СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
Часть 1. Архитектура распределенных сервисов 7
1.1. Разновидности сетевых приложений 7
1.1.1. Определение. 7
1.1.2. Варианты расположения приложений. 7
1.1.3.Средства, инструменты, спецификации для построения и сопровождения приложений. 11
1.1.4. Современные решения 19
Контрольные вопросы 20
1.2. Распределенное приложение базы данных 21
1.2.1. Принципы создания системы обработки информации в масштабе предприятия…. 21
1.2.2. Основные уровни архитектуры распределенного приложения 22
1.2.3. Расширение базовых уровней 24
Контрольные вопросы. 24
1.3. Сервис-ориентированные архитектуры (SOA) 25
1.3.1. Сервис-ориентированные архитектуры: элементы и характеристики 25
1.3.2. Общая схема SOA 26
1.3.3. Сервис-ориентированный анализ и проектирование. 28
1.3.4. Концепция управления распределенными службами 31
Контрольные вопросы. 33
Часть 2. Облачная обработка данных как концепция 34
2.1. Основные понятия и модели расположения приложений 34
2.1.1. Понятие Gloud computing 34
2.1.2. Характеристики облачных вычислений 34
2.1.3. Три модели расположения приложений 35
2.1.4. Модель облачных вычислений 36
Контрольные вопросы. 38
2.2. Сервисы, представляемые облаком 38
2.2.1. Облачные вычисления и предоставляемые ими сервисы. 38
2.2.2. Границы управляемости. Модели развертывания собственной инфраструктуры 39
2.2.3. Проблема создания неконтролируемых данных 43
Контрольные вопросы. 45
2.3. Модели обслуживания 45
2.3.1. Программное обеспечение как услуга (SaaS) 45
2.3.2. Платформа как услуга (PaaS) 46
2.3.3. Инфраструктура как услуга (IaaS) 46
Контрольные вопросы. 47
Часть 3. Архитектура web-сервисов 48

3.1. Web-ориентированная архитектура (WOA) 48
3.1.1. Устройство web-приложений 48
3.1.2. Службы web. 51
3.1.3. Протокол SOAP. 53
3.1.4. Язык описания Web-служб WSDL. 58
3.1.5. Пространство имен XML …. 60
Контрольные вопросы. 67
3.2. Аспектно-ориентированное и функциональное программирование. . 67
3.2.1. Основные понятия и определения. 67
3.2.2. Разделение функциональности. 70
3.2.3. F# - типовые приемы функционального программирования....
Реализация web-приложений на F#. 71
Контрольные вопросы. 75
Итоговый тест…. 76
Литература 80

ВВЕДЕНИЕ

Настоящее учебное пособие строится на материале, который читается для студентов специальности 230101 по дисциплине «Интернет-технологии в ГА» и направления 230100 (бакалавриат) «Разработка Internet-приложений».

Большинство задач, которые приходится решать программисту в повседневной жизни, связано с выполнением локальных действий в системе (управлением файловой системой, клавиатурой, экраном). Для координации работы множества компонентов приложения, в том числе обмен данными и управление вводом-выводом, а также обеспечения максимальной производительности, необходимо владеть навыками сетевого программирования, позволяющего взаимодействовать нескольким программам, работающим на разных компьютерах, подключенных к сети.

Архитектура, основанная на сервисах, - это новый шаг в направлении комплексного решения проблемы создания распределенных приложений, выполняющихся на многих компьютерах. Архитектура, основанная на сервисах (service oriented architecture), отличается от архитектуры, основанной на объектах (object oriented architecture), тем, что при необходимости выполнить какие-то действия пользователь просто идентифицирует контракт, описывает, как выглядят данные, имеет возможность динамически переключаться между сервисами, реализующими нужную функциональность, и выбирать себе сервис из нескольких доступных. При этом пользователя не интересует, как именно реализована функциональность сервиса.

Первая волна Web-сервисов и соответствующие стандарты - SOAP, UDDI, WSDL позволяли создавать полнофункциональные интероперабельные приложения. Вторая волна Web-сервисов должна привести к появлению новой модели программирования, которая позволит использовать много различных инструментов и продуктов для создания приложений, одна часть которых функционирует под управлением J2EE-серверов приложений, а другая - под управлением Windows или .NET.

Если раньше представление о Web-сервисах было во многом ограничено предположением об обязательном применении протокола HTTP, широко распространенного и очень популярного, то сейчас рассматриваются возможности доставки данных с применением других протоколов, более подходящих для решения ряда задач с точки зрения надежности.

Облачные вычисления (cloud computing) - новая парадигма, предполагающая распределенную и удаленную обработку, и хранение данных. Термин «облако» используется как метафора, основанная на изображении Интернета на схемах компьютерных сетей, или как образ сложной инфраструктуры, за которой скрываются все технические детали. Облако есть не что иное, как некий крупный дата-центр (или сеть взаимосвязанных между собой серверов), в котором могут храниться файлы и совершаться все вычислительные операции. Это автоматически снимает проблемы с

производительностью компьютера и количеством свободного места на жестком диске.

В данном учебном пособии, которое разделено на три части, рассматривается архитектура распределенных сервисов, приводятся варианты расположения приложений, рассмотрены средства, инструменты, спецификации для построения и сопровождения приложений, приведены элементы и характеристики сервис-ориентированной архитектуры (SOA).

Во второй части данного пособия рассмотрена концепция облачной обработки данных, приведены характеристики облачных вычислений (в соответствии с NIST); модели расположения приложений: on premises, hosting (ASP), cloud, а также рассмотрены модели обслуживания: программное обеспечение как услуга (SaaS), платформа как услуга (PaaS), инфраструктура как услуга (IaaS).

В третьей части приведена архитектура web-сервисов (WOA), устройство web-приложений и интерфейсные методы web-служб. В следующих разделах третьей части рассмотрены вопросы аспектно-ориентированного и функционального программирования.

Пособие рассчитано на студентов специальности 230101 и направления 230100, а также слушателей высших учебных заведений, обучающихся по техническим дисциплинам. Может быть использовано при проведении практических занятий по дисциплине «Разработка Internet-приложений», выполнении выпускной квалификационной работы, а также при самостоятельном решении задач разработки современных приложений.

ЧАСТЬ 1. АРХИТЕКТУРА РАСПРЕДЕЛЕННЫХ СЕРВИСОВ

1.1. Разновидности сетевых приложений

1.1.1. Определение

Одной из составляющих программного обеспечения вычислительных сетей является сетевое (распределенное) приложение. Все сетевые службы, включая файловую службу, службу печати, электронной почты, службу удаленного доступа, Интернет-телефонию, по определению относятся к классу распределенных приложений. Действительно, любая сетевая служба включает в себя клиентскую и серверную части, которые могут и обычно выполняются на разных компьютерах.

Под термином *Распределенное приложение (Distributed application)* понимается приложение, которое выполняется в среде распределенных вычислений, модули такого приложения могут выполняться на разных вычислительных системах.

Под *распределенными вычислениями* понимается способ выполнения расчетов путем их разделения между множеством компьютеров, которые разбросаны по всему миру. Идея заключается в том, что любую задачу, требующую большого объема вычислений, можно поделить на много маленьких частей, каждая их которых окажется по зубам любой домашней машине.

Распределенная обработка данных (Didtributed Data Processing) - методика выполнения прикладных программ группой систем, при этом пользователь получает возможность работать с сетевыми службами и прикладными процессами, расположенными в нескольких взаимосвязанных абонентских системах.

1.1.2. Варианты расположения приложений

Существует два типа сетевых приложений: чисто сетевые (pure) и обособленные (standalone). Чисто сетевые приложения разработаны для применения в сетях, и на отдельных компьютерах использовать не имеет смысла. Наоборот, обособленные приложения призваны работать на отдельном компьютере. Для расширения возможностей они перестроены для работы в сетях. Примерами обособленных приложений могут служить текстовый процессор и редактор электронных таблиц.

Чисто сетевые приложения(pure)

Эти приложения были созданы для использования возможностей сетей. Каждое из них имеет свой отдельный пользовательский интерфейс и требует выполнения некоторой последовательности "сетевых" команд, индивидуальных для каждого приложения. К чисто сетевым приложениям относятся:

- эмуляция терминала;
- передача файла;
- электронная почта;
- групповые приложения.

Одним из первых pure-приложений была эмуляция терминала. До появления сетей терминалы использовались для доступа к прикладным программам на больших ЭВМ и миникомпьютерах. С переходом на персональные компьютеры потребовался метод доступа к прикладным программам на существующих больших ЭВМ и мини-компьютерах. Эмуляция терминала предоставляет пользователю преимущества двух сред компьютерного мира. Приложения больших ЭВМ и миникомпьютеров могут выполняться на ПК наряду с обычными обособленными приложениями типа текстовых процессоров и электронных таблиц.

Передача файла является основным приложением практически во всех сетях. В некоторых случаях файлы, передаваемые от ПК одного типа к ПК другого типа, требуют перевода из одного формата данных в другой.

Электронная почта дает возможность пользователю ввести сообщение на ПК или локальной рабочей станции и оправить его к кому-нибудь по сети. Электронная почта является, возможно, одним из наиболее важных сетевых приложений. Она предоставляет путь, по которому сеть может улучшить межкорпоративные коммуникации.

Групповые приложения используют сети для электронной автоматизации административных функций современного офиса, они позволяют пользователям координировать календарь, встречи, телефонные звонки и другие задачи электронным путем, могут предлагать pure - либо обособленные административные функции. Например, групповые приложения могут включать электронную почту как средство отправления и получения сообщений между сотрудниками. Так же может использоваться календарь для координации расписания работы сотрудников. Групповые приложения стремятся интегрировать эти функции без потерь для каждой из них.

Обособленные приложения (standalone)

Все приложения, перечисленные выше, предназначены для работы в сетевой среде. В последнее время многие известные обособленные приложения были адаптированы для функционирования в среде клиент-сервер, например, текстовые процессоры, табличные процессоры, базы данных, презентационная графика и управление проектами. Для адаптации *standalone* –приложений для работы в сетевой среде, они разбиваются на две части: первая часть включает пользовательский интерфейс и связующую обработку и работает на станции-клиенте; вторая часть, работающая на сервере, включает операции, требующие значительных процессорных затрат. Основанием к переводу традиционных обособленных приложений в сетевую среду послужило:

- разделение файлов: пользователи могут получать доступ к файлам, таким, как большие базы данных, сохраняемым в общем разделяемом пространстве. Наличие только одной копия файла на сервере, позволяет исключить опасность дублирования файлов с различными датами модификации;

- простота использования: в сетевых версиях приложений используется тот же пользовательский интерфейс, включая команды оператора, что и в предыдущих обособленных версиях. В отличие от чисто сетевых приложений пользователям нет необходимости изучать новые команды для обеспечения нормальной работы;

- ограничение ресурсов: персональные компьютеры с ограниченными ресурсами не могут обрабатывать целиком современные большие приложения, однако если приложение разбивается на две части, то ПК может обрабатывать одну из этих частей (архитектура "клиент-сервер"). Персональный компьютер ("клиент") в общем случае обрабатывает часть пользовательского интерфейса от всего приложения, а более мощный компьютер ("сервер") обрабатывает интенсивную процессорную часть и ввод-вывод информации;

- экономия от масштабирования: новое серверное приложение не требуется для каждого пользователя.

Ниже, на рис.1.1 и рис.1.2, иллюстрируются различные схемы взаимодействия централизованного и распределенного сетевого приложения.

Централизованное сетевое приложение целиком выполняется на данном компьютере, но обращается в процессе своего выполнения к ресурсам других компьютеров сети. На рис.1.1 приложение, которое выполняется на клиентском компьютере, обрабатывает данные из файла, хранящегося на файл-сервере, а затем распечатывает результаты на принтере, подключенном к серверу печати. Очевидно, что работа такого типа приложений невозможна без участия сетевых служб и средств транспортировки сообщений.

Распределенное (сетевое) приложение состоит из нескольких взаимодействующих частей, каждая из которых выполняет какую-то определенную законченную работу по решению прикладной задачи, причем каждая часть может выполняться и, как правило, выполняется на отдельном компьютере сети. Части распределенного приложения взаимодействуют друг с другом, используя сетевые службы и транспортные средства ОС.

Очевидным преимуществом распределенных приложений является [7] возможность распараллеливания вычислений, а также специализация компьютеров.

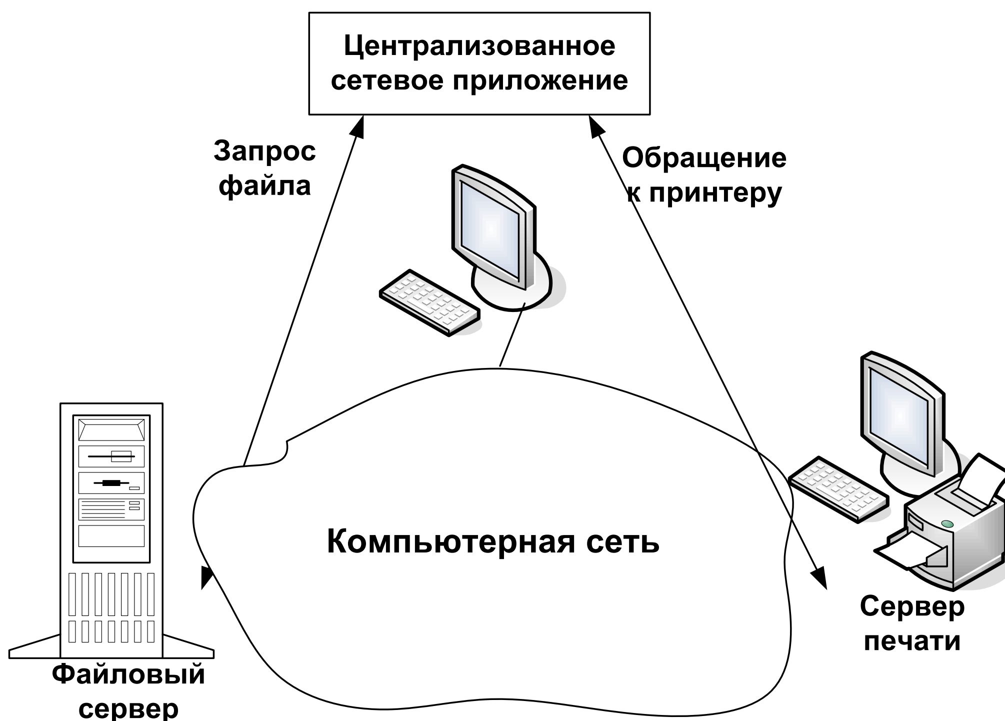

Рис.1.1. Схема взаимодействия централизованного приложения

Так, в приложении, предназначенном, скажем, для анализа изменений полетной информации, можно выделить три достаточно самостоятельные части (рис.1.2), допускающие распараллеливание.

Рис.1.2. Схема взаимодействия распределенного приложения

Первая часть приложения, выполняющаяся на сравнительно маломощном персональном компьютере, могла бы поддерживать специализированный графический пользовательский интерфейс, вторая - заниматься статистической обработкой данных на высокопроизводительном мэйнфрейме, а третья - генерировать отчеты на сервере с установленной стандартной СУБД.

В общем случае каждая из частей распределенного приложения может быть представлена несколькими копиями, работающими на разных компьютерах. Скажем, в данном примере часть первую, ответственную за поддержку специализированного пользовательского интерфейса, можно было бы запустить на нескольких персональных компьютерах, что позволило бы работать с этим приложением нескольким пользователям одновременно.

Многочисленные примеры распределенных приложений можно встретить и в такой области, как обработка данных научных экспериментов, так как многие эксперименты порождают такие большие объемы данных, генерируемых в реальном масштабе времени, которые просто невозможно обработать на одном, даже очень мощном, суперкомпьютере. Кроме того, алгоритмы обработки экспериментальных данных часто легко распараллеливаются, что также важно для успешного применения взаимосвязанных компьютеров с целью решения какой-либо общей задачи.

Одним из последних и очень известных примеров распределенного научного приложения является программное обеспечение обработки данных большого адронного коллайдера (Large Hadron Collider, LHC), запущенного 10 сентября 2008 года в CERN- это приложение работает более чем на 30 тысячах компьютеров, объединенных в сеть.

При очевидных достоинствах распределенных приложений разработчикам необходимо решать такие проблемы, как: какие функции и сколько функций выделить как отдельные части, как организовать взаимодействия различных частей, корректность завершения оставшихся частей при сбое и отказов.

1.1.3. Средства, инструменты, спецификации для построения и сопровождения приложений

1.1.3.1. Язык Java

В настоящее время основным средством для создания сетевых приложений в сети Internet является язык Java. Java – это объектно-ориентированный язык, внешне схожий с языком программирования C++. Основные преимущества данного языка состоят в том, что это простой, объектно-ориентированный, распределяемый, интерпретируемый, надежный, защищенный, не зависящий от архитектуры, переносимый, высокопроизводительный, многопоточный и динамичный язык. Для удобства программистов из языка C++ в язык Java были перенесены некоторые языковые конструкции и инструменты, а также были удалены инструменты, служащие источником проблем в программировании (перегрузка операторов, множественное наследование и автоматическое согласование типов данных), а

также инструменты, связанные с управлением памятью. Java содержит объемную библиотеку классов объектов для доступа к Internet, поддерживающих протоколы TCP/IP, HTTP и FTP. Java-приложения могут работать с объектами по сети через URL точно так же, как и с локальными объектами. Для создания собственных приложений можно использовать объекты из любой точки сети. Также можно создать объект, к которому может получить свободный доступ программное обеспечение других разработчиков. Для обеспечения кросс-платформенности компилятор Java создает код, не зависящий от какой-то конкретной архитектуры. Java позволяет выполнять разнообразные действия непосредственно на клиентском компьютере (в то время как программа CGI выполняется на сервере сети по запросу клиента). Java осуществляет динамическую компоновку, загружая классы тогда, когда он в них нуждается; процесс компоновки в Java отложен до времени выполнения. Интерпретатор Java ограничивает программы рамками приемлемого поведения, что увеличивает безопасность языка Java. В настоящее время для облегчения создания Java-приложений непрограммистами создан простой язык сценариев JavaScript [1].

В [2] рассмотрены возможности языка Java в области представления параллельных процессов и механизмов их взаимодействия с целью сравнения с языком сетевых моделей в этой области. Аналогом процесса в языке Java являются так называемые потоки. Потоки дают возможность программистам писать программное обеспечение, которое одновременно является и более эффективным, и более простым в понимании. Однако потоки имеют свои ограничения, в том числе большие накладные расходы на межпроцессные взаимодействия.

Запуск нового потока иллюстрируется следующим фрагментом Java-кода:

Thread thread = new Thread();
thread.start();

Первая строка реализует новый объект класса Thread. Эта реализация всего лишь создает поток. Метод *start()* приводит к началу выполнения потока. Поток может находиться в одном из следующих состояний: "рождения", выполнения, "смерти", приостановки, "сна" (временной приостановки), блокировки. Три последних состояния относятся к невыполняемым. Для приостановки выполнения потока применяются методы *suspend( )* – "приостановить" и *resume( )* – "продолжить". Другой способ изменения состояния потока на невыполняемое состояние – использование метода *sleep(время приостановки)*. Метод использует аргумент, определяющий продолжительность приостановки. По истечении этого времени поток автоматически "просыпается". Метод *wait( )* переводит поток в состояние блокировки. Поток будет пребывать в этом состоянии до тех пор, пока не будет выполнено некоторое условие, называемое иногда охранным. Данное условие представляется как булево выражение. Для перевычисления значения охранного условия используется метод *notifyAll( )*. Метод *yield( )* приводит к передаче управления диспетчеру потоков, который

проверяет, нет ли в очереди к процессору другого потока с таким же приоритетом. Если да, то начинает выполняться данный поток. Метод *stop( )* уничтожает поток, а метод *run( )* реализует другие методы. Для управления порядком выполнения потоков в языке Java используется механизм приоритетов.

Под синхронизацией в языке Java понимается действие по запиранию данных или метода, гарантирующее непрерываемый доступ к ним. Такую возможность дает ключевое слово *synchronized*. Этот модификатор можно добавить к объявлению метода. Синхронизируемый метод является разделяемым ресурсом. Обладать данным методом может только один поток, остальные потоки-претенденты на данный метод ожидают в очереди. Кроме методов синхронизированы могут быть данные.

При использовании многопотокового программирования возможно появление тупиковых ситуаций типа взаимной блокировки. На языке Java потенциальная возможность взаимной блокировки определяется следующим кодом:

```
synchronized(a) {
  synchronized(b) {
    .......
  }
}
synchronized(a) {
  synchronized(b) {
    .......
  }
}
```

Здесь два потока взаимно блокируются по данным a и b в результате неправильной последовательности блокировки этих данных.

Отметим некоторые недостатки языка Java: отсутствуют средства графического представления многопоточных программ, нет средств структуризации взаимодействия параллельных процессов, неявно представляется параллелизм, недетерминизм, конфликты; отсутствуют развитые механизмы межпроцессного взаимодействия, например, механизм взаимодействия на основе разделения общих глобальных переменных; не формализована семантика языка, что затрудняет проведение верификации многопоточных программ на языке Java.

1.1.3.2. Платформа .Net как основа создания новых электронных интерактивных сервисов

Современное развитие Интернет дает широкие возможности получения и использования деловой информации; возникли новые модели бизнеса (электронные биржи, электронные системы бронирования и продажи билетов, системы интеграции каналов сбыта, сообщества поставщиков и потребителей, электронные аукционы и др.), позволяющие сочетать глобальный охват рынка с

персональным подходом к каждому клиенту. Развертывание и внедрение сложной информационной системы сегодня обходится существенно дешевле и может быть выполнено значительно (иногда - в несколько раз) быстрее, чем это было еще несколько лет назад [2].

За последние годы произошло значительное увеличение пропускной способности сетей благодаря реализации многочисленных высокоскоростных каналов. Впервые появилась возможность организации по-настоящему распределенных систем, обслуживающих миллионы пользователей, например, приложение Napster представляет собой многофункциональный клиент, который взаимодействует со службой каталогов в Интернете и использует в качестве серверов компьютеры других пользователей этого приложения. Еще один пример распределенного приложения — система мгновенного обмена сообщениями, где многофункциональный клиент использует находящийся в Интернете «список приятелей» и взаимодействует с другими клиентами (Instant Messenger и Windows) в сети.

В то же время внедрение Интернет-технологий в корпорациях проходило не всегда гладко. Упрощенные или узкоспециализированные решения, не способные адаптироваться к реальной рыночной ситуации, расти вместе с развитием бизнеса, взаимодействовать с другими информационными системами, вызывали немало разочарований. Стала очевидна настоятельная потребность в надежной масштабируемой платформе для создания новых электронных интерактивных сервисов. Она должна:

- быть доступной самому широкому кругу предприятий - от малого бизнеса до транснациональных корпораций;
- отличаться гибкостью и широтой функций;
- работать на наиболее распространенном оборудовании;
- постоянно развиваться;
- быть удобной в изучении, развертывании и сопровождении.

Задача создания подобной платформы уникальна в целом ряде отношений:

- требуется опыт разработки самых различных решений - от серверных операционных систем и баз данных до офисных приложений и Интернет-обозревателей;
- ее компоненты должны строиться в соответствии с открытыми стандартами и уметь взаимодействовать с другими внешними системами - от систем корпоративного планирования на мэйнфреймах и хранилищ данных на кластерах RISC/Unix до пользовательских интерфейсов карманных компьютеров и сотовых телефонов;
- ИТ-специалисты должны сосредоточиться на создании новых сервисов, освободившись от узкоспециальных работ (масштабировании, обеспечение надежности и совместимости низкоуровневых протоколов);
- создаваемые технологии должны использоваться для развертывания многих критически важных приложений в самых разных областях.

Для скорейшего развития распределенных систем нового поколения должны быть выполнены три условия:

1) Web-службы. Первое условие заключается в том, что все компоненты системы должны быть реализованы в виде веб-служб. Это в равной степени относится к компонентам программного обеспечения и к сетевым ресурсам (например, хранилищам).

2) Объединение и интеграция. Вторым условием является наличие простых и удобных способов объединения и интеграции веб-служб.

3) Простота и удобство работы пользователя. Третье условие — это наличие простой и удобной рабочей среды для конечных пользователей и потребителей.

Корпорация Microsoft считает, что платформа .NET позволяет выполнить эти условия. Платформа .NET стимулирует развитие распределенных систем нового поколения и строится на признанных концепциях и стандартах.

Платформу .NET образуют следующие компоненты (рис.1.3): средства разработки; серверные системы; службы .NET Building Block Services — «строительные блоки»; программное обеспечение для устройств; специализированные рабочие среды (реализованы в виде приложений на платформе .NET).

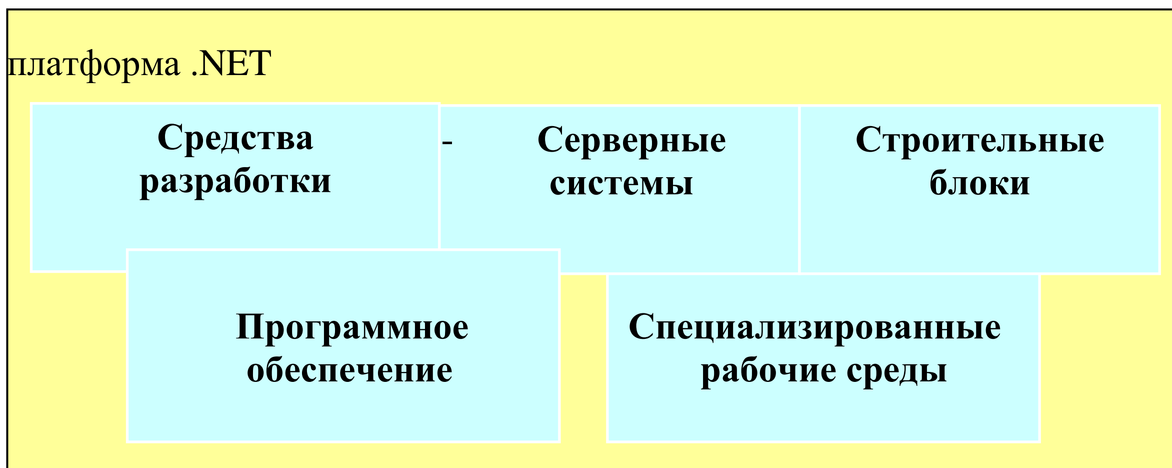

Рис.1.3. Компоненты .NET

Первый компонент упрощает создание веб-служб. Он представлен платформой .NET Framework и набором инструментальных средств Visual Studio. .NET Framework и Visual Studio .NET обеспечивают самый простой, быстрый и эффективный способ разработки веб-служб.

Вторым компонентом .NET является набор серверов, отвечающих за объединение и интеграцию веб-служб. Эти серверные системы можно условно разделить на две категории. Первая включает программные продукты, которые обеспечивают базовые средства для работы с XML, например, Windows 2000/2007, SQL Server 2000/2008 и Exchange 2000/2007. Использование языка XML является самым простым и «открытым» способом интеграции веб-служб. Вторая категория включает специальные серверные системы (такие как BizTalk

Server), которые обеспечивает эффективные и универсальные возможности объединения и интеграции. Например, сервер BizTalk Server 2000 предлагает встроенный язык XLANG, позволяющий определять бизнес-процессы, транзакции и контракты и обеспечивающий глубокую интеграцию разнородных сред.

Третьим компонентом платформы .NET является набор служб, играющих роль «строительных блоков» (Building Block Service), которые повышают простоту и удобство работы пользователя. Сегодня пользователям часто приходится вводить одни и те же учетные данные для доступа к веб-узлам и приложениям. Создатели .Net работают над созданием небольшого набора служб (таких, как службы идентификации, оповещения и схематизированные хранилища), которые значительно упростят переход от одной службы к другой или переход из одной среды в другую. Такая интеграция имеет ключевое значение в мире распределенных вычислительных систем.

Службы — «строительные блоки» предлагают широкие возможности не только пользователям, но и разработчикам, позволяя им использовать готовые службы, доступные через Интернет.

Четвертый компонент платформы .NET представлен набором программного обеспечения для устройств и клиентских систем. Его роль заключается в том, чтобы предложить пользователю удобную и интегрированную среду для работы. Платформа .NET предполагает использование ни одного устройства или клиента, а целого семейства дополняющих друг друга устройств. Объединяет эти устройства то, что все они являются «интеллектуальными». Чтобы поддержка этих устройств стала возможной, для них создается программное обеспечение с новыми функциями, которые делают работу пользователя более удобной и эффективной. Они запоминают вашу личную информацию и используют в качестве платформы для обработки данных Web-, а не отдельные серверы.

Пятым компонентом платформы .NET являются удобные рабочие среды, ориентированные на определенную категорию пользователей, которые интегрируют Web-службы и объединяют различные функциональные возможности. Предлагается несколько таких сред:

- MSN для потребителей;
- bCentral для предприятий малого бизнеса;
- Office для офисных работников;
- Visual Studio .NET для разработчиков.

Наиболее впечатляющая и наиболее обозримая часть .NET — новое семейство корпоративных серверов. Состав продуктов семейства .NET Enterprise Servers подобран так, чтобы удовлетворить в равной степени как практиков, развертывающих и сопровождающих информационные системы современных предприятий, так и теоретиков, заботящихся об элегантности концепций и перспективах развития отрасли.

Концепция «компонентного» программирования является развитием «объектно-ориентированного» подхода и позволяет создавать сложные информационные системы из многократно используемых «строительных блоков». Можно выделить следующие блоки:

- *Identity* - аутентификация, построенная на средствах Windows .NET (Whistler) и Passport .NET. Поддержка карточек доступа и биометрических устройств;
- *Notification and Messaging* - интегрированная поддержка всех типов сообщений от корпоративных и e-mail до факсов и SMS;
- *Personalization* - управление доступом и синхронизацией на основе групп, ролей и правил;
- *XML Store* - использование XML и SOAP для определения контента и функциональности, контроля полноты и корректности транзакций, взаимодействия с SQL Server, MSN Communities, NTFS и т.д.

Новая архитектура .NET гарантирует масштабируемость, необходимую самым крупным предприятиям.

1.1.3.4. Открытая инфраструктура для распределенных вычислений университета Беркли

При проведении большого объема вычислений все чаще используют распределенные вычисления. Для организации распределенных вычислений используются различные инструменты. Первые проекты распределенных вычислений представляли собой автономные программы. Однако, когда на компьютере установлено несколько проектов, то управлять ими становилось достаточно сложно. Поэтому вполне закономерным оказалось появление программы-менеджера, которая позволяла существенно упростить пользователю процесс подключения к новому проекту и свести к минимуму его проблемы по управлению несколькими проектами.

На сегодняшний день для упрощения процесса организации и управления распределенными вычислениями создано несколько программных комплексов, среди которых есть как коммерческие, так и абсолютно бесплатные. Самый известный из комплексов - BOINC-Berkeley Open Infrastructure for Network Computing (Открытая инфраструктура для распределенных вычислений университета Беркли), он используется в значительной части существующих исследований. Проекты, работающие под управлением BOINC-платформы, получили название boinc-проекты.

При работе с boinc-проектами имеется возможность всё управление и контроль осуществлять из одного "контрольного центра", которым является BOINC-менеджер, т.е. можно расставить приоритетность считаемых проектов, выделив каждому необходимую долю ресурсов; разрешить или запретить проектам принимать новые задания, или вообще приостановить работу проекта. При этом имеется возможность видеть свои результаты (сколько очков и по какому проекту) как в таблично-цифровой форме, так и в виде графиков. Менеджер учитывает время, необходимое для окончания расчета, и, при

необходимости, самостоятельно начнет считать тот проект, срок выполнения заданий по которому (т.н. "дедлайн") приближается.

Самый первый проект, в котором был использован метод распределенных вычислений - это проект SETI@Home [1]. Его целью ставился поиск внеземных цивилизаций (Search for Extraterrestrial Inteliegence), для чего исследователями обрабатывается космический шум, записываемый радиотелескопом Arecibo. Сигналы, полученные радиотелескопом, исследуются на предмет наличия особых характеристик - признаков разумной радиоактивности. Программа-клиент выглядит как скринсейвер (рис. 1.4).

Рис.1.4. Вид программы-клиента

При этом каждый из участвующих в вычислениях компьютеров получает небольшую "порцию" расчетов, а назад возвращает полученный результат. При подобной организации возможно с использованием количества обычных ПК достичь вычислительной мощности мощнейших суперкомпьютеров. Сами программы, установленные на компьютеры пользователей и осуществляющие расчеты, написаны так, чтобы работать только в моменты наименьшей загрузки компьютера и не оказывать влияние на работу других программ.

Проект Climate Prediction занимается прогнозированием изменений климата на Земле, чтобы определить, насколько точны существующие методы долговременного предсказания погоды и насколько сильно на их точность влияют погрешности в исходных данных. Проект призван помочь в предсказании погоды на пятьдесят лет вперед. Вычисления, связанные с климатом Земли, настолько ресурсоемки (в них может фигурировать до 100 миллионов переменных), что при работе с суперкомпьютером в ходе сеанса симуляции одновременно можно изменять лишь несколько параметров.

Система распределенных вычислений позволяет запускать сразу множество симуляций одновременно, но с различными параметрами. Каждая программа-клиент работает на компьютерах участников несколько месяцев, прежде чем отправит обработанные данные обратно на центральный сервер. Так как все участники получат немного разные входные данные, то результат работы каждого из них уникален. Затем среди полученных результатов выбираются те, которые наиболее точно отражают реально происходившие в мире климатические события, и делаются выводы о наиболее вероятных изменениях, ожидающих нас в будущем.

Участники проекта Einstein@Home занимаются составлением атласа излучаемых звездами-пульсарами гравитационных полей для всего неба. Это делается с целью проверки одной из гипотез Эйнштейна, которая предсказывает теоретическую возможность существования гравитационных волн, возникающих при столкновениях черных дыр и взрывах звезд. Однако пока никому из ученых так и не удалось зафиксировать эти загадочные волны. Данные для анализа поступают с Лазерно-интерферометрной обсерватории гравитационных полей (LIGO).

1.1.4. Современные решения

Идея перейти от продажи приложений к подписке и аренде возникла уже достаточно давно, однако, ее широкое внедрение тормозилось как консерватизмом потребителей, так и отсутствием необходимых технологий. Существующие сегодня решения позволяют создавать компоненты и сервисы, которые могут адаптивным и прозрачным для пользователя образом настраиваться на разнообразные источники контента и обновлений кода. Хорошо спроектированный сервис «экранирует» подробности своей реализации от потребителя, передавая вопрос о выборе схемы его поставки от пользователей в компетенцию корпоративных ИТ-специалистов и лиц, принимающих бизнес-решения.

К новым возможностям следует отнести:

1) Natural Interface - технологии, обеспечивающие «естественное» общение с компьютером, включая распознавание и синтез речи, рукописный ввод и другие, сделают наиболее удобным использование самых различных устройств.

2) Universal Canvas - новая основанная на XML-схемах архитектура, превращающая Интернет из платформы «для чтения» в платформу для «чтения/записи». Кроме того, она поддерживает для документа создание, объединение источников, чтение, редактирование, комментирование, аннотирование, анализ.

3) Information Agent - представитель пользователя в Интернет, имеет историю, привычки и потребности конкретного пользователя, позволяет Интернет-сайтам и другим сервисам оптимально приспособиться к нему. Поддерживает P3P-технологии (приватные личные контакты). В отличие от

существующих технологий персонализации, Information Agent остается под полным контролем пользователя.

4) Smarttags - обобщение технологии Intellisense на Web-контент. Позволяет «предугадывать» и корректировать действия пользователя. Наиболее известные примеры использования технологии Intellisense: «помощники» в Office и Internet Explorer, автозамена, автоформат, автозаполнение и т.д.

Главными программными компонентами в комплексе этих средств являются .NET Framework и Visual Studio .NET. Именно поэтому точкой отсчета "эпохи .NET" считают начало выпуска этих продуктов.

Говоря о перспективах .NET, нужно отметить два момента. С одной стороны, это действительно новая технологическая платформа, существенно отличающаяся от сегодняшней Windows. В упрощенном виде Windows = Win API + + COM, а .NET = CLR + XML Web Services. С другой стороны, .NET — это очевидный ответ Microsoft идеологии Java.

Общее мнение аналитиков по вопросу будущего данной технологии выразила компания Gartner: NET — это действительно новая платформа, а не очередная модификация Windows. Она очень тесно связана с нынешней архитектурой COM+ и, более того, две эти платформы будут еще длительное время существовать параллельно (хотя бы потому, что многих средств, реализованных с помощью COM+, в рамках .NET пока не существует). Тем не менее .NET - самостоятельная платформа со своими сильными и слабыми сторонами:

- она переносит фокус с отдельных Web-серверов или специализированных информационных систем на создание среды, обеспечивающей их эффективное взаимодействие между собой и с пользователями;

- Microsoft .NET развивает и объединяет концепции, как операционных систем, так и Интернета, превращая построенную по открытым стандартам глобальную Сеть в операционную систему нового поколения. .NET позволяет приложениям и сервисам преодолевать ограничения отдельных физических устройств;

- разработчики могут расширять свои решения и инструментальные средства наиболее подходящими многократно используемыми компонентами, импортируемыми из глобальной сети, уделив внимание проектированию элегантной архитектуры решения, а не утомительному воспроизведению вспомогательных фрагментов кода, необходимого сегодня для простого связывания компонентов.

Контрольные вопросы

1. Что понимается под термином «распределенные приложения»?
2. Поясните варианты расположения приложений.
3. Назовите отличия pure- и standalone - приложений.
4. Что понимается под платформой .NET?

5. Какие условия способствовали возникновению .NET?
6. Назовите основное отличие .NET от других современных технологий, в том числе COM+?
7. Перечислите основные компоненты технологии .NET.
8. Что понимается под термином «boinc-проект»?
9. Какие новые возможности предоставляет boinc-технология?

1.2. Распределенное приложение базы данных

1.2.1. Принципы создания системы обработки информации в масштабе предприятия

Идея создания объединенных вычислительных мощностей компьютеров для решения сложных задач обработки информации, непосильных для каждого из них в отдельности, многие годы не была реализована по причине отсутствия технических решений (отсутствие стандартов и протоколов взаимодействия). Примером первой распределенной обработки можно назвать компьютерную сеть ARPANet. Появление локальных сетей привело к развитию новой области программного обеспечения - созданию распределенных приложений, интерес к которым сразу проявили крупные компании, структура бизнеса которых требовала подобных решений. Именно на этапе создания корпоративных распределенных приложений были сформированы основные требования и разработаны основные архитектуры подобных систем, используемые и в настоящее время [3]:

- *пространственное разделение* (подразделения организации разнесены в пространстве и зачастую имеют слабо унифицированное программное обеспечение);

- *структурное соответствие* (программное обеспечение должно адекватно отражать информационную структуру предприятия и соответствовать основным потокам данных);

- ориентация на внешнюю информацию (в своей работе современные предприятия и вынуждены уделять повышенное внимание работе с заказчиками. Поэтому программное обеспечение, используемое предприятием, должно уметь работать с новым типом пользователей, и их запросами).

Схема распределения вычислений в распределенной системе приведена на рис.1.5.

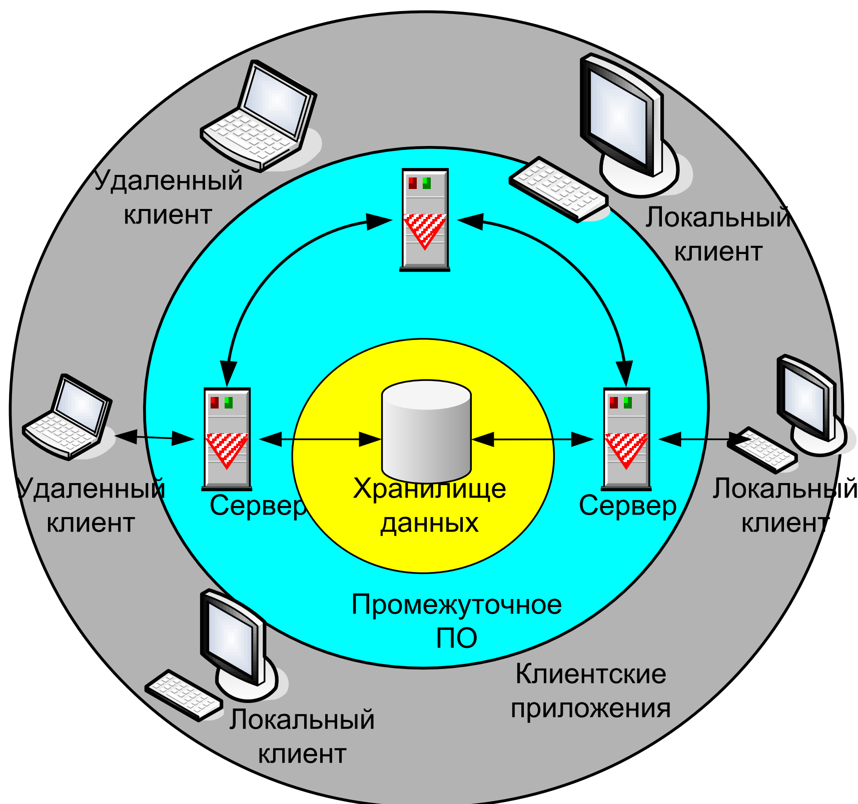

Рис.1.5. Схема распределения вычислений

Идеология распределенных вычислений подразумевает наличие нескольких центров (серверов) хранения и обработки информации, реализующих различные функции и разнесенных в пространстве. Эти центры помимо запросов клиентов системы должны выполнять и запросы других центров, так как для решения одной задачи могут потребоваться совместные ресурсы других серверов. Для управления сложными запросами и работой системы в целом необходимо специализированное управляющее программное обеспечение. Кроме этого, для обеспечения взаимодействия частей должна быть некая транспортная среда. Доступ к данным в распределенных приложениях может быть организован на различных уровнях - от клиентского ПО и транспортных протоколов до защиты серверов БД.

1.2.2. Основные уровни архитектуры распределенного приложения

Рассмотрим архитектуру распределенного приложения, позволяющую ему выполнять сложные и разнообразные функции. В разных источниках приводятся различные варианты построения распределенных приложений. И все они имеют право на существование, ведь такие приложения решают самый широкий круг задач во многих предметных областях, в том числе в гражданской авиации, а неудержимое развитие средств разработки и технологий подталкивает к непрерывному совершенствованию.

Общая архитектура распределенного приложения разбивается на несколько логических слоев (уровней обработки данных), в зависимости от трех важнейших функций приложений:

- представление данных (пользовательский уровень): пользователи приложения могут просмотреть необходимые данные, отправить запрос на выполнение, ввести в систему новые данные или отредактировать их;

- обработка данных (промежуточный уровень): на этом уровне сконцентрирована бизнес-логика приложения, осуществляется управление потоками данных и организуется взаимодействие частей приложения. Именно концентрация всех функций обработки данных и управление на одном уровне считается основным преимуществом распределенных приложений;

- хранение данных (уровень данных): это уровень серверов баз данных. На этом уровне расположены сами серверы, базы данных, средства доступа к данным, различные вспомогательные инструменты.

Часто такую архитектуру называют трехзвенной или трехуровневой, при этом каждый уровень может быть дополнительно разбит на несколько подуровней. В такую архитектуру можно вписать любое распределенное приложение, однако следует учитывать еще одну характерную особенность, присущую именно распределенным приложениям, - это управление данными. Важность этой функции очевидна, поэтому распределенное приложение должно обладать еще одним логическим уровнем – уровнем управления данными. Следовательно, целесообразно промежуточный уровень разделить на два самостоятельных: уровень обработки данных (концентрация бизнес-правил обработки данных) и уровень управления данными (контроль выполнения запросов, обслуживание работы с потоками данных и организация взаимодействия частей системы).

Таким образом, можно выделить четыре основных уровня распределенной архитектуры (рис. 1.6):

- представление данных (пользовательский уровень);

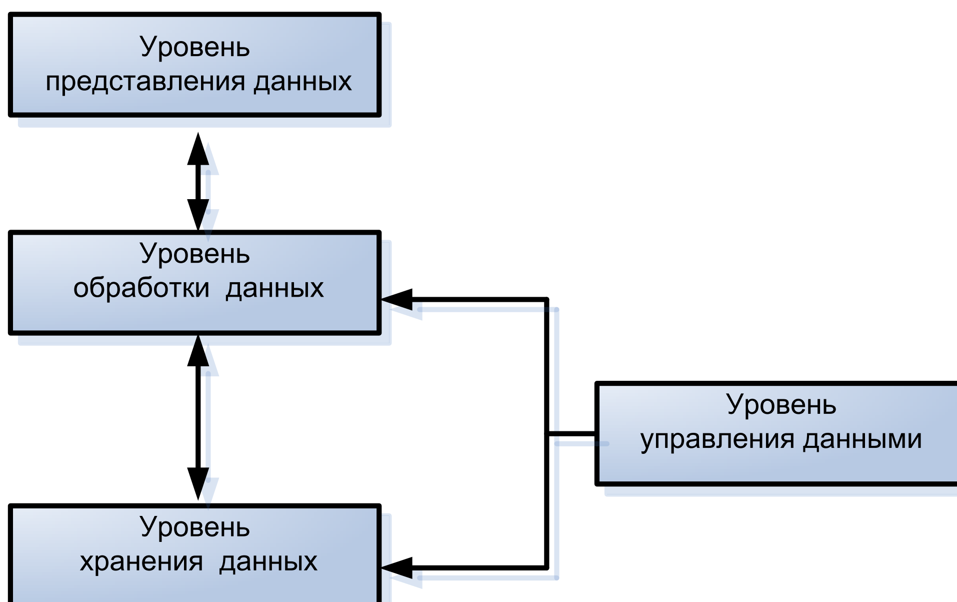

Рис.1.6. Уровни архитектуры

- правила бизнес-логики (уровень обработки данных);
- управление данными (уровень управления данными);
- хранение данных (уровень хранения данных).

Три уровня из четырех, исключая первый, занимаются непосредственно обработкой данных, а уровень представления данных позволяет визуализировать и редактировать их. Этот уровень позволяет пользователям получать данные от уровня обработки данных, который извлекает информацию из хранилищ и осуществляет все необходимые преобразования данных. После ввода новой информации или редактирования существующей, потоки данных проходят обратный путь: от пользовательского интерфейса через уровень бизнес-правил в хранилище. В случае просмотра данных в режиме «только для чтения» уровень обработки данных не используется в общей схеме передачи данных, так как какие-либо изменения не вносятся, и сам поток информации является однонаправленным – из хранилища на уровень представления данных.

1.2.3. Расширение базовых уровней

Рассмотренные выше уровни архитектуры распределенного приложения являются базовыми. Они формируют структуру создаваемого приложения в целом, но при этом, естественно, не могут обеспечить реализацию любого приложения - предметные области и задачи слишком обширны и многообразны. Для таких случаев архитектура распределенного приложения может быть расширена за счет дополнительных уровней, которые призваны отобразить особенности создаваемого приложения.

Можно выделить два наиболее часто применяемых расширения базовых уровней:

- уровень бизнес-интерфейса – располагается между уровнем пользовательского интерфейса и уровнем обработки данных. Данный уровень обеспечивает абстрагирование программного кода клиентского приложения от особенностей реализации логики приложения;
- уровень доступа к данным – располагается между уровнем хранения данных и уровнем обработки данных. Данный уровень позволяет сделать структуру приложения не зависящей от конкретной технологии хранения данных.

Контрольные вопросы

1. Перечислите основные требования к системе обработки информации в масштабе предприятия.
2. Поясните схему распределения вычислений.
3. Перечислите базовые уровни архитектуры распределенного приложения.
4. Назовите основную функцию уровня представления данных.
5. Для каких целей используются дополнительные уровни?
5. Перечислите часто применяемые расширения базовых уровней.

6. Что обеспечивает уровень бизнес-интерфейса?
7. Назначение уровня доступа к данным.

1.3. Сервис-ориентированные архитектуры (SOA)

1.3.1. Сервис-ориентированные архитектуры: элементы и характеристики

Архитектура, основанная на сервисах - это новый шаг в направлении комплексного решения проблемы создания распределенных приложений, выполняющихся на многих компьютерах. Она отличается от архитектуры, основанной на объектах (object oriented architecture), тем, что при необходимости выполнить какие-либо действия необходимо просто идентифицировать контракт, описать, как выглядят данные, и получить возможность динамически переключаться между сервисами, реализующими нужную пользователю функциональность, и выбирать сервис из нескольких доступных.

Service-Oriented Architecture (SOA) - это архитектура, обеспечивающая интеграцию и связность, и предназначенная для использования естественно возникающей гетерогенности систем и инфраструктур [4]. Существует много взглядов на концепцию SOA, она развивается, превращая Интернет в утилиту, доступную для всех способов обмена данными, коммерции и коммуникации. Архитектура включает в себя стандарты для процессов, технологий и интерфейсов. Одной из целей сервис-ориентированной архитектуры является обеспечение реакции на такое подвижное, деятельное видение предприятия путем формирования предложений с обоснованной ценностью для бизнеса. Предприятие не реализует SOA по принципу «все сразу», т.к. существует потребность в постепенных стратегиях внедрения, которые решают следующие вопросы:

- составление экономического обоснования внедрения архитектуры;
- оценка имеющихся технологий;
- выбор пилотного бизнес-проекта и стороны бизнеса;
- распространение в масштабе предприятия;
- распространение в сети партнеров.

Эти измерения показывают, что для создания SOA необходимо рассмотреть широкую сферу вопросов, включающую бизнес, организацию, архитектуру и технологию.

Элементами архитектуры SOA являются домены, в которых могут существовать службы и функции, которые они представляют. Характеристики определяют, как разные домены взаимодействуют друг с другом. К ним относятся: платформа, местоположение, протоколы, язык программирования, структура вызова, безопасность, управление версиями служб, модель служб, информационная модель, формат данных.

Для служб предприятия общими являются все или часть этих характеристик, и они влияют на то, какие действия выполняет служба, как она

их выполняет. В [10] выделены следующие четыре архитектурных домена с субдоменами, которые влияют на то, где может существовать служба и какие функции она может выполнять:

- домен инфраструктурных служб с субдоменами:
  - бизнес-службы утилиты;
  - автоматизация на уровне служб и оркестровка;
  - виртуализация ресурсов;
- домен промежуточного программного обеспечения;
- домен бизнес-служб;
- домен прикладных служб с субдоменами:
  - субдомен модели прикладного программирования;
  - субдомен готового коммерческого программного обеспечения;
  - субдомен управления информацией.

На предприятии домены могут быть по-разному реализованы с использованием любых комбинаций готовых приложений, специально написанных приложений, существующей инфраструктуры, а также внешних и получаемых по аутсорсингу служб. Конкретные службы, которые входят в субдомены автоматизации уровня служб (service-level automation, SLA) и оркестровки, представляют собой автоматизированные службы для преодоления проблем, восстановления после сбоя системы, регулирования рабочей нагрузки, управления ресурсами, а также службами системы безопасности и обеспечения данными (инсталляция и размещение служб в системе).

1.3.2. Общая схема SOA

В общем виде SOA предполагает наличие трех основных участников: поставщика сервиса, потребителя сервиса и реестра сервисов (рис.1.7). Взаимодействие участников выглядит достаточно просто: поставщик сервиса регистрирует свои сервисы в реестре, а потребитель обращается к реестру с запросом. Отсутствие любого из этих элементов недопустимо, а добавление других составляющих на практике не только возможно, но и неизбежно. Среди таких элементов могут быть всевозможные программные средства промежуточного слоя, контролирующие порядок и контекст взаимодействия, осуществляющие мониторинг и управление сервисами, а также управление метаданными и другие вспомогательные процессы.

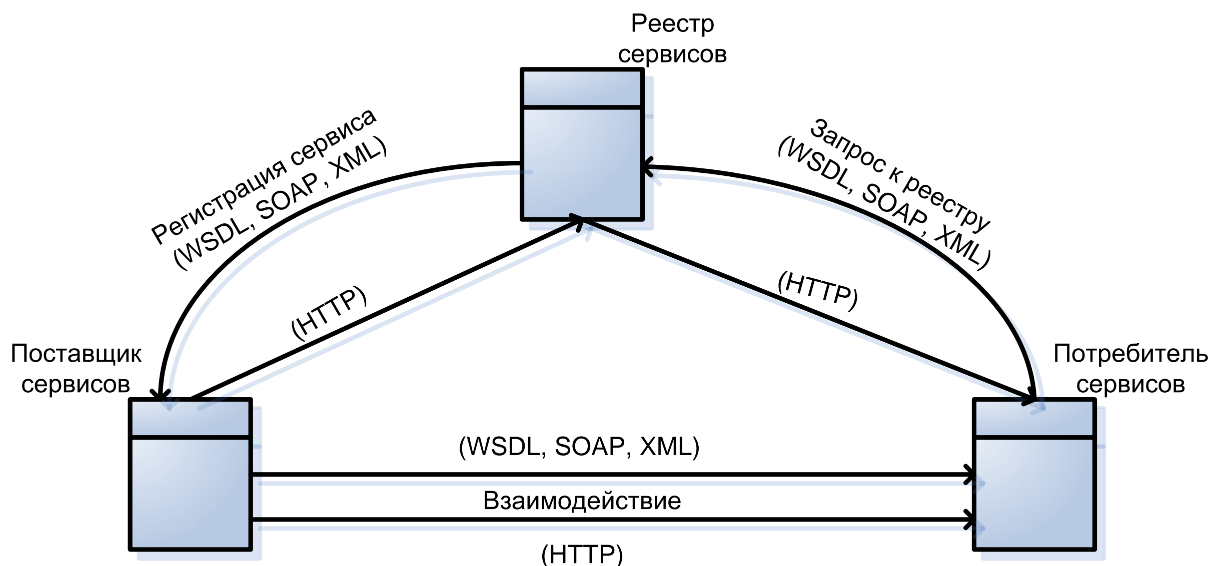

Рис.1.7. Общая схема SOA

Для использования сервиса необходимо следовать соглашению об интерфейсе для обращения к сервису - интерфейс должен не зависеть от платформы. SOA реализует масштабируемость сервисов - возможность добавления сервисов, а также их модернизацию. Поставщик сервиса и его потребитель оказываются несвязанными - они общаются с помощью сообщений. Поскольку интерфейс должен не зависеть от платформы, то и технология, используемая для определения сообщений, также должна не зависеть от платформы. Поэтому, как правило, сообщения являются XML-документами, которые соответствуют XML-схеме.

Модель SOA не зависит от технологий, использующихся для реализации SOA, а основным методологически значимым ее компонентом является реестр сервисов. В обозначенном на схеме асинхронном протоколе общения провайдера и потребителя сервисов он выполняет функции посредника. Провайдер размещает информацию о своих сервисах в реестре, что дает возможность потребителю в любой момент найти необходимый ему сервис. За этим процессом общения скрывается основное качество SOA - слабая связанность. Благодаря этому свойству сервисы обретают мобильность, способность перемещаться с одного сервера на другой, не требуя согласования и координации со всеми потребителями. Естественно, что потребители сервисов в ряде случаев не способны и не должны принимать во внимание регулярное перераспределение ресурсов, обеспечивающих функционирование сервисов. Позднее связывание также позволяет отложить момент конечной сборки связей до времени исполнения, а не времени разработки программы, что характерно для традиционных монолитных систем. Можно также во время исполнения менять параметры связи (такие как адрес, протокол и канал взаимодействия). Это придает несколько измерений гибкости самой связке между провайдером и потребителем сервиса, соответственно вызываемым и

осуществляющим вызов объектами. В частности, провайдер и потребитель могут исполняться на сколь угодно физически удаленных инфраструктурах. Каждая из систем может иметь собственные параметры жизненного цикла, а любые изменения в них, не затрагивающие интерфейс сервиса, не требуют остановки ни одной из них.

В SOA сервисы рассматриваются как автономные объекты, управление которыми не централизовано. Это позволяет взаимодействующим посредством сервисов информационным системам развиваться в соответствии с потребностями бизнеса, которые потребителям сервисов, как правило, не только не известны, но и не интересны. Однако это было бы невозможно, если бы интерфейс сервиса не был прочно закреплен обоюдным соглашением провайдера и потребителя сервиса.

Одной из отличительных черт SOA является наличие контрактов, описывающих интерфейсы сервисов. Такой контракт представляет собой документ, специфицирующий ожидания сервиса по отношению к его потребителям и наоборот. Контракты Web-сервисов описываются WSDL-документом, определяющим в нотации XML, как потребители должны обращаться к сервису. Использование XML на этом этапе имеет принципиальное значение, позволяя и провайдеру, и потребителю сервиса не зависеть от определенной платформы. Следует заметить, что подобные контракты существовали и до появления Web-сервисов. Например, в архитектуре CORBA для описания интерфейса объектов использовался язык IDL, который уступает WSDL по ряду существенных параметров. Главный из них - отсутствие поддержки XML и XML-Schema, ставших наиболее распространенными языками разметки передаваемых по сети сообщений и представления моделей данных. Технические контракты, формулируемые провайдером сервисов, должны быть доступны потенциальным потребителям для интерпретации, анализа и реализации интеграции. Для этого используется специальный реестр, каталогизирующий доступные сервисы.

1.3.3. Сервис-ориентированный анализ и проектирование

Методы анализа и проектирования за последние четыре десятилетия сделали значительный шаг в развитии. Последние разработки связаны с моделированием и использованием метаданных в качестве основы для методов и инструментов, удовлетворяющих широкому диапазону задач, от моделирования бизнес-процессов до автоматизированной генерации исполняемого программного обеспечения.

При проектировании SOA необходимо принять во внимание две перспективы в SOA – потребителя и поставщика сервиса. Посредник сервиса на данный момент не участвует в основном потоке. Стратегия проектирования SOA не начинается снизу вверх, как это обычно бывает в случаях с проектированием Web-сервисов, так как архитектура SOA является стратегической и ориентированной на бизнес. Количество важных действий и решений влияет не только на архитектуру интеграции, но и на архитектуры

предприятия и приложения. Сюда включены мероприятия с двух ключевых позиций потребителя и поставщика (табл.1.1).

Таблица 1.1

Мероприятия по обеспечению сервис-ориентированного моделирования

Роль	**Проводимые мероприятия**				
Потребитель	Идентифика ция сервиса	Категориза ция сервиса	Решения по раскрытию сервиса	Хореография или компози-ция	Качество обслужива-ния
Поставщик	Идентифика ция компонента	Специфика ция компонен-та	Реализация сервиса	Управление сервисом	Реализация стандартов
	Привязка сервиса к компонен-там	Разбиение SOA на уровни	Создание техническо-го прототипа	Выбор продукта	Архитектур-ные решения

Как следует из табл.1.1, мероприятия поставщика являются расширенным набором мероприятий потребителя. К примеру, поставщик так же, как и потребитель, будет вовлечен в идентификацию сервиса, его категоризацию и т.д. Во многих случаях, разграничение ролей происходит по причине того, что поставщики сами определяют необходимые для них сервисы (обычно путем их поиска). После того, как потребитель будет уверен в соответствии спецификаций искомого сервиса со спецификацией сервиса, предоставляемого поставщиком, он может вызвать необходимый сервис. Поставщику в данном случае необходимо опубликовать сервисы, которые он желает поддерживать, как с позиций функциональности, так и с позиций обеспечения требуемого качества обслуживания (QoS). Это неявно выраженное соглашение между поставщиком и потребителем могло бы сформироваться в явно выраженные условия соглашений об уровне услуг (SLA), установленные или электронным способом или путем подписания юридических документов.

Мероприятия, описанные выше, можно отобразить в виде потоков в сервис-ориентированном моделировании и архитектурном методе, как это показано на рис.1.8 [5].

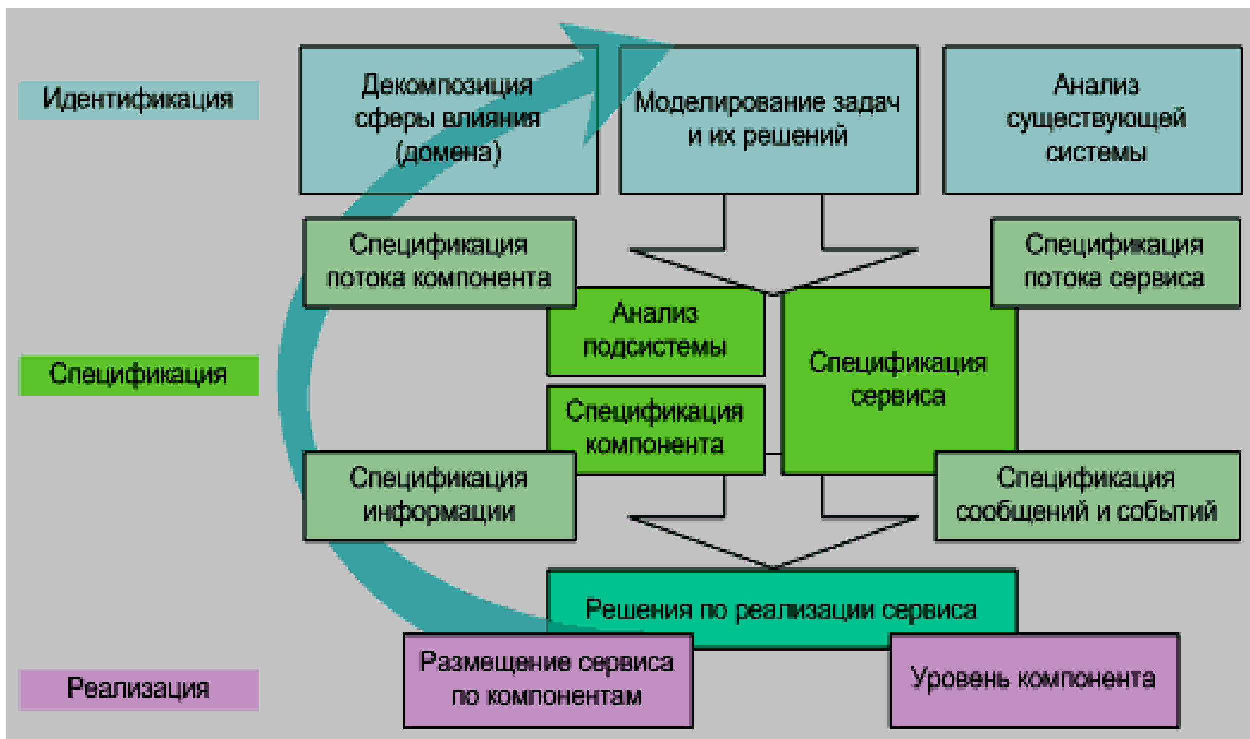

Рис.1.8. Метод сервис-ориентированного моделирования
и архитектуры

Процесс сервис-ориентированного моделирования и построения архитектуры состоит из трех основных шагов: идентификации, спецификации и реализации сервисов, компонентов и потоков (обычно путем объединения сервисов).

1) *Идентификация сервиса.* Данный процесс состоит из комбинации нисходящих, восходящих и исходящих методик декомпозиции сферы влияния (домена), анализа существующих средств и моделирования задач, и средств для ее решения, а также моделирования сервисов. При *нисходящем анализе* проект возможных случаев использования бизнеса обуславливает спецификацию для бизнес-сервисов. Такой нисходящий процесс часто называют *декомпозицией домена*. Он заключается в декомпозиции сферы влияния (домена) бизнеса в его функциональные области и подсистемы, включая потоки или декомпозицию процесса в процессы, подпроцессы и высокоуровневые случаи использования бизнеса. Это способствует раскрытию бизнес-сервисов на стороне предприятия, или использованию последних в пределах предприятия во всей бизнес стратегии.

В *восходящей части* процесса (*анализе существующей системы*) анализируется унаследованная система. При этом выбираются подходящие кандидаты для обеспечения менее затратных решений для реализации функциональности сервиса, лежащей в основе. Здесь анализируются и используются для достижения цели различные API, транзакции и модули от унаследованных и упакованных приложений. В некоторых случаях, с целью поддержки функциональности сервиса необходимо использовать компоненты унаследованных систем для перемоделирования существующих средств.

Исходящий подход заключается в *моделировании типа задача-сервис* для подтверждения и извлечения сервисов, не найденных при проведении восходящей и нисходящей идентификации. Он разбивает сервисы на задачи и подзадачи, ключевые показатели производительности и исходные параметры.

При идентификации сервисов очень важно начать классификацию сервиса в иерархии сервисов, отражая композитную или фрактальную природу сервисов – сервисы могут и должны быть скомпонованы из более тонкоструктурных компонентов и сервисов. Классификация помогает определить композицию и иерархическое представление, а также скоординировать построение взаимозависимых сервисов и их иерархии.

При идентификации выполняется анализ подсистемы. В данном мероприятии берутся подсистемы, найденные в результате декомпозиции домена, и определяются взаимозависимости и потоки между ними. В нем также выявляются случаи использования, идентифицированные во время декомпозиции домена как раскрытые сервисы в интерфейсе подсистемы. Анализ подсистемы заключается в создании моделей объекта для представления внутренних выработок и конструкций подсистем, раскрывающих и анализирующих сервисы. Проектная конструкция "подсистемы" впоследствии реализуется, как конструкция реализации крупно структурного компонента, реализующего сервисы в данном мероприятии.

2) *Спецификация компонента.* В этом важном мероприятии определяются следующие детали компонента, реализующего сервисы: данные, правила, сервисы, настраиваемый профиль, вариации. Здесь также задаются спецификации обмена сообщениями, спецификации событий и дается определение управления.

3) *Размещение сервиса.* Размещение сервиса заключается в распределении сервисов по подсистемам, которые уже были идентифицированы. В этих подсистемах присутствуют корпоративные компоненты, реализующие их опубликованную функциональность. Структурирующие компоненты появляются при использовании шаблонов для создания корпоративных компонентов в комбинации с медиаторами, фасадами, объектами Rule, настраиваемыми профилями, фэктори.

1.3.4. Концепция управления распределенными службами

На предприятии управление и мониторинг информационных технологий включают в себя использование в центрах управления хорошо известных механизмов и стратегий. Эти механизмы можно свободно разделять на разные модели управления, включая сортировку, решение элементарных проблем и операции, управляемые ресурсами. Эти модели меняются при переходе на SOA. Каждый следующий шаг в развитии моделей управления связан с более абстрактным видением конкретной проблемы, с переходом от отдельных событий к состояниям ресурсов (потенциально агрегируемые ресурсы), потом – к потокам транзакций (агрегируемые ресурсы) и, наконец, - к службам (агрегированные потоки) с соглашениями на уровне обслуживания.

Первая модель управления обычно выполняет сортировку событий, концентрируясь на приеме событий и направлении их в соответствующие группы сортировки для решения. Сортировочная модель управления может эволюционировать в следующий уровень – *модель решения элементарных проблем* за короткое время, еще до передачи сведений о событиях. Независимо от модели управления работа большинства центров управления IT направляется событиями, которые запускают выполнение действий. Это могут быть события Tivoli Event Console™ или новые уведомления об ошибках (trouble tickets), создаваемые функциями автоматизации-интеграции. Эти системы также могут устанавливать приоритет событий, определять временную последовательность, приоритет, влияние на бизнес и географию.

В некоторых организациях операции по управлению информационными технологиями эволюционировали до модели, *направляемой ресурсами*. Рабочий процесс в этой модели направляется изменением состояния ресурсов. Ресурсами здесь могут быть виртуализованные ресурсы или кластеры.

Более высокий уровень IT- управления – это *операции, направляемые бизнес-службами.* Главная проблема управления на этом уровне заключается в понимании бизнес-служб и их компонентов.

Основные области IT-управления в SOA показаны на рис. 1.9.

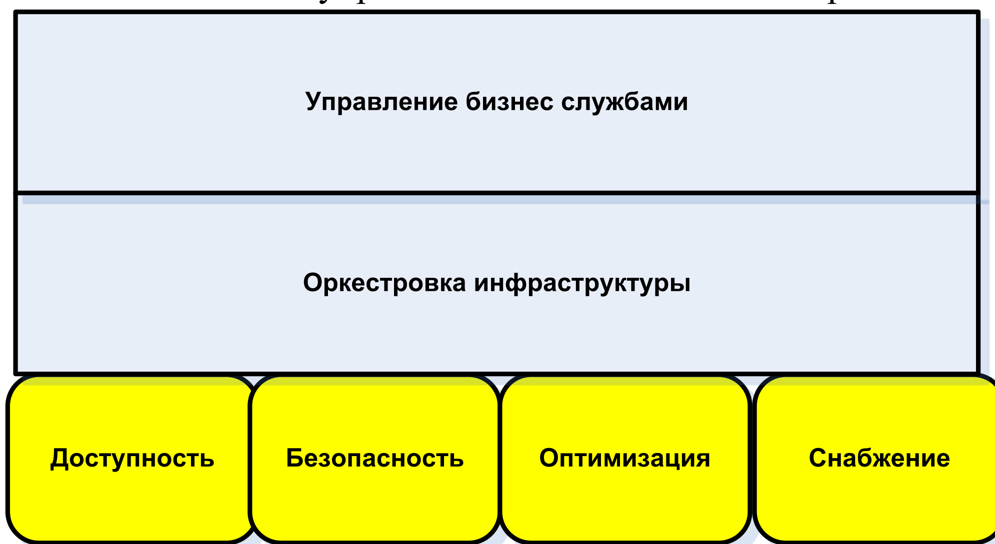

Рис.1.9. Ключевые области управления SOA

Инфраструктура на основе SOA и соответствующие инструменты управления IT позволяют справиться с такими традиционными проблемами интеграции несовместимых бизнес-процессов и приложений, основанных на службах, как: мониторинг производительности и доступности уровня службы; обеспечение выполнения соглашений об уровне обслуживания; отслеживание динамической взаимосвязи слабо связанных компонентов системы и т.д.

Контрольные вопросы

1. Что включает в себя термин «сервис-ориентированные архитектуры»?
2. Чем отличается архитектура SOA от архитектуры, основанной на объектах?
3. Перечислите основные архитектурные элементы SOA.
4. Назовите основные функции SLA и оркестровки.
5. Поясните схему взаимодействия основных элементов SOA.
6. Назначение контрактов в SOA.
7. Какой компонент модели SOA является методологически значимым?
8. Перечислите основные этапы процесса сервис-ориентированного моделирования.
9. Перечислите основные модели IT-управления.

ЧАСТЬ 2. ОБЛАЧНАЯ ОБРАБОТКА ДАННЫХ КАК КОНЦЕПЦИЯ

2.1. Основные понятия и модели расположения приложений

2.1.1. Понятие Gloud computing

В соответствии с определением, предложенным Национальным Институтом Стандартов и Технологий США (NIST) под термином *Cloud Computing* (облачные вычисления) понимается модель предоставления повсеместного и удобного сетевого доступа по мере необходимости к общей совокупности конфигурируемых вычислительных ресурсов (например, сетей, серверов, систем хранения, приложений и сервисов), которые могут быть быстро предоставлены и освобождены с минимальными усилиями по управлению и необходимостью взаимодействия с провайдером услуг (сервис-провайдером).

Облачные технологии представляют собой технологии вида «клиент-сервер», которые состоят из виртуального сервера (или группы серверов) и нескольких клиентов, которые подключаются к нему с помощью сети Интернет. Обозначение «облаков» в данном случае используется как основная ассоциация при обозначении структуры работы данной системы.

Облачная модель поддерживает высокую доступность сервисов и описывается пятью основными характеристиками (essential characteristics), тремя сервисными моделями/моделями предоставления услуг (service models) и четырьмя моделями развертывания (deployment models).

2.1.2.Характеристики облачных вычислений

NIST были разработаны следующие обязательные характеристики облачных вычислений (Essential Characteristics):

1. Самообслуживание по требованию (self service on demand) - потребитель может самостоятельно обеспечивать себя вычислительными возможностями (средствами и ресурсами), такими как серверное время и сетевые хранилища, по мере необходимости запрашивая их у сервис-провайдера в одностороннем автоматическом режиме, без необходимости взаимодействия с персоналом, представляющим сервис-провайдера.

2. Свободный сетевой доступ (Broad network access) - запрашиваемые сервисы доступны по сети через стандартные механизмы, поддерживающие использование гетерогенных платформ вне зависимости от используемого терминального устройства (например, мобильных телефонов, ноутбуков, и т.д.).

3. Обьединение ресурсов (Resource pooling) - вычислительные ресурсы провайдера организованы в единый пул для обслуживания различных потребителей в многопользовательской модели с возможностью динамического назначения и переназначения различных физических и виртуальных ресурсов в соответствии с требованиями потребителей. Особое значение имеет независимость размещения ресурсов, при котором пользователь, в общем случае, не знает и не контролирует точное физическое местоположение предоставляемых ресурсов, но может специфицировать их расположение на более высоком уровне абстракции (например, страна, штат или центр

обработки данных). Примерами таких ресурсов являются системы хранения, обработка данных, память, пропускная способность сети, виртуальные машины.

4. Быстрая эластичность (Rapid elasticity) - вычислительные возможности могут быть предоставлены быстро и в ряде случаев автоматически для оперативного повышения масштабируемости и быстрого освобождения для уменьшения масштабов потребления. Для потребителя эти ресурсы часто представляются как доступные в неограниченном объеме, и могут быть приобретены в любой момент времени в любом количестве.

5. Измеримый сервис (Measured Service) - облачные системы автоматически контролируют и оптимизируют использование ресурса, за счет использования его на определенном уровне абстракции (например, обьем хранимых данных, пропускная способность, количество активных учетных записей пользователей, количество транзакций).

2.1.3. Три модели расположения приложений

В настоящее время существует три основных модели расположения приложений: в инфраструктуре заказчика, у компании-хостера, в облаке.

Расположение в инфраструктуре заказчика (on premises) - наиболее традиционная модель развертывания приложений, существующая уже десятки лет. Размещение приложений в локальной инфраструктуре предполагает существенные начальные инвестиции в аппаратные ресурсы, программное обеспечение, сетевую инфраструктуру и персонал. Такая модель - оплата, приобретение, владение - напрямую связана с высокими капитальными затратами, но, в тоже время, она обеспечивает полный контроль за инфраструктурой, аппаратным и программным обеспечением.

Модель развертывания приложений «Расположение у компании-хостера (hosting)», называвшаяся ранее Application Services Prodiver (ASP), а затем - SaaS или просто «хостинг», получила свое развитие несколько лет назад и является одним из наиболее популярных способов снижения расходов на информационные технологии. Она основана на аренде аппаратной платформы, программного обеспечения, соответствующей инфраструктуры и персонала, выполняющего ее обслуживание. Такая модель отличается меньшим контролем за инфраструктурой, аппаратным и программным обеспечением и базируется на оплате фиксированного числа ресурсов, что обычно предполагает оплату даже в тех случаях, когда арендуемые ресурсы не используются.

Расположение в облаке (cloud) - данная модель появилась совсем недавно, она предполагает оплату по факту использования арендуемых аппаратных и программных ресурсов, что приводит к существенному снижению начальных расходов и переходу от капитальных инвестиций к операционным расходам. Такая модель отличается практически отсутствием контроля за инфраструктурой и аппаратным обеспечением, а при аренде программного обеспечения - еще и отсутствием контроля за ним.

Каждый из рассмотренных подходов имеет свои достоинства и недостатки, но, с точки зрения экономики, самой важной характеристикой

является оплата по факту использования, реализуемая именно облачными вычислениями.

2.1.4. Модель облачных вычислений

Модель облачных вычислений состоит из *внешней* (front end) и *внутренней* (back end) частей. Эти два элемента соединены по сети, в большинстве случаев через Интернет. Посредством внешней части пользователь взаимодействует с системой; внутренняя часть – это собственно само облако. Внешняя часть состоит из клиентского компьютера или сети компьютеров предприятия и приложений, используемых для доступа к облаку. Внутренняя часть предоставляет приложения, компьютеры, серверы и хранилища данных, создающие облако сервисов.

Концепция облака основана на уровнях**,** каждый из которых предоставляет определенную функциональность (рис.2.1). Такое деление компонентов облака позволяет сделать уровни облачных вычислений коммунальным ресурсом, аналогичным электричеству, услугам телефонии или природному газу. Товар "облачные вычисления" - это более дешевые и менее затратные для пользователя вычислительные ресурсы. В перспективе облачные вычисления могут стать еще одним коммунальным ресурсом.

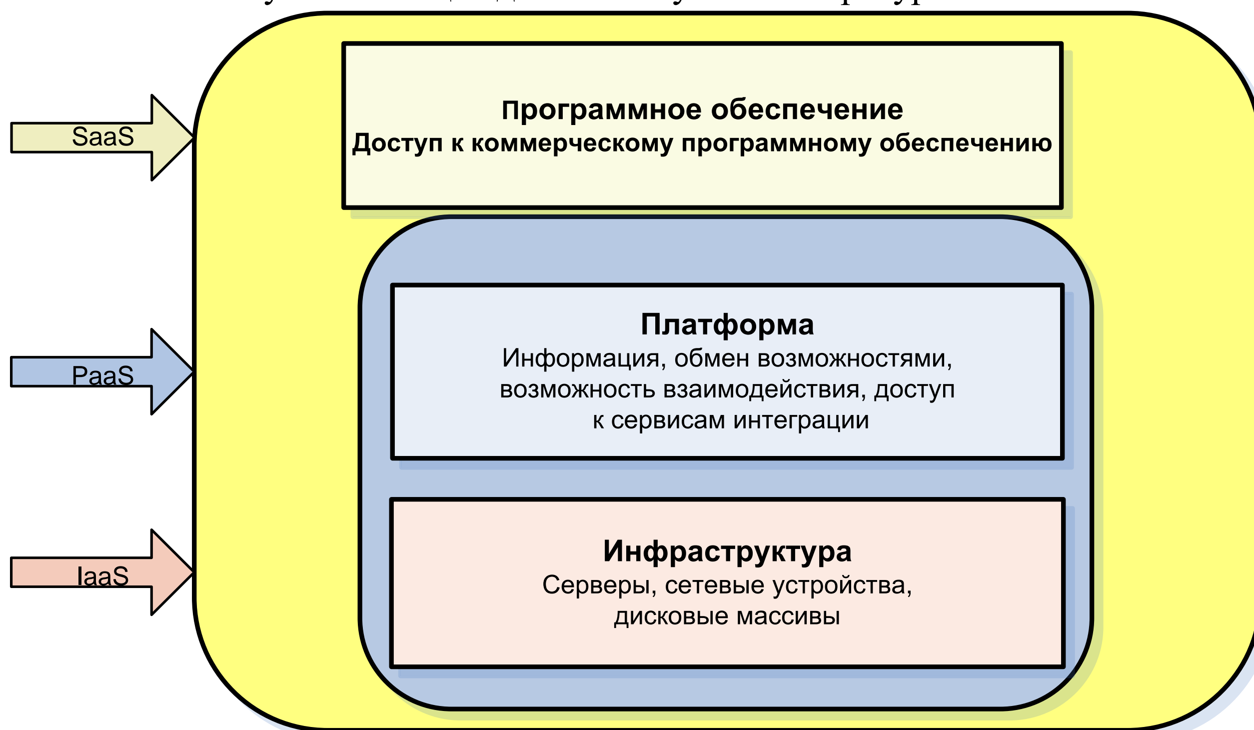

Рис.2.1. Уровни облачных вычислений

Основа облака - уровень *инфраструктуры*. Он состоит из физических активов - серверов, сетевых устройств, дисков и т.д. Существуют поставщики инфраструктуры как сервиса (Infrastructure as a Service - IaaS), например IBM® Cloud.

Промежуточным уровнем является *платформа*. Она предоставляет инфраструктуру приложений. Платформа как сервис (Platform as a Service - PaaS) предоставляет доступ к операционным системам и соответствующим

сервисам. Она дает способ развертывания приложений в облаке при помощи языков программирования и инструментальных средств, поддерживаемых поставщиком. Существуют поставщики PaaS, например Elastic Compute Cloud (EC2) от Amazon.

Верхний уровень - это уровень *приложений*, который обычно и изображают в виде облака. Приложения, выполняющиеся в нем, предоставляются пользователям по требованию. Существуют поставщики программного обеспечения как сервиса (Software as a Service - SaaS), например, Google Pack. Google Pack содержит доступные через Интернет приложения - Calendar, Gmail, Google Talk, Docs и многие другие.

На рис.2.2 представлены модели развертывания облака (Deploument Models).

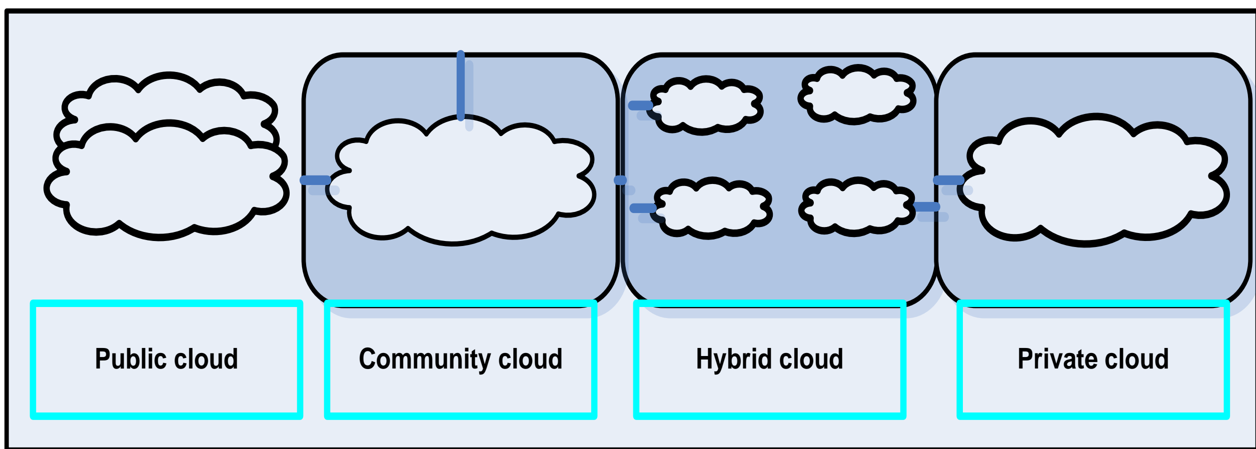

Рис.2.2. Модели развертывания

Публичное облако (Public cloud) - облачная инфраструктура создана в качестве общедоступной или доступной для большой группы пользователей. Такая инфраструктура находится во владении организации, продающей соответсвующие облачные услуги/сервисы.

Облако сообщества или общее облако (Community cloud) - облачная инфраструктура используется совместно несколькими организациями и поддерживает ограниченное сообщество, разделяющее общие принципы. Такая облачная инфраструктура может управляться самими организациями или третьей стороной и может существовать как на стороне пользователя, так и у внешнего провайдера.

Гибридное облако (Hybrid cloud) - облачная инфраструктура является композицией (сочетанием) двух и более облаков (частных, общих или бубличных), остающихся уникальными сущностями, но объединенными вместе стандартизированными или частными технологиями, обеспечивающими портируемость данных и приложений (например, такими технологиями, как пакетная передача данных для баланса загрузки между облаками).

Частное облако (Private cloud) - облачная инфраструктура функционирует целиком в целях обслуживания одной организации. Инфраструктура может управляться самой организацией или третьей стороной и может существовать как на стороне потребителя, так и внешнего провайдера.

Каждая из моделей частного облака и облака сообщества допускают два варианта сценария развертывания, которые должны рассматриваться по отдельности, по причине влияния на периметр безопасности: будет ли он собственный (on-site) или на аутсоурсинге (outsourced). Гибридная модель развертывания является комбинацией других моделей и, поэтому, гибридное развертывание может предполагать и влияние на периметр безопасности его элементов - "строительных блоков", и уникальные аспекты влияния, возникающие в результате объединения множества систем в комплексные интегрированные системы.

Контрольные вопросы

1. Что понимается под термином «облачные вычисления»?
2. Поясните модели развертывания облака.
3. Что является основой облака?
4. Перечислите и поясните основные характеристики облака.
5. Какой уровень предоставляет доступ к операционным системам и соответствующим сервисам?
6. Что представляет уровень инфраструктуры?
7. Перечислите сценарии развертывания облака.
8. Что понимается под арендой сервиса?

2.2. Сервисы, представляемые облаком

2.2.1. Облачные вычисления и предоставляемые ими сервисы

Существуют следующие модели представления сервисов:

- *Storage-as-a-Service* (“хранение как сервис”). Данный сервис является базовым для нижеперечисленных сервисов, поскольку входит в состав практически каждого из них. Он является самым простым из сервисов, представляющим собой дисковое пространство по требованию. Услуга Storage-as-a-Service дает возможность сохранять данные во внешнем хранилище, в “облаке”, который будет выглядеть, как дополнительный логический диск или папка. Примером может служить Google Drive и другие схожие сервисы.

- *Database-as-a-Service* (“база данных как сервис”). Данный сервис предоставляет возможность работать с базами данных, как если бы СУБД была установлена на локальном ресурсе.

- *Information-as-a-Service* (“информация как сервис”) - предоставляет возможность удаленно использовать любые виды информации, которая может меняться ежесекундно.

- *Process-as-a-Service (“*управление процессом как сервис”) - представляет собой удаленный ресурс, который может связать воедино

несколько ресурсов (таких, как услуги или данные, содержащиеся в пределах одного “облака” или других доступных “облаков”) для создания единого бизнес-процесса.

- *Application-as-a-Service* (“приложение как сервис”) или *Software-as-a-Service* (“программное обеспечение как сервис”) - позиционируется как «программное обеспечение по требованию», которое развернуто на удаленных серверах и каждый пользователь может получать к нему доступ посредством Интернет, причем все вопросы обновления и лицензий на данное обеспечение регулируется поставщиком данной услуги.

- *Platform-as-a-Service* (“платформа как сервис”) - данный сервис предоставляет пользователю компьютерную платформу с установленной операционной системой и некоторым программным обеспечением.

- *Integration-as-a-Service* (“интеграция как сервис”) - предоставляет возможность получать из “облака” полный интеграционный пакет, включая программные интерфейсы между приложениями и управление их алгоритмами. Сюда входят известные услуги и функции пакетов централизации, оптимизации и интеграции корпоративных приложений (EAI), но предоставляемые как “облачный” сервис.

- *Security-as-a-Service* (“безопасность как сервис”) - данный вид услуги предоставляет возможность пользователям быстро развертывать продукты, позволяющие обеспечить безопасное использование web-технологий, электронной почты и т.п., что позволяет пользователям данного сервиса экономить на развертывании и поддержании своей собственной системы безопасности.

- *Management/Governace-as-a-Service* (“администрирование и управление как сервис”) - данный сервис предоставляет возможность управлять и задавать параметры работы одного или многих “облачных” сервисов (такие параметры, как топология, использование ресурсов, виртуализация).

- *Infrastructure-as-a-Service* (“инфраструктура как сервис”) - пользователю предоставляется компьютерная инфраструктура, обычно виртуальные платформы (компьютеры), связанные в сеть, которые он самостоятельно настраивает под собственные цели.

- *Testing-as-a-Service* (“тестирование как сервис”) - данный сервис предоставляет возможность тестирования локальных или “облачных” систем с использованием тестового программного обеспечения из “облака” (при этом дополнительного оборудования или программного обеспечения на предприятии не требуется).

2.2.2. Границы управляемости. Модели развертывания собственной инфраструктуры

Обсуждая различные типы облачных сервисов - программное обеспечение, платформу и инфраструктуру как сервис, следует обращать внимание на так называемые границы управляемости - т.е. на то, чем, в сравнении с традиционными моделями развертывания в собственной

инфраструктуре, можно управлять при переходе на облачную платформу. По понятным причинам, инфраструктура как сервис предоставляет большие возможности по настройке отдельных компонентов, тогда как платформа как сервис и программное обеспечение как сервис практически минимизируют эти возможности.

Отличия в границах управляемости показаны на рис.2.3. Как видно из рисунка, при развертывании собственной инфраструктуры потребитель управляете всеми ее компонентами - от сетевых ресурсов до выполняющихся приложений. Тогда как при использовании модели IaaS потребитель можете контролировать такие компоненты, как среда исполнения кода, безопасность и интеграция, базы данных, и т.п. При переходе к модели PaaS, все компоненты платформы предоставляются как сервисы с ограниченными возможностями для управления ими. Это сделано, чтобы предоставить в распоряжение потребителей оптимально сконфигурированную платформу, не требующую дополнительных настроек. При использовании облачных вычислений затраты потребителя смещаются в сторону операционных, таким образом классифицируются расходы на оплату услуг облачных провайдеров.

Для объяснения экономической составляющей облачных подходов к вычислениям часто используется аналогия с услугами электроснабжения, предоставляемыми в развитых инфраструктурах по соответствующим коммунальным сетям, легкодоступными и оплачиваемыми по мере потребления, в сравнении с разработкой каждым потребителем собственного водозабора или монтированием собственной электроустановки. На самом деле наиболее близкая аналогия - это банковская система, когда предпринимателям даются финансовые средства в виде кредитов, позволяющие осуществлять экономическую деятельность большего масштаба с малым собственным капиталом.

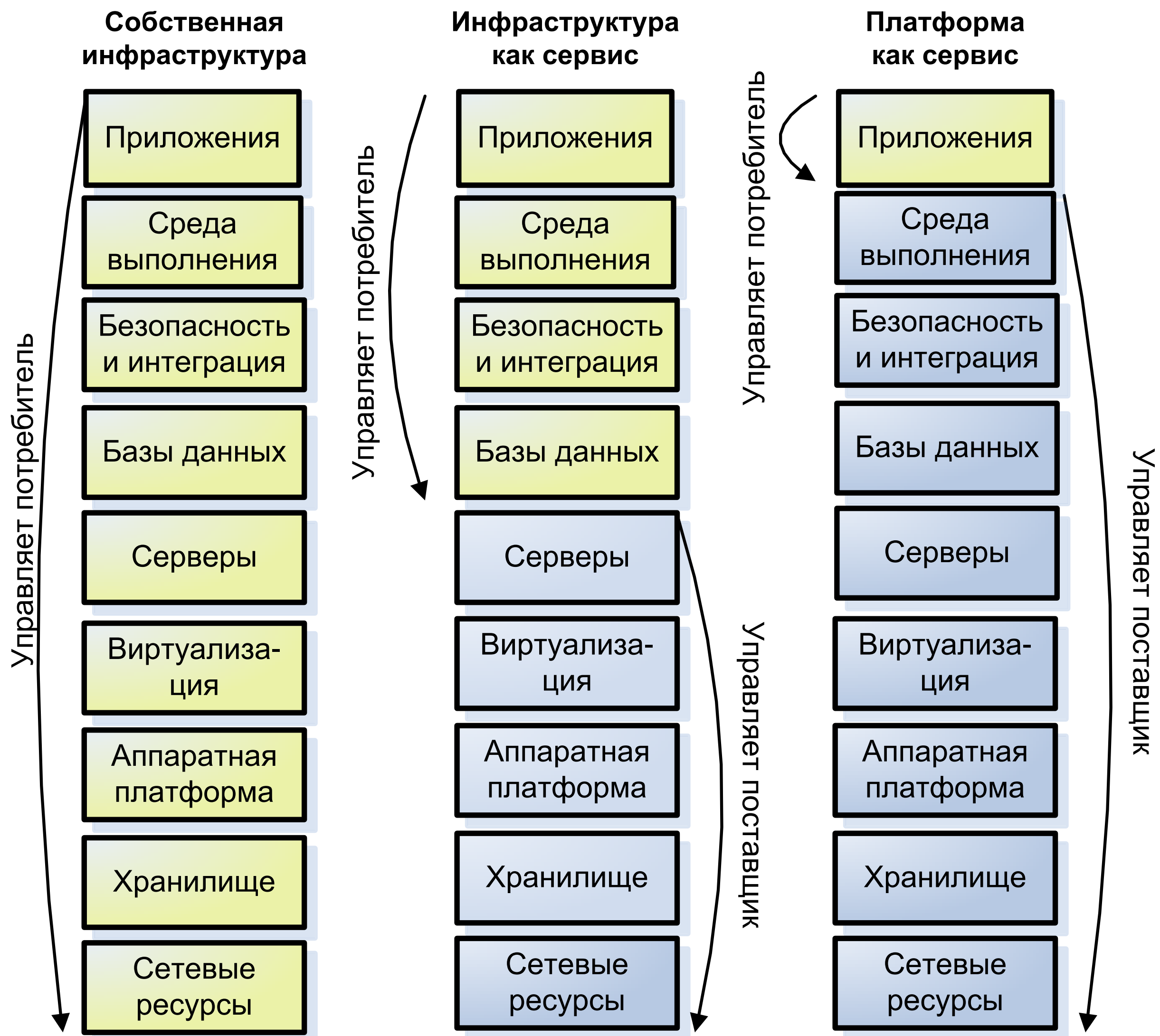

Рис. 2.3. Границы управляемости

Для обеспечения согласованной работы узлов вычислительной сети на стороне облачного провайдера используется специализированное промежуточное программное обеспечение, обеспечивающее мониторинг состояния оборудования и программ, балансировку нагрузки, обеспечение ресурсов для решения задачи.

Одним из основных решений для сглаживания неравномерности нагрузки на услуги является размещение слоя серверной виртуализации между слоем программных услуг и аппаратным обеспечением. В условиях виртуализации балансировка нагрузки может осуществляться посредством программного распределения виртуальных серверов по реальным серверам. Перенос виртуальных серверов происходит посредством живой миграции.

С точки зрения клиента облачная служба должна предоставлять вычислительные мощности по запросу, а единственным ограничителем мощности является сумма, которую клиент готов заплатить. Более высокий уровень доступности и устойчивости обеспечивается в облаке путем замены

традиционной модели физической избыточности на программные средства.

Первым из таких программных средств является виртуализация. Она позволяет отделить службу от определенного сервера, повышая тем самым возможность переноса службы на другие ресурсы. Второй программный инструмент - гипервизор. Технологии, предоставляемые гипервизором, могут обеспечить либо прозрачный перенос, либо перезапуск нагрузки на других виртуальных серверах.

Устойчивость и доступность повышаются без использования каких-либо прочих специализированных программ, выполняющихся в составе нагрузки. Избыточность больше не нужна для вычислительных компонентов, но по-прежнему необходима для хранилищ. Кроме того, требуется избыточность и сетевых компонентов (чтобы поддерживать потребности хранилищ). Текущие требования к сетям и хранилищам исключают полный отказ от избыточности, но все же можно сэкономить немало средств, отказавшись от избыточности вычислительных систем.

В отличие от традиционных центров обработки данных, облако восстанавливает работоспособность за считанные секунды. Поэтому такой показатель, как доступность (количество времени, в течение которого система работала в штатном режиме, за год), уже не является главным показателем успешности предоставления ИТ-услуг. Показателями успеха стало ощущение доступности и влияние недоступности на работу компаний. Это показано на приведенной ниже иллюстрации [6] (рис. 2.4).

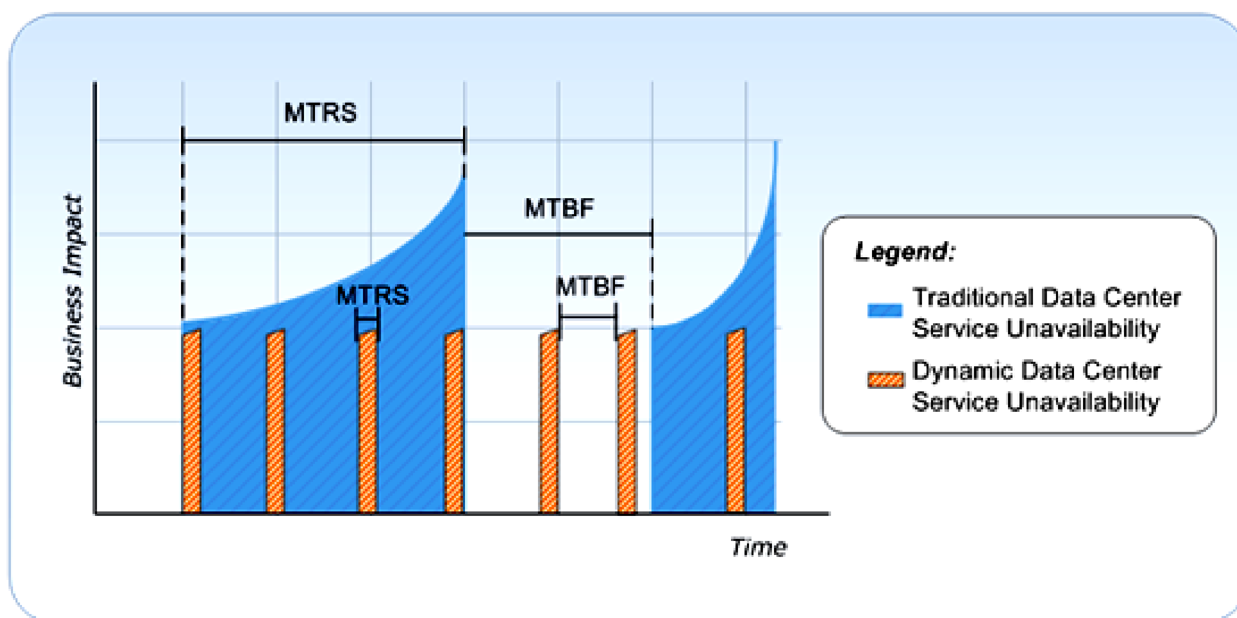

Рис. 2.4. Показатели успеха

2.2.3. Проблема создания неконтролируемых данных

Существует вероятность, что с повсеместным приходом облачной технологии станет очевидной проблема создания неконтролируемых данных, когда информация, оставленная пользователем, будет храниться годами без его ведома, или он будет не в состоянии изменить какую-то её часть. Примером того могут служить сервисы Google, где пользователь не в состоянии удалить неиспользуемые им сервисы и даже удалить отдельные группы данных, созданные в некоторых из них (в качестве альтернативы «очистке» своего профиля предлагается создать новый). Поскольку облачные вычисления будут всецело проприетарны (открытый API не исправляет ситуацию), рассчитывать на то, что пользователю предоставят средство для удаления своих же данных на подобных серверах не приходится.

Некоторые аналитики, например, Марк Андерсон, руководитель отраслевого IT-издания Strategic News Service, считает, что из-за значительного притока пользователей сервисов, использующих облачные вычисления, растёт стоимость ошибок и утечек информации с подобных ресурсов, а также возможны крупные «катастрофы типа выхода из строя, или катастрофы, связанные с безопасностью». Тем не менее, многие эксперты придерживаются точки зрения, что преимущества и удобства перевешивают возможные риски использования облачных сервисов.

Архитектура облачных вычислений содержит пять главных действующих субъектов – актеров. Каждый актер выступает в роли и выполняет действия и функции. Возможны следующие сценарии использования:

1) облачный потребитель может запросить услугу (сервис) у облачного брокера вместо прямого контактирования с облачным провайдером. Облачный брокер может создать новый сервис, комбинируя набор сервисов или расширяя существующий сервис. В этом примере облачный провайдер невидим облачному потребителю;

2) облачный оператор связи предоставляет услуги подключения и транспорт (доставку) облачных услуг от облачного провайдера к облачному потребителю. Облачный провайдер устанавливает соглашение об уровне обслуживания (SLA – Service Level Agreement) с облачным оператором и может запрашивать выделенные и защищенные соединения;

3) облачный аудитор проводит независимую оценку обслуживания и безопасности реализации облачной услуги.

Подробное описание актеров эталонной архитектуры облачных вычислений и их взаимодействия предоставлены в табл.2.1.

Таблица 2.1

Актеры эталонной архитектуры Cloud Computing

Актер	Определение
облачный потребитель (Cloud Consumer)	Лицо или организация, поддерживающая бизнес отношения и использующая услуги облачных провайдеров
облачный провайдер (Cloud Provider)	Лицо, организация или сущность, отвечающая за доступность облачной услуги для облачных потребителей
облачный аудитор (Cloud Auditor)	Участник, который может выполнять независимую оценку облачных услуг, обслуживания информационных систем, производительности и безопасности реализации облака
облачный брокер (Cloud Broker)	Сущность, управляющая использованием, производительностью и предоставлением облачных услуг, а также устанавливающая отношения между облачными провайдерами и облачными потребителями
облачный оператор Связи (Cloud Carrier)	Посредник, предоставляющий услуги подключения и транспорт (доставки) облачных услуг от облачных провайдеров к облачным потребителям

Среди представленных пяти актеров облачный брокер (cloud broker) – опционален, так как облачные потребители (cloud consumers) могут получать услуги напрямую от облачного провайдера (cloud provider).

Облачные потребители подразделяются на три группы, основанные на их приложениях/различных сценариях использования (табл. 2.2).

Таблица 2.2.

Деятельность облачного пользователя

Тип потребителя	Основная деятельность	Примеры пользователей
SaaS	Использует приложения/сервисы для автоматизации бизнес-процессов	Бизнес-пользователи, администраторы приложений
PaaS	Разрабатывает, тестирует, развертывает и управляет приложениями, развернутыми в облачном окружении	Разработчики приложений, тестировщики, администраторы
IaaS	Создает/устанавливает, управляет и мониторит сервисы для управления ИТ-инфраструктурой	Системные разработчики, администраторы, ИТ-менеджеры

Облачные провайдеры выполняют различные задачи в различных сервисных моделях (табл. 2.3).

Таблица 2.3

Деятельность облачного провайдера

Тип провайдера	Основная деятельность
SaaS	Устанавливает, управляет, сопровождает и поддерживает программное обеспечение, развернутое на облачной инфраструктуре
PaaS	Предоставляет и управляет облачной инфраструктурой и связующим программным обеспечением платформы для потребителей; предоставляет инструменты разработки, развертывания и администрирования потребителям платформы
IaaS	Предоставляет и управляет физическими вычислительными процессами, системами хранения, сетями и хостинг-окружением, а также облачной инфраструктурой для IaaS-потребителей

Контрольные вопросы

1. Перечислите модели представления сервисов.
2. Какой облачный сервис является базовым?
3. Какой сервис позволяет тестировать локальные системы без дополнительного оборудования и ПО?
4. Что понимается под границей управляемости?
5. Поясните механизм согласованной работы узлов вычислительной сети на стороне облачного провайдера.
6. Что понимается под приетарностью вычислений?

2.3. Модели обслуживания

Одним из ключевых отличий облака от традиционных ЦОД и серверных инфраструктур является абстрагирование от таких физических ресурсов, как серверы, сети и дисковые ресурсы. Они объединяются на более высоком уровне в пулы ресурсов, дефектные домены, домены обновления и т.д. Эти логические объединения относятся к физической инфраструктуре и помогают принимать информированные решения по подготовке и управлению ресурсами. Провайдеры Cloud Computing предлагают свои услуги в соответствии с тремя основными моделями: SaaS (Cloud Software as a Service), PaaS (Cloud Platform as a Service), IaaS (Cloud Infrastructure as a Service).

2.3.1. Программное обеспечение как услуга (SaaS)

При модели обслуживания «Программное обеспечение как услуга - SaaS (Cloud Software as a Service)» пользователю предоставляются программные средства (приложения провайдера), выполняемые на облачной инфраструктуре. Приложения доступны с различных клиентских устройств через интерфейс тонкого клиента (например, браузер). Пользователь не управляет и не контролирует саму облачную инфраструктуру, на которой выполняется приложение, будь то сети, серверы, операционные системы,

системы хранения или даже некоторые специфичные для приложений возможности. В ряде случаев пользователю может быть предоставлена возможность доступа к некоторым пользовательским конфигурационным настройкам.

2.3.2. Платформа как услуга (PaaS)

Платформа как услуга - PaaS (Cloud Platform as a Service). Пользователю предоставляются платформы с определенными характеристиками для разработки, тестирования, развертывания, поддержки Web-приложений и т.д. Благодаря модели PaaS весь перечень операций по разработке, тестированию и разворачиванию Web-приложений можно выполнить в одной интегрированной среде, тем самым исключив затраты на поддержку отдельных сред для конкретных этапов. Это позволяет существенно снизить затраты как на приобретение и поддержку оборудования, так и на обслуживание самого сервиса. Примером использования такой модели - предоставление услуги хостинга для Web-сайтов.

2.3.3. Инфраструктура как услуга (IaaS)

При модели обслуживания «Инфраструктура как услуга – IaaS (Cloud Infrastructure as a Service)» пользователю предоставляются средства обработки данных, хранения, сетей и других базовых вычислительных ресурсов, на которых пользователь может разворачивать и выполнять произвольное программное обеспечение, включая операционные системы и приложения. Пользователь не управляет и не контролирует саму облачную инфраструктуру, но в отличие от модели SaaS может контролировать операционные системы, средства хранения, развертываемые приложения и, возможно, обладать ограниченным контролем над выбранными сетевыми компонентами.

Различные модели развертывания облаков, описанные в определении облачных вычислений NIST, подразумевают размещение контролируемого подписчиком периметра безопасности и, следовательно, уровень контроля, который подписчики могут осуществлять в отношении ресурсов, доверяемых облаку (т.е. передаваемых под управление провайдера облака с определенным уровнем доверия этому провайдеру).

При сравнении использования облаков с традиционной моделью внутренних” (on premises) вычислений, облачная модель требует от подписчиков передачи провайдеру двух важных возможностей, предполагающих высокий уровень доверия подписчика к провайдеру:

- контроль, т.е. возможность решать, кто и что может получать доступ к данным и программам подписчика, и выполнять действия, которые должны быть произведены, но без дополнительных действий, не соответствующих намерениям подписчика (например, подписчик запрашивает удаление объектов данных, при этом не должно выполняться их неявное копированием по инициативе провайдера);

- видимость или явность действий, т.е. возможность осуществления мониторинга статуса программ и данных подписчика и того, как к ним осуществляется доступ.

Однако границы контроля и видимости, которые необходимо передать провайдеру со стороны подписчиков, зависят от многих факторов, включая физическое владение и возможность конфигурирования (с высоким уровнем доверия) механизмов защиты границ доступа к вычислительным ресурсам подписчиков. Именно стандарты и рекомендации NIST в области компьютерной/информационной безопасности, в большой степени, определили фундамент и продолжают определять концептуальную основу и подходы к теории и практическому обеспечению безопасности в индустрии информационных технологий.

Типичные контроллеры границ включают сетевые экраны (firewalls), средства блокирования доступа (guards) и виртуальные частные сети. Организация может добиться оценки количественных показателей как в отношении контроля использования ресурсов, так и мониторинга доступа к ним. Более того, переконфигурируя периметр безопасности, организация может адаптировать его к меняющимся потребностям (например, блокирования или разрешения протоколов или форматов данных, исходя из изменений бизнес-условий).

Контрольные вопросы

1. Назовите отличие облака от традиционных ЦОД.
2. Перечислите модели обслуживания в облаке.
3. Какая модель обслуживания в облаке позволяет выполнить в одной интегрированной среде операции по разработке, тестированию и разворачиванию Web-приложений?
4. Какая инфраструктура позволяет пользователю обладать ограниченным контролем над выбранными сетевыми компонентами?
5. Перечислите факторы, влияющие на границы контроля и видимости.
6. Что понимается под термином «контроллер границ»?

ЧАСТЬ 3. АРХИТЕКТУРА WEB-СЕРВИСОВ

3.1. Web-ориентированная архитектура (WOA)

3.1.1. Устройство web-приложений

Web-приложение - это приложение, разработанное по архитектуре «клиент-сервер», использующее в качестве клиента web-браузер и работающее с использованием протокола HTTP на стороне web-сервера (рис. 3.1).

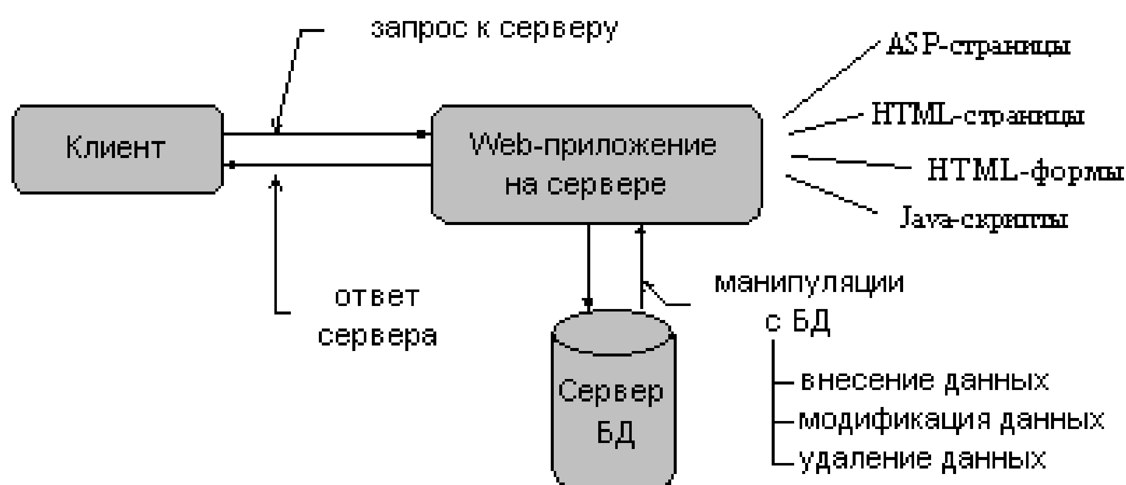

Рис. 3.1. Общая схема взаимодействия пользователя с web-приложением

Пользователь с помощью браузер отправляет HTTP-запрос по определенному URL-адресу, который сопоставляются с ресурсами на web-сервере. Если данный URL-адрес указывает на динамический ресурс (web-приложение), то сервер запускает некоторую внешнюю программу (web-приложение), формирующую HTML-страницу, которая может быть показана браузером, и возвращает ее клиенту. Основной частью web-приложения является его логика на стороне web-сервера.

Чаще всего web-приложение не является самостоятельной программой (кроме технологии CGI), а выполняется под управлением некоторой внешней программы - среды выполнения. В качестве такой программы могут выступать: сервер приложений или контейнер приложений. Среда выполнения делает следующие действия:

- принимает от web-сервера всю информацию, связанную с HTTP запросом;
- определяет, какое приложение, созданное для данной среды, должно быть выполнено;
- загружает запрашиваемое приложений и передает ему управление;

- создает все вспомогательные объекты, которые могут потребоваться для работы данного приложения (такие, как, например, Application, Session, Request, Response, User и т.п.);

- web-приложение выполняется и создает HTML страницу, которую записывает в специально созданный для этого объект (например, объект Response) и после этого возвращает управление web-серверу;

- web-сервер формирует http-ответ, включает в него сформированную web-приложением HTML-страницу и отправляет его клиенту, который прислал запрос к данному web-приложению.

Последовательность вызовов web-приложения пользователем составляют сеанс его работы.

Можно выделить следующие примеры типичных web-приложений: поисковые системы, новостные системы, видеоколлекции, социальные сети, Интернет-магазины, хостинги файлов, онлайн-органайзер, системы онлайн-покупки билетов на различные виды транспорта и т.п.

Приложения обычно делятся на логические части, называемые «слоями», при этом, каждому слою назначается своя роль. Локальные приложения могут состоять только из одного слоя, который размещается на компьютере клиента, а web-приложения по своей природе следуют N-слойному подходу. Хотя возможны разные варианты, наиболее распространенными являются приложения, использующие три слоя: слой представление; слой бизнес-логика; слой доступ к данным (хранилище). Каждый слой включает набор компонент (наборов классов), выполняющих специальные функции.

На рис.3.2 показана типовая архитектура web-приложения с набором наиболее часто применяемых компонент, сгруппированных по функциональным областям.

Слой представления обычно включает компоненты пользовательского интерфейса (UI) и логику представления. Слой бизнес-логики включает компоненты бизнес-логики, бизнес-процесса и бизнес-сущностей. Слой доступа к данным включает компоненты, реализующие доступ к требуемым данным и web-сервисам.

Web-приложение также можно разделить на базовые и функциональные подсистемы. К базовым подсистемам web-приложения относятся:

- единообразное оформление web-страниц, составляющих приложение;
- поддержка состояния сеанса работы пользователей;
- персонализация web-страниц;
- навигация между web-страницами;
- обеспечение безопасности (аутентификация и авторизация, регистрация пользователей);
- доступ к источникам данных.

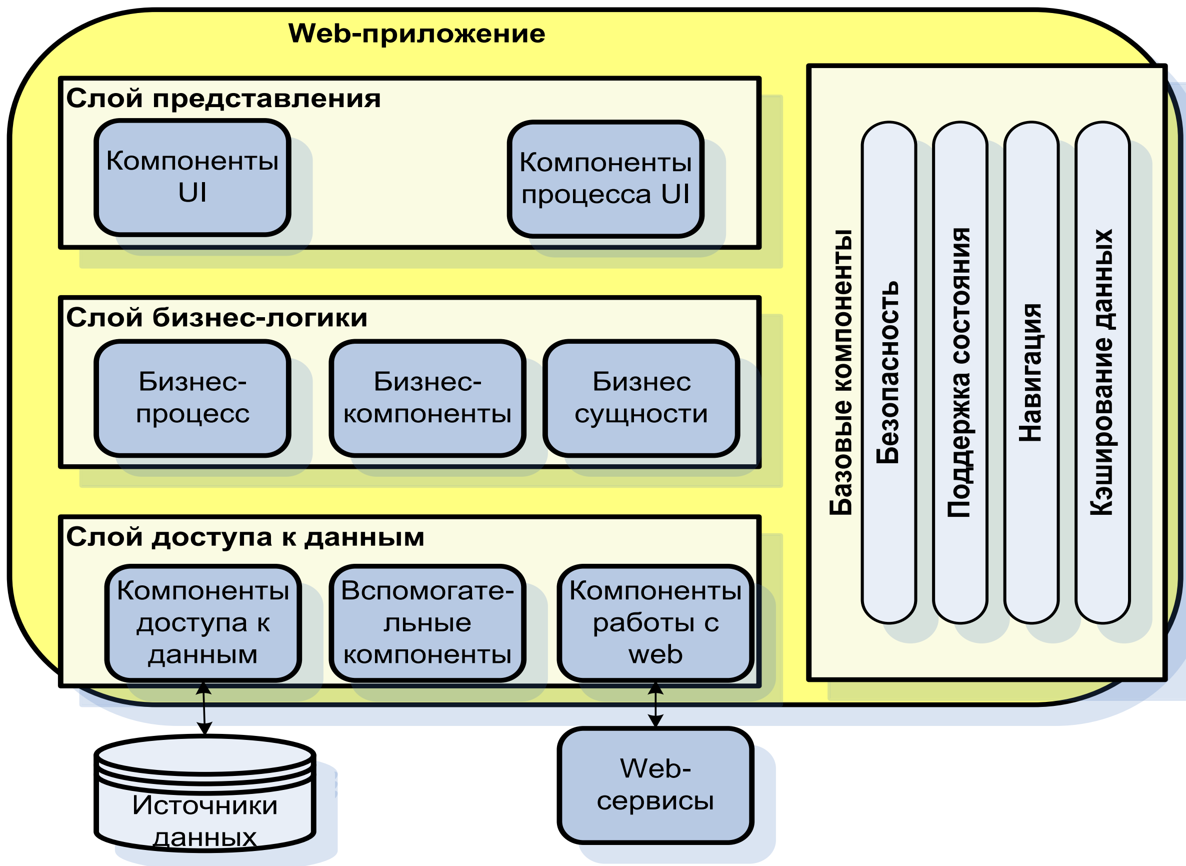

Рис.3.2. Типовая структура web-приложения

К функциональным подсистемам относятся:

- управление контентом web-приложения. Web-приложение должно предоставлять возможности загрузки контента на сервер (uploading), классификации, поиска и получения с сервера (downloading);
- поиск информации в контенте приложения;
- поддержка взаимодействия пользователей (например, форумы);
- специальные возможности (например, расчеты в соответствии с бизнес-логикой).

Базовая функциональность во многом реализуется средой выполнения web-приложения, а для создания функциональных подсистем требуется составление программного кода.

Web-приложения в сравнении с локальными приложениями имеют следующие преимущества:

- простота доступа к приложению. Любой человек, имеющий компьютер, подключенный к сети Интернет, может использовать web-приложение;
- простота развертывания (установки). В отличие от локальных приложений, web-приложения после завершения разработки не требуется устанавливать на компьютерах пользователей (достаточно сообщить URL-адрес приложения). При изменении приложения все пользователи сразу начинают работать с измененной версией;
- наличие большого количества обученных пользователей;

-высокий уровень развития и надежности сетевых соединений и web-технологий.

Однако web-приложения имеют и свои недостатки:

- слабосвязанная архитектура web-сети: отсутствие поддержки состояния сеанса работы и задержка при перезагрузке каждой страницы. Это создает прерывистый режим работы пользователя;

-ограниченный набор элементов управления для проектирования форм приложения, так как отсутствие встроенной поддержки браузеров ведет к разнообразию реализаций с несогласованными представлениями и способами работы с ними;

- несогласованные подходы к оформлению и способу взаимодействия пользователями. Это вызвано тем, что пользователи сталкиваются с большим разнообразием стилей визуальных интерфейсов и способов взаимодействия, каждый из которых предлагает свой словарь объектов, действий и визуальных представлений, смешанных в одном и том же приложении.

Кроме web-приложений другим важным видом используемого в web-сети программного обеспечения являются web-сервисы (web services).

Web-сервисы представляют собой наборы методов, которые можно вызывать на выполнение с заданными параметрами с помощью http-запросов, а результаты их выполнения получать в виде http-ответов. Web-сервисы поддерживаются, как и web-приложения, web-серверами. Они предназначены для использования любыми программами, которые могут формировать правильные http-запросы и понимать полученные http-ответы, но не пользователями, так как они не имеют графического пользовательского интерфейса. Такими программами могут быть web-приложения, либо другие web-сервисы, или даже локальные приложения.

3.1.2. Службы web

Wеб-службы - это распределенные компоненты приложений, доступ к которым осуществляется извне (протокол http), а обмен данными происходит в формате XML или JSON, используя один из трех наиболее распространенных архитектурных стилей проектирования приложений: Remote Procedure Call (RPC), SOAP или REST. Их можно использовать для интеграции компьютерных приложений, написанных на различных языках программирования и выполняемых на различных платформах. Wеб-службы не зависят от языка и платформы, так как между поставщиками существует договоренность об общих стандартах wеб-служб.

В отличие от обычных динамических библиотек такой подход обладает рядом плюсов:

- wеб-служба находится на серверах компании, которая её создала. Поэтому в любой момент пользователю доступна актуальная версия данных и нет необходимости заботиться об обновлениях и вычислительных мощностях, требуемых для выполнения операции;

- инструменты для работы с http и XML есть в любом современном языке программирования, поэтому web-службы переходят в разряд платформонезависимых.

В качестве примера можно привести проект Oracle java.net под названием Metro. В состав Metro входит поддержка технологий взаимодействияweб-служб (Web Services Interoperability Technologies – WSIT). WSIT поддерживает функции уровня корпораций, такие как обеспечение безопасности, надежности и оптимизация сообщений.

Можно выделить три инстанции, взаимодействующие в рамках web-службы: заказчик (service requestor); исполнитель (service provider); каталог (service broker). Архитектура web-службы представлена на рис.3.3.

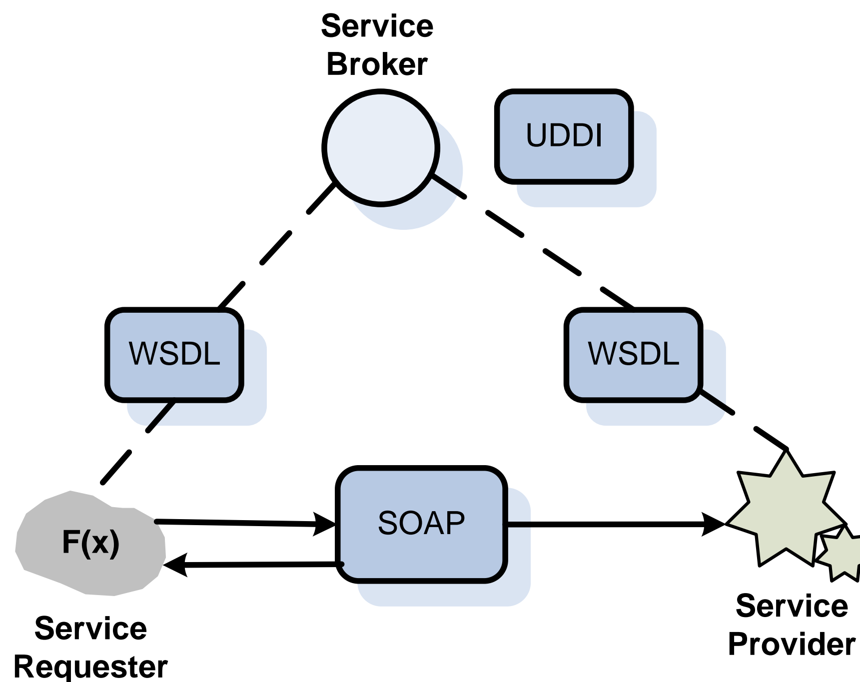

Рис.3.3. Архитектура web-службы

Когда служба разработана, исполнитель регистрирует её в каталоге, где её могут найти потенциальные заказчики. Заказчик, найдя в каталоге подходящую службу, импортирует оттуда её WSDL-спецификацию и разрабатывает в соответствии с ней свое программное обеспечение. WSDL описывает формат запросов и ответов, которыми обмениваются заказчик и исполнитель в процессе работы. Для обеспечения взаимодействия используются следующие стандарты:

- XML - расширяемый язык разметки, предназначенный для хранения и передачи структурированных данных;

- SOAP - протокол обмена сообщениями на базе XML;

- WSDL - язык описания внешних интерфейсов web-службы на базе XML;

- UDDI -универсальный интерфейс распознавания, описания и интеграции (Universal Discovery, Description and Integration).

Разработчики wеб-служб используют несколько моделей программирования. Эти модели можно разделить на следующие две категории, каждая из которых поддерживается в среде IDE:

- *на основе REST* (REpresentational State Transfer - передача состояния представления) - это новый способ создания и взаимодействия с wеб-службами. В REST ресурсы имеют идентификаторы URI, и управление ими происходит через операции с заголовками http.

- *на основе SOAP/WSDL.* В стандартных моделях wеб-служб, интерфейсы цеб-служб предоставляются с помощью документов WSDL (тип XML) с URL-адресами. Последующий обмен сообщениями осуществляется через SOAP, другой тип документа XML.

- *wеб-службы RESTful* - это коллекция веб-ресурсов, идентифицируемых по своим URI. Каждый документ и каждый процесс смоделирован как wеб-ресурс с уникальным идентификатором URI. Управление осуществляется с помощью действий, указанных в заголовке HTTP. Стандарты SOAP, WSDL и WS-* не используются, т.к. обмен сообщениями может быть проведен в любом формате – XML, JSON, HTML и т.д. Во многих случаях клиентом может служить веб-браузер.

Службы REST предоставляют весьма сложные функциональные возможности. Wеб-службы RESTful применяются такими компаниями, как Flickr, Google Maps и Amazon. Программное обеспечение IDE NetBeans как "программное обеспечение как услуга" (SaaS) позволяет использовать Facebook, Zillow и другие службы сторонних производителей в своих собственных приложениях.

В wеб-службах на основе SOAP служебные программы Java создают в wеб-службе файл WSDL, основанный на коде Java. Обмен сообщениями происходит в формате SOAP. Спектр операций, которые могут быть переданы в SOAP, намного шире спектра REST, особенно в области безопасности. Wеб-службы на основе SOAP подходят и для крупных приложений, использующих сложные операции, и для приложений, требующих усложненной защиты, надежности или других поддерживаемых стандартами функций WS-*. Они подходят также в тех случаях, когда необходимо использовать транспортный протокол, отличный от HTTP. Многие веб-службы Amazon, в частности те, которые связаны с коммерческими транзакциями, и wеб-службы, используемые банками и правительственными агентствами, основаны на SOAP.

3.1.3. Протокол SOAP

На сегодняшний день существует множество технологий и протоколов, позволяющих объединять элементы распределенных систем между собой. Одна из наиболее известных технологий - DCOM, позволяющая эффективно осуществлять RPC-вызовы, передавать и принимать данные, распределять нагрузку между несколькими back-end серверами. Однако у систем,

построенных на DCOM, есть очень важный недостаток, затрудняющий взаимодействие уровня представления и уровня бизнес-логики через Internet. Большинство современных сетевых экранов будут запрещать передачу таких пакетов из соображений безопасности. В распределенных системах для обеспечения взаимодействия разных уровней используется SOAP.

Simple Object Access Protocol (SOAP) – основанный на XML протокол, предназначенный для обмена информацией в распределенных системах. SOAP устанавливает стандарт взаимодействия клиент-сервер и регламентирует, как должен осуществляться вызов, передаваться параметры и возвращаемые значения. Для представления любой информации, передаваемой от клиента к серверу и наоборот, используется XML [7,8].

SOAP нельзя рассматривать как полную замену DCOM, так как DCOM поддерживает специфические технологии, которые сложно или практически невозможно реализовать для SOAP-вызовов – управление временем жизни объектов (DCOM ping), передача объектных ссылок, COM-события. Поэтому в общем случае для перехода с DCOM на SOAP может потребоваться частичное (или даже полное) изменение архитектуры приложения.

К достоинству протокола SOAP следует отнести то, что он нейтрален к платформе, т.е. не накладывает ограничений на платформы, которые используются клиентом и сервером.

SOAP описывает преобразование в XML следующих элементов вызова:

- запрос (SOAP Request);
- отклик (SOAP Response);
- сообщение об ошибке (SOAP Fault).

1) SOAP Request: вызов метода по протоколу SOAP преобразуется в XML следующим образом:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope ...>
 <SOAP-ENV:Body ...>
 <SOAPSDK4:Add ...>
   <x>1</x>
   <y>2</y>
 </SOAPSDK4:Add>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В приведенном примере запроса вызывается метод Add, которому передаются два параметра: 1 и 2. Информация о том, какой метод и у какого объекта необходимо вызвать, передается в заголовке. В случае протокола HTTP заголовок может выглядеть следующим образом:

```
<HTTPHeaders>
```

```
<soapaction>"http://tempuri.org/Sample1/action/Adder.Add"</soapaction>
  <content-type>text/xml; charset="UTF-8"</content-type>
  <user-agent>SOAP Toolkit 3.0</user-agent>
  <host>aida:8080</host>
  <content-length>516</content-length>
  <connection>Keep-Alive</connection>
  <cache-control>no-cache</cache-control>
  <pragma>no-cache</pragma>
</HTTPHeaders>
```

В теге SoapAction указывается, какое действие необходимо выполнить на сервере.

2) SOAP Response: Ответ сервера содержит значения возвращаемых параметров.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body ...>
  <SOAPSDK4:AddResponse ...>
    <Result>3</Result>
  </SOAPSDK4:AddResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В данном случае сервер ответил на вызов метода Add с параметрами 1 и 2 значением 3.

3) SOAP Fault: в случае возникновения ошибок до или во время обработки запроса сервер возвращает информацию об ошибке. Ошибки могут быть двух типов:

- клиентские ошибки: неправильный запрос к серверу; запрос содержит неверные значения параметров; неправильно сформирован и т.д.;

- серверные ошибки. Ошибки могут быть связаны как с функционированием самого сервера (закончились свободные ресурсы), так и с обработкой запроса (компонент вернул ошибку).

Серверный компонент может использовать несколько стратегий для того, чтобы сообщить клиенту об ошибках, возникающих во время обработки запроса. Если метод возвращает код ошибки, SOAPServer предпримет попытку получить от компонента расширенную информацию об ошибке:

- ISoapError – если компонент поддерживает этот интерфейс, SOAPServer установит значение элементов отклика SOAPFault в соответствии с информацией, полученной вызовом методов интерфейса ISoapError.

- IErrorInfo – если компонент поддерживает стандартный интерфейс для предоставления информации об ошибке в COM, SOAPServer установит

некоторые значения отклика SOAPFault в соответствии с информацией в IErrorInfo.
Пример серверной ошибки:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body ...>
  <SOAP-ENV:Fault>
    <faultcode>SOAP-ENV:Server</faultcode>
    <faultstring>Can add no more numbers</faultstring>
    <faultactor>http://aida:8080/Sample1/Sample1.ASP</faultactor>
    <detail>
    <mserror:errorInfo ...>
      <mserror:returnCode>-2147467259 : Unspecified error
      </mserror:returnCode>
      <mserror:serverErrorInfo>
      <mserror:description>Can add no more numbers</mserror:description>
      <mserror:source>Sample1.Adder.1</mserror:source>
      </mserror:serverErrorInfo>
      <mserror:callStack>
      <mserror:callElement>
        <mserror:component>WSDLOperation</mserror:component>
        <mserror:description>Executing method Add failed
        </mserror:description>
        <mserror:returnCode>-2147352567 : Exception occurred.
        </mserror:returnCode>
      </mserror:callElement>
      ...
      </mserror:callStack>
    </mserror:errorInfo>
    </detail>
  </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В теге faultcode указывается источник ошибки – клиент или сервер. Faultstring содержит строковое описание ошибки. Faultfactor – указывает URL, к которому обращался клиент. Тег detail содержит дополнительную информацию об ошибке (например, call stack). Спецификация SOAP не накладывает на нее каких-бы то ни было ограничений. SOAP Toolkit помещает в тег detail элемент <mserror:errorInfo ...>, содержащий следующие дочерние элементы:

- returnCode содержит код ошибки, которую вернул компонент;

- serverErrorInfo появится в случае, если серверный компонент поддерживает IErrorInfo, и будет содержать значения всех составляющих IErrorInfo – описание ошибки, источник, helpfile и helpcontext;

- callStack – коллекция элементов callElement – трассировка вызова по внутренним компонентам SOAPServer. Каждый callElement содержит имя компонента, код возврата, описание ошибки. С помощью этой информации можно увидеть, на каком из входящих в SOAPServer компонентов обработка вызова завершилась с ошибкой, и как выглядела цепочка вызываемых компонентов.

Любой вызов SOAP включает в себя следующие этапы:

- разбор WSDL-файла и определение операции и параметров, которые необходимо передать на сервер;
- преобразование значений параметров в текстовую форму;
- формирование XML-запроса;
- передача запроса на сервер;
- получение отклика сервера;
- преобразование выходных параметров, разбор сообщения об ошибке.

За каждый из этапов отвечает специальный компонент SOAP Toolkit 3.0. За разбор WSDL-файла и предоставление информации о количестве и типах параметров отвечает компонент WSDLReader30, результатом работы которого являются WSDLOperation, WSDLPort, WSDLService. С помощью этих компонентов можно получить всю информацию, содержащуюся в WSDL-файле.

Преобразование параметров в текстовую форму осуществляется с помощью так называемых mapper-ов. В состав SOAP Toolkit входят mapper-ы для всех oleautomation-типов и некоторых других. Кроме того, есть Generic Custom Type Mapper, который подходит для большинства составных типов данных (структур). Для тех случаев, когда стандартные mapper-ы не подходят, можно использовать свои собственные.

Передача запроса на сервер и получение отклика осуществляется с помощью так называемого коннектора.

Для разработки распределенных приложений, использующих протокол SOAP, существует большое количество наборов компонентов, облегчающих процесс разработки и берущих на себя обработку различных этапов вызова и получения отклика от сервера, например, SOAP Toolkit. Microsoft SOAP Toolkit в значительной степени облегчает процесс создания серверных компонентов SOAP, обеспечивающих прием и обработку запросов, и предоставляет стандартный Proxy, реализующий динамический IDispatch - интерфейс, что позволяет легко преобразовывать вызовы COM в SOAP-запросы.

SOAP Toolkit включает в себя следующие части:

- SOAP Listener – серверные компоненты, предназначенные для обработки входящих запросов, представлены в 2-х разновидностях – ASP-страница и ISAPI-расширение;

- Proxy – клиентский компонент, позволяющий преобразовать вызов метода, осуществляемый через IDispatch-интерфейс, в SOAP-вызов;
- утилита для трассировки SOAP-вызовов, позволяет перехватывать вызовы к серверу и анализировать запросы и ответы сервера;
- WSDL-генератор. По библиотеке типов компонента генерирует файлы, необходимые для работы SOAP Listener'а и Proxy. В этих файлах находятся описания методов, параметров, типов параметров и т.д.;
- набор низкоуровневых компонентов, предназначенных для создания SOAP-запросов, преобразования различных типов данных в текстовый формат, приема и передачи пакетов по протоколу HTTP и преобразования ответа сервера;
- документация и примеры.

3.1.4. Язык описания Web-служб WSDL

WEB Service Description Language (WSDL) основан на XML и предназначен для описания тех сервисов, которые поддерживает сервер. Для каждого сервиса в WSDL-файле перечисляется набор операций, поддерживаемых данным сервисом, а также определяется формат сообщения, которого должен придерживаться клиент, чтобы сформировать правильный запрос.

WSDL-файл задает контракт между сервером и клиентом, который клиент обязан выполнять, чтобы быть обслуженным. WSDL-файла включает следующие основные разделы:

1) Схема – <schema> – описывает сложные типы данных, используемые в параметрах методов.

```
<types>
  <schema ...>
  ...
  <complexType  name ='Recordset'>
    ...
  </complexType>
  </schema>
</types>
```

2) Описание SOAP-сообщений – <message>

```
<message name='TView.GetModules'>
  <part name='processID' type='xsd:int'/>
</message>

<message name='TView.GetModulesResponse'>
  <part name='Result' type='typens:Recordset'/>
</message>
```

3) Описание SOAP-портов и операций, которые поддерживаются этими портами:

```
<portType name='TViewSoapPort'>
 ...
 <operation name='GetModules' parameterOrder='processID'>
   <input message='wsdlns:TView.GetModules'/>
   <output message='wsdlns:TView.GetModulesResponse'/>
 </operation>
</portType>
```

4) Описание формата операций:

```
<binding name='TViewSoapBinding' type='wsdlns:TViewSoapPort' >
 ...
 <operation name='ShutdownMachine'>
   <soap:operation .../>
     <input>
       <soap:body ...parts='nFlags'/>
     </input>
     <output>
       <soap:body .../>
     </output>
   </sopa:operation>
 </operation>
</binding>
```

5) Описание сервисов и портов, входящих в эти сервисы:

```
<service name='TView' >
 <port name='TViewSoapPort' binding='wsdlns:TViewSoapBinding' >
   <soap:address location='http://evm/TView/TView.ASP'/>
 </port>
</service>
```

В WSDL-файле могут быть описаны один или несколько сервисов, для каждого из которых задан свой URL-адрес. WSDL-файл может находиться локально по отношению к клиенту или загружаться по протоколу HTTP (как в данном случае). Если WSDL-файл загружается по HTTP, и Web-сервер требует аутентификации, имя пользователя и пароль необходимо передать через URL, например, http://user:pwd@evm/fakult/fakult.wsdl/. Каждый сервис включает в себя один или несколько портов. Генератор WSDL-файлов создает один порт для каждого из указанных интерфейсов COM-объекта. Каждый порт включает в

себя одну или несколько операций. Если сравнивать порт с интерфейсом, то операцию можно сравнить с методом интерфейса. Каждая операция включает несколько частей. Часть (part) можно сравнить с параметром метода.

WSML-файл описывает связь операций с конкретными методами COM-объектов, которые будут обслуживать запросы. Ниже приведены основные разделы WSML-файла.
Ссылки на используемые COM объекты:

```
<service name='TView'>
  <using PROGID='TView.TView.1' cachable='0' ID='TViewObject' />
```

2) Связь сложных типов данных с мэпперами, которые будут отвечать за преобразование сложных типов в XML:

```
<types>
  <type name='Recordset' ... targetPROGID='ADODB.Recordset' iid='{00000535-0000-0010-8000-00aa006d2ea4}'/>
</types>
```

3) Описание связи между SOAP-операцией и методами COM-объекта:

```
<port name='TViewSoapPort'>
  <operation name='ShutdownMachine'>
    <execute uses='TViewObject' method='ShutdownMachine' dispID='1'>
      <parameter callIndex='1' name='nFlags' elementName='nFlags' />
    </execute>
  </operation>
</port>
```

На основе информации, содержащейся в этих файлах, компоненты SOAP Toolkit осуществляют подготовку SOAP-запросов, преобразование и передачу параметров, анализ ответа сервера.

3.1.5. Пространство имен XML

Пространства имен (XML Namespaces) в XML-рекомендации [9] - одна из самых коротких спецификаций XML, состоящая из 10 страниц, не включая приложения. Путаница, однако, касается семантики пространства имен, в отличие от синтаксиса, описанного спецификацией. Чтобы полностью понять пространства имен XML, пользователи должны иметь представление о том, что такое пространство имен, как определяются пространства имен и как они используются.

Пространство имен - это набор имен, в котором все имена уникальны. Например, имена студентов можно рассматривать как пространство имен, так же как и названия факультетов, имена идентификаторов типов C++ или имена

Internet-доменов. Любой логически связанный набор имен, в котором каждое имя должно быть уникальным, является пространством имен.

Пространства имен облегчают поиск уникальных имен. Помещение уникальности в более ограниченные рамки, как набор имен студентов, чрезвычайно все упрощает. Когда называется следующий студент, необходимо учитывать только то, чтобы повторно не использовать имя, которое использовалось для другого студента. Другие факультеты могут выбрать такое же имя для одного из своих студентов, но эти имена будут частью различных пространств имен и, следовательно, легко различимы.

Перед тем, как добавить в пространство имен новое имя, администрация пространства имен должна убедиться, что в пространстве имен такого имени еще нет. В некоторых сценариях это просто сделать, как, например, в присваивании имен дочерним элементам. В других - чрезвычайно трудно. Многие современные Internet-администрации присваивания имен являются хорошим примером. Однако, если упустить это, дублирование имен в конце концов приведет к нарушению пространства имен и сделает невозможным однозначно ссылаться на определенные имена. Когда это происходит, набор имен официально больше не считается пространством имен - по определению пространство имен должно обеспечивать уникальность своих членов.

Самим пространствам имен имена также должны присваиваться с точки зрения практического применения. Если у пространства имен есть имя, можно обращаться к его членам.

Пространства имен используются во многих языках программирования, чтобы избегать конфликта имен. Это именно то решение, которое было необходимо для завершения спецификации XML 1.0.

При определении пространства имен в таком языке программирования как C++, есть ограничения на символы, которые могут использоваться в имени. Идентификаторы пространства имен XML также должны соответствовать определенному синтаксису - синтаксису для ссылок Универсального идентификатора ресурса (Uniform Resource Identifier (URI)) в соответствии с RFC 2396. URI определяется как составная строка символов для определения абстрактного или физического ресурса. В большинстве ситуаций ссылки URI используются для определения физических ресурсов (Web-страниц, файлов для загрузки и т.д.), но в случае с пространствами имен XML ссылки URI определяют абстрактные ресурсы, а именно, пространства имен.

Согласно URI-спецификации существует две основные формы URI: Унифицированные указатели информационного ресурса (Uniform Resource Locators (URL)) и Унифицированные имена информационного ресурса (Uniform Resource Names (URN)). Любой тип URI может использоваться как идентификатор пространства имен. Ниже приведен пример двух URL, которые могут использоваться в качестве идентификаторов пространства имен:

http://www.develop.com/student
http://www.ed.gov/elementary/students

и примеры URN, которые могут использоваться как идентификаторы пространства имен:

urn:www-develop-com:student

urn:www.ed.gov:elementary.students

urn:uuid:E7F73B13-05FE-44ec-81CE-F898C4A6CDB4

Наиболее важным атрибутом пространства имен является то, что он уникален. Регистрируя имя домена в Internet-администрации присваивания имен, авторы могут гарантировать уникальность URL. Затем автор отвечает за обеспечение уникальности всех строк, используемых после имени домена. Обработчик XML интерпретирует идентификаторы пространства имен как непрозрачные строки и никогда как разрешимые ресурсы.

Использование пространства имен можно рассматривать как процесс использования одного или более элементов или атрибутов из данного пространства имен в XML-документе. Имена элементов и атрибутов на самом деле образуются из двух частей: имени пространства имен и локального имени. Такие двойные имена известны как составные имена или QName.

В XML-документе используется префикс пространства имен, чтобы определить локальные имена элементов или атрибутов. На самом деле префикс- это аббревиатура идентификатора пространства имен (URI), который обычно слишком длинный. Сначала префикс преобразовывается в идентификатор пространства имен через описание пространства имен. Синтаксис описания пространства имен следующий:

xmlns:<prefix>="<namespace identifier>"

Описание пространства имен выглядит просто как атрибут (элемента), но в рамках логической структуры документа нет официально принятых атрибутов (т.е. они не появятся в коллекции атрибутов элемента при использовании DOM).

Префикс пространства имен рассматривается в области видимости элемента описания, так же, как и наследующих от него элементов. Будучи объявленным, префикс может использоваться перед именем любого элемента или атрибута, отделенным от него двоеточием (как s:student). Это полное имя, включая префикс, является лексической формой составного имени (QName):

QName = <prefix>:<local name>

Префикс ассоциирует элемент или атрибут с идентификатором пространства имен, в настоящее время преобразованным в префикс в области видимости.

В настоящее время большинство пространств имен XML определены в формальных документах спецификации, которые описывают имена элементов, а также атрибутов вместе с их семантикой. Именно так формально определены все пространства имен W3C [10]. Как только пространство имен определено, разработчик программного обеспечения реализовывает пространство имен, как предписывает спецификация. Например, MSXML 3.0, Xalan и Saxon - это реализации спецификации XSLT 1.0, которые жестко запрограммированы на поиск элементов, принадлежащих пространству имен XSLT 1.0

(http://www.w3.org/1999/XSL/Transform). Чтобы использовать эти реализации, необходимо предоставить XML-документ, правильно использующий имена из пространства имен XSLT 1.0. Если в пространстве имен XSLT 1.0 были внесены изменения, то обеспечивающие программные средства должны быть обновлены.

Рабочая группа XML Schema [11,12] собрала новую спецификацию (XML Schema), которая определяет синтаксис на базе XML для описания элементов, атрибутов и типов в пространстве имен. Кроме того, XML Schema делает возможным обеспечить синтаксическое описание для пространства имен, как показано ниже.

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
  targetNamespace="http://www.develop.com/student"
  elementFormDefault="qualified"
>
 <element name="student">
   <complexType>
     <sequence>
       <element name="id" type="long"/>
       <element name="name" type="string"/>
       <element name="language" type="string"/>
       <element name="rating" type="double"/>
     </sequence>
   </complexType>
 </element>
</schema>
```

В этом примере описывается пространство имен http://www.develop.com/student, содержащее пять именованных элементов: student, id, name, language и rating. Кроме простого предоставления пространства имен, эта схема также предоставляет дополнительные метаданные, такие как порядок дочерних элементов элемента student, а также их типы.

XML Schemas не определяют семантику элементов и атрибутов и, следовательно, нуждаются в сопроводительной спецификации.

Следующий XML-документ показывает, как использовать элементы из описания XML Schema, показанного ранее:

```
<d:student xmlns:d="http://www.develop.com/student">
 <d:id>3235329</d:id>
<d:name>Jeff Smith</d:name>
<d:language>C#</d:language>
<d:rating>9.5</d:rating>
</d:student>
```

Следует заметить, что независимо от того, как определены пространства имен, синтаксис обращения к ним одинаковый.

В случае, если документы используют элементы или атрибуты из нескольких пространств имен, общепринятым считается иметь описания нескольких пространств имен в данном элементе:

```
<d:student xmlns:d="http://www.develop.com/student"
 xmlns:i="urn:schemas-develop-com:identifiers"
 xmlns:p="urn:schemas-develop-com:programming-languages"
>
<i:id>3235329</i:id>
 <name>Jeff Smith</name>
<p:language>C#</p:language>
<d:rating>9.5</d:rating>
</d:student>
```

Здесь элементы student и rating из одного пространства имен, id и language из разных, а name не принадлежит ни одному пространству имен.

Префиксы пространства имен также могут быть изменены путем переобъявления префикса во вложенном контексте:

```
<d:student xmlns:d="http://www.develop.com/student">
<d:id>3235329</d:id>
 <d:name xmlns:d="urn:names-r-us">Jeff Smith</d:name>
<d:language>C#</d:language>
<d:rating>35</d:rating>
</d:student>
```

В этом примере все элементы из одного пространства имен, кроме элемента *name,* который относится к пространству имен *urn:names-r-us*.

Существует еще один тип описания пространства имен, который может использоваться для ассоциирования идентификаторов пространств имен с именами элементов. Они известен как описание стандартного пространства имен, которое использует следующий синтаксис:

```
xmlns="<namespace identifier>"
```

Следует заметить, что здесь нет префикса. Когда используется описание стандартного пространства имен, все неопределенные имена элементов автоматически ассоциируются с установленным идентификатором пространства имен. Однако описания стандартных пространств имен абсолютно никак не влияют на атрибуты. Единственный способ ассоциировать атрибут с идентификатором пространства имен - через префикс.

Рассмотрим следующий пример:

```
<d:student  xmlns:d="http://www.develop.com/student"
    xmlns="urn:foo" id="3235329"
>
<name>Jeff Smith</name>
 <language xmlns="">C#</language>
 <rating>35</rating>
</d:student>
```

Здесь "student" из пространства имен http://www.develop.com/student, а "name" и "rating" из стандартного пространства имен urn:foo. Атрибут id не принадлежит пространству имен, поскольку атрибуты автоматически не ассоциируются с идентификатором стандартного пространства имен.

Этот пример иллюстрирует, что можно отменить объявление стандартного пространства имен, просто установив его идентификатор опять в пустую строку, как показано в элементе language (помните, что вы не можете делать этого с описаниями префиксов). В результате элемент language также не принадлежит пространству имен.

Сегодня широко распространены API, SAX и DOM, реализующие эту абстрактную модель данных. SAX создает элементы с помощью вызовов метода startElement/endElement в ContentHandler:

```
public interface contentHandler
{
...
void startElement(String namespaceURI, String localName,
  String qName, Attributes atts) throws SAXException;
Void endElement(String namespaceURI, String localName,
 String qName) throws SAXException;
...
}
```

Обратите внимание, что элементы идентифицируются комбинацией идентификатора пространства имен и локального имени. Атрибуты также идентифицируются через ряд знающих пространство имен методов интерфейса Attributes. Это говорит о том, что при использовании SAX программно различить различные типы элементов student нетрудно.

```
...
void startElement(String namespaceURI, String localName,
  String qName, Attributes atts)
{
   if ( namespaceURI.equals("urn:dm:student") &&
       localName.equals("student") )
     {
      // обрабатываем элемент student из пространства имен urn:dm:student
   }
   else if ( namespaceURI.equals("urn:www.ed.gov:student")
       && localName.equals("student") )
     {
      // обрабатываем элемент student из пространства имен
urn:www.ed.gov:student
   }
}
…
```

Поскольку имя пространства имен (идентификатор пространства имен + локальное имя) автоматически разбирается синтаксическим анализатором SAX, не имеет значения, какой префикс (если он имеется) использовался в конкретном элементе или атрибуте исходного документа, это, по большей мере, деталь сериализации. Однако это не значит, что префиксы после синтаксического анализа могут быть отброшены.

Существует другая XML-спецификация -XPath, которая определяет, как идентифицировать узлы в абстрактной структуре документа. Выражения XPath делают возможным идентифицировать элементы и атрибуты по определенным пространством имен именам. Поскольку проверки имен XPath - это простые строковые выражения, единственным способом ассоциировать проверку имени XPath с идентификатором пространства имен является использование префикса пространства имен.

Если узел не содержит префикс, он как бы запрашивает данное имя, которое принадлежит "никакому пространству имен". Например, возьмем следующее выражение XPath:

/student/name

Это выражение идентифицирует все элементы name, которые не принадлежат ни одному пространству имен и являются потомками корневого элемента student, который не относится ни к одному пространству имен. Чтобы идентифицировать элементы student и name, принадлежащие пространству имен urn:dm:student, сначала необходимо ассоциировать префикс пространства имен с urn:dm:student. Затем этот префикс может использоваться в выражении XPath.

Принимая, что "dm" был ассоциирован с urn:dm:student в контексте XPath, следующее выражение будет идентифицировать элементы name, принадлежащие пространству имен urn:dm:store и являющиеся потомками корневого элемента student, также принадлежащего пространству имен urn:dm:store:

/dm:student/dm:name

Если запрашиваемый документ выглядит как приведенный ниже код, тогда предыдущее выражение будет идентифицировать все элементы name дерева, являющиеся потомками student, независимо от их префикса, потому что все они из одного рассматриваемого пространства имен.

```
<s:student xmlns:s="urn:dm:student">
    <s:name/>
   <n:name xmlns:n="urn:dm:student"/>
   <s:name/>
   </s:student>
```

Префиксы преобразовываются в контекст XPath в зависимости от реализации.

Контрольные вопросы

1. Перечислите основные функции протокола SOAP.
2. Назовите основные достоинства протокола SOAP.
3. Каким образом описывается связь операций с конкретными методами COM-объектов?
4. В каком файле содержатся описания сервисов, поддерживаемых сервером?
5. Перечислите элементы вызова SOAP.
6. Назовите основные этапы вызова SOAP.
7. Какие основные разделы включает WSDL-файл?
8. С помощью каких компонентов можно получить всю информацию, содержащуюся в WSDL-файле?
9. Назначение SOAP Toolkit.
10. Как осуществляется связь сложных типов данных

3.2. Аспектно-ориентированное и функциональное программирование

3.2.1. Основные понятия и определения

Аспектно-ориентированное программирование (АОП) - парадигма программирования, основанная на идее разделения функциональности, для улучшения разбиения программы на модули. С увеличением сложности программы следует внимательнее подходить к её архитектуре в целом, и выделению общей функциональности в частности. Практика показывает, что для выделения некоторой общей функциональности существующих парадигм программирования недостаточно. Такую функциональность называют «сквозной» или «разбросанной», в виду того, что её реализация действительно разбросана по разным частям приложения. Примерами сквозной функциональности могут служить:

- логирование;
- обработка транзакций;
- обработка ошибок;
- авторизация и проверка прав;
- кэширование.

Основной задачей аспектно-ориентированного программирования (АОП) является модуляризация сквозной функциональности, выделение её в аспекты. Для этого языки, поддерживающие концепцию АОП, реализуют следующие средства для выделения сквозной функциональности:

- аспект (aspect) - модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом. Так же аспект может использоваться для внедрения функциональности;

- совет (advice) - дополнительная логика - код, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения;

- точка соединения (join point) - точка в выполняемой программе (вызов метода, создание объекта, обращение к переменной), где следует применить совет;

- срез (pointcut) - набор точек соединения. Срез определяет, подходит ли данная точка соединения к заданному совету;

- внедрение (introduction) - изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код;

- цель (target) – объект, к которому будут применяться советы;

- переплетение (weaving) – связывание объектов с соответствующими аспектами (возможно на этапе компиляции, загрузки или выполнения программы).

AspectJ является аспектно-ориентированным расширением framework'ом для языка Java. В [13] рассмотрена реализация аспекта логирования с его помощью:

```
@Aspect
public class WebServiceLogger {
  private final static Logger LOG =
    Logger.getLogger(WebServiceLogger.class);

  @Pointcut("execution(* example.WebService.*(..))")
  public void webServiceMethod() { }

  @Pointcut("@annotation(example.Loggable)")
  public void loggableMethod() { }

  @Around("webServiceMethod() && loggableMethod()")
  public Object logWebServiceCall(ProceedingJoinPoint thisJoinPoint) {
    String methodName = thisJoinPoint.getSignature().getName();
    Object[] methodArgs = thisJoinPoint.getArgs();

    LOG.debug("Call method " + methodName + " with args " + methodArgs);

    Object result = thisJoinPoint.proceed();

    LOG.debug("Method " + methodName + " returns " + result);
      return result;
  }
}
```

Сначала создаётся аспект логирования методов сервисов - класс WebServiceLogger, помеченный аннотацией @Aspect. Далее определяются два среза точек соединения: webServiceMethod (вызов метода, принадлежащего классу WebService) и loggableMethod (вызов метода, помеченного аннотацией @Loggable). В завершении объявляется совет (метод logWebServiceCall), который выполняется вместо (аннотация @Around) точек соединения, удовлетворяющих срезу («webServiceMethod() && loggableMethod()»).

В коде совета происходит получение информации о текущем методе (точке соединения), логирование начала выполнения метода, непосредственный вызов запрошенного метода, логирование и возвращение результата работы.

AspectJ обладает довольно большим объёмом поддерживаемых срезов точек соединения. Ниже приведены основные из них:

- execution(static * com.xyz..*.*(..)) - выполнение кода любого статического метода в пакете com.xyz;

- call(void MyInterface.*(..)) - вызов любого метода, возвращающего void, интерфейса MyInterface;

- initialization(MyClass || MyOtherClass) - инициализация класса MyClass или MyOtherClass;

- staticinitialization(MyClass+ && !MyClass) - статическая инициализация класса, имя которого начинается на MyClass, но не сам MyClass;

- handler(ArrayOutOfBoundsException) - выполнение обработчика исключения ArrayOutOfBoundsException;

- get/set(static int MyClass.x) - чтение / запись свойства *x* класса MyClass;

- this/target(MyClass) - выполнение точки соединения, соответствующей объекту типа MyClass;

- args(Integer) - выполнение точки соединения, в которой доступен аргумент типа Integer;

- if(thisJoinPoint.getKind().equals(«call»)) - совпадает со всеми точками соединения, в которых заданное выражение истинно;

- within/withincode(MyClass) - совпадает со всеми точками соединения, встречающимися в коде заданного класса;

- cflow/cflowbelow(call(void MyClass.test())) - совпадает со всеми точками соединения, встречающимися в потоке выполнения заданного среза;

- @annotation(MyAnnotation) – выполнение точки соединения, цель которой помечена аннотацией @MyAnnotation.

Количество советов намного меньше, но они полностью покрывают всё необходимое множество ситуаций:

- before - запуск совета до выполнения точки соединения;

- after returning - запуск совета после нормального выполнения точки соединения;

- after throwing - запуск совета после выброса исключения в процессе выполнения точки соединения;

- after - запуск совета после любого варианта выполнения точки соединения,

- around - запуск совета вместо выполнения точки соединения (выполнение точки соединения может быть вызвано внутри совета).

3.2.2. Разделение функциональности

Основная цель АОП - вынос «общей» (сквозной) функциональности «за скобки» (модуляризация сквозной функциональности). Следует заметить, что для Java AOP доступен через проект AspectJ, для .NET- через PostSharp. Наиболее простая и проверенная реализация AOP - Spring AOP.

PostSharp является аспектно-ориентированным framework'ом для платформы .NET. Существуют и другие реализации АОП для .NET, однако, судя по сравнениям с сайта PostSharp, лидирующую позицию занимает именно он. Рассмотрим описание аспекта обработки исключений. Первым делом необходимо создать класс, расширяющий соответствующий аспект:

```
public class ExceptionDialogAttribute : OnExceptionAspect
{
  public override void OnException(MethodExecutionEventArgs eventArgs)
  {
  string message = eventArgs.Exception.Message;
 Window window =
Window.GetWindow((DependencyObject)eventArgs.Instance);
  MessageBox.Show(window, message, "Exception");
 eventArgs.FlowBehavior = FlowBehavior.Continue;
}
}
```

Следует заметить, что аспекты в терминологии PostSharp - это аспект и совет в терминологии АОП.

Для того, что бы указать срез точек пересечения для данного аспекта, необходимо в файл настроек сборки (AssemblyInfo.cs) добавить следующую строку:

```
[assembly: ExceptionDialog ( AttributeTargetTypes="Example.WorkflowService.*",
    AttributeTargetMemberAttributes = AttributeTargetElements.Public )]   ,
```

или же явно пометить интересующие методы атрибутом ExceptionDialog:

```
[ExceptionDialog] public BookDTO GetBook(Integer bookId).
```

Классическая ситуация введения аспекта - реализация в отдельном классе-источнике или получателе данных алгоритма отслеживания взаимодействия других пар объектов и синхронизации с ними исключительно затруднена. Цель - обеспечить синхронизацию обновления данных; желаемое поведение аспекта - реализовать "конвейерную передачу" обновлений по уровням иерархии и между поддеревьями БД.

Сначала необходимо определить "точку пересечения". Она одна - аспект перехватывает управление при попытке выполнения каким-либо из объектов-

источников записи одного или нескольких новых значений; блокирует обработку новых значений в объектах-получателях, дает завершиться перехваченным методам SetValue(…), после чего ожидает в течение заданного интервала времени выполнения аналогичных действий другими объектами-источниками (рис.3.4). По истечении времени ожидания происходит разблокирование всех объектов-получателей, в которые были записаны новые значения.

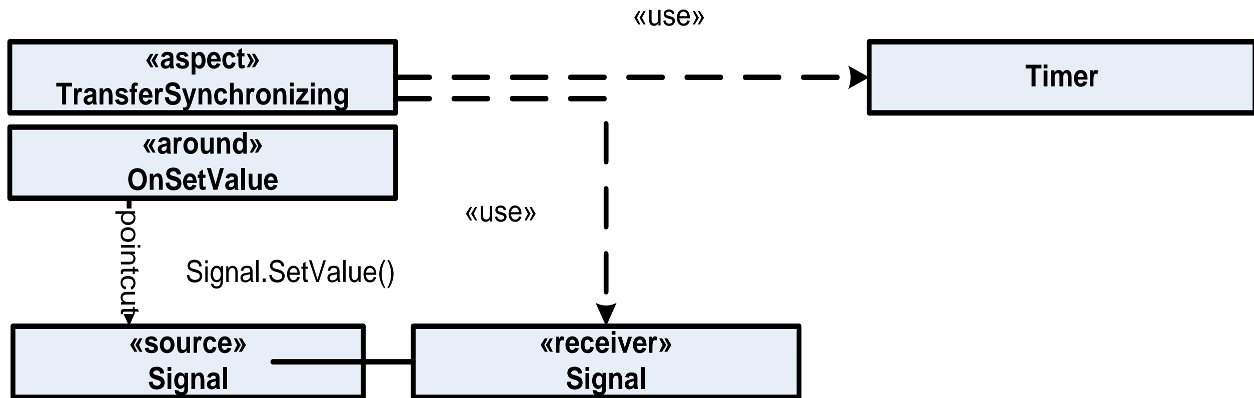

Рис.3.4. Введение аспекта

3.2.3. F# - типовые приемы функционального программирования

Реализация web-приложений на F#

F# - это язык для написания практических приложений, уменьшающий объем стереотипного кода, упрощающий чтение и сопровождение моделей и моделей представлений.

Хотя F# совсем недавно появился в семействе продуктов Visual Studio, он уже помог многим .NET-разработчикам раскрыть всю мощь функционального программирования. Язык F# завоевал прочную репутацию в таких областях, как, например, параллельное и асинхронное программирование, обработка данных и финансовое моделирование. Однако это не означает, что F# является узкоспециализированным языком; он отлично подходит и для решения рутинных задач.

Концепции, которые делают F# языком для упрощения сложнейших алгоритмов, можно применять и для сокращения стереотипного кода в моделях представлений (view models).

Сокращение стереотипного кода

На примере моделей представлений можно наглядно убедиться, насколько F# сокращает стереотипный код, т.е. улучшает «отношение сигнал/шум» по сравнению с языком C#.

Ниже приведен код на F# для простой модели видеоклипа с полями, в которые записываются название, жанр и необязательный рейтинг:

```
type Movie = {
  Name: string
```

```
Genre: string
Rating: int option
}
```

Типы option в F# более мощные и выразительные аналоги C#-типов, допускающих значения null (nullables). Тип option позволяет естественным образом выразить тот факт, что в модели видеоклипа рейтинг может как присутствовать, так и отсутствовать, что может вызвать проблемы при связывании с данными. Например, попытка получить значение типа option, установленного в None (эквивалент null для nullable-типов), приведет к исключению.

Модель представления с примером дополнительной логики отображения Модель представления для видеоролика с логикой отображения рейтингов представлена ниже:

```
type MovieViewModel(movie:Movie) =
member this.Name = movie.Name

 member this.Genre = movie.Genre

 member this.Rating =
   match movie.Rating with
  Some x -> x
  None -> 0

member this.HasRating = movie.Rating.IsSome
```

Если рейтинг есть, его значение передается представлению; в ином случае рейтингу присваивается значение по умолчанию, равное 0. Эта простая логика предотвращает исключение, если Rating установлен в None. Модель представления также содержит второе свойство для обработки связывания с визуальными элементами. Оно просто возвращает true, если Rating равен Some, и false, если он - None. Логика в этой модели представления проста, следует обратить внимание, как четко F# выражает определение модели представления. В C# модели представлений зачастую переполнены стереотипным кодом, который сильно мешает пониманию логики представления.

Ниже показана та же модель представления, написанная на C#:

```
class MovieViewModelCSharp
{
 Movie movie;

 public MovieViewModelCSharp(Movie movie)
 {
  this.movie = movie;
 }
```

```
 public string Name
 {
  get { return movie.Name; }
 }

 public string Genre
 {
  get { return movie.Genre; }
 }

 public int Rating
 {
  get
  {
   if(OptionModule.IsSome(movie.Rating))
   {
    return movie.Rating.Value;
   }
   else
   {
    return 0;
   }
  }
 }

 public bool HasRating
 {
  get
  {
   return OptionModule.IsSome(movie.Rating);
  }
 }
}
```

Увеличение стереотипного кода очевидно. На C# нужно написать примерно в 4 раза больше строк кода, чем на F#. Большая часть этого увеличения строк кода связана с фигурными скобками, но даже значимые элементы вроде аннотаций типов, выражений return и модификаторов уровня доступа мешают восприятию логики, которую должна инкапсулировать модель представления. F# уменьшает этот шум и переводит в фокус код самой логики. Иногда к функциональному программированию предъявляют претензии за чрезмерную краткость и сложность в чтении кода, но в этом примере совершенно очевидно, что F# не жертвует ясностью во имя краткости.

Предыдущий пример иллюстрирует одно преимущество F# в написании моделей представлений, но F# также позволяет устранить другую распространенную проблему с этими моделями в MVVM-приложениях. Допустим, предметная область сменилась и нужно обновить свою модель. Genre теперь является списком тегов, а не одной строкой. В коде модели достаточно заменить строку:

Genre: string

на строку:

Genre: string list.

Поскольку представление взаимодействовало с этим свойством через модель представления, возвращаемый тип в модели тоже надо изменить. В C# это требует внесения изменений вручную, а в F# осуществляется автоматически благодаря логическому распознаванию типов (type inference). Genre не требует никакой логики отображения, поэтому модель представления просто передает это поле представлению безо всякой модификации. Иначе говоря, нет необходимости заботиться о возвращаемом типе свойства в модели представления (ViewModel), если только оно соответствует возвращаемому типу свойства в модели (Model). Следует подчеркнуть, что F# по-прежнему является статически типизируемым языком, поэтому любое неправильное применение поля в ViewModel или Model (с сопутствующим XAML) даст ошибку при компиляции.

Структуризация MVVM-приложений в F#

Рассмотрим пример интеграции F#-кода в приложения Silverlight и WPF. В C# есть много способов структуризации MVVM-приложений; то же самое относится и к F#. Можно использовать два способа: использование только F# и метод полиглота, при котором для представлений применяется C#, а для ViewModel и Model — F#. Второй способ считается предпочтительнее по нескольким причинам. Во-первых, это подход, рекомендованный группой разработчиков языка F#. Во-вторых, инструментальная поддержка WPF и Silverlight в C# гораздо обширнее, чем в F#. Наконец, этот подход позволяет включать F#-код в существующие приложения и предоставляет способ с малым риском опробовать применение F# в MVVM-приложениях.

Подход «только F#» интересен, потому что дает возможность писать приложение на одном языке, но накладывает ряд ограничений. Подход полиглота не дает пользователю использовать F# в коде представления, но продуманные MVVM-приложения должны содержать в представлениях минимум логики. Более того, C# - язык, который подходит для написания (при необходимости) логики представления, так как по своей природе она императивна и обременена множеством побочных эффектов.

Подход полиглота

При таком подходе создать MVVM-приложение очень легко. Сначала создайте новый проект WPF на C#, используя шаблон проекта WPF Application. Этот проект отвечает за любые представления и отделенный код (codebehind),

который нужен в приложении. Затем добавьте в решение новый проект библиотеки на F#, который будет содержать модели представлений, модели и любой код, не относящийся к представлениям. Наконец, добавьте ссылку на проект F#-библиотеки из проекта C# WPF. Вот и все, что нужно для подготовки к работе с использованием подхода полиглота.

F# рассчитан на взаимодействие с любым .NET-языком, а это означает, что можно подключить модели представлений на F# к представлениям на C# любым методом, традиционным для моделей представлений на C#.

Рассмотрим пример, где для простоты используется отделенный код. Сначала создается простая модель представления, переименуем Module1.fs в проекте на F# в MainWindowViewModel.fs. Заполняем модель представления кодом. Далее подключаем модель представления на F# к представлению на C# с помощью кода. Пример модели представления на F# приведен ниже:

```
namespace Core.ViewModels
type MainWindowViewModel() =
 member this.Text = "hello world!"
```

Добавьте текстовое поле к MainWindow.xaml и установите Binding в Text. И вновь все ведет себя так, будто приложение написано исключительно на C#. Проверьте привязку, запустив приложение: вы должны увидеть стандартное приветствие:

«hello world!»

Контрольные вопросы

Что понимается под WOA?

1. Поясните структуру web-приложений.
2. Перечислите службы webю.
3. Что понимается под протоколом SOAP?
4. Поясните особенности язык описания Web-служб?
5. Что такое пространством имен XML?

7. Поясните отличие аспектно-ориентированного и функционального программирования.

8. Перечислите приемы в программировании на F#.

ИТОГОВЫЙ ТЕСТ

1) SOAP Toolkit включает в себя следующие части:
А) серверные компоненты, предназначенные для обработки входящих запросов
Б) Утилита для трассировки SOAP-вызовов
В) WSDL-генератор
Г) Набор низкоуровневых компонентов, предназначенных для создания SOAP-запросов
Д) Proxy – клиентский компонент
Е)ASP-страница и ISAPI-расширение
Ж) WSDL- и WSML-файлы

1) А Б В Г Д Ж
2) Б В Г Е
3) А Б В Г Д Е
4) А Б Д Е Ж
5) А Б В Г Е Ж
6) В Г Е Ж

2) Что является универсальными «строительными» блоками .NET?
1. PERSONALIZATION, NOTIFICATION AND MESSAGING, XML STORE, NATYRALE INTERFACE
2. XML STORE, SMARTTAGS, IDENTIFY,
3. IDENTIFY, PERSONALIZATION, NOTIFICATION AND MESSAGING, XML STORE, UNIVERSAL CANVAS
4. UNIVERSAL CANVAS, NOTIFICATION AND MESSAGING, XML STORE,
5. IDENTIFY, PERSONALIZATION, NOTIFICATION AND MESSAGING, XML STORE
6. MSN.NET, PERSONALIZATION, , XML STORE, NATYRALE INTERFACE

3) Укажите номер правильного ответа для нижеприведенных утверждений.
1) C# обладает всеми основными свойствами языка для современных серверных приложений
2) C#   объектно-ориентированный язык ,без множественного наследования.
3) C# объектно-ориентированный язык, с развитыми средствами безопасности.
4) C# объектно-ориентированный язык, со ссылками, но без указателей
5) C# объектно-ориентированный язык, с объектной обработкой исключений, с множественным наследованием
   1. утверждения 1,3,5 - верные,   2,4 - ложные

2. утверждения 1,2, 4 - верные, 3,5 - ложные
3. утверждения 1,2,3 - верные, 4,5 - ложные
4. утверждения 3,4,5- верные, 1,2 - ложные
5. утверждения 2,3,4 - верные, 1,5 - ложные
6. утверждения 1-4 - верные, 5 - ложное

4) Основные разделы WSDL-файла:

1. <schema> <message> <portType><service name='TView'> <types> <binding> <service>

2. <schema> <message> <service name='TView'> <types> <port name='TViewSoapPort'>

3. <service name='TView'> <types> <portType><binding>

4. <schema> <message> <portType> <binding> <service>

5. <portType> <service name='TView'> <types> <port name='TViewSoapPort'> <message>

5) Укажите правильную последовательность этапов получения данных в ASP.NET:

А) открыть созданное соединение
Б) заполнить компоненту DataSet требуемыми данными
В) установить соединение с БД
Г) установить компоненту DataView для отображения данных
Е) Используя привязку данных, связать серверный элемент управления с компонентой Data View
Ж) Импортирование пространства имен, содержащие компоненты работы с данными

1. А Б В Г Д Ж
2. В Б А Г Е
3. А Б В Г Д Е
4. А В Б Д Е Ж
5. Ж В А Б Г Е
6. А В Г Е Ж

6) Приведенный ниже пример содержит:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Body ...>
<SOAPSDK4:Add ...>
<x>1</x>
<y>2</y>
```

```
</SOAPSDK4:Add>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1. Вызов метода Add с параметрами 1 2 по протоколу SOAP
2. Ответ сервера на вызов метода Add значениями 1 2 по протоколу SOAP
3. Указания, какие действия необходимо выполнить на сервере, используя методы со значением 1,2 SOAP
4. Описание сложных типов данных, используемых в параметрах методов по протоколу SOAP

7) Укажите номер неверного высказывания:

1. assembly .NET – набор файлов, модулей и дополнительной информации, обеспечивающих простую установку приложений
2. манифест assembly позволяет скрыть от потребителя детали реализации, не требует привлечения таких средств как реестр.
3. каждый assembly   имеет уникальное имя, состоящее из префикса, простого текстового имени, номера версии и информации о локализации.
4. манифест assembly содержит все файлы и модули, а также описание всех интерфейсов с внешним миром.
5. некоторые assembly могут иметь только простое текстовое имя, при использовании их как части другого приложения

8) Модель облачных вычислений состоит из:

1.SaaS
2. IaaS
3. Слой бизнес-логики
4. Нет правильного ответа.

9) В нижеприведенном коде укажите номер строки, содержащей аннотацию:

1 @Aspect
2 public class WebServiceLogger {
3 private final static Logger LOG =
4 Logger.getLogger(WebServiceLogger.class);

1. 1,2
2. 1,3
3. 1
4. 1,4
5. совокупность всех строк

10) В нижеприведенном коде укажите номер строки, где создается аспект логирования методов сервисов:

```
1 @Aspect
2 public class WebServiceLogger {
3 private final static Logger LOG =
4  Logger.getLogger(WebServiceLogger.class);
```

1. 1,2
2. 1,3
3. 1
4. 1,4
5. совокупность всех строк

11) Укажите синтаксис описания пространства имен, который может использоваться для ассоциирования идентификаторов пространств имен с именами элементов:

1. *QName = <prefix>:<local name>*
2. *xmlns="<namespace identifier>"*
3. *xmlns=="<namespace identifier>"*
4. *QName = "<namespace identifier>"local name>*

ЛИТЕРАТУРА

1. Федоров А., Елманова Н. Архитектура распределенных вычислений // Компьютерпресс. - 2001. - № 1.
2. Информационно-юридический центр. http://shans-i.narod.ru/Disk_PC/IndexPC.htm.
3. Е. Марков Архитектура распределенных приложений. http://www.pcweek.ru/infrastructure/article/detail.php?ID=66147, (411)45, 2003.
4. Биберштейн Н., Боуз С., Джонс К., Фиаммант М., Ша Р. Компас в мире сервис-ориентированной архитектуры (SOA): ценность для бизнеса, планирования и план развития предприятия /пер. с англ.- М.: КУДИЦ-ПРЕСС, 2007. - 256с.
5. Ali Arsanjani. Советы по программированию Web-сервисов: Сервис-ориентированное моделирование и архитектура - материалы онлайн-конференции developerWorks/2004- http://www.ibm.com/developerworks/ru/library/ws-soa-design1/
6. Private Cloud Principles, Concepts, and Patterns. http://social.technet.microsoft.com/wiki/contents/articles/4346.private-cloud-principles-concepts-and-patterns.aspx.
7. Сайт ОАО Телкомнет. - http://www.telcomnet.ru/setevye-prilozheniya.html
8. Приложения сетевых технологий: сетевой журнал. http://alice.pnzgu.ru/~dvn/complex/applic.htm
9. Recommendations of the National Institute of Standards and Technology. http://csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf.
10. Спецификация XSLT 1.0 - http://www.w3.org/TR/xslt.
11. Сайт компании W3- http://www.w3.org/XML/Schema.
12. Сайт компании W3- http://www.w3.org/TR/REC-xml-names/
13.Стив Шварц, Клеменс Вастерс. Будущее распределенных вычислений: взгляд Microsoft и ее партнеров. //Компьютерпресс.- 2003. - № 6.