. Умрихин, Е. Д. Основы разработки iOS-приложений на C# с помощью Xamarin : учебное пособие для вузов / Е. Д. Умрихин. — Санкт-Петербург : Лань, 2021. — 384 с. — ISBN 978-5-8114-6930-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/173095 (дата обращения: 27.10.2021). — Режим доступа: для авториз. пользователей.
6. Черников, В. Разработка мобильных приложений на C# для iOS и Android : учебное пособие / В. Черников. — Москва : ДМК Пресс, 2020. — 188 с. — ISBN 978-5-97060-805-0. — Текст : электронный // Лань :

электронно-библиотечная система. — URL: https://e.lanbook.com/book/140592 (дата обращения: 27.10.2021). — Режим доступа: для авториз. пользователей.

7. Сейерс, Э. Х. Docker на практике / Э. Х. Сейерс, А. Милл ; перевод с английского Д. А. Беликов. — Москва : ДМК Пресс, 2020. — 516 с. — ISBN 978-5-97060-772-5. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/131719 (дата обращения: 27.10.2021). — Режим доступа: для авториз. пользователей.
8. Архитектурные решения информационных систем : учебник / А. И. Водяхо, Л. С. Выговский, В. А. Дубенецкий, В. В. Цехановский. — 2-е изд., перераб. — Санкт-Петербург : Лань, 2021. — 356 с. — ISBN 978-5-8114-2556-3. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/167464 (дата обращения: 27.10.2021). — Режим доступа: для авториз. пользователей.

У ч е б н о е   и з д а н и е

Михаил Андреевич **Кузнецов**
Андрей Евгеньевич **Андреев**

Разработка распределенных кроссплатформенных программных систем

Учебно-методическое пособие

Редактор *Л. Н. Рыжих*

Темплан 2021 г. (учебники и учебные пособия). Поз. № 19.
Подписано в печать 00.00.2021. Формат 60x84 1/16. Бумага газетная.
Гарнитура Times. Печать офсетная. Усл. печ. л. 4,0. Уч.-изд. л. 5,05.
Тираж 100 экз. Заказ

Волгоградский государственный технический университет.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 1.

Отпечатано в типографии ИУНЛ ВолгГТУ.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 7.