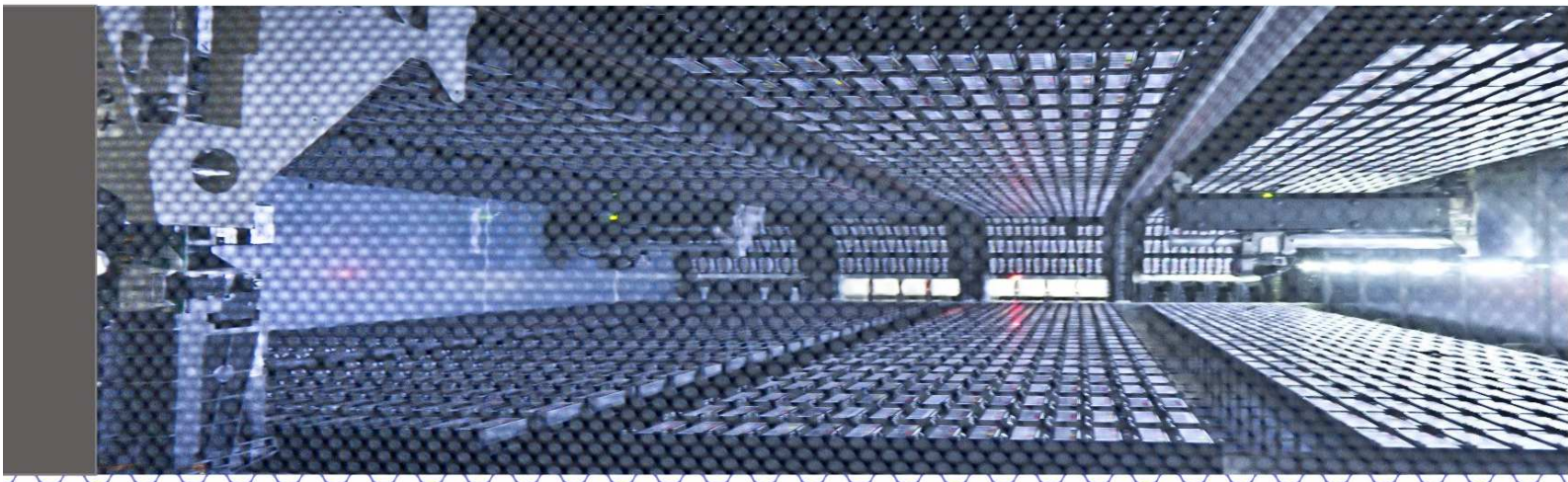
Министерство образования и науки Российской Федерации
Южно-Уральский государственный университет
Кафедра системного программирования

Г.И. Радченко

Распределенные вычислительные системы

Учебное пособие



Министерство образования и науки Российской Федерации
Южно-Уральский государственный университет
Кафедра системного программирования

004.75

Г.И. Радченко

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Учебное пособие

Челябинск
2012

УДК 004.75

Одобрено
учебно-методической комиссией
факультета вычислительной математики и информатики

Рецензент:
доктор физ.-мат. наук, профессор Л.Б. Соколинский

Радченко, Г.И.

Распределенные вычислительные системы / Г.И. Радченко. –Челябинск:: Фотохудожник, 2012. – 184 с.

ISBN 978-5-89879-198-8

В учебном пособии рассмотрены основные технологии организации распределенных вычислительных систем. Представлены основные подходы к распределенной обработке информации в вычислительных сетях и организации распределенных приложений. Проводится обзор основных подходов к организации распределенных вычислительных систем: методы удаленных вызовов процедур, многослойные клиент-серверные системы, многоагентные системы, технологии одноранговых вычислений. Рассматривается сервис-ориентированный подход к построению распределенных вычислительных систем. Приводится описание концепции грид-вычислений. Рассматривается технология и архитектура построения систем облачных вычислений.

Пособие предназначено для студентов бакалавриата и магистратуры по направлению 010300 «Фундаментальная информатика и информационные технологии» при изучении курсов «Объектная распределенная обработка», «Введение в сервис-ориентированные архитектуры», «Распределенные объектные технологии».

УДК 004.75

Содержание

1. Введение в распределенные вычислительные системы 7
1.1 Определение распределенной вычислительной системы 8
1.2 Промежуточное программное обеспечение 9
1.3 Терминология РВС 10
1.4 Классификация РВС 11
1.5 Связь в РВС 13
2. История развития распределенных вычислений 15
2.1 Первое поколение систем распределенных вычислений 16
2.2 Второе поколение систем распределенных вычислений 18
2.3 Современные РВС 21
2.4 Заключение 26
3. Веб 27
3.1 Рассвет Веба 28
3.2 Единообразное именование ресурсов 29
3.3 Общее представление ресурсов 33
3.4 Протокол передачи гипертекста 34
3.5 Заключение 40
4. Модель «Клиент-Сервер» 41
4.1 Разделение приложений по уровням 42
4.2 Типы клиент-серверной архитектуры 44
5. Объектные распределенные системы 51
5.1 Вызов удаленных процедур. 51
5.2 Организация связи с использованием удаленных объектов 54
5.3 CORBA 57
6. Агентные технологии 63
6.1 Понятие программного агента 63
6.2 Мультиагентные системы 67
6.3 Безопасность в системах мобильных агентов 69

7. Компонентные системы ... 73
7.1 Основы компонентных программных систем ... 73
7.2 Концепция JavaBeans ... 74
8. Сервис-ориентированная архитектура ... 81
8.1 Концепция СОА ... 81
8.2 Связанность программных систем ... 82
8.3 Принципы построения СОА ... 83
8.4 Подход СОА ... 85
9. Веб-сервисы ... 93
9.1 Веб-сервисы первого поколения ... 93
9.2 Стандарт WSDL ... 95
9.3 Стандарт SOAP ... 99
9.4 Второе поколение стандартов веб-сервисов ... 102
9.5 Адресация и WS-Addressing ... 109
9.6 Состояние веб-сервисов и WSRF ... 111
10. Технологии одноранговых сетей ... 117
10.1 Основы технологии одноранговых сетей ... 117
10.2 Алгоритмы работы P2P сетей ... 121
10.3 Применение технологий P2P ... 124
10.4 Достоинства и недостатки P2P ... 129
11. Технологии Грид ... 131
11.1 Архитектура Грид ... 131
11.2 Стандарты Грид ... 134
11.3 Система Globus ... 136
11.4 Система UNICORE ... 137
11.5 Параметрические модели производительности Грид ... 139
12. Облачные вычисления ... 145
12.1 Определение облачных вычислений ... 146
12.2 Многослойная архитектура облачных приложений ... 149

12.3 Компоненты облачных приложений 152
12.4 Достоинства и недостатки облачных вычислений 156
12.5 Классификация облаков 158
12.6 Наиболее распространенные облачные платформы 160
12.7 Сравнение Грид и Облачных вычислений 168
Список литературы 173
Предметный указатель 179

1. Введение в распределенные вычислительные системы

Область распределенных вычислительных систем в настоящее время характеризуется быстрыми темпами изменения идеологий и подходов. За короткую историю существования систем такого типа появилось множество различных парадигм реализации распределенных вычислений, набравших большой вес и общее признание, но практически исчезнувших впоследствии под давлением более новых и модных подходов. Однако когда технология исчезает из виду, очень часто она появляется вновь под новым именем. В результате происходит непрерывное перемешивание базовых концепций с новейшими подходами к разработке.

В середине 1990-х существовало два основных подхода к разработке распределенных вычислительных систем. С одной стороны, концепция *Веб* представляла собой ориентированное на человека распределенное информационное пространство. С другой стороны, *технологии распределенных объектов*, такие как CORBA [58] и DCOM [68] были в первую очередь ориентированы на создание распределенных сред, которые эмулировали процесс разработки и исполнения локальных приложений, обеспечивая преимущества доступа к сетевым ресурсам. Но, несмотря на первоначальную идею Веб как пространства, которое позволяло многим людям обмениваться информацией, большинство пользователей просто потребляли информацию, не публикуя ничего взамен. Между тем системы распределенных объектов росли с точки зрения предоставляемых возможностей, но становились все более тяжелыми в плане разработки и использования.

Сразу после начала нового тысячелетия произошел взрыв развития новых методов и *промежуточного программного обеспечения* для распределенных вычислительных систем, включая технологии *одноранговых сетей* (*peer-to-peer* или *P2P*) и *грид-технологии*. Применение P2P позволило множеству пользователей, которые раньше были простыми потребителями информации, поучаствовать в предоставлении контента. С другой стороны, применение технологии грид позволило интегрировать крупные комплексы обработки и хранения данных, обеспечивая их доступность для различных правительственных и научных пользователей. Концепция грид-вычислений была ориентирована на построение инфраструктуры, обеспечивающей «вычисления по требованию», аналогично тому, как мы сейчас получаем доступ к коммунальным услугам, например, к электричеству.

В то же время, бизнес-сообщество занялось разработкой следующего поколения спецификаций, призванных решить проблемы ранних стандартов распределенных объектных технологий посредством *Веб-сервисов* и *сервис-ориентированной архитектуры*. Слияние бизнес подхода к предоставлению вычислительных ресурсов в виде сервисов и концепции грид-вычислений привело к появлению в конце 2010-х новой концепции получившей название *Облачных вычислений*.

Далее в этой главе мы попытаемся дать определение распределенных вычислительных систем (РВС), рассмотрим их основы и базовые понятия.

1.1 Определение распределенной вычислительной системы

Формального определения распределенной вычислительной системы в настоящее время не существует. Из множества различных определений, можно выделить ироничное определение Лесли Лампорта[1] [44]:

> ***«Распределенной вычислительной системой*** *можно назвать такую систему, в которой отказ компьютера, о существовании которого вы даже не подозревали, может сделать ваш собственный компьютер непригодным к использованию».*

Это определение он дал в мае 1987 года, в своем письме коллегам по поводу очередного отключения электроэнергии в машинном зале.

Эндрю Таненбаум[2], в своем фундаментальном труде «Распределённые системы. Принципы и парадигмы» [2] предложил следующее (чуть более строгое) определение, которого мы будем придерживаться в рамках данной книги:

> ***«Распределенная вычислительная система (РВС)*** *– это набор соединенных каналами связи независимых компьютеров, которые с точки зрения пользователя некоторого программного обеспечения выглядят единым целым».*

В этом определении фиксируются два существенных момента: автономность узлов РВС и представление системы пользователем, как единой структуры. При этом, основным связующим звеном распределенных вычислительных систем является программное обеспечение.

---

[1] Лесли Лампорт (родился в 1941 году) – американский ученый в области теории вычислительных систем, первый лауреат премии Дейкстры за достижения в области распределенных вычислений (2000 год), разработчик системы LaTeX.

[2] Эндрю Стюарт Таненбаум (родился в 1944 году) – профессор Амстердамского свободного университета, создатель операционной системы Minix, автор множества учебных трудов в области информатики и вычислительной техники.

1.2 Промежуточное программное обеспечение

Распределенная вычислительная система представляет собой программно-аппаратный комплекс, ориентированный на решение определенных задач. С одной стороны, каждый вычислительный узел является автономным элементом. С другой стороны, программная составляющая РВС должна обеспечивать пользователям видимость работы с единой вычислительной системой. В связи с этим выделяют следующие важные характеристики РВС:

- возможность работы с различными типами устройств:
  - с различными поставщиками устройств;
  - с различными операционными системами,
  - с различными аппаратными платформами.

  Вычислительные среды, состоящие из множества вычислительных систем на базе разных программно-аппаратных платформ, называются *гетерогенными*;
- возможность простого расширения и масштабирования;
- перманентная (постоянная) доступность ресурсов (даже если некоторые элементы РВС некоторое время могут находиться вне доступа);
- сокрытие особенностей коммуникации от пользователей.

Для обеспечения работы гетерогенного оборудования РВС в виде единого целого, стек программного обеспечения (ПО) обычно разбивают на два слоя. На верхнем слое располагаются *распределенные приложения*, отвечающие за решение определенных прикладных задач средствами РВС. Их функциональные возможности базируются на нижнем слое – *промежуточном программном обеспечении (ППО)* . ППО взаимодействует с системным ПО и сетевым уровнем, для обеспечения прозрачности работы приложений в РВС (см. рис. 1).

Для того чтобы РВС могла быть представлена пользователю как единая система, применяют следующие типы прозрачности в РВС:

- *прозрачный доступ к ресурсам* – от пользователей должна быть скрыта разница в представлении данных и в способах доступа к ресурсам РВС;
- *прозрачное местоположение ресурсов* – место физического расположения требуемого ресурса должно быть несущественно для пользователя;
- *репликация* – сокрытие от пользователя того, что в реальности существует более одной копии используемых ресурсов;

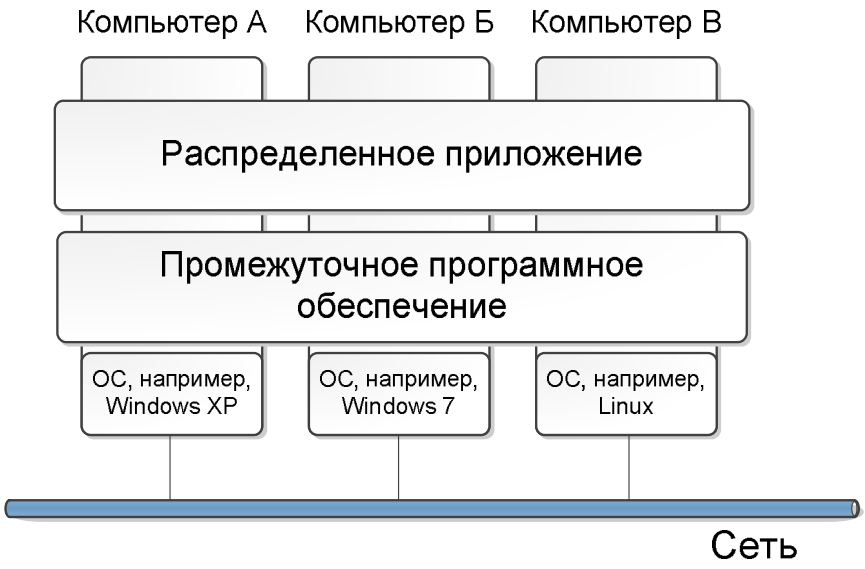


Рис. 1. Слои программного обеспечения в РВС

- *параллельный доступ* – возможность совместного (одновременного) использования одного и того же ресурса различными пользователями независимо друг от друга. При этом факт совместного использования ресурса должен оставаться скрытым от пользователя;
- *прозрачность отказов* – отказ (отключение) каких-либо ресурсов РВС не должен оказывать влияния на работу пользователя и его приложения.

1.3 Терминология РВС

1. *Ресурсом* называется любая программная или аппаратная сущность, представленная или используемая в распределенной сети. Например, компьютер, устройство хранения, файл, коммуникационный канал, сервис и т.п.
2. *Узел* – любое аппаратное устройство в распределенной вычислительной системе.
3. *Сервер* – это поставщик информации в РВС (например, веб-сервер).
4. *Клиент* – это потребитель информации в РВС (например, веб-браузер).
5. *Пир* – это узел, совмещающий в себе как клиентскую, так и серверную часть (т.е. и поставщик, и потребитель информации одновременно).
6. *Сервис* – это сетевая сущность, предоставляющая определенные функциональные возможности [30] (например, веб-сервер может предоставлять сервис передачи файлов по протоколу HTTP). В рамках одного узла могут предоставляться несколько различных сервисов.

На рисунке 2 приведена схема, устанавливающая взаимоотношения между данными терминами. Из схемы видно, что каждый компьютер или устройство

представляет собой сущность в распределенной вычислительной системе в виде узла. При этом на каждом узле может располагаться несколько клиентов, серверов, сервисов или пиров. Важно заметить, что любой узел, сервер, пир или сервис (*но не клиент!*) являются ресурсами распределенной вычислительной системы.

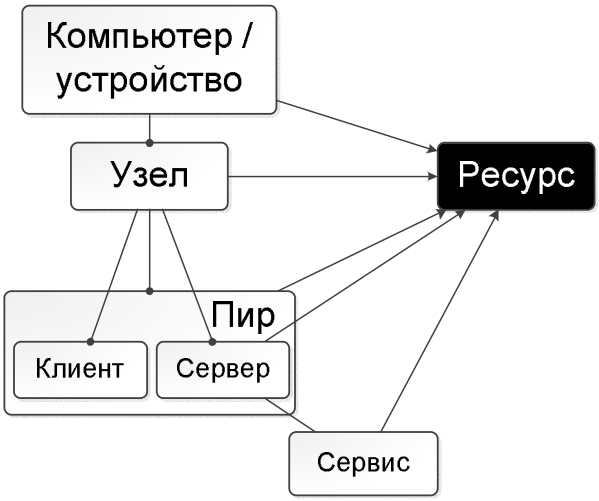


Рис. 2. Схема взаимоотношений между терминами РВС

Сервис получает запрос на предоставление определенных данных (почти как аргументы, передаваемые при вызове локальной функции) и возвращает ответ. Таким образом, сервис можно определить как некую замену вызова функции на локальном компьютере. Существует множество технологий, обеспечивающих создание и сопровождение сервисов в распределенных вычислительных системах: технология XML веб-сервисов, сервисы REST и др.

1.4 Классификация РВС

Выделяют следующие признаки классификации РВС по шкале «централизованный – децентрализованный»:

– методы обнаружения ресурсов;
– доступность ресурсов;
– методы взаимодействия ресурсов.

Существует множество различных технологий, обеспечивающих поиск и обнаружение ресурсов в РВС (например, такие *службы обнаружения ресурсов* как DNS, Jini Lookup, UDDI и др.). Примером *централизованного метода обнаружения* ресурсов может служить служба DNS (англ. Domain Name System – система доменных имен). Данная служба работает по принципам, чрезвычайно похожим на принцип работы телефонной книги. На основе указанного имени сайта (например, www.susu.ac.ru) DNS возвращает его IP-адрес (например,

85.143.41.59). Таким образом, сервер DNS представляет собой большую базу данных ресурсов, расположенных в РВС. Существует ограниченное количество серверов, которые предоставляют службу DNS. Обычно пользователь указывает ограниченное количество (1 или 2) таких серверов для работы. И если указанные сервера отключаются, то процесс обнаружения ресурсов останавливается, если вручную не указать альтернативные сервера.

При использовании *децентрализованного метода обнаружения* ресурсов (например, в сети Gnutella [33]) запрос на поиск отправляется всем узлам, известным отправителю. Эти узлы производят поиск ресурса у себя, и транслируют запрос далее. Таким образом, отсутствуют выделенные узлы для обнаружения и централизованное хранилище информации о ресурсах, доступных в сети.

Другим важным фактором является *доступность ресурсов* РВС. Примером *централизованной доступности ресурсов* в РВС может являться технология веб-сервисов. Существует только один сервер с выделенным IP-адресом, который предоставляет определенный веб-сервис или сайт. Если данный узел выйдет из строя или будет отключен от сети, данный сервис станет недоступна. Естественно, можно применить методы репликации для расширения доступности определенного сайта или сервиса, но доступность определенного IP-адреса останется прежней.

Существуют системы, предоставляющие *децентрализованные подходы к доступности ресурсов* посредством множественного дублирования сервисов, которые могут обеспечить функциональность, необходимую пользователю. Наиболее яркими примерами децентрализованной доступности ресурсов могут служить одноранговые вычислительные системы (BitTorrent, Gnutella, Napster), где каждый узел играет роль, как клиента, так и сервера, который может предоставлять ресурсы и сервисы, аналогичные остальным устройствам данной сети (поиск, передача данных и др.)

Еще одним критерием классификации РВС могут служить *методы взаимодействия узлов. Централизованный подход к взаимодействию узлов* основан на том, что взаимодействие между узлами всегда происходит через специальный центральный сервер. Таким образом, один узел не может обратиться к другому непосредственно.

Децентрализованный подход к взаимодействию реализуется в одноранговых вычислительных системах. Такой подход основывается на прямом взаимодействии между узлами РВС, т.к. каждый узел играет как роль клиента, так и роль сервера.

1.5 Связь в РВС

Понятие «распределенная вычислительная система» подразумевает, что компоненты такой системы распределены, т.е. удалены друг от друга. Очевидно, что функционирование подобных систем невозможно без эффективной связи между ее компонентами.

Задачи организации обмена между распределенными (территориально, административно и т.д.) компонентами давно и в значительной мере успешно решаются в вычислительных сетях, и, естественно, что РВС используют наработанный опыт.

Взаимодействие в вычислительных сетях базируется на протоколах. *Протокол* – это набор правил и соглашений, описывающих процедуру взаимодействия между компонентами системы (в том числе и вычислительной).

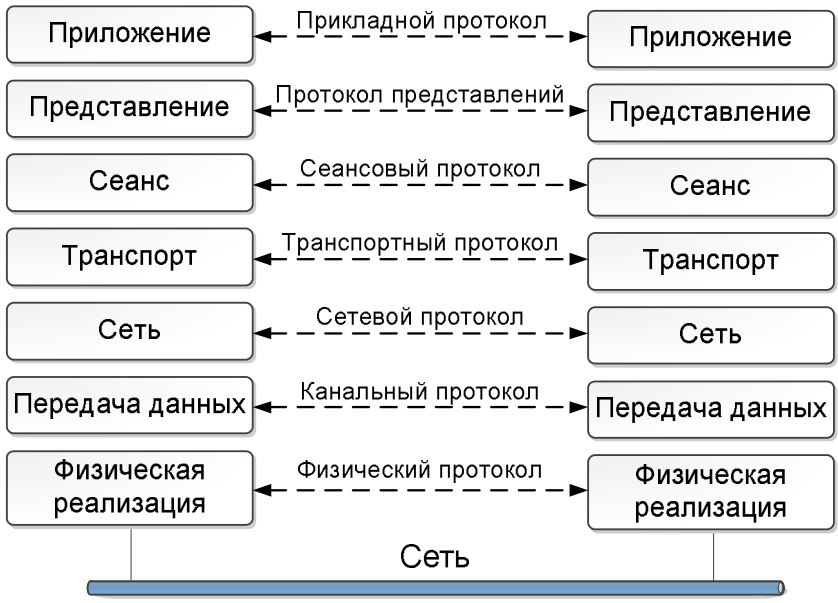


Рис. 3. Уровни модели OSI

Если система поддерживает определённый протокол, она, с большой долей вероятности, окажется способна взаимодействовать с другой системой, которая так же поддерживает данный протокол. В области вычислительных коммуникаций уже длительное время существует общепринятая система протоколов – сетевая модель OSI (англ. Open Systems Interconnection basic reference model – базовая эталонная модель взаимодействия открытых систем). Эта модель представляет собой стек протоколов разного уровня, которые позволяют описать практически все аспекты взаимодействия компонентов РВС.

Детальное рассмотрение стека протоколов OSI и особенностей его различных уровней лежит за рамками данной дисциплины.

2. История развития распределенных вычислений

Проблема выбора между централизованной и распределенной моделями предоставления вычислительных ресурсов является одной из ключевых проблем организации вычислительных систем. Примером этого может являться статья «Централизованные и децентрализованные вычислительные системы: организационные соображения и варианты управления», опубликованная еще 1983 г. [43].

С того момента, как появилась возможность соединять компьютеры между собой, многие группы исследователей занялись изучением возможностей, предоставляемых системами распределенных вычислений, создавая множество библиотек и промежуточного программного обеспечения. Разработанное ППО предназначалось для того, чтобы обеспечить организацию и управление географически-распределенными ресурсами таким образом, чтобы они представляли собой единую виртуальную параллельную вычислительную машину.

До середины 70-х годов прошлого века по причине высокой стоимости телекоммуникационного оборудования и относительно слабой мощности вычислительных систем доминировала централизованная модель. В конце 70-х годов появление систем разделения времени и удаленных терминалов, явилось предпосылкой возникновения *клиент-серверной архитектуры*, обеспечивающей предоставление ресурсов мейнфреймов (больших вычислительных машин) конечным пользователям посредством удаленного соединения. Дальнейшее развитие телекоммуникационных систем и появление персональных компьютеров дало толчок развитию клиент-серверной парадигмы обработки данных. На последующих этапах произошло уточнение и переосмысление задачи распределенных вычислений.

В 2000-м году Ян Фостер[1] определил задачу распределенных вычислительных систем как «гибкое, безопасное, координированное распределение ресурсов среди динамических наборов пользователей, организаций и ресурсов» [31]. Он предложил называть такие распределенные вычислительные системы термином *грид* (более формальное определение грид будет дано в главе 11 «Технологии Грид»). Коммерческие разработчики, развивая идею грид, в 2007-2008 годах представили концепцию «облачных вычислений», в основе ко-

[1] Ян Фостер – директор института вычислений, старший научный сотрудник Арагонской национальной лаборатории (США). Является автором множества научных работ, посвященных параллельным и распределенным вычислениям. Руководил разработкой нескольких международных распределенных вычислительных систем. Автор концепции грид.

торой было предоставление высокомасштабируемых виртуальных вычислительных ресурсов конечному пользователю через интернет в виде услуг.

На текущий момент, использование распределенных вычислений в виде технологий грид и облачных вычислений набирает обороты. Они применяются для решения различных задач, их использование становится все проще. Распределенные вычисления становятся неотъемлемой частью научных и коммерческих высокопроизводительных вычислений.

2.1 Первое поколение систем распределенных вычислений

Первые проекты по распределенным вычислениям, появившиеся в начале 1990-х годов, основывались на объединении вычислительных возможностей суперкомпьютеров. Основной целью данных проектов было предоставление вычислительных ресурсов для определенного ряда высокопроизводительных приложений. В качестве типичных проектов того времени можно рассмотреть проекты FAFNER [34] и I-WAY [61]. Эти проекты стали базовыми, для всей отрасли распределенных вычислений в дальнейшем. На них основывались первые попытки стандартизации распределенных вычислений в гетерогенных вычислительных средах.

2.1.1 Проект FAFNER

Проект FAFNER был создан для решения задачи разложения больших чисел на основе мощностей географически-распределенных вычислительных систем. Нахождение простых множителей больших чисел является позволяет расшифровать данные, зашифрованные на основе алгоритма RSA.

Для шифрования секретной информации широко используется метод кодирования, основанный на публичном ключе RSA (аббревиатура из первых букв фамилий разработчиков данного метода: Rivest, Shamir и Adleman). Метод работы данного ключа основан на том, что разложение на множители больших чисел (сто и более знаков) – чрезвычайно сложная вычислительная задача. В марте 1991 корпорация RSA Data Security основала конкурс по поиску и реализации методов разложения больших чисел на множители. Это состязание обеспечило создание крупнейшей библиотеки по методикам поиска простых множителей от крупнейших ученых со всего земного шара.

Все алгоритмы поиска простых множителей, известные на сегодняшний день, требуют чрезвычайно большого количества вычислений (поэтому этот метод и используется для шифрования). Но особенность параллельной реализации этих алгоритмов состоит в том, что процессы поиска делителей вычисли-

тельно независимы, и не требуют обмена информации во время расчета. Первые попытки реализовать подобный алгоритм на распределенных вычислительных системах основывались на обмене электронными письмами.

В 1995 г. консорциумом организаций в области информационных технологий был запущен проект FAFNER – Factoring via Network-Enabled Recursion (Сетевое разложение на множители посредством рекурсии) по решению задачи разложения больших чисел посредством Веб-серверов.

Можно выделить следующие особенности, отличавшие этот проект от многих других [34]:

– реализация NFS – Network File System (Сетевая Файловая Система) позволяла даже малым рабочим станциям (с 4 Мб оперативной памяти) выполнять полезную работу, рассчитывая свой маленький фрагмент задачи;
– проект FAFNER поддерживал анонимную регистрацию участников. Пользователи могли поделиться своими вычислительными ресурсами без боязни раскрытия своей личной информации;
– консорциум сайтов, представлявших костяк вычислительной системы, формировали иерархическую структуру веб-серверов, что уменьшало возможность возникновения «узкого места» в вычислительной системе.

Данная система доказала свою надежность и качественность, заняв первое место по производительности в конкурсе, проводимом в рамках конференции Supercomputing’95.

2.1.2 Проект I-WAY

I-WAY – Information Wide Area Year (Год Информации Глобальных Сетей) – экспериментальная высокопроизводительная сеть, которая объединила множество высокопроизводительных компьютеров и передовые средства визуализации в США. Она была спроектирована в начале 1995, с целью объединения высокоскоростных сетей, существующих на тот момент. Данные и компьютерные ресурсы были распределены по 17-и локациям в США и объединены 10-ю сетями, с различной пропускной способностью, различными протоколами соединения и с использованием различных сетевых технологий для их построения [23].

В рамках проекта, была построена аппаратная инфраструктура, посредством которой осуществлялся доступ к ресурсам сети I-WAY. Она состояла из базовых рабочих станций под управлением операционной системы UNIX, на которые было установлено специальное ППО (сервер I-POP). Система I-POP брала на себя функции шлюза к ресурсам сети I-WAY. Каждый такой сервер

поддерживал стандартные процедуры аутентификации, резервирования ресурсов создания процессов и коммуникации.

Также, в рамках данного проекта был разработан планировщик ресурсов, известный как «Брокер Вычислительных Ресурсов» (CRB – Computing Resources Broker). Он обеспечивал выполнения функций управления очередями задач, распределения заданий между компьютерами и слежения за ходом решения.

Для поддержки пользовательских приложений была адаптирована библиотека передачи данных Nexus. В нее были включены механизмы, поддерживающие автоматическое конфигурирование работы пользовательского приложения, в зависимости от методов передачи данных, базовой операционной системы и т. п.

Проект I-WAY использовался для решения следующих задач [23]:

- суперкомпьютерные вычисления;
- доступ к удаленным ресурсам;
- задачи виртуальной реальности.

2.2 Второе поколение систем распределенных вычислений

2.2.1 Грид

Проекты высокопроизводительных вычислительных систем, реализованные в начале 1990-х годов, выявили основные проблемы, которые необходимо было решить для развертывания стабильных распределенных вычислительных систем. Для решения всех указанных задач была разработана концепция *грид* – РВС, обеспечивающей «гибкое, безопасное, координированное распределение ресурсов среди динамических наборов пользователей, организаций и ресурсов». Основной задачей грид было построение инфраструктуры, обеспечивающей «*вычисления по требованию*» (*utility computing*), аналогично тому, как мы сейчас получаем доступ к коммунальным услугам, например, к электричеству. В конце 1990-х, в результате исследований систем распределенных вычислений, Ян Фостер вывел три основных требования, которым должны соответствовать РВС для того чтобы называться грид [27].

1. *Гетерогенность*. Вычислительная среда грид может состоять из множества различных ресурсов, обладающих различными характеристиками и параметрами.
2. *Масштабируемость*. Грид может состоять из сколь угодно большого числа ресурсов. И если поставленная задача решается на 10-и компьютерах за определенное время, существует опасность, что на 100 компьютерах она

будет решаться в 100 раз медленнее, в связи с многократным возрастанием расходов на передачу данных между узлами.

3. *Приспособляемость*. При работе с грид-системами необходимо учитывать, что ошибки при работе с ресурсами – это не исключение, а правило. Среда грид может состоять из сотен компьютеров, и ошибки в работе десятка из них не должны влиять на полученные результаты решения.

Дальнейшее развитие грид происходило в рамках определенных Фостером требований. Применение технологии грид позволило объединить крупные комплексы обработки и хранения данных, обеспечивая их доступность для различных пользователей, включая правительственные и научные организации. Для реализации предложенных концепций в 1997 году был запущен проект Globus [20], направленный на разработку и предоставление инфраструктуры для грид-вычислений. Первоначально, Globus был развитием проекта I-WAY, который мы рассмотрели выше. Но в процессе эволюции проекта Globus, основной акцент разработчиков был перенесен с поддержки высокопроизводительных вычислений в сторону сервисов поддержки *виртуальных организаций* и предоставление возможности приложениям работать с распределенными разнородными вычислительными ресурсами как с единой виртуальной машиной. Более подробно о конкретных технологии грид и особенностях их разработки мы поговорим в главе 11 «Технологии Грид».

2.2.2 Проект Legion

Проект Legion – это объектно-ориентированная система, предоставляющая программную оболочку для организации однородного взаимодействия гетерогенных распределенных высокопроизводительных вычислительных систем [35]. Первая реализация системы была выпущена в ноябре 1997-го. Основной целью проекта было предоставление пользователям единой интегрированной инфраструктуры РВС, независимо от масштаба, географического положения, языка или операционной системы. В отличие от Globus, система Legion основывалась на объектно-ориентированном подходе, включая обязательную поддержку абстракций данных, инкапсуляции, наследования и полиморфизма.

Legion предоставлял пользователю набор объектов, предоставляющих базовые сервисы:

- *объекты вычислителей* – абстракции, реализующие базовые принципы работы с вычислительными ресурсами;
- *объекты систем хранения данных* – абстракции, предоставляющие базовые методы работы с системами хранения данных;

- *объекты связывания* – объекты, обеспечивающие связи между абстрактным идентификатором объекта и его физическим адресом;
- *объекты контекста* – объекты, реализующие проекцию пользовательских имен объектов на абстрактные идентификаторы объектов в системе Legion.

В течение года система развилась настолько, что было принято решение о коммерциализации проекта, и в августе 1998 г компания Applied Metacomputing выпустила коммерческую версию проекта. Проект активно разрабатывался вплоть до 2001 года, после чего развитие проекта остановилось.

2.2.3 Распределенные объектные системы

При рассмотрении второго поколения распределенных вычислительных систем, нельзя не затронуть такой класс, как распределенные объектные системы. Они предоставляют базовые методы для регистрации, сериализации и десериализации объектов обеспечивая удаленный вызов методов.

В середине 1990-х годов одним из наиболее распространенных методов построения распределенных объектных систем являлась архитектура *CORBA* (*Common Object Request Broker Architecture* — общая архитектура брокера объектных запросов) [37]. В 1997-1998 годах консорциум OMG (Object Management Group) опубликовал вторую версию спецификации стандарта CORBA, который обеспечивал стандартный протокол взаимодействия объектно-ориентированных систем. Вместе со стандартом были выпущены отображения для наиболее распространенных ОО-языков: C++ и Java. В состав решения CORBA было включено множество сервисов, которые можно было применить к отраслям электронной коммерции и науки. Также, на базе CORBA возможно было реализовать концептуальные модели архитектуры распределенной вычислительной среды, т.к. она обеспечивала взаимосвязь с языком UML (Unified Modeling Language – Унифицированный Язык Моделирования). Продержавшись на рынке около 5 лет, популярность CORBA пошла на спад. Частично это объясняют сложностью процесса разработки на основе объектно-ориентированной модели, предлагаемой CORBA, частично – высокой стоимостью приобретения и поддержки таких систем. Также, многие считают, что CORBA ушла в тень, в связи с тем, что не соответствовала стандартам Веб-систем, мода на которые пришла в начале 2000-х. Более подробно роли Веб в области распределенных вычислительных систем будет рассказано в главе 3 «Веб».

Также следует упомянуть о модели Java. Тогда как, для борьбы с гетерогенностью вычислительной среды CORBA реализует высокоуровневые стан-

дарты взаимодействия, модель Java использует единую виртуальную среду. В определенной степени JVM (Java Virtual Machine – Виртуальная Машина Java), в совокупности с Java-приложениями и сервисами, преодолевают проблемы гетерогенности вычислительных систем, предоставляя методы удаленного вызова процедур посредством технологии RMI (Remote Method Invocation – Удаленный Вызов Методов).

Более подробно об архитектуре CORBA и методах работы с удаленными объектами мы поговорим в главе 5 «Объектные распределенные системы».

2.3 Современные РВС

На сегодняшний день, РВС отходят от традиционных понятий высокопроизводительных распределенных вычислений в сторону развития виртуального сотрудничества и виртуальных организаций. *Виртуальная организация* – это ряд людей и/или организаций, объединенных общими правилами коллективного доступа к определенным вычислительным ресурсам [31]. Методы предоставления доступа к вычислительным ресурсам становятся сервисно-ориентированными, что позволяет гибко использовать одни и те же вычислительные ресурсы различными потребителями.

Значительно расширились области автоматизированного управления ресурсами. Человек не в силах вручную решить задачу распределения вычислений в системах такого масштаба и гетерогенности. Таким образом, необходимо использование автоматизированных систем управления задачами, которые берут на себя задачи управления предоставляемой системой. Также, с возрастанием масштаба вычислительных сетей, необходимы автоматизированные средства обработки ошибок и восстановления вычислительного процесса.

2.3.1 Одноранговые (peer-to-peer) сети

В 1999 году, в Северо-восточном Университете (Массачусетс, США) первокурсник Шон Фэннинг написал систему обмена MP3 файлами между пользователями. Этот проект получил название Napster. Он стал первым проектом, положившим начало технологии одноранговых (Peer-to-peer (P2P) – англ. «равный-к-равному») распределенных вычислительных сетей. Через 2 года Napster был закрыт совместными усилиями владельцев авторских прав на музыкальные произведения, распространявшиеся через эту сеть. Но по примеру Napster развился целый класс P2P систем нового, децентрализованного типа, которые закрыть было значительно сложнее.

В 2000 году Джастин Франкел (20-ти летний хакер из США, в 1997 году выпустил бесплатный MP3 плеер WinAmp) написал Gnutella – P2P протокол передачи файлов. В отличие от Napster, который использовал центральный сервер для установления связи между пирами, Gnutella полагалась исключительно на системы конечных пользователей для организации сети. Таким образом, в отличие от Napster, сеть Gnutella оказалось невозможно «закрыть», отключив центральный сервер. Миллионы человек до сих пор пользуются данной системой.

При работе в рамках парадигмы P2P, компьютеры обмениваются ресурсами непосредственно друг с другом, без использования центрального сервера. Подход P2P обеспечивает решение проблем, возникших в результате экспоненциального роста интернет и Веб. Применение P2P позволило множеству пользователей, которые раньше были простыми потребителями информации, поучаствовать в предоставлении контента. В момент своего появления, P2P было скорее модным словом, чем продуманной концепцией. В результате мощного продвижения с помощью средств массовой информации, технология P2P распространилась в академических и промышленных кругах.

В рамках концепции P2P все входящие в сеть компьютеры взаимодействуют друг с другом напрямую, без использования централизованных серверов. Основные достоинства одноранговых вычислительных систем:

- упрощается поддержка масштабируемости при значительном росте количества узлов в вычислительной сети;
- повышается отказоустойчивость сети, т.к. сбой любого вычислительного узла не может привести к остановке функционирования сети целиком.

Тем не менее, существует ряд препятствий при построении P2P сетей:

1. При работе с P2P приложениями, вычислительный узел берет на себя функции, как клиента, так и сервера. Это приводит к *увеличению требований к производительности* каждого компьютера, включенного в такую сеть.
2. *Низкая степень защищенности машин*, участвующих в P2P сети объясняется тем, что они предоставляют открытый доступ к своим ресурсам (таким как такты процессора, определенные папки на жестком диске и т. п.). Таким образом, при отсутствии средств защиты, компьютеры, включенные в P2P подвержены риску взлома или заражения со стороны недобросовестных участников.

3. При построении P2P сети приходится преодолевать возможную *гетерогенность аппаратного и программного обеспечения* ее потенциальных участников. Этот вопрос может быть решен путем применения таких технологий как XML или Java.
4. Основная проблема P2P сетей – это *поиск доступных ресурсов*, без использования централизованной точки управления. Каждому узлу приходится производить поиск среди сотен и тысяч ресурсов внутри сети, что является очень трудоемкой и ресурсоемкой задачей.

Шумиха и споры о легальности P2P привели к недоверию к этой технологии, хотя на самом деле, она обладает существенным потенциалом. В настоящее время P2P – это общепринятая концепция построения распределенных высокомасштабируемых вычислительных систем. Несмотря на все трудности, происходит бурное развитие и использование P2P сетей. Среди наиболее известных и значимых примеров, стоит отметить такие проекты как BitTorrent, Napster, Skype, SETI@home (хотя принадлежность SETI@home к роду P2P систем вызывает серьезные споры среди разработчиков распределенных приложений).

Несмотря на то, что базовая философская концепция грид и P2P различается, обе технологии пытаются решить одну и ту же проблему – создание виртуального слоя [69] над существующей инфраструктурой Интернет для обеспечения совместной работы и использования совместных ресурсов [24]. Тем не менее, в реализации, подходы сильно отличаются. Грид скорее ориентирована на объединение виртуальных организаций для обеспечения совместной работы их участников, а P2P связывает отдельных пользователей находящихся за конечными узлами сети интернет (то есть за NAT, брандмауэрами и др.). Более подробно о концепции P2P вычислений и будет рассказано в главе 10 «Технологии одноранговых сетей».

2.3.2 Сервис-ориентированная архитектура

В начале 2000-х годов бизнес-сообщество занялось разработкой следующего поколения спецификаций, призванных решить проблемы ранних стандартов распределенных объектных технологий посредством *веб-сервисов* и *сервис-ориентированной архитектуры (Service-Oriented Architecture – SOA).*

Стандарты веб-сервисов были разработаны по инициативе организаций, занимающихся предоставлением удаленного доступа к определенным вычислительным ресурсам, и закреплены консорциумом W3C. К основным стандартам разработки и функционирования веб-сервисов можно отнести:

- SOAP – основанный на XML протокол взаимодействия веб-сервисов;
- WSDL (Web Services Description Language – Язык описания веб-сервисов) – это методология описания ресурсов, предоставляемых веб-сервисом;
- UDDI (Universal Description Discovery and Integration – Универсальный метод поиска и интеграции) – метод описания, поиска, взаимодействия и использования веб-сервисов.

На сегодняшний день, сервис-ориентированный подход является стандартом «де-факто» при разработке распределенных вычислительных систем. Более подробно о сервис-ориентированных системах и технологиях веб-сервисов будет рассказано в главе 8 «Сервис-ориентированная архитектура» и главе 9 «Веб-сервисы».

2.3.3 Агенты

Несмотря на все преимущества технологии веб-сервисов, они не предоставляют новых методологий и решений построения широкомасштабных вычислительных сетей. Для поиска решений в этом направлении, необходимо рассмотреть агентно-ориентированную парадигму построения РВС.

Вычислительные сети на основе так называемых *агентов* – это принципиально иной подход к организации РВС. *Программный агент* – это автономный процесс, способный реагировать на среду исполнения и вызывать изменения в среде исполнения, возможно, в кооперации с пользователями или другими агентами. Рассмотрим основные принципы работы агентных сетей [40]:

- *автономность* – агенты функционируют автономно, без возможности постороннего вмешательства в их внутреннее состояние;
- *социальное поведение* – агенты взаимодействуют друг с другом посредством определенного языка;
- *активность* – агенты взаимодействуют с окружающей средой, получая определенные сигналы и отвечая на них;
- *про-активность* – агенты действуют целенаправленно.

Агентные сети принципиально приспособлены для функционирования в динамически-изменяемой окружающей среде. В этом случае, автономность агентов позволяет организовать динамическую подстройку вычислительного алгоритма под условия вычислительной среды. Таким образом, РВС можно представить как набор взаимодействующих компонентов, а информация, которой они обмениваются, разбивается на определенные категории:

- *информация о компонентах и их функциональных возможностях*, в рамках определенной области;
- *информация о взаимодействиях между компонентами*;
- обобщенная *информация о рабочем процессе* и более конкретная информация по той или иной задаче.

Для обеспечения функционирования такой системы, необходима стандартизация методов взаимодействия между компонентами. Для решения этой задачи разрабатываются и стандартизуются языки взаимодействия агентов (Agent Communication Languages, ACLs). Одним из наиболее известных, является архитектура взаимодействия FIPA (Foundation for Intelligent Physical Agents, базис интеллектуальных физических агентов). Эта архитектура стандартизует методы взаимодействия агентов и агентных систем.

Более подробно, принципы организации агентных систем будут рассмотрены в главе 6 «Агентные технологии».

2.3.4 Облачные вычисления

Облако – это парадигма крупномасштабных распределенных вычислений, основанная на эффекте масштаба, в рамках которой пул абстрактных, виртуализованных, динамически-масштабируемых вычислительных ресурсов, ресурсов хранения, платформ и сервисов предоставляется по запросу внешним пользователям через Интернет [29].

Не смотря на то, что метафора «облако» уже давно используется специалистами в области сетевых технологий для изображения на сетевых диаграммах сложной вычислительной инфраструктуры (или же Интернета как такового), скрывающей свою внутреннюю организацию за определенным интерфейсом, термин «Облачные вычисления» появился на свет совсем недавно. Согласно результатам анализа поисковой системы Google, термин «Облачные вычисления» («Cloud Computing») начал набирать вес в конце 2007 – начале 2008 года, постепенно вытесняя популярное в то время словосочетание «Грид-вычисления» («Grid Computing»). Судя по заголовкам новостей того времени, одной из первых компаний, давших миру данный термин, стала компания IBM, развернувшая в начале 2008 года проект «Blue Cloud» и ставшая спонсором Европейского проекта «Joint Research Initiative for Cloud Computing».

На сегодняшний день уже можно говорить о том, что облачные вычисления прочно вошли в повседневную жизнь каждого пользователя Интернета (хотя многие об этом и не подозревают). Однако до сих пор нет единого мнения о

том, что такое «Облачные Вычисления» и каким образом они соотносятся с парадигмой «Грид-вычислений».

Более подробно, принципы организации агентных систем будут рассмотрены в главе 12 «Облачные вычисления».

2.4 Заключение

Распределенные вычислительные системы – это сформировавшаяся сфера высокопроизводительных вычислений, обладающая своей спецификой, ярко выраженным классом решаемых задач и методами их решения. Разрабатываются и внедряются новые концепции построения распределенных систем, расширяется круг решаемых ими задач, упрощается процесс организации, разрабатываются более простые методы использования ресурсов конечными пользователями.

3. Веб

Пожалуй, Веб стал настолько распространенным не только из-за того, что он делает, но из-за того, что он не пытается делать. Так, Веб формирует очень простую основу, на базе которой можно легко создавать новые концепции. Однако сложность заключается в том, чтобы новые идеи не блокировали развитие самого изобретения. Именно поэтому основным направлением деятельности Консорциума Всемирной Паутины (W3C) и других заинтересованных групп является определение архитектуры Веба, способов ее поддержания и совершенствования. Это своеобразный баланс между естественным эволюционным развитием и сохранением уже накопленного опыта.

В основе веб-концепции лежит понятие *ресурса*. Ресурсом может быть что угодно – фотография, налоговая накладная, страна, политическое движение, алгоритм, мысль, человек – все, что имеет некоторые границы и поэтому может быть идентифицировано. Веб не задает никаких ограничений на допустимые ресурсы; все, что он определяет – это то, как эти ресурсы могут передаваться между компьютерами и, следовательно, людьми. Самая основная возможность любой распределенной системы – перемещать ресурсы с одной машины на другую. Для этого Веб поддерживает несколько очень простых технологий.

– *Именование ресурсов.* Веб определяет гибкие и расширяемые способы именования произвольных ресурсов, именуемые унифицированными идентификаторами ресурсов (URI – Uniform Resource Identifier).
– *Представление ресурсов.* Для передачи произвольных ресурсов между компьютерами, необходимо такое представление ресурса, которое может быть преобразовано в поток битов и передано по сети. В идеале формат представления должен быть принят всеми, что обеспечивает простоту создания и интерпретации ресурса. В настоящее время существует несколько типов представления ресурса. Первоначальный и наиболее важный из них – это язык гипертекстовой разметки (HTML – Hypertext Markup Language).
– *Передача ресурсов.* Hypertext Transfer Protocol (HTTP) стал основным механизмом передачи данных в Интернете. HTTP – это протокол передачи по типу клиент/сервер, который поддерживает минимальный необходимый набор операций передачи данных.

Вместе эти технологии стали очень мощным механизмом для создания широкого спектра распределенных приложений. После краткого введения о том, как зарождался Веб, мы более внимательно рассмотрим каждую из них.

3.1 Рассвет Веба

«Сайты должны обладать возможностью взаимодействия
в едином, универсальном пространстве».
Тим Бернерс-Ли

В 1990 году Тим Бернерс-Ли[1] создал первый веб-браузер. Это произошло во время его работы в лаборатории ЦЕРН в Швейцарии, где несколько тысяч ученых-ядерщиков использовали различные компьютеры. Бернерса-Ли не устраивал способ передачи данных между компьютерами сотрудников: для получения доступа к необходимой информации нужно было осуществить вход на компьютер другого пользователя. После написания нескольких программ для преобразования и передачи информации между различными системами, он задумался о более эффективных способах реализации. Вскоре возникла идея создания некоторой воображаемой информационной системы, которая могла бы использоваться всеми. На базе уже существующей концепции гипертекста был создан первый веб-браузер, который подключался через DNS и TCP, ставший началом развития Всемирной паутины (World Wide Web). По словам Тима Бернерса-Ли, это был лишь первый шаг, основная сложность заключалась в том, чтобы заинтересовать и привлечь пользователей.

Первый браузер, называвшийся WorldWideWeb, работал в среде NeXT, а первым веб-сервером стал `nxoc01.cern.ch`, позже переименованный в `info.cern.ch`. С этого момента начинается широкое распространение ПО, и к 1992 году насчитывается уже 50 веб-серверов по всему миру, а в 1993 NCSA выпускает альфа-версию браузера «*Mosaic for X*». Позже, в октябре 1994, Бернерс-Ли основал Консорциум Всемирной паутины (W3C), цель которого – стандартизация общих протоколов для содействия развитию Веба и обеспечения его совместимости. В это время происходит настоящий взрыв Всемирной паутины, так, число пользователей Интернета возросло с 40 миллионов в 1995 году до 150 млн. в 1998 году и 320 млн. в 2000 году. На сегодняшний день насчитывается 1 миллиард 350 миллионов пользователей Всемирной паутины по всему миру.

[1] Сэр Тимоти Джон Бернерс-Ли (родился в 1955 году) – британский учёный, изобретатель URI, URL, HTTP, HTML, изобретатель Всемирной паутины (Wold Wide Web), и действующий глава Консорциума Всемирной паутины (W3C).

3.2 Единообразное именование ресурсов

Центральное место в архитектуре веба занимает унифицированный (единообразный) идентификатор ресурса (URI – Uniform Resource Identifier). URI представляет собой последовательность символов, которая однозначно идентифицирует ресурс в Интернете. URI обеспечивает единый механизм именования произвольных ресурсов и имеет очень простой синтаксис.

Рассмотрим некоторые примеры:

1. http://www.google.com
2. urn:isbn:0-395-36341-1
3. urn:jxta:uuid:59616261646162614A78746150325033F3BC76
4. mailto:john.doe@example.com
5. http://www.bbc.co.uk/weather/
6. http://www.google.com/search?client=safari&q=web

Каждый URI начинается со схемы обращения к ресурсу (часто под схемой обращения к ресурсу подразумевается сетевой протокол). В приведенных выше примерах мы видим схемы http, схему mailto, используемую почтовыми клиентами, и две схемы urn. Также доступны такие схемы, как ftp для протокола передачи файлов, sip для протокола установления сеанса, который в основном используется для передачи голоса по IP-телефонии, и даже схема tel, предназначенная для идентификации телефонных номеров.

После схемы обращения к ресурсу в URI указывается адрес хоста и номер порта, где расположен необходимый ресурс. По умолчанию HTTP использует 80-й порт, и, если номер порта не указан, подразумевается, что ресурс находится именно на этом порту. Если ресурс расположен на нестандартном порту, например, 8080, то необходимо после адреса хоста через двоеточие прописать нужный порт, к примеру, http://www.bbc.co.uk:8080/weather/.

Стандарт URI включает в себя стандарты *имени ресурса* (URN – Uniform Resource Name) и *унифицированного локатора ресурса* (URL – Uniform Resource Locator).

URL описывает ресурс с точки зрения протокола, который используется для доступа к нему. Например, вышеприведенные идентификаторы http и mailto. *URN* описывает ресурс в некотором пространстве имен, к примеру, уникальный номер книжного издания ISBN.

Традиционно URL и URN рассматривались как формальные подмножества URI-пространства. Однако в настоящее время понятия URI, URL и URN часто используются как синонимы. Основное различие между URN и URL состоит в том, что URL содержит некоторый сетевой адрес ресурса. Это мы можем наблюдать в примерах с http и mailto. В таких адресах указано имя хоста

(Google.com, bbc.co.uk и example.com), которое используется приложением для получения физического адреса сервера, где расположен ресурс. Получение физического адреса происходит при помощи системы доменных имён (DNS – Domain Name System). Однако URN также может быть использован для получения физического сетевого адреса, для этого проводят некоторую предварительную обработку URI запроса перед его непосредственным выполнением. К примеру, вышеприведенный идентификатор urn:jxta. Платформа JXTA преобразовывает уникальный JXTA-идентификатор в сетевой адрес вычислительного узла JXTA.

Схема HTTP является одной из наиболее распространенных в Интернете. Она классифицируется как иерархический URI, поскольку в ней можно выделить несколько отдельных частей. Так, в примере №5, мы наблюдаем три части. Первая – это схема http, которую мы уже упомянули. Вторая часть – это сетевой адрес ресурса (www.bbc.co.uk). Он используется браузером для получения физического IP-адреса сервера, где расположен ресурс, при помощи DNS. Последняя часть – это непосредственный путь к ресурсу, в нашем примере, weather. Путь определяется относительно хоста, найденного браузером при помощи DNS. Другими словами, символьная строка weather ссылается на ресурс, расположенный на хосте BBC.

URI типа HTTP может также содержать строку запроса. Примером такого идентификатора является пункт №6 из приведенного ранее списка. Можно наблюдать URI такого типа, если ввести поисковый запрос на главной странице Google. Начало запроса обозначается вопросительным знаком (?). Далее идут последовательности пар ключ/значение, каждая из которых отделена знаком амперсанд (&). Между ключом и значением ставится знак равенства (=). Рассматриваемый нами пример URI в строке запроса содержит две пары ключ/значение:

– client = safari
– q = web

Ключи и их значения распознаются сервером, получающим запрос. В этом примере ключ client указывает на браузер, где формировался запрос. Здесь это браузер Safari от компании Apple. Сервер может выдавать ответ в разных форматах в зависимости от указанного браузера. Ключ q отражает текст поискового запроса, введенного пользователем. В нашем случае это текст web.

URI может также содержать идентификатор необходимого фрагмента, представляющий собой последовательность символов, отделенную от пути к ресурсу решеткой (#).

Идентификатор фрагмента в URI позволяет адресовать определенную часть ресурса. Например, очень большой документ, который структурно поделен на разделы. Использование идентификатора фрагмента позволяет указать браузеру, какую часть документа необходимо отобразить, при этом избавив пользователя от необходимости прокрутки всего документа: http://server.com/documents/bigdocument.html#section3.

Для возможности навигации по HTML-документу соответствующий раздел должен быть помечен тегом со значением section3, уникальным в пределах этого документа.

Иерархический URI, может быть абсолютным или относительным. Последний указывается относительно другого URI, который тоже в свою очередь может быть относительным. На веб-страницах часто можно встретить такие относительные ссылки, например: ../images/fido.jpg.

Как правило, такой URI определяется от места расположения документа, в котором он упоминается. Следовательно, если вышеупомянутая ссылка находится в документе http://server.com/pets/dogs/index.html, то она будет обрабатываться следующим образом: сначала будет определен путь к родительскому элементу страницы index.html, т.е. dogs. Далее для каждого вхождения символов ../ будет получен путь к родительскому элементу абсолютного URI. Как и в Unix-подобных системах, символы ../, содержащиеся в пути, означают подъем на один уровень вверх. В данном случае мы поднимаемся на один уровень вверх к элементу pets. Наконец, к абсолютному URI добавляем относительный, чтобы получить следующий адрес: http://server.com/pets/images/fido.jpg.

Обратите внимание, что относительный URI не может адресовать ресурс, находящийся по другому сетевому адресу. Следовательно, вы не можете задавать URI относительно идентификатора, указывающего на другой хост.

3.2.1 Шаблоны URI

По словам Бернерса-Ли, URI должен быть непрозрачным для пользователя, даже если составляющие его компоненты могут быть легко выделены [6]:

«Идентификатор может быть использован только для обращения к объекту. Не следует смотреть на содержимое URI-строки с целью извлечь дополнительную информацию».

Аргументом в пользу такого подхода является то, что чем больше пользователь понимает и вчитывается в URI, тем более хрупкой становится система, так как пользователь может сформировать подобные URI, которые не существуют в пространстве имен сервера. Данный метод также делает систему более связанной, поскольку появляется возможность совместного использования зна-

чений в URI, которые подаются на сервер. Это в свою очередь раскрывает внутреннюю логику сервера и затрудняет ее изменение без последующего влияния на механизм внешней идентификации.

Однако, помимо непосредственной идентификации ресурсов, различные компоненты URI интуитивно задают отношения между частями идентификатора. Например, путь к ресурсу в URI представляет собой последовательность символов, разделенных косой чертой. Порядок расположения компонентов пути в URI по структуре напоминает граф или дерево, подобно иерархии файлов и папок на локальном компьютере. Этот порядок может быть использован для логической группировки ресурсов. Например, URI http://server.com/media/photos и http://server.com/media/music показывают, что раздел media содержит два подмножества: photos и music. URI сам по себе не дает дополнительных сведений о внутренней работе или организации ресурсов на стороне сервера. Тогда как сервер, использующий URI, предоставляет пользователю возможность строить логические связи между существующими ресурсами. Такое устройство подобно супермаркету, где однотипные товары выложены таким образом, что поблизости с чайным прилавком можно найти и кофе. В ином случае, посетитель может растеряться.

Запрос в URI также задает некоторый алгоритм, который будет выполняться на сервере в ответ на конкретные значения, введенные в поле запроса, что не следует путать с вызовом методов на сервере, которые принимают определенные параметры. Возвращаясь к примеру с супермаркетом, это как попросить консультанта показать, где представлен кофе, и натуральный, и крепкий одновременно. Продавец может показать отдельно натуральный кофе, отдельно крепкий, но такой подход может стать обременительным, если представлен очень широкий ассортимент.

В последнее время специалисты полагают, что такая логическая организация URI полезна, особенно в связи с широким распространением веб-сервисов. Частью этого движения является создание спецификации шаблона URI, которая в настоящее время находится на стадии разработки в Инженерном совете Интернет (IETF Network Working Group).

Шаблон URI – это идентификатор с внедренными в него переменными. Переменная заключена в фигурные скобки (фигурные скобки не допускаются в URI, поэтому такой идентификатор не может быть спутан с действительным URI). Чтобы шаблон URI преобразовать в стандартный идентификатор, необходимо заменить переменные их фактическими символьными значениями, характерными для конкретной ситуации. Например, шаблон

http://www.server.com/users/{userId} будет заменен на http://www.server.com/users/1234, если значение идентификатора конкретного пользователя равно 1234. Шаблоны URI особенно подходят для машинной обработки, поскольку они позволяют веб-сервису описать способ доступа к ресурсам в абстрактных терминах. Личные страницы каждого из пользователей вымышленной службы на www.server.com, одинаковы и описаны службой только один раз до тех пор, пока переменная в URI не будет заменена на идентификатор конкретного пользователя. Это в свою очередь позволяет службам публиковать свои интерфейсы в общем, машиночитаемом виде, чего зачастую не хватает в Вебе.

URI оказался невероятно универсальным. Эта универсальность обеспечивает простой и достаточно гибкий синтаксис для именования любых веб-объектов. URI со схемой HTTP сегодня используется для именования практически бесконечного количества ресурсов. Идентификаторы URI не только связывают разрозненные веб-элементы, позволяя ссылаться на ресурсы, географически расположенные в разных концах Земли, но и прочно вошли в нашу жизнь и широко используются вне Интернета: в беседах, на обратной стороне конвертов, в газетах, на рекламных щитах и автомобилях, в поездах и самолетах.

Перенос понятия URI из цифрового мира в реальный, естественно, повлиял на то, как мы воспринимаем и понимаем Веб с точки зрения его архитектуры. Например, Веб никогда не был нацелен на надежное, понятное обнаружение ресурса. В целом, обнаружение ресурса либо излишне завязано на Веб, например, осуществляется через поисковую систему, или услуги, такие как Technorati (поисковая машина для блогов), либо заключается в случайном переходе по ссылкам.

Тем не менее, тот факт, что URI встречается практически повсеместно, означает, что процесс обнаружения ресурсов может стать настолько децентрализованным, что выходит за пределы Интернета. Обедая в кафе, вы можете натолкнуться на рекламную брошюру печи для пиццы, на которой будет ссылка на официальный сайт производителя. Однако вы никогда не встретите ссылку на объект CORBA, или даже WSDL-документ веб-сервиса, в аналогичных обстоятельствах.

3.3 Общее представление ресурсов

Бернерс-Ли выдвинул и описал принцип наименьшей силы (the Principle of Least Power), который гласит, что *для выражения чего-либо необходимо использовать наименее мощный язык из доступных*. Этот принцип заключается

в следующем: чем сложнее выражение, тем меньше шансов, что оно будет понятным другим людям, и они смогут преобразовать его в язык или представление, наиболее подходящее для их нужд. При совместном использовании представлений различными организациями и сообществами, важно, чтобы формат этого представления был наименее централизованным и эксклюзивным.

Язык гипертекстовой разметки HTML обеспечивает очень низкий порог вхождения в описание ресурсов. Он в первую очередь нацелен на способ представления ресурса в человеческом восприятии, а не на описание возможностей или внутренней структуры ресурса. Это очень отличается от обмена сериализованными Java-объектами, например, в Jini, где представление содержит информацию о возможностях и внутренней структуре ресурса. Обратная сторона медали состоит в том, что только клиенты, знающие внутреннюю структуру и представление ресурса, могут восстановить его. С другой стороны, структура HTML не привязана к структуре ресурса – они ортогональны. Автор представления вынужден адаптировать свою концепцию ресурса (автомобиля, ходатайства, собаки) к ограниченному словарю HTML. Пусть полученное представление не будет совершенным, но, по крайней мере, все желающие, будь то автомеханик, политик или любитель собак, смогут получить к нему доступ. И в зависимости от сферы их деятельности, они могут извлечь из представления необходимую информацию, чтобы преобразовать ее в свой собственный формат, определяющий внутреннюю структуру ресурса. HTML действует как метаязык типа «многие-ко-одному» для преобразования ресурсов. В результате, он накладывает массу ограничений.

Язык гипертекстовой разметки HTML, в отличие от любого другого языка разметки, превосходит традиционное понятие линейного, плоского текста, содержание которого определено автором и неизменяемо читателем. Напротив, HTML предоставляет пользователю множество вариантов.

3.4 Протокол передачи гипертекста

Протокол передачи гипертекста (HTTP – Hypertext Transfer Protocol) стал «де-факто» способом обмена ресурсами в Интернете. На самом деле, он настолько полезный, гибкий и повсеместно распространенный, что при определенных обстоятельствах используется в других системах, таких как JXTA и Gnutella. Кроме того, веб-сервисы на основе SOAP почти всегда используют туннелирование HTTP для SOAP-конвертов.

HTTP является простым протоколом прикладного уровня. Он поддерживает несколько методов, аналогичных операциям, которые выполняются в базе данных или хранилище на основе ключ/значение. Наряду с именем метода, клиент посылает серверу идентификатор ресурса и несколько заголовков HTTP. Идентификатором ресурса является часть URI, расположенная после имени хоста. Заголовки HTTP представляют собой пары ключ/значение. Наиболее широко используемые методы HTTP.

- **GET** извлекает ресурс по данному URI. Этот запрос не содержит представления ресурса.
- **DELETE** удаляет ресурс по данному URI. Этот запрос не содержит представления ресурса.
- **PUT** обновляет или изменяет ресурс по данному URI. Если требуемого ресурса не существует, сервер может создать новый ресурс по данному URI. Этот запрос содержит представление ресурса.
- **POST** используется по-разному в разных контекстах. В целом, этот метод предназначен для создания нового ресурса, представление которого направляется серверу в запросе. Созданный ресурс логически подчинен тому ресурсу, на URI которого пришел POST-запрос. На практике именно сервер решает, что значит «подчиненный». Запрос содержит представление ресурса.
- **HEAD** получает метаданные о ресурсе. Запрос возвращает все те же данные клиенту, что и GET, за исключением тела ресурса, то есть описание не включено в ответ. Этот запрос не содержит представления ресурса.
- **OPTIONS** возвращает список методов HTTP, которые доступны по данному URI. Этот запрос не содержит представления ресурса.

Итак, как выглядит запрос HTTP? На рис. 4 приведен простой пример запроса HTTP, который может быть отправлен на страницу погоды BBC http://www.bbc.co.uk/weather/.

```
GET / weather / HTTP/1.1
User-Agent: curl/7.16.3
Host: www.bbc.co.uk
Accept: * / *
```

Рис. 4. Пример HTTP-запроса GET

Первая строка запроса определяет метод – GET – путь ресурса, а именно, часть полного URI после адреса хоста, и используемую версию HTTP – в данном случае версия 1.1. Далее следуют заголовки. Заголовок User-Agent указы-

вает серверу, какое программное обеспечение используется для создания запроса. В заголовке Host указан адрес хост-сервера из полного URI. Заголовок Accept определяет тип данных, который готов принять клиент. Это может быть любой из типов многоцелевого расширения почты Интернет (MIME – Multipurpose Internet Mail Extensions): text/html, text/plain или image/jpeg и image/png. Как видим, MIME-тип имеет тип – значение перед косой чертой, и подтип – значение после косой черты. Таким образом, html и plain – подтипы text, так же, как jpeg и png являются подтипами типа image. В нашем примере, звездочка (*) используется, чтобы обозначить любой тип. Следовательно, клиент будет рад принять ответ любого типа и любого подтипа.

```
HTTP/1.1 200 OK
Date: Ср, 26 Mar 2008 16:03:01 GMT
Server: Apache/2.0.59 (Unix)
Content-Length: 12345
Content-Type: text/html
```

Рис. 5. Пример ответа на GET-запрос

На рис. 5 представлен пример ответа на указанный ранее GET-запрос. В первой строке указана используемая версия HTTP, затем код статуса. Код ответа 200 означает, что все прошло удачно и запрашиваемый ресурс содержится в сообщении. Другими распространенными кодами ответа являются 201 Created, если ресурс успешно создан на сервере, 404 Not Found, если сервер не может отобразить ресурс по заданному пути, и 403 Forbidden, если клиент не имеет прав для доступа к ресурсу. Всего имеется 40 стандартных кодов ответа, покрывающих все возможные ситуации. В приведенном выше примере сервер возвращает заголовки, описывающие дату и время выполнения запроса, серверное программное обеспечение, длину возвращаемого клиенту представления в байтах и MIME-тип представления. После заголовков следуют фактические данные, если таковые имеются.

Методы HTTP и условия, в которых они используются, часто рассматриваются с точки зрения двух свойств – *безопасность* и *идемпотентность* (метод называется идемпотентным, если при множественном вызове возвращает тот же результат, что и при единичном вызове).

Безопасная операция не производит побочного эффекта. С точки зрения пользователя, побочный эффект – это изменение состояния ресурса, с которым происходит взаимодействие. В соответствии с соглашением, методы GET, HEAD и OPTION считаются безопасными, и поэтому не должны применяться в тех случаях, когда их использование может вызвать изменение состояния на сервере. Даже если такое изменение происходит, клиент, делающий запрос, не

несет ответственности за побочный эффект. Методы PUT, POST и DELETE, как правило, не безопасны. Они должны использоваться в ситуациях, сопровождающихся изменением состояния ресурса на сервере. Когда такие методы используются, безопасность подразумевает негласный контракт между сервером и клиентом. Выполнение безопасного метода, например, GET, не подразумевает наличие контракта; сервер не ожидает от клиента последующих запросов и на сервере не произойдет никаких изменений, за которые ответственен клиент. Поэтому, если сервер использует безопасный метод в ситуации, когда происходит побочный эффект, подразумевается, что он нарушает договор с клиентом.

Поддержание этого контракта помогает сохранить Веб слабосвязанным, нарушение же может привести к нежелательным результатам. Например, обычно после регистрации пользователь получает письмо, содержащее ссылку для подтверждения своей подписки. Он переходит по этому адресу, и браузер выполняет GET запрос на сервер. После возвращения контента браузеру, пользователь видит страницу с поздравлениями о вступлении в список рассылки. Однако, клиент посылал безопасный GET запрос, не вступая в договорные отношения с сервером. В таком случае на GET-запрос сервер должен вернуть страницу с кнопкой «Нажмите здесь, чтобы подтвердить подписку». И уже при нажатии этой кнопки, будет сформирован POST-запрос к серверу, который добавляет пользователя в базу данных списка рассылки.

Как уже было сказано выше, *идемпотентный метод* – это такой метод, который при множественном вызове возвращает результат, как будто был вызван единожды. Этим свойством, как правило, обладают методы GET, HEAD, OPTIONS, PUT и DELETE. Безопасные методы по своему определению идемпотентны. Свойство идемпотентности особенно полезно в распределенном контексте, поскольку позволяет повторять запрос при возникновении ошибки во время выполнения. Например, в случае отказа сервера клиент остается без ответа. Если запрос ни идемпотентный, ни безопасный, то клиенту предстоит принять серьезное решение о том, стоит ли повторять запрос, потому что повторение может вызвать нежелательный побочный эффект.

URI, HTTP и HTML составляют основу веб-технологий. Однако не существует программного обеспечения, которое бы понимало и объединяло их. На стороне сервера, требуется веб-сервер, который может в ответ на HTTP-запросы обслуживать контент ресурсов. На стороне клиента, необходимо программное обеспечение, такое как браузер, для отображения HTML и предоставления возможности пользователю переходить по ссылкам.

3.4.1 HTTP и безопасность

Безопасные веб-соединения снабжены разнообразными методами защиты в зависимости от их потребностей. HTTP поддерживает аутентификацию пользователя непосредственно через определенные заголовки и коды ответов. В типичном сценарии браузер переходит по ссылке на конкретный ресурс. Если ресурс ограничен правами пользователей, то сервер возвращает код состояния 401 Unauthorized. Вместе с кодом ответа, сервер посылает заголовок WWW-Authenticate, где указывается ожидаемый тип и название аутентификации. Существует два типа аутентификации – базовая и дайджест. Для аутентификации обоих типов необходимы имя пользователя и пароль.

В базовом типе аутентификации имя и пароль кодируются при помощи кодировки Base64. Base64 – это система кодирования, предназначенная для представления бинарных объектов в тексте письма. Эта кодировка отнюдь не является безопасной – с помощью простого алгоритма можно преобразовать закодированную строку обратно в обычный текст.

Дайджест-аутентификация использует хэш-функции для шифрования учетных данных. С помощью хэш-функции получают необратимый ключ фиксированной длины. Распространенным примером такой функции является MD5. Имя аутентификации представляет собой защищенную область веб-сайта, как правило, относящуюся к идентификатору домена сервера.

Когда браузер получает код состояния 401, он предоставляет пользователю поля для ввода логина и пароля. Когда пользователь заполняет эту информацию, браузер ретранслирует первоначальный запрос, на этот раз, добавляя заголовок Authenticate, содержащий тип аутентификации и учетные данные, закодированные с помощью либо Base64 для базовой аутентификации, либо хэш-функции, в случае использования дайджест-аутентификации. Если сервер принимает учетные данные, он предоставляет право доступа, и браузер получает содержимое защищенной страницы.

Механизм авторизации HTTP имеет ряд ограничений. Во-первых, при базовой аутентификации учетные данные отправляются в виде обычного текста, который может быть перехвачен третьими лицами и легко декодирован. Во-вторых, даже если для аутентификации используется безопасная хэш-функция, все последующие обмены данными производятся в открытом виде. Когда передаются такие конфиденциальные данные, как номера кредитной карты, это недопустимо. В таких ситуациях веб-серверы обычно используют уровень защищенных сокетов (*SSL – Secure Socket Layer*) и/или протокол безопасности транспортного уровня (*TLS – Transport Layer Security*), являющийся его преем-

ником. Эти протоколы являются «прослойкой» между протоколами TCP и HTTP, располагаясь на 6-м уровне (уровне представления) в рамках модели OSI. Ресурс, который требует безопасного соединения, использует протокол HTTPS в схеме URI, для того, чтобы указать, что он защищен.

Зачастую сервер с поддержкой TLS имеет пару открытый/закрытый ключ и цифровой сертификат, подтвержденный удостоверяющим центром (CA – Certification Authority). Этот сертификат включает в себя открытый ключ с подписью CA, которая гарантирует, что это на самом деле сервер. Когда клиент запрашивает ресурс через защищенное соединение, он посылает серверу ряд параметров, включающих случайное число, используемое в дальнейшем для генерации ключа сеанса, а также версию TLS и поддерживаемые алгоритмы шифрования (cyphers) и хэш-функции. Сервер выбирает наиболее устойчивые алгоритмы шифрования и хэш-функции, доступные и клиенту, и серверу, и сообщает клиенту, какой алгоритм будет использован во время последующего обмена сообщениями. Наряду с этой информацией, сервер также отправляет клиенту случайное число, а затем и сертификат.

На этом этапе клиент может обратиться в удостоверяющий центр для проверки подлинности сертификата сервера. Браузеры обычно обладают списком известных центров сертификации, где они могут проверить подлинность подписи сертификата. Если подпись не известна браузеру, как в случае самоподписанного сертификата, он предложит пользователю принять или отклонить данный сертификат.

Если сертификат считается надежным, то клиент генерирует еще одно случайное число, известное как PreMasterKey, шифрует его с помощью открытого ключа сервера, и отправляет на сервер. Только сервер может расшифровать это число, используя свой закрытый ключ. Затем обе стороны генерируют ключ сеанса на основе случайных чисел, первоначально посланных друг другу, и PreMasterKey. Ключ сеанса используется для шифрования и дешифрования всех сообщений обмена в пределах открытой сессии.

TLS также поддерживает взаимную аутентификацию, при которой клиент также должен обладать парой открытый/закрытый ключ, хотя этот способ менее распространен в Интернете. В таком случае, сервер запрашивает сертификат от клиента. После отправки PreMasterKey на сервер, клиент подписывает сообщения, которыми они обменялись во время процедуры handshake (рукопожатия). Сервер может проверить эту подпись с помощью открытого ключа клиента, убедившись, что клиент является тем, за кого он себя выдает.

3.5 Заключение

Методы именования, предоставления и передачи ресурсов, предоставляемые Веб, на сегодняшний день, применяются множеством различных распределенных вычислительных систем. Такой подход позволят разработчикам РВС реализовывать решения, совместимые с существующими стандартами и методами взаимодействия удаленных вычислительных систем.

4. Модель «Клиент-Сервер»

Согласно парадигме *клиент-серверной архитектуры* несколько клиентов и несколько серверов совместно с промежуточным программным обеспечением и средой взаимодействия образуют единую систему, обеспечивающую распределенные вычисления, анализ и представление данных. Использование клиент-серверного подхода позволило пользователю персонального компьютера получить доступ к различным ресурсам удаленных серверов, таких как базы данных, файлы, принтеры, процессорное время и др.

В базовой модели клиент-сервер все процессы в распределенных системах делятся на две возможно перекрывающиеся группы. Процессы, реализующие некоторый сервис, например, сервис файловой системы или базы данных, называются *серверами*. Процессы, запрашивающие сервисы у серверов путем посылки запроса и последующего ожидания ответа от сервера, называются *клиентами*.

Если базовая сеть так же надежна, как локальные сети, взаимодействие между клиентом и сервером может быть реализовано посредством простого протокола, не требующего установления соединения (например, протокол UDP). В этом случае клиент, запрашивая сервис, облекает свой запрос в форму сообщения с указанием в нем сервиса, которым он желает воспользоваться, и необходимых для этого исходных данных. Затем сообщение посылается серверу. Последний, в свою очередь, постоянно ожидает входящего сообщения, получив его, обрабатывает, упаковывает результат обработки в ответное сообщение и отправляет его клиенту.

Использование не требующего соединения протокола дает существенный выигрыш в эффективности. До тех пор пока сообщения не начнут пропадать или повреждаться, можно вполне успешно применять протокол типа запрос-ответ. К сожалению, создать протокол, устойчивый к случайным сбоям связи, – нетривиальная задача. Все, что мы можем сделать – это дать клиенту возможность повторно послать запрос, на который не был получен ответ. Проблема, однако, состоит в том, что клиент не может определить, действительно ли первоначальное сообщение с запросом было потеряно или ошибка произошла при передаче ответа. Если потерялся ответ, повторная посылка запроса может привести к повторному выполнению операции. Если операция представляла собой что-то вроде «снять 10 000 долларов с моего банковского счета», понятно, что было бы гораздо лучше, если бы вместо повторного выполнения операции вас

просто уведомили о произошедшей ошибке. С другой стороны, если операция была «сообщите мне, сколько денег у меня осталось», запрос прекрасно можно было бы послать повторно. Нетрудно заметить, что у этой проблемы нет единого решения.

В качестве альтернативы во многих системах клиент-сервер используется надежный протокол с установкой соединения (например, протокол TCP). Хотя это решение в связи с его относительно низкой производительностью не слишком хорошо подходит для локальных сетей, оно великолепно работает в глобальных системах, для которых ненадежность является «врожденным» свойством соединений. Так, практически все прикладные протоколы Интернета основаны на надежных соединениях на основе стека TCP/IP. В этом случае всякий раз, когда клиент запрашивает сервис, до посылки запроса серверу он должен установить с ним соединение. Сервер обычно использует для посылки ответного сообщения то же самое соединение, после чего оно разрывается. Проблема состоит в том, что установка и разрыв соединения в смысле затрачиваемого времени и ресурсов относительно дороги, особенно если сообщения с запросом и ответом невелики. Мы обсудим альтернативные решения, в которых управление соединением объединяется с передачей данных, в следующей главе.

4.1 Разделение приложений по уровням

Модель клиент-сервер была предметом множества дебатов и споров. Один из главных вопросов состоял в том, как точно разделить клиента и сервера. Рассматривая множество приложений типа клиент-сервер, предназначенных для организации доступа пользователей к базам данных, многие рекомендовали разделять их на три уровня.

- уровень представления (пользовательского интерфейса);
- уровень бизнес-логики (обработки);
- уровень данных.

Уровень пользовательского интерфейса содержит все необходимое для непосредственного общения с пользователем, например, для управления дисплеем. Уровень обработки обычно содержит приложения, а уровень данных – собственно данные, с которыми происходит работа. В следующих пунктах мы обсудим каждый из этих уровней.

4.1.1 Уровень представления

Уровень пользовательского интерфейса обычно реализуется на клиентах. Этот уровень содержит программы, посредством которых пользователь может взаимодействовать с приложением.

Сложность программ, входящих в пользовательский интерфейс, весьма различна. Простейший вариант программы пользовательского интерфейса не содержит ничего, кроме символьного (не графического) дисплея. Такие интерфейсы обычно используются при работе с мейнфреймами. В том случае, когда мейнфрейм контролирует все взаимодействия, включая работу с клавиатурой и монитором, мы вряд ли можем говорить о модели клиент-сервер («однозвенная» архитектура). Однако во многих случаях терминалы пользователей производят некоторую локальную обработку, осуществляя, например, эхопечать вводимых строк или предоставляя интерфейс форм, в котором можно отредактировать введенные данные до их пересылки на главный компьютер. Современные пользовательские интерфейсы значительно более функциональны.

4.1.2 Уровень бизнес-логики

Бизнес-логика – это совокупность правил, принципов и зависимостей поведения объектов предметной области системы. Синонимом данного понятия является *логика предметной области* (анг. Domain Logic).

Бизнес-логика – это реализация предметной области (например, бухгалтерского учета, обучения студентов, методов управления предприятием и др.) в информационной системе. К ней относятся, например, формулы расчёта ежемесячных выплат по ссудам (в финансовой индустрии), автоматизированная отправка сообщений электронной почты руководителю проекта по окончании выполнения частей задания всеми подчиненными (в системах управления проектами), отказ от отеля при отмене рейса авиакомпанией (в туристическом бизнесе) и т. д.

4.1.3 Уровень данных

Уровень данных в модели клиент-сервер содержит программы, которые предоставляют данные обрабатывающим их приложениям. Специфическим свойством этого уровня является *требование сохранности.* Это означает, что когда приложение не работает, данные должны сохраняться в определенном месте в расчете на дальнейшее использование. В простейшем варианте уровень данных реализуется файловой системой, но чаще для его реализации задействуется полномасштабная база данных. В модели клиент-сервер уровень данных обычно находится на стороне сервера.

Кроме простого хранения информации уровень данных обычно также отвечает за *поддержание целостности* данных для различных приложений. Для базы данных поддержание целостности означает, что метаданные, такие как описания таблиц, ограничения и специфические метаданные приложений, также хранятся на этом уровне.

Обычно в деловой среде уровень данных организуется в форме реляционной базы данных. Ключевым здесь является независимость данных. Данные организуются независимо от приложений так, чтобы изменения в организации данных не влияли на приложения, а приложения не оказывали влияния на организацию данных. Использование реляционных баз данных в модели клиент-сервер помогает нам отделить уровень обработки от уровня данных, рассматривая обработку и данные независимо друг от друга.

Однако существует обширный класс приложений, для которых реляционные базы данных не являются наилучшим выбором. Характерной чертой таких приложений является работа со сложными типами данных, которые проще моделировать в понятиях объектов, а не отношений. Примеры таких типов данных – от простых наборов прямоугольников и окружностей до проекта самолета в случае систем автоматизированного проектирования. Также и мультимедийным системам значительно проще работать с видео- и аудио потоками, используя специфичные для них операции, чем с моделями этих потоков в виде реляционных таблиц.

В тех случаях, когда операции с данными значительно проще выразить в понятиях работы с объектами, имеет смысл реализовать уровень данных средствами объектно-ориентированных баз данных. Подобные базы данных не только поддерживают организацию сложных данных в форме объектов, но и хранят реализации операций над этими объектами. Таким образом, часть функциональности, приходившейся на уровень обработки, мигрирует в этом случае на уровень данных.

4.2 Типы клиент-серверной архитектуры

Исторически, первым вариантом клиент-серверной архитектуры являлась так называемая «Однозвенная» архитектура, характеризующаяся тем, что клиент не нес никакой функциональной нагрузки, кроме отображения информации, предоставляемой мейнфреймом (сервером). Примером такой архитектуры может являться терминальный доступ к удаленному серверу или удаленный рабочий стол. В этом случае весь объем вычислительной нагрузки приходился на сервер.

Следующим шагом в развитии клиент-серверной архитектуры было появление так называемой двухзвенной архитектуры (рис. 6).

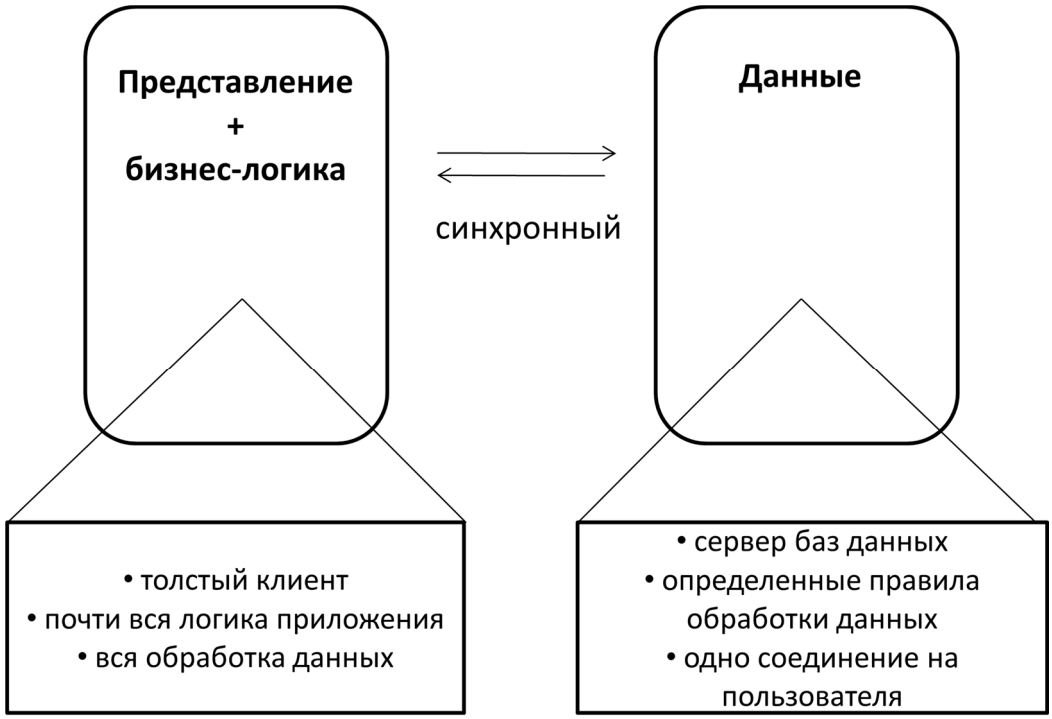


Рис. 6. Двухзвенная клиент-серверная архитектура

Особенностью данного подхода является использование «толстых» клиентов, на которые возлагались основные задачи по отображению информации пользователю и обработке всех данных. Центральный сервер реализовывал лишь функции хранения и предоставления данных. Этот подход являлся наиболее распространенным решением для корпоративных распределенных вычислительных систем вплоть до начала 2000-х годов.

Поскольку большинство логики клиент-серверного приложения находится в клиентской части, клиентская рабочая станция несет ответственность за большую часть обработки. Для оценки разделения объемов работ часто используется соотношение 80/20: сервер базы данных обычно выполняет порядка двадцати процентов работы. Несмотря на это, база данных часто становится узким местом производительности в этих средах.

Двухуровневая клиент-серверная система часто требует, чтобы каждый клиент устанавливал свои собственные соединения с базой данных. Постоянная поддержка такого множества соединений с базой данных стоят дорого, и потребность в ресурсах иногда может привести к перегрузке сервера баз данных и задержки обработки запросов всех пользователей.

Одной из основных причин отказа от двухзвенного клиент-серверного подхода было постоянное увеличение расходов на поддержание логики работы

приложения на рабочих станциях пользователей. Поскольку код приложения реализуется в каждом клиенте, каждое обновление для приложения требует переустановки клиентского программного обеспечения на всех рабочих станциях. В больших средах это приводит к высокой сложности администрирования.

Также, поскольку рабочие станции могут иметь различный набор установленного ПО или, возможно, были приобретены у различных поставщиков оборудования, возникает ряд вопросов по поддержанию работоспособности клиентских частей. Кроме того, при расширении возможностей клиентской части приложения, устаревший парк рабочих станций может стать препятствием для обновления до новой версии системы в связи с ограниченными возможностями старых клиентских машин.

В ответ на затраты и ограничения, связанные с двухуровневой клиент-серверной архитектурой, зародилась концепция многоуровневой клиент-серверной архитектуры (рис. 7).

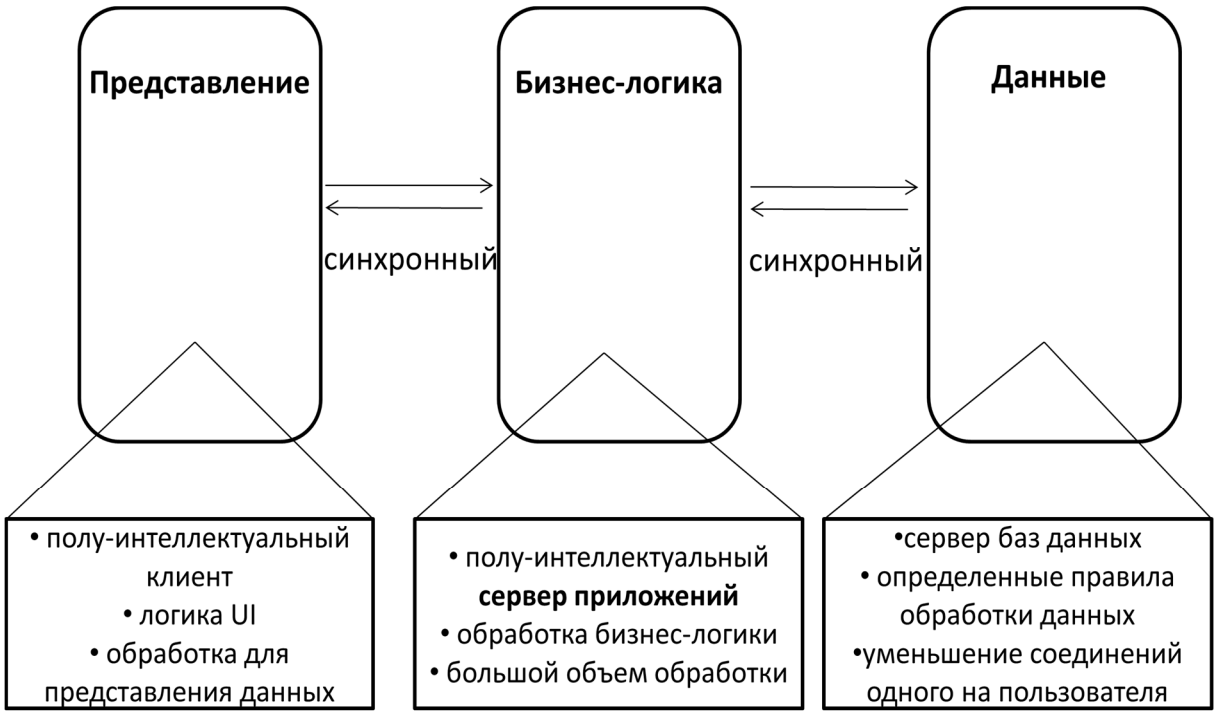


Рис. 7. Обобщенная организация трехзвенной архитектуры

Применение такого подхода позволило распределить уровень бизнес-логики среди нескольких компонентов (часть – на клиенте, часть – на сервере) и уменьшить количество проблем при развертывании системы путем централизации большего количества логики на серверах. Серверные компоненты, перешедшие на выделенные сервера приложений, обеспечили возможность управления пулами соединений с базой данных, облегчая задачу серверу баз данных

посредством значительного уменьшения одновременного количества соединений, так как одно соединение может обеспечить работу нескольким клиентам.

В качестве примера рассмотрим поисковую машину в Интернете. Пользовательский интерфейс поисковой машины очень прост: пользователь вводит строку, состоящую из ключевых слов, и получает список заголовков веб-страниц. Результат формируется из гигантской базы просмотренных и проиндексированных веб-страниц. Ядром поисковой машины является программа, трансформирующая введенную пользователем строку в один или несколько запросов к базе данных. Затем она помещает результаты запроса в список и преобразует этот список в набор HTML-страниц. В рамках модели клиент-сервер часть, которая отвечает за выборку информации, обычно находится на уровне обработки (рис. 8).

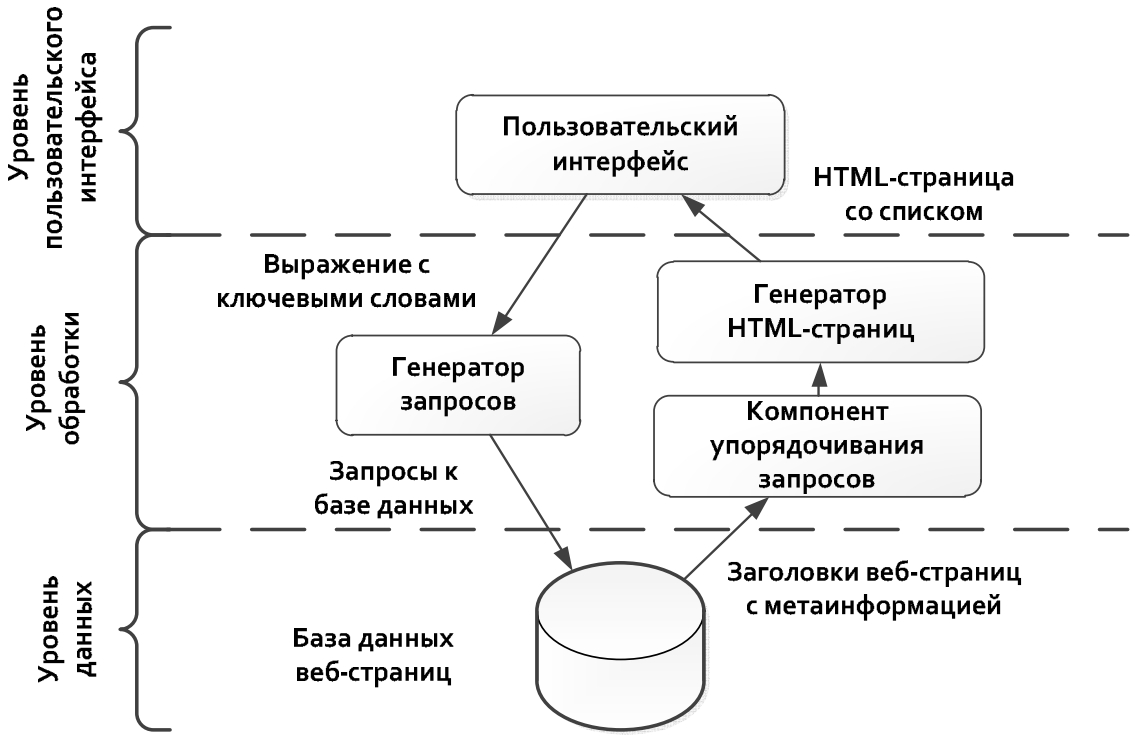


Рис. 8. Обобщенная организация трехуровневой поисковой машины для Интернета

4.2.1 Методы горизонтального распределения

Многозвенные архитектуры клиент-сервер являются прямым продолжением разделения приложений на уровни пользовательского интерфейса, компонентов обработки и данных. Различные звенья взаимодействуют в соответствии с логической организацией приложения. Во множестве бизнес-приложений распределенная обработка эквивалентна организации многозвенной архитектуры приложений клиент-сервер. Мы будем называть такой тип распределения *вертикальным распределением*.

Характеристической особенностью вертикального распределения является то, что оно достигается *размещением логически различных компонентов на разных машинах*. Это понятие связано с концепцией вертикального разбиения, используемой в распределенных реляционных базах данных, где под этим термином понимается разбиение по столбцам таблиц для их хранения на различных машинах.

Однако вертикальное распределение – это лишь один из возможных способов организации приложений клиент-сервер, причем во многих случаях наименее интересный. В современных архитектурах распределение на клиенты и серверы происходит способом, известным как *горизонтальное распределение*. При таком типе распределения клиент или сервер могут содержать физически разделенные части логически однородного модуля, причем работа с каждой из частей может происходить независимо. Это делается для выравнивания загрузки.

В качестве распространенного примера горизонтального распределения рассмотрим веб-сервер, реплицированный на несколько машин локальной сети (рис. 9).

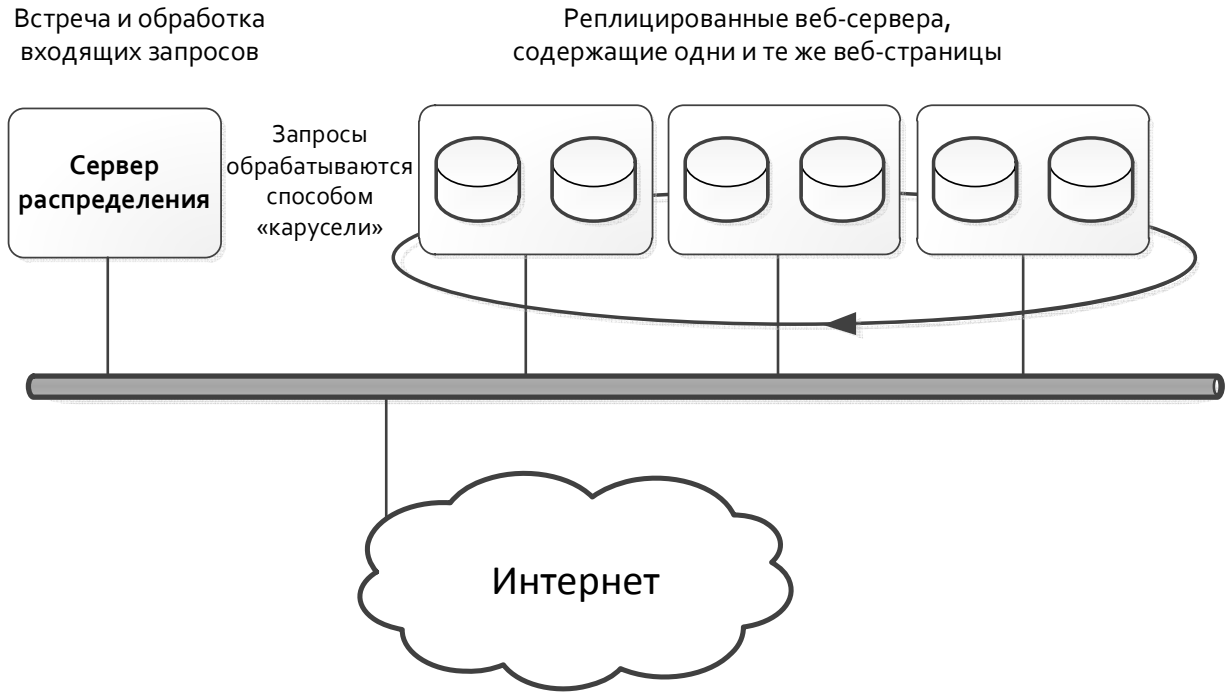


Рис. 9. Пример горизонтального распределения веб-сервера

На каждом из серверов содержится один и тот же набор веб-страниц. Всякий раз, когда одна из веб-страниц обновляется, ее копии незамедлительно рассылаются на все серверы. Сервер, которому будет передан приходящий запрос, выбирается по правилу «карусели». Эта форма горизонтального распределения

весьма успешно используется для выравнивания нагрузки на серверы популярных веб-сайтов.

Таким же образом, хотя и менее очевидно, могут быть распределены и клиенты. Для несложного приложения, предназначенного для коллективной работы, мы можем не иметь сервера вообще. В этом случае мы обычно говорим об одноранговом распределении. Подобное происходит, например, если пользователь хочет связаться с другим пользователем. Оба они должны запустить одно и то же приложение, чтобы начать сеанс. Третий клиент может общаться с одним из них или обоими, для чего ему нужно запустить то же самое приложение.

Самым ярким примером применения горизонтального клиент-серверного распределения могут служить программные системы, созданные на основе концепции облачных вычислений. Более подробно об этом мы поговорим в главе 12 «Облачные вычисления».

5. Объектные распределенные системы

5.1 Вызов удаленных процедур.

Идея вызова удаленных процедур (Remote Procedure Call – RPC) состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства вызова удаленных процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными [2].

Реализация удаленных вызовов существенно сложнее реализации вызовов локальных процедур. Возникает множество проблем: организация передачи данных с адресного пространства одной машины на другую, включая прозрачное использование нижележащей системы связи; обработку экстренного завершения родительского или дочернего удаленного процесса и др.

Кроме того, существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно так же во всех других языках.

Эти и некоторые другие проблемы решает широко распространенная технология RPC, лежащая в основе многих распределенных операционных систем.

5.1.1 Базовые операции RPC

Чтобы понять работу RPC, рассмотрим вначале выполнение вызова локальной процедуры в обычном компьютере, работающем автономно. Чтобы осуществить вызов, вызывающая процедура помещает параметры в стек в обратном порядке. После того, как вызов выполнен, он помещает возвращаемое значение в регистр, перемещает адрес возврата и возвращает управление вызывающей процедуре, которая выбирает параметры из стека, возвращая его в исходное состояние.

Идея, положенная в основу RPC, состоит в том, чтобы сделать вызов удаленной процедуры выглядящим по возможности так же, как и вызов локальной процедуры. Другими словами – вызывающей процедуре не требуется знать, что вызываемая процедура находится на другой машине, и наоборот.

RPC достигает прозрачности следующим путем. Когда вызываемая процедура действительно является удаленной, в библиотеку помещается вместо локальной процедуры другая версия процедуры, называемая клиентским *стабом* (англ. stub – заглушка). Подобно оригинальной процедуре, стаб вызывается с использованием вызывающей последовательности, так же происходит прерывание при обращении к ядру. Только в отличие от оригинальной процедуры он не помещает параметры в регистры и не запрашивает у ядра данные, вместо этого он формирует сообщение для отправки ядру удаленной машины.

5.1.2 Этапы выполнения RPC

Взаимодействие программных компонентов при выполнении удаленного вызова процедуры иллюстрируется рисунком 10.

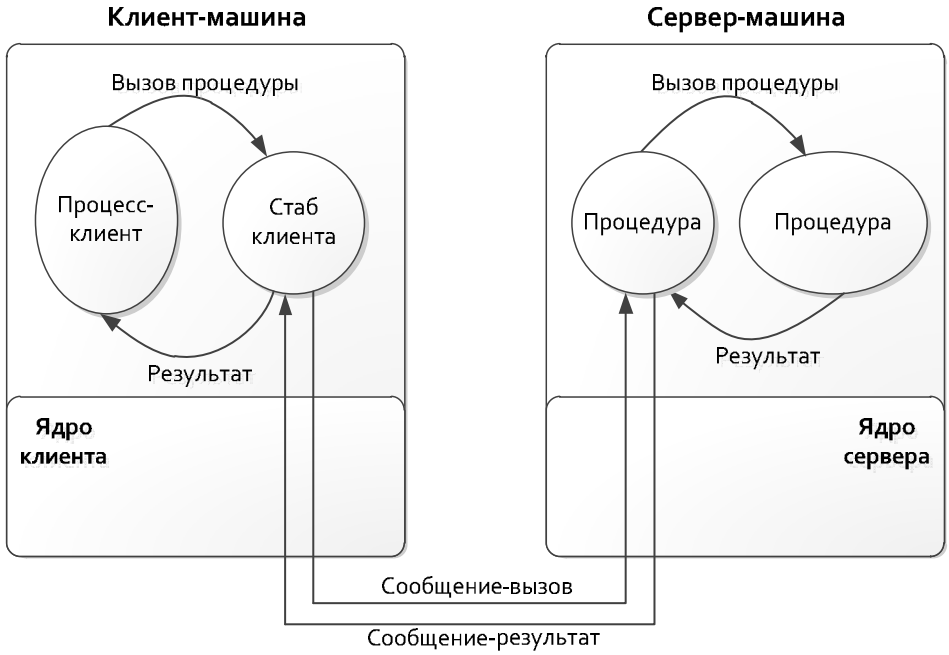


Рис. 10. Порядок удаленного вызова процедуры

После того, как клиентский стаб был вызван программой-клиентом, его первой задачей является заполнение буфера отправляемым сообщением. В некоторых системах клиентский стаб имеет единственный буфер фиксированной длины, заполняемый каждый раз с самого начала при поступлении каждого нового запроса. В других системах буфер сообщения представляет собой пул буферов для отдельных полей сообщения, причем некоторые из этих буферов уже заполнены. Этот метод особенно подходит для тех случаев, когда пакет имеет формат, состоящий из большого числа полей, но значения многих из этих полей не меняются от вызова к вызову.

Затем параметры должны быть преобразованы в соответствующий формат и вставлены в буфер сообщения. К этому моменту сообщение готово к передаче, поэтому выполняется прерывание по вызову ядра.

Рис. 11. Этапы выполнения процедуры RPC

Когда ядро получает управление, оно переключает контексты, сохраняет регистры процессора и карту памяти (дескрипторы страниц), устанавливает новую карту памяти, которая будет использоваться для работы в режиме ядра. Поскольку контексты ядра и пользователя различаются, ядро должно скопировать сообщение в свое собственное адресное пространство, запомнить адрес назначения, после чего передать его сетевому интерфейсу. На этом завершается работа на клиентской стороне. Включается таймер передачи, и ядро может либо выполнять циклический опрос наличия ответа, либо передать управление планировщику, который выберет какой-либо другой процесс на выполнение. В первом случае ускоряется выполнение запроса, но отсутствует мультипрограммирование.

На стороне сервера поступающие биты помещаются принимающей аппаратурой либо во встроенный буфер, либо в оперативную память. Когда вся информация будет получена, генерируется прерывание. Обработчик прерывания проверяет правильность данных пакета и определяет, какому стабу следует их передать. Если ни один из стабов не ожидает этот пакет, обработчик должен либо поместить его в буфер, либо вообще отказаться от него. Если имеется ожидающий стаб, то сообщение копируется ему. Наконец, выполняется переключение контекстов, в результате чего восстанавливаются регистры и карта

памяти, принимая те значения, которые они имели в момент, когда стаб сделал вызов receive.

Теперь начинает работу серверный стаб. Он распаковывает параметры и помещает их соответствующим образом в стек. По завершении работы, выполняется вызов сервера. После выполнения процедуры сервер передает результаты клиенту. Для этого выполняются все описанные выше этапы, только в обратном порядке.

5.2 Организация связи с использованием удаленных объектов

Объектно-ориентированная технология в настоящее время широко применяется при разработке приложений, в том числе и распределенных. Одним из наиболее важных свойств объекта является то, что он скрывает особенности реализации, предоставляя для взаимодействия строго описанный интерфейс. Это позволяет заменять или изменять объекты, оставляя интерфейс неизменным.

Развитие клиент-серверной архитектуры в начале 1990-х годов привело к формированию *объектно-ориентированной концепции* распределенных систем, ориентированной на инкапсуляцию механизма распределенных взаимодействий и уменьшение сложности разработки распределенных приложений посредством методов объектно-ориентированной разработки и *удаленных вызовов методов* объектов. Основными достоинствами данного подхода стали:

- простота разработки распределенных приложений по сравнению с классическим клиент/серверным подходом;
- возможность разработки приложений для гетерогенных вычислительных сред (обеспечивалась применением виртуальных машин, например, Java, и независимым описанием интерфейсов взаимодействующих компонентов);
- возможность отделения интерфейса удаленного объекта от его непосредственной реализации.

Удаленный объект представляет собой некоторые данные, совокупность которых определяет его *состояние*. Это состояние можно изменять путем вызова его *методов*. Если возможен прямой доступ к данным удаленного объекта, это происходит посредством неявного удаленного вызова, необходимого для передачи значения поля данных объекта между процессами. Методы и поля объекта, которые могут использоваться через удаленные вызовы, доступны через некоторый *внешний интерфейс* класса объекта.

В момент, когда клиент начинает использовать удаленный объект, на стороне клиента создается клиентская заглушка, называемая *посредником* (англ. proxy). Посредник реализует тот же интерфейс, что и удаленный объект.

Для передачи параметров по сети используется *сериализация* объектов и данных. *Сериализация* – это перевод состояния объекта в последовательность битов (чаще всего, бинарный или XML-файл), после чего его копия может быть передана в другой процесс. Обратный процесс – *десериализация* – это восстановление состояния объекта из принятой последовательности битов.

Вызывающий процесс использует методы посредника, который сериализует их параметры для передачи по сети, и передает их по сети серверу. Промежуточная среда на стороне сервера десериализует параметры и передает их заглушке на стороне сервера, которую называют *каркасом* (skeleton) или, как и в удаленном вызове процедур, заглушкой. Каркас связывается с некоторым экземпляром удаленного объекта. Это может быть как вновь созданный, так и существующий экземпляр объекта, в зависимости от применяемой модели использования удаленных объектов, которые будут рассмотрены ниже.

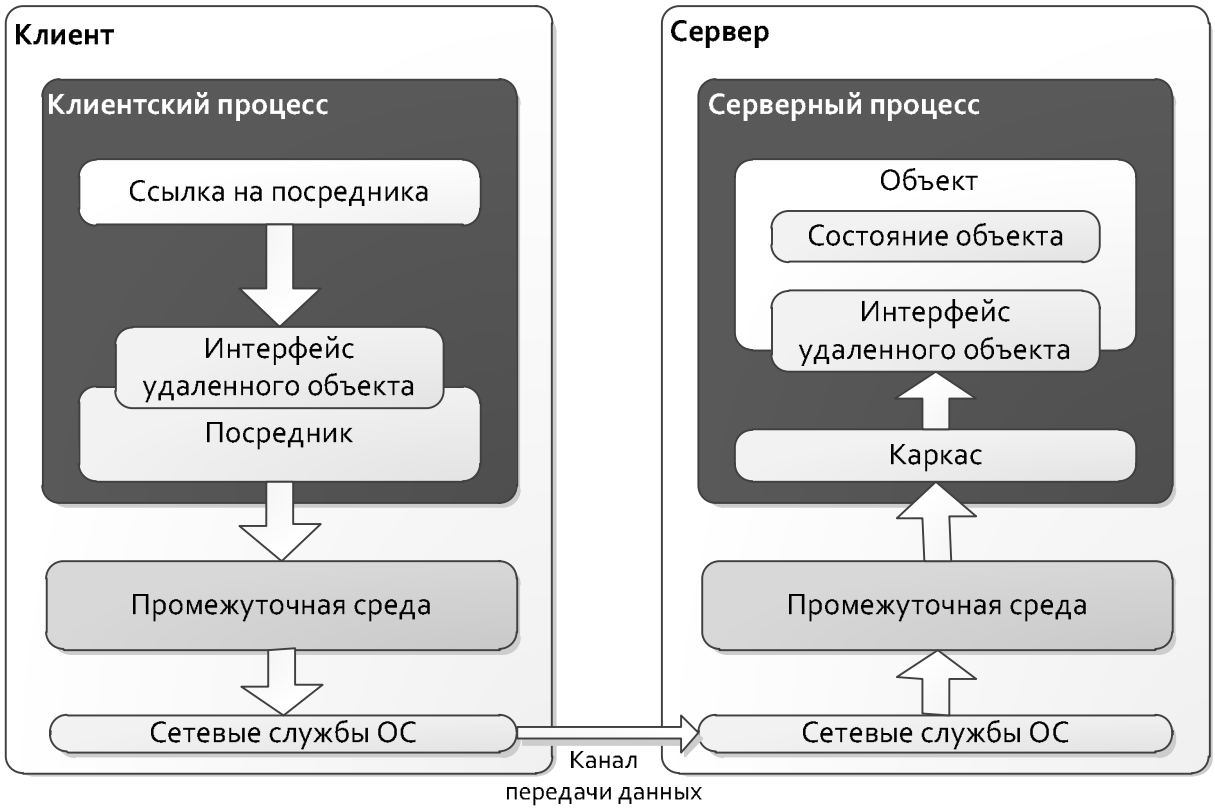


Рис. 12. Использование удаленных объектов

При использовании удаленных объектов проблемными являются вопросы о времени их жизни:

– в какой момент времени создается экземпляр удаленного объекта;

– в течение какого промежутка времени он существует.

Для описания жизненного цикла в системах с удаленными объектами используются два дополнительных понятия:

– активация объекта: процесс перевода созданного объекта в состояние обслуживания удаленного вызова, то есть связывания его с каркасом и посредником.
– деактивация объекта: процесс перевода объекта в неиспользуемое состояние.

Технология *Java RMI (Remote Method Invocation – вызов удаленных методов)* позволяет обеспечить прозрачный доступ к методам удаленных объектов, обеспечивая доставку параметров вызываемого метода, сообщение объекту о необходимости выполнения метода и передачу возвращаемого значения клиенту обратно.

Распределенное приложение, разработанное на базе технологии Java RMI, состоит из двух отдельных программ: клиента и сервера. Серверное приложение создает удаленный объект, публикует ссылки на него и ожидает, когда клиенты произведут вызов метода данного удаленного объекта. Приложение-клиент получает с сервера ссылку на удаленный объект на сервере, после чего может вызывать его методы. Технология RMI обеспечивает механизм, при помощи которого производится обмен информацией между клиентом и сервером. Процесс публикации ссылки на удаленный объект может быть реализован с помощью специального регистра или же посредством передачи удаленной объектной ссылки как части обычной операции.

Достоинствами использования технологии Java RMI для разработки распределенного приложения можно назвать возможность разрабатывать систему целиком основываясь на объектно-ориентированной концепции, не погружаясь в разработку собственных протоколов взаимодействия между распределенными компонентами систем, а также кроссплатформенность, предоставляемую виртуальной машиной Java. К недостаткам данного подхода можно отнести:

– строгую ограниченность данной технологии платформой Java;
– необходимость обработки соединений между распределенными компонентами приложения ограничивает масштабируемость используемого подхода.

5.3 CORBA

5.3.1 Основные понятия CORBA

CORBA (Common Object Request Broker Architecture – общая архитектура брокера объектных запросов) – это технология разработки распределенных приложений, ориентированная на интеграцию распределенных изолированных систем.

В начале 1990-х гг. ночным кошмаром разработчиков было обеспечение общения программ, выполняемых на разных машинах, особенно, если использовались разные аппаратные средства, операционные системы и языки программирования. Программистам приходилось использовать технологию сокетов и самостоятельно реализовывали весь стек протоколов взаимодействия, либо их программы вовсе не взаимодействовали (другие ранние средства промежуточного программного обеспечения были ограничены средой C и Unix и не подходили для использования в неоднородных средах).

Для решения данной проблемы в 1989 г. был создан консорциум OMG (Object Management Group), основной задачей которого стала разработка и продвижение объектно-ориентированных технологий и стандартов. Это некоммерческое объединение, разрабатывающее стандарты для создания корпоративных платформо-независимых приложений.

Концептуальной инфраструктурой, на которой базируются все спецификации OMG, является Object Management Architecture (OMA). В состав OMA входят разнообразные стандартизованные или в настоящий момент стандартизируемые OMG сервисы, программные образцы и шаблоны, язык определения интерфейсов распределенных объектов IDL (Interface Definition Language), стандартизованные или стандартизируемые отображения IDL на языки программирования и, наконец, объектная модель CORBA. Главной особенностью CORBA является использование компонента ORB (Object Resource Broker – брокер ресурсов объектов) для создания экземпляров объектов и вызова их методов. Данный компонент формирует «мост» между приложением и инфраструктурой CORBA.

В 1997 г. консорциум OMG опубликовал спецификацию CORBA 2.0. В ней определялись стандартный протокол и отображение для языка C++, а в 1998 г. было определено отображение для Java. В результате разработчики получили инструментальное средство, позволяющее им относительно легко создавать неоднородные распределенные приложения. CORBA быстро завоевала популяр-

ность, и с использованием этой технологии был создан ряд критически важных приложений.

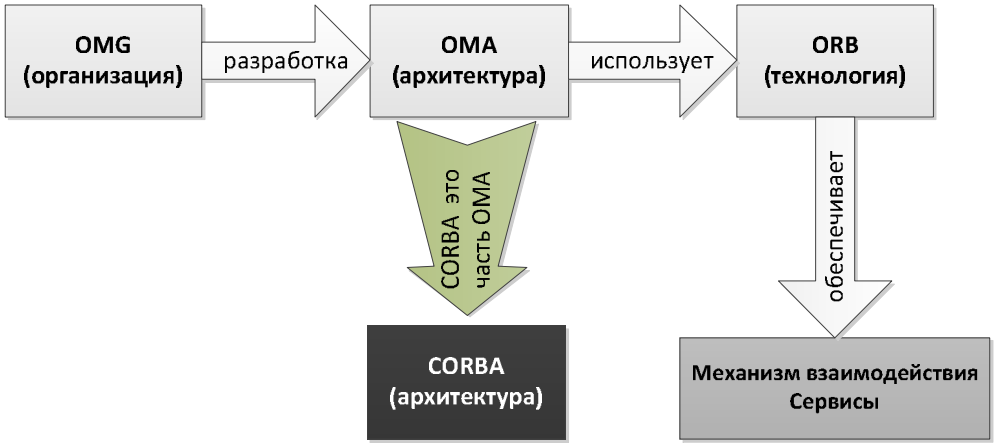


Рис. 13. Основные понятия технологии CORBA

5.3.2 Технология CORBA

Основные компоненты, составляющие архитектуру CORBA, представлены на рисунке 14. Технологический стандарт CORBA определяет язык IDL, применяемый для унифицированного описания интерфейсов распределенных объектов, и его отображения на языки Ada, C, C++, Java, Python, COBOL, Lisp, PL/1 и Smalltalk. Для преобразования описания интерфейса на языке IDL на требуемый язык программирования используется специальный компилятор. В дальнейшем построенный с его помощью программный код может быть преобразован любым стандартным компилятором в исполняемый код.

Главной особенностью CORBA является использование компонента *ORB* (Object Resource Broker – брокер ресурсов объектов) для создания экземпляров объектов и вызова их методов. Данный компонент формирует «мост» между приложением и инфраструктурой CORBA. ORB поддерживает удаленное взаимодействие с другими ORB, а также обеспечивает управление удаленными объектами, включая учет количества ссылок и времени жизни объекта. Для обеспечения взаимодействия между ORB используется протокол *GIOP* (General Inter-ORB Protocol – общий протокол для коммуникации между ORB) [42]. Наиболее распространенной реализацией данного протокола является протокол IIOP (Internet Inter-ORB Protocol – протокол взаимодействия ORB в сети интернет), обеспечивающий отображение сообщений GIOP на стек протоколов TCP/IP.

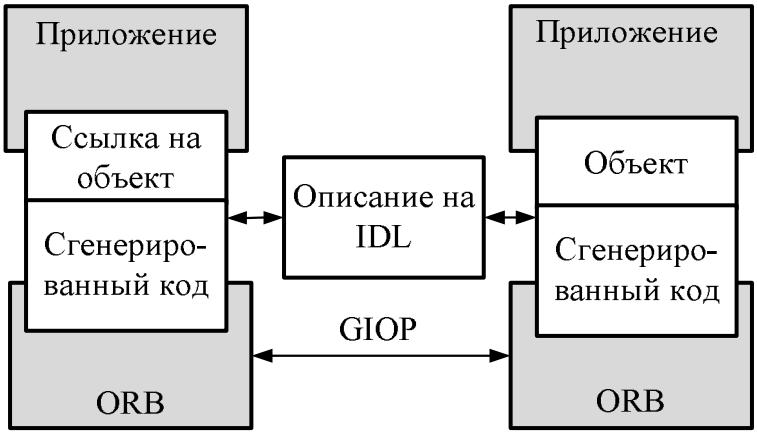


Рис. 14. Ядро архитектуры CORBA

Изначально, технология CORBA ориентирована на предоставление готовой проблемно-ориентированной инфраструктуры для создания РВС в рамках определенной проблемной области. Для этого, в состав CORBA включают набор *стандартных объектных сервисов* и *общих средств*. Спецификация CORBA предусматривает также ряд стандартизованных сервисов (CORBA Services) и горизонтальных и вертикальных Общих Средств (Common Facilities). Сервисы представляют собой обычные CORBA-объекты со стандартизованными (и написанными на IDL) интерфейсами. К таким сервисам относится, например, сервис имен NameService, сервис сообщений, позволяющий CORBA-объектам обмениваться сообщениями, сервис транзакций, позволяющий CORBA-объектам организовывать транзакции. В реальной системе не обязательно должны присутствовать все сервисы, их набор зависит от требуемой функциональности. На сегодня разработано 14 объектных сервисов.

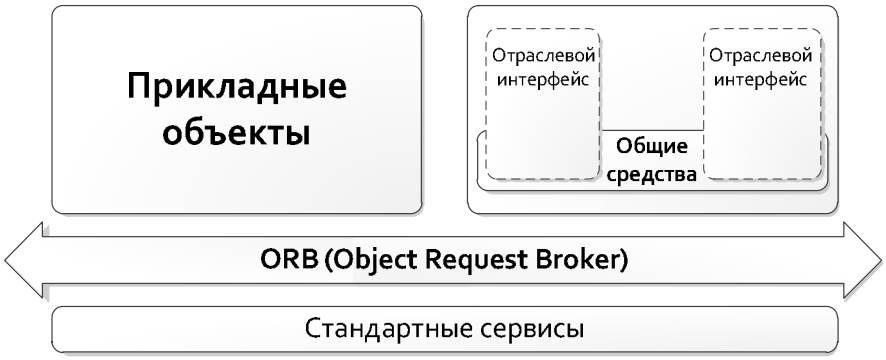


Рис. 15 Схема взаимодействия объектов ОМА

Между объектными сервисами и общими средствами CORBA нет четкой границы. Последние тоже представляют собой CORBA-объекты со стандартизованными интерфейсами. Общие средства делятся на горизонтальные (общие для всех прикладных областей) и вертикальные (для конкретной прикладной

области). Например, разработаны общие средства для медицинских организаций, для ряда производственных сфер и т.п.

5.3.3 Разработка на основе CORBA

Процесс разработки приложения с использованием технологии CORBA состоит из следующих 4-х этапов:

1. Определение интерфейса на IDL.
2. Обработка IDL для создания кода заглушки и скелетона.
3. Создание кода реализации объекта (сервер).
4. Создание кода использования данного объекта (клиент).

Язык определения IDL позволяет независимо от используемого языка программирования создать универсальное описание интерфейса будущей системы.

Созданный на IDL код должен специальным компилятором преобразовываться в код интерфейса объекта на требуемом языке программирования. После чего, на клиенте автоматически генерируется заглушка, преобразующая вызовы методы данного интерфейса в обращения к ORB. На сервере программист на основе сгенерированного интерфейса создает собственную реализацию данного класса. Скелетон автоматизирует получение и обработку удаленного вызова методов, поступающих через ORB.

```
// Модуль системы ценовых предложений
module QuoteSystem
{
  // Структура данных цены
  struct Quote
  {
    string value;
  }

  // Интерфейс сервера ценовых предложений
  interface QuoteServer
  {
    // Атрибут определяющий фондовую биржу
    string exchange;
    // Исключение «Неизвестный идентификатор»
    exception UnknownSymbolException { string message; };
    // Поиск предложения по идентификатору
    Quote getQuote (in string symbol)
                          raises (UnknownSymbolException);
  };
};
```

Рис. 16. Пример описания интерфейса на языке IDL

По сравнению с классическим клиент-серверным подходом, использование технологии CORBA для разработки распределенных приложений имеет следующие преимущества:

- использование IDL для описания интерфейсов позволяет разрабатывать программные компоненты независимо от языка программирования и базовой операционной системы;
- поддержка богатой инфраструктуры распределенных объектов;
- прозрачность вызова удаленных объектов.

Однако программные решения на базе технологии CORBA редко выходят за рамки отдельных предприятий. Разработка крупномасштабных межорганизационных систем на базе технологии CORBA сопряжена со следующими трудностями:

- плохая совместимость различных реализаций технологии CORBA от различных поставщиков;
- проблемы взаимодействия узлов CORBA через Интернет;
- несогласованность многих архитектурных решений CORBA и отсутствие компонентной модели, которая могла бы значительно упростить разработку.

На смену технологии CORBA, пришли стандартизованные протоколы вебсервисов, такие как XML, WSDL, SOAP и др. В настоящее время CORBA используется для реализации узкого круга унаследованных приложений.

6. Агентные технологии

6.1 Понятие программного агента

В соответствии с определением, данным Э. Таненбаумом, *программный агент* – это автономный процесс, способный реагировать на среду исполнения и вызывать изменения в среде исполнения, возможно, в кооперации с пользователями или другими агентами [2].

Агенты могут применяться при решении следующих задач:

- *мобильные вычисления*: миграция агентов может поддерживаться не только между постоянно подсоединенными к сети узлами, но и между мобильными платформами, подключаемыми к постоянной сети на некоторые промежутки времени и возможно по низкоскоростным каналам. Клиент может подсоединяться к постоянной сети на короткий промежуток времени с мобильной платформы, отсылать агента для выполнения задачи и отключаться от сети. Затем клиент подсоединяется к другой точке сети и забирает результаты работы агента. Второй вариант – сервер, на который должен переместиться агент, подсоединяется к сети, а затем отсоединяется. В этом случае агент должен уметь переместиться на такой временно подсоединяемый сервер и вернуться в постоянную сеть;
- *поиск информации:* один человек может быть не в состоянии за короткий срок найти и проанализировать всю необходимую ему информацию. Использование агента позволяет автоматизировать данный процесс. Агент может странствовать по сети и собирать информацию, лучше всего удовлетворяющую поставленной задаче. Поисковые агенты могут содержать сведения о различных информационных источниках (включая тип информации, способ доступа к ней, а также такие характеристики информационного источника, как надежность и точность данных);
- *отбор (обработка) информации*. Из всех данных, приходящих к клиенту, агент может выбирать только те данные, которые могут быть интересны клиенту;
- *мониторинг данных*. Агент может осуществлять извещение пользователя об изменениях в различных источниках данных в реальном времени (например, мобильный агент перемещается на вычислительный узел, на котором расположен источник данных; это эффективнее, чем использовать статического агента, посылающего запросы источнику данных);

– *универсальный доступ к данным*. Агенты могут быть посредниками для работы с различными источниками данных, имеющими механизмы для взаимодействия друг с другом (например, агент создает несколько агентов, каждый из которых работает со своим источником данных).

С. Франклин и А. Грэссер в 1996 году предложили следующее обобщенное определение автономного агента [32]:

Автономный агент – это система, находящаяся внутри окружения и являющаяся его частью, воспринимающая это окружение (его сигналы) и воздействующая на окружение для выполнения собственной программы действий.

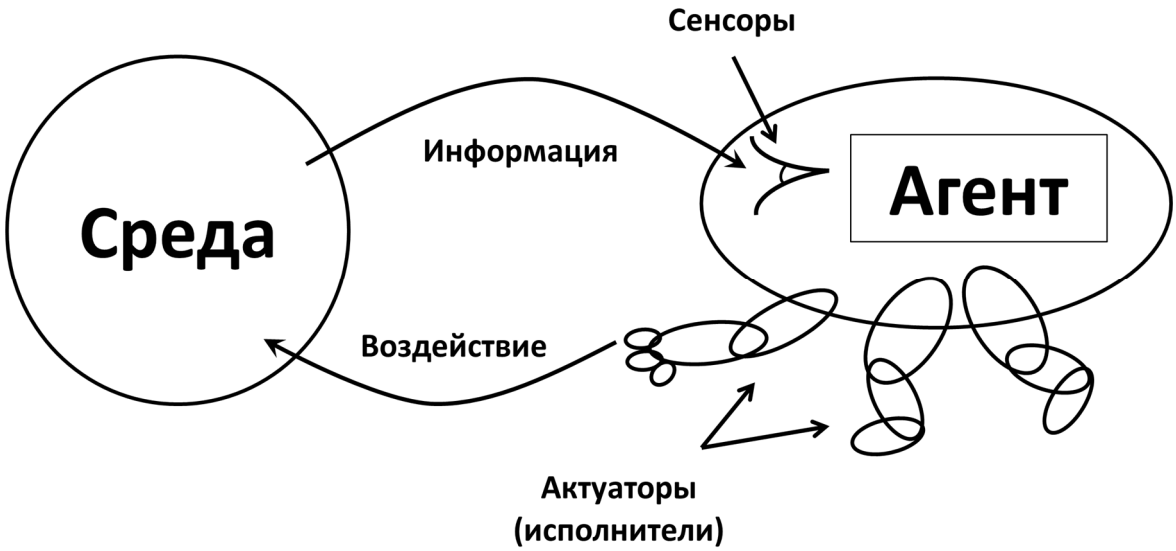


Рис. 17. Автономный агент

Можно выделить следующие основные составляющие автономного агента (рис. 17):

– *Сенсоры*: блоки агента, обеспечивающие получение информации об окружающей среде и других агентах;
– *Актуаторы*: блоки агента, обеспечивающие воздействие на окружающую среду.

При работе простой автономный агент руководствуется стандартным набором правил «Если-то» (рис. 18). Автономный агент должен обладать следующими свойствами:

– реактивность;
– автономность;
– целенаправленность;
– коммуникативность.

Разные авторы не совсем одинаково трактуют перечисленные свойства. Попытаемся объяснить их подробнее [1].

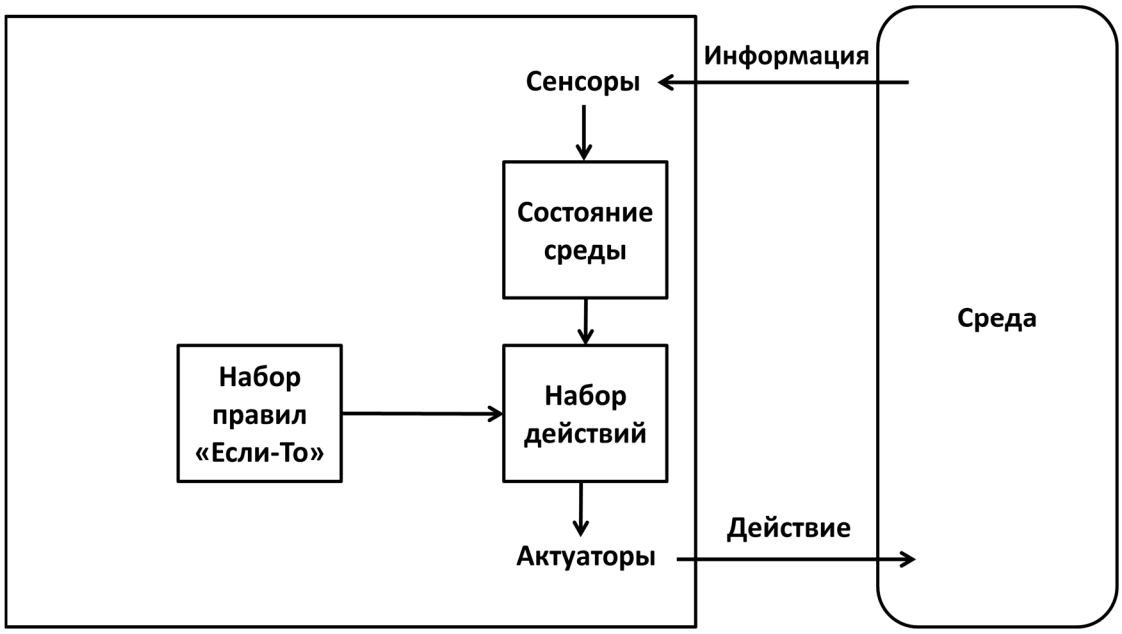


Рис. 18. Структура автономного агента

Свойство реактивности означает, что агент временами отвечает на изменения в окружении. Агент имеет сенсоры, с помощью которых получает информацию от окружения. Сенсоры могут быть самыми различными. Это могут быть микрофоны, воспринимающие акустические сигналы и преобразующие их в электрические, видеокарты захвата изображений, клавиатура компьютера или общая область памяти, в которую окружение помещает данные и из которой программный агент берет данные для вычислений. Не все изменения окружения становятся известными (доступными) сенсорам агента. Это вполне естественно. Ведь и человек не воспринимает звуки сверхвысокой частоты, радиоволны и т.д. Таким образом, окружение не является полностью наблюдаемым для агента. Аналогично, агент воздействует на окружение, путем разнообразных исполнительных механизмов, включая общую память. Разумеется, степень воздействия, как и степень восприятия, является ограниченной. Агент может перевести окружение из некоторого состояния в некоторое другое, но не из любого в любое.

Свойство автономности означает, что агент является самоуправляющимся, сам контролирует свои действия. Программный агент, находящийся на некотором сервере, обладает возможностью «самозапуска». Он не требует от пользователя каких-либо специальных действий по обеспечению его старта (подобно тому, как мы «кликаем» два раза по иконке некоторого файла).

Свойство целенаправленности означает, что у агента имеется определенная цель и его поведение (воздействие на окружение) подчинено этой цели, а не является простым откликом на сигналы из окружения. Иначе говоря, агент является управляющей системой, а не управляемым объектом.

Свойство коммуникативности означает, что агент общается с другими агентами (включая людей), используя для этого некоторый язык. Это не обязательно единый язык для всех агентов. Достаточно, чтобы у пары общающихся агентов был общий язык. Язык может быть сложным как, например, естественный язык. Но может быть и примитивным: обмен числами или короткими словами. Если многословные фразы сложного языка несут всю информацию, как правило, в себе, то слова простого языка предполагают «умолчание»: обе стороны диалога «знают», о чем идет речь (как в известном анекдоте о занумерованных анекдотах).

В отдельную категорию интеллектуальных агентов выделяют автономные агенты, обладающие свойством обучаемости. *Свойство обучаемости* означает, что агент может корректировать свое поведение, основываясь на предыдущем опыте. Это не просто накопление в памяти параметров окружения, т.е. использование исторических данных, но сопоставление истории собственных действий с историей их влияния на окружение, и изменение в связи с этим своей программы действий.

Одна из главнейших особенностей агента – это интеллектуальность. *Интеллектуальный агент* владеет определенными знаниями о себе и об окружающей среде, и на основе этих знаний он способен определять свое поведение (рис. 19). Интеллектуальные агенты являются основной сферой интересов агентной технологии. Важна также среда существования агента: это может быть как реальный мир, так и виртуальный, что становится важным в связи с широким распространением сети Интернет. От агентов требуется способность к обучению и даже самообучению [12]. Способность планировать свои действия делит агентов на *регулирующие* и *планирующие*. Если умение планировать не предусмотрено (регулирующий тип), то агент будет постоянно переоценивать ситуацию и возобновлять свое воздействие на окружающую среду. Планирующий агент может запланировать несколько действий на разные промежутки времени. При этом агент может моделировать развитие ситуации, что дает возможность более адекватно реагировать на текущие ситуации. При этом агент должен принимать во внимание не только свои действия и реакцию на них, но и сохранять модели объектов и агентов окружающей среды для прогнозирования их возможных действий и реакций.

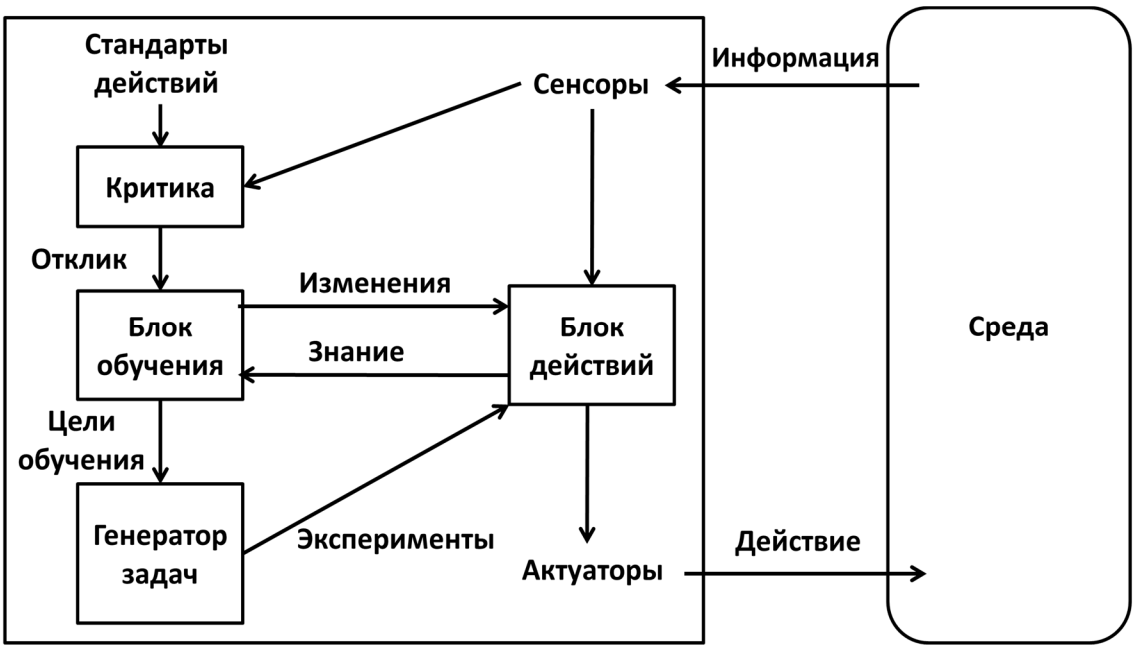


Рис. 19. Структура интеллектуального агента

6.2 Мультиагентные системы

Мультиагентная система (МАС, англ. Multi-agent system) — это система, образованная несколькими взаимодействующими агентами.

Мультиагентные системы могут быть использованы для решения таких проблем, которые сложно или невозможно решить с помощью одного агента или монолитной системы.

В мультиагентной системе необязательно все агенты взаимодействуют (общаются) между собой. В крайнем случае, общения нет вообще. Такие системы назовем дискретными мультиагентными системами. Второй крайний случай – каждый агент общается с каждым. Такую систему можно назвать полносвязной мультиагентной системой [1].

Мультиагентная система, действующая как единый агент, должна характеризоваться и некоторой общей для всех субагентов целью и координацией действий по достижению этой цели. Поскольку встречаются и другие ситуации, когда агенты не связаны столь тесно, то такие системы можно назвать обществами агентов. Отсутствие единой цели, однако, не отрицает возможного группового поведения агентов. Но оно является, скорее, эпизодическим, чем систематическим.

Важным отличием мультиагентной системы от программы или одного агента является то, что входящие в систему программные агенты (по крайней мере, некоторые) не были спроектированы специально для этой системы. Может быть, это – повторно используемые агенты, или агенты, разработанные для

решения более универсальных задач. В этих случаях агенты имеют собственные цели, не совпадающие полностью с целями системы (организации), но совместимые с ними. Тем не менее, они могут быть полезны друг другу для решения стоящих перед ними задач и, поэтому, очень важным для них с этой точки зрения является свойство коммуникативности [1].

6.2.1 Агентные платформы

Агентная платформа – это промежуточное программное обеспечение, поддерживающее создание, интерпретацию, запуск, перемещение и уничтожение агентов. Как «воздух» для человека, агентная платформа обеспечивает агентам среду для жизни и работы.

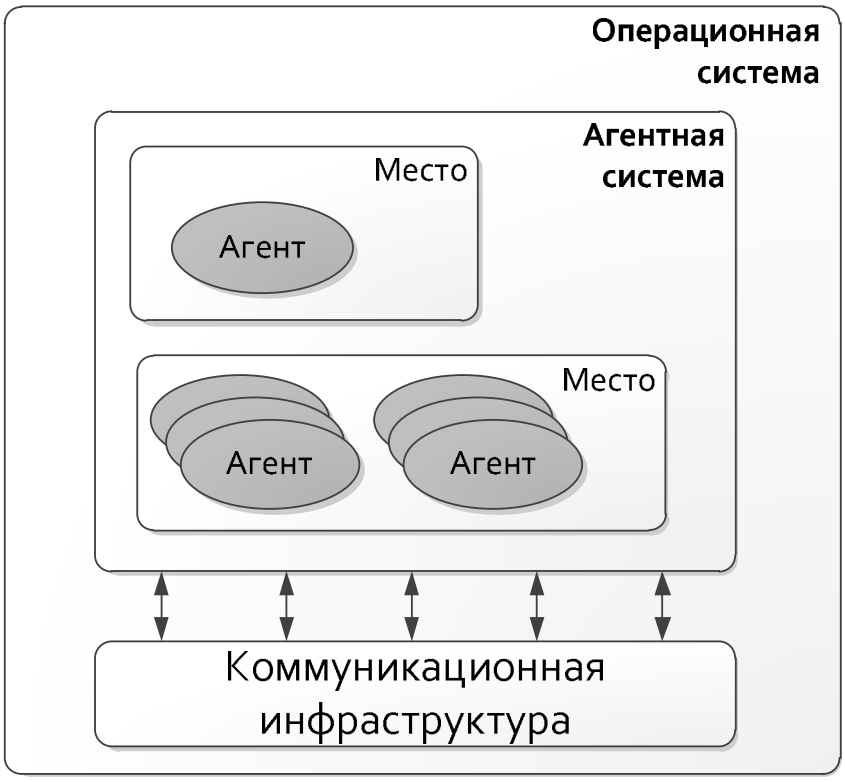


Рис. 20. Агентная система

Агентная система – это приложение, однозначно идентифицируемое именем и адресом, которое обеспечивает жизненный цикл агентов на конкретном узле РВС. На одной машине могут размещаться несколько агентных систем. Тип агентной системы определяется агентной платформой и описывает совокупность параметров агента.

Все общение между агентными системами осуществляется через коммуникационную инфраструктуру.

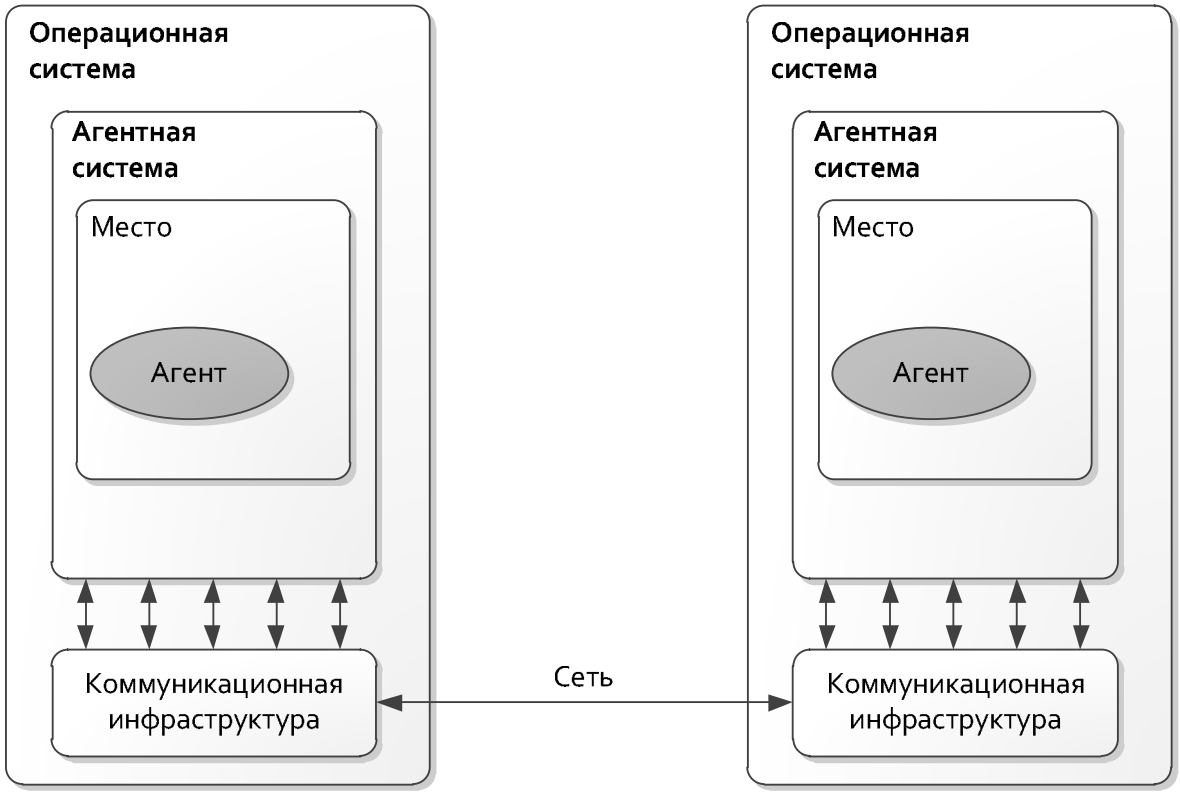


Рис. 21. Связи между агентными системами

Коммуникационная инфраструктура обеспечивает транспортные службы связи (например, RPC), службу имен и службу безопасности для агентных систем.

На сегодняшний день, можно выделить следующие наиболее распространенные агентные платформы.

1. JADE (Java Agent DEvelopment Framework) [39] – Инфраструктура для разработки агентов на языке Java. JADE включает в себя:
   – Динамическую среду, где JADE агенты могут «жить»;
   – Библиотеку классов для разработки агентов;
   – Набор графических инструментов, позволяющий управлять и следить за активностью запущенных агентов.
2. Cougaar (Cognitive Agent Architecture) [12] – это основанная на Java архитектура для построения высокомасштабируемых распределенных агентных приложений.

6.3 Безопасность в системах мобильных агентов

Агентные распределенные вычислительные системы нашли определенную нишу в сфере РВС, но их повсеместному распространению препятствует сам принцип их работы: вряд ли здравомыслящий администратор вычислительной сети позволит странствовать по компьютерам пользователей *автономным ин-*

теллектуальным изменяющимся самообучающимся программным системам, которые могут мигрировать и выполняться на любом вычислительном узле. В связи с этим возникает целый ряд серьезных проблем, связанных с безопасностью агентных платформ.

С одной стороны, агенты могут нести личную информацию для обеспечения собственной работы. Например, агент в системе электронной коммерции может содержать номер кредитной карточки и паспортные данные пользователя для того, чтобы от его имени совершать сделки. Соответственно, необходимо, чтобы среда обеспечивала безопасную для агента среду исполнения.

С другой стороны, сторонний агент может попытаться атаковать базовую среду и извлечь данные или заполучить иные ресурсы. В этом случае агентная платформа должна быть достаточно хорошо защищена и иметь возможность противостоять таким атакам.

Выделяют следующее возможные проблемы безопасности при работе агентных платформ:

- *Агент атакует Хост*: Агент может украсть или модифицировать данные хоста
- *Хост атакует Агента*: Хост может украсть или модифицировать данные Агента, изменить его состояние или код
- *Злонамеренный агент атакует другого агента*;
- *Атака другими элементами*.

Вариант атаки «Агент атакует Хост» является стандартной атакой, в которой код, полученный из ненадежного источника, пытается получить полный доступ к системе или же помешать нормальному выполнению задач, повысив свои полномочия на исполнение. В этом случае от атаки помогают традиционные средства защиты, как то:

- контроль уровня доступа;
- песочницы;
- аутентификация;
- криптография.

Также, существуют специфические для агентных платформ методы обеспечения безопасности, например анализ истории движения агента. В этом случае платформа может узнать об истории передвижения агента из лога и сделать вывод о его качестве на основе информации о том, на каких платформах он побывал до этого.

При варианте атаки «Хост атакует агента» традиционные средства обеспечения не работают, так как хост должен иметь полную информацию о коде агента для его исполнения. Соответственно, традиционные методы обеспечения безопасности оказываются бессильны. В этом случае могут быть применены следующие средства:

- *Мобильная криптография*: функции и данные агента шифруются таким образом, что хост не может разобрать, каким образом функции исполняются и извлечь код. Недостатком данного метода является необходимость поиска схемы шифрования для произвольных функций, а также необходимость переноса ключа кадрирования.
- *Безопасное перемещение*: миграция только на определенные (доверенные) хосты.
- *Использование фиктивных данных*: в базе данных системы, анализирующей работу агентов, хранится набор фиктивных данных, которые не изменяются в ходе нормальной работы агента.
- *Использование доверенного аппаратного обеспечения*: это могут быть смарт-карты, интегрированные микросхемы, и т.п.

7. Компонентные системы

7.1 Основы компонентных программных систем

Компонентно-ориентированный подход к проектированию и реализации программных систем и комплексов является в некотором смысле развитием объектно-ориентированного подхода и практически более пригоден для разработки крупных и распределенных программных систем (например, корпоративных приложений).

С точки зрения компонентно-ориентированного подхода программная система – это набор компонентов с четко определенным интерфейсом. В отличие от других подходов программной инженерии, изменения в систему вносятся путем создания новых компонентов или изменения старых, а не путем рефакторинга существующего кода.

Программный компонент – это автономный элемент программного обеспечения, предназначенный для многократного использования, который может распространяться для использования в других программах в виде скомпилированного кода. Подключение к программным компонентам осуществляется с помощью открытых интерфейсов, а взаимодействие с программной средой осуществляется по событиям, причём в программе, использующей компонент, можно назначать обработчики событий, на которые умеет реагировать компонент.

Как видно из определения, применение компонентного программирования призвано обеспечить более простую быструю и прямолинейную процедуру первоначальной инсталляции прикладного программного обеспечения, а также увеличить процент повторного использования кода, т.е. усилить основные преимущества ООП.

Говоря о свойствах компонентов, следует, прежде всего, отметить, что это существенно более крупные единицы, чем объекты (в том смысле, что объект представляет собой конструкцию уровня языка программирования). Другими отличиями компонентов от традиционных объектов являются возможность содержать множественные классы и (в большинстве случаев) независимость от языка программирования.

Заметим, что, автор и пользователь компонента, вообще говоря, территориально распределены и, вполне возможно, они не только пишут программы, но и говорят на разных языках.

Можно выделить следующие основные преимущества применения компонентно-ориентированного подхода при проектировании и разработке ПО:

- снижение стоимости программного обеспечения;
- повышение повторного использования кода;
- унификация обработки объектов различной природы;
- менее человеко-зависимый процесс создания программного обеспечения.

Так как программный компонент подразумевает полноценное автономное использование в виде «черного ящика», к разработке программных компонентов предъявляются серьезные требования:

- *полная документированность интерфейса*: все методы, предоставляемые в интерфейсе программного компонента, должны быть качественно документированы, с учетом всех возможных вариантов их использования в сторонних приложениях;
- *тщательное тестирование:* необходимо учесть все возможные и невозможные варианты использования программного компонента в сторонних системах на всех возможных значениях входных данных;
- *тщательный анализ входных значений*: необходимо учитывать возможность передачи в программный компонент входных данных, не соответствующих его спецификации и адекватно обрабатывать такие ситуации;
- *возврат адекватных и понятных сообщений об ошибках:* так как один программный компонент может быть использован в большом числе сторонних программных систем, необходимо обеспечить сторонним разработчикам возможность получения информации об ошибках программного компонента и возможных вариантах их решения;
- *необходимость предусмотреть возможность неправильного использования.*

7.2 Концепция JavaBeans

JavaBeans — классы в языке Java, написанные по определённым правилам. Они используются для объединения нескольких объектов в один (bean) для удобной передачи данных. JavaBeans обеспечивают основу для многократно используемых, встраиваемых и модульных компонентов ПО.

Спецификация Sun Microsystems определяет JavaBeans, как «универсальные программные компоненты, которые могут управляться с помощью графического интерфейса».

Компоненты JavaBeans могут принимать различные формы, но наиболее широко они применяются в элементах графического пользовательского интерфейса. Одна из целей создания JavaBeans – взаимодействие с похожими компонентными структурами. Например, Windows-программа, при наличии соответствующего моста или объекта-обёртки, может использовать компонент JavaBeans так, будто бы он является компонентом COM или ActiveX.

7.2.1 Правила описания JavaBean

Чтобы класс мог работать как bean, он должен соответствовать определённым соглашениям об именах методов, конструкторе и поведении. Эти соглашения дают возможность создания инструментов, которые могут использовать, замещать и соединять JavaBeans.

Правила описания гласят:

- *Класс должен иметь public конструктор без параметров*. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
- *Свойства класса должны быть доступны через get, set и другие методы (так называемые методы доступа)*, которые подчинятся стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание bean. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
- *Класс должен быть сериализуем*. Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние bean независимым от платформы и виртуальной машины способом.
- *Класс не должен содержать никаких методов обработки событий*.

Т.к. требования в основном изложены в виде соглашения, а не интерфейса, некоторые разработчики рассматривают JavaBeans, как Plain Old Java Object, которые следуют определённым правилам именования.

7.2.2 Enterprise JavaBeans

Enterprise JavaBeans – это высокоуровневая, базирующаяся на использовании компонентов технология создания распределенных приложений, которая использует низкоуровневый API для управления транзакциями. Первый вариант спецификации Enterprise JavaBeans появился в марте 1998 г. За время своего существования, технология прошла большой путь и продолжает развиваться.

Enterprise JavaBeans – больше, чем просто инфраструктура. Ее использование подразумевает еще и технологию (процесс) создания распределенного при-

ложения, навязывает определенную архитектуру приложения, а также определяет стандартные роли для участников разработки. Применение данных техник обеспечивает решение следующих стандартных проблем масштабируемых и эффективных серверов приложений с использованием Java:

- организация удаленных вызовов между объектами, работающими под управлением различных виртуальных машин Java;
- управление потоками на стороне сервера;
- управление циклом жизни серверных объектов (создание, взаимодействие с пользователем, уничтожение);
- оптимизация использования ресурсов (время процессора, память, сетевые ресурсы);
- создание схемы взаимодействия контейнеров и операционных сред;
- создание схемы взаимодействия контейнеров и клиентов, включая универсальные средства создания разработки компонентов и включения их в состав контейнеров;
- создание средств администрирования и обеспечение их взаимодействия с существующими системами;
- создание универсальной системы поиска клиентом необходимых серверных компонентов;
- обеспечение универсальной схемы управления транзакциями;
- обеспечение требуемых прав доступа к серверным компонентам;
- обеспечение универсального взаимодействия с СУБД.

Технология Enterprise JavaBeans определяет набор универсальных и предназначенных для многократного использования компонентов, которые называются Enterprise beans (далее – *компоненты EJB*). При создании распределенной системы, ее бизнес-логика будет реализована в этих компонентах. Каждый компонент EJB состоит из *удаленного интерфейса*, *собственного интерфейса* и *реализации EJB-компонента*. Удаленный интерфейс (remote-интерфейс) определяет бизнес-методы, которые может вызывать клиент EJB. Собственный (домашний) интерфейс (home-интерфейс) предоставляет методы `create` для создания новых экземпляров компонентов EJB, методы поиска (`finder`) для нахождения экземпляров компонентов EJB и методы `remove` для удаления экземпляров EJB. Реализация EJB-компонента определяет бизнес-методы, объявленные в удаленном интерфейсе, и методы создания, удаления и поиска собственного интерфейса.

После завершения разработки, наборы компонентов EJB помещаются в специальные файлы (архивы, jar), по одному или более компоненту, вместе со специальными *параметрами развертывания* (deployment). Затем они устанавливаются в специальной операционной среде, в которой запускается контейнер EJB. *Контейнер EJB* предоставляет окружение выполнения и средства управления жизненным циклом EJB-компонентам.

Клиент осуществляет поиск компонентов в контейнере с помощью home-интерфейса соответствующего компонента. После того, как компонент создан и/или найден, клиент выполняет обращение к его методам с помощью remote-интерфейса.

Контейнеры EJB выполняются под управлением *сервера* EJB, который является связующим звеном между контейнерами и используемой операционной средой. Сервер EJB обеспечивает доступ контейнерам EJB к системным сервисам, таким как управление доступом к базам данных или мониторинг транзакций.

Все экземпляры компонентов EJB выполняются под управлением контейнера EJB. Контейнер предоставляет системные сервисы размещенным в нем компонентам и управляет их (компонентов) жизненным циклом. В общем случае контейнер предназначен для решения следующих задач:

– *Обеспечение безопасности*. Дескриптор развертывания (deployment descriptor) определяет права доступа клиентов к бизнес-методам компонентов. Обеспечение защиты данных обеспечивается за счет предоставления доступа только для авторизованных клиентов и только к разрешенным методам.
– *Обеспечение удаленных вызовов*. Контейнер берет на себя все низкоуровневые вопросы обеспечения взаимодействия и организации удаленных вызовов, полностью скрывая все детали процесса, как от разработчика компонентов, так и от клиентов. Это позволяет производить разработку компонентов точно так же, как если бы система работала в локальной конфигурации, т.е. вообще без использования удаленных вызовов.
– *Управление жизненным циклом*. Клиент создает и уничтожает экземпляры компонентов, однако контейнер для оптимизации ресурсов и повышения производительности системы может самостоятельно выполнять различные действия, например, активизацию и деактивацию этих компонентов, создание их пулов и т.д.
– *Управление транзакциями*. Все параметры, необходимые для управления транзакциями, помещаются в дескриптор поставки. Все вопросы по обес-

печению управления распределенными транзакциями в гетерогенных средах и взаимодействия с несколькими базами данных берет на себя контейнер EJB. Контейнер обеспечивает защиту данных и гарантирует успешное подтверждение внесенных изменений; в противном случае транзакция откатывается.

7.2.3 Типы компонентов EJB

Существуют два типа компонентов EJB: сессионные и сущностные компоненты (session- и entity-компоненты).

Сессионный компонент представляет собой объект, созданный для обслуживания запросов *одного* клиента. В ответ на удаленный запрос клиента контейнер создает экземпляр такого компонента. Сессионный компонент всегда сопоставлен с одним клиентом, и его можно рассматривать как «представителя» клиента на стороне EJB-сервера. Такие компоненты могут «знать» о наличии транзакций – они могут отвечать за изменение информации в базах данных, но сами они непосредственно не связаны с представлением данных в БД.

Сессионные компоненты являются временными объектами. Обычно сессионный компонент существует, пока создавший его клиент поддерживает с ним сеанс связи. После завершения связи с клиентом компонент уже никак не сопоставляется с ним. Объект считается временным, так как в случае завершения работы или краха сервера клиент должен будет создать новый компонент.

Обычно сессионный компонент содержит параметры, которые характеризуют состояние его взаимодействия с клиентом, т.е. он сохраняет некоторое состояние между вызовами удаленных методов в процессе сеанса связи с клиентом.

Сущностные компоненты представляют собой объектное представление данных из базы данных. Например, компонент сущности может моделировать одну запись из таблицы реляционной базы данных (или набор записей из связанных таблиц). Ключевым отличием сущностного компонента от сессионного является то, что несколько клиентов могут одновременно обращаться к одному экземпляру сущностного компонента. Сущностные компоненты изменяют состояние сопоставленных с ними баз данных в контексте транзакций.

Состояние компонентов-сущностей в общем случае нужно сохранять, и живут они столько, сколько существуют в базе данных те данные, которые они представляют, а не столько, сколько существует клиентский или серверный процесс. Остановка или крах контейнера EJB не приводит к уничтожению содержащихся в нем сущностных компонентов.

7.2.4 Составные части EJB-компонента

EJB-компонент физически состоит из нескольких частей, включая сам компонент, реализацию некоторых интерфейсов и информационный файл. Все это собирается вместе в специальный jar-файл – модуль развертывания.

- *Enterprise Bean* является Java-классом, разработанным поставщиком Enterprise Bean. Он реализует интерфейс Enterprise Bean и обеспечивает реализацию бизнес-методов, которые выполняет компонент. Класс не реализует никаких методов авторизации, многопоточности или поддержки транзакций.
- *Домашний интерфейс*. Каждый создающийся Enterprise Bean должен иметь ассоциированный домашний интерфейс. Домашний интерфейс применяется как фабрика для компонента EJB. Клиент использует домашний интерфейс для нахождения экземпляра компонента EJB или создания нового экземпляра компонента EJB.
- *Удаленный интерфейс* является Java-интерфейсом, который отображает через рефлексию те методы Enterprise Bean, которые необходимо показывать внешнему миру. Удаленный интерфейс играет ту же роль, что и IDL-интерфейс в CORBA, и обеспечивает возможность обращения клиента к компоненту.
- *Описатель развертывания* является XML-файлом, который содержит информацию относительно компонента EJB. Использование XML позволяет установщику легко менять атрибуты компонента. Конфигурационные атрибуты, определенные в описателе развертывания, включают:
  - имена домашнего и удаленного интерфейса;
  - имя JNDI для публикации домашнего интерфейса компонента;
  - транзакционные атрибуты для каждого метода компонента;
  - контрольный список доступа для авторизации.
- *EJB-Jar-файл* – это обычный java-jar-файл, который содержит компонент (компоненты) EJB, домашний и удаленный интерфейсы, а также описатель развертывания.

7.2.5 Инфраструктура Enterprise JavaBean

Инфраструктура EJB обеспечивает удаленное взаимодействие объектов, управление транзакциями и безопасность приложения. Спецификация EJB оговаривает требования к элементам инфраструктуры и определяет Java Application Programming Interface (API); она не касается вопросов выбора платформ, протоколов и других аспектов, связанных с реализацией.

Ниже показаны различные элементы инфраструктуры EJB. Она должна обеспечивать канал связи с клиентом и другими компонентами EJB. Инфраструктура должна также обеспечить соблюдение прав доступа к компонентам EJB.

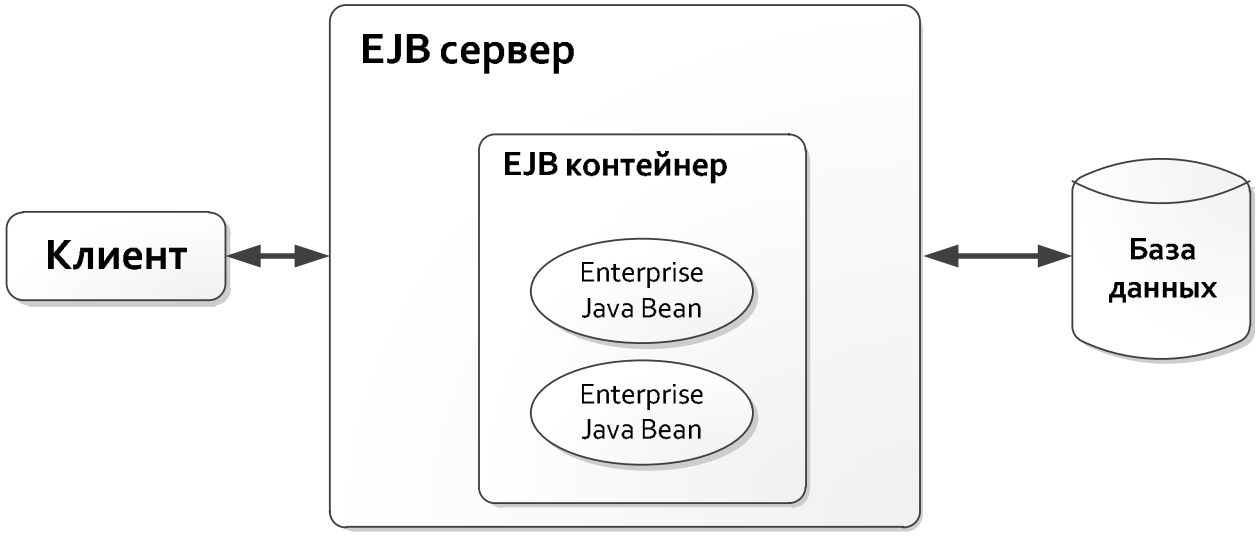


Рис. 22. Компоненты, контейнеры и серверы EJB

В общем случае необходимо гарантировать сохранение состояния компонентов в контейнерах. Инфраструктура EJB обязана предоставить возможности для интеграции приложения с существующими системами и приложениями – без этого нельзя говорить о пригодности приложения для функционирования в сложной информационной среде. Все аспекты взаимодействия клиентов с серверными компонентами должны происходить в контексте транзакций, управление которыми возлагается на инфраструктуру EJB. Для успешного выполнения процесса поставки компонентов инфраструктура EJB должна обеспечить возможность взаимодействия со средствами управления приложениями.

Таким образом, спецификация Enterprise JavaBeans – это существенный шаг к стандартизации модели распределенных объектов в Java. В настоящее время существует большое количество инструментов, поддерживающих этот подход и помогающих ускорить разработку EJB-компонентов.

8. Сервис-ориентированная архитектура

8.1 Концепция СОА

По определению организации OASIS (Organization for the Advancement of Structured Information Standards), *сервис-ориентированная архитектура (Service-Oriented Architecture – SOA)* – это парадигма организации и использования распределенных возможностей, которые могут принадлежать различным владельцам [56]. В основе сервис-ориентированной архитектуры лежит сущность «действия» (в противоположность сущности «объекта» объектно-ориентированной архитектуры). Типичными составляющими сервис ориентированной архитектуры являются: *сервисные компоненты* (сервисы); *контракты сервисов* (интерфейсы); *соединители сервисов* (транспорт) и *механизмы обнаружения сервисов* (регистры) [53].

Сервисные компоненты (или сервисы) описываются программными компонентами, обеспечивающими прозрачную сетевую адресацию. Существует множество различных определений сервиса. В статье [51] дается следующее определение: сервисами называются открытые, самоопределяющиеся компоненты, поддерживающие быстрое построение распределенных приложений. При этом нет четкого определения, какой объем услуг должен предоставлять отдельный сервис. С одной стороны, *мелкомодульный сервис* может предоставлять элементарный объем функциональной нагрузки и обеспечивать высокую степень повторного использования [63]. В этом случае, для получения желаемого результата, необходимо обеспечить координированную работу нескольких сервисов. С другой стороны, использование *крупномодульных сервисов* позволяет обеспечить хорошую инкапсуляцию функциональности, но затрудняет повторное использование таких сервисов, в связи с их узкой специализацией.

Контракт сервиса (или интерфейс) обеспечивает описание возможностей и качества предоставляемых услуг, предоставляемых конкретным сервисом [51]. В интерфейсе определяется формат сообщений, используемый для обмена информацией, а также входные и выходные параметры методов, поддерживаемых сервисным компонентом. От выбора языка и способа описания интерфейса зависят возможности программной совместимости различных реализаций сервис-ориентированной архитектуры.

Соединитель сервисов (или транспорт) обеспечивает обмен информацией между отдельными сервисными компонентами. Наряду с открытыми стандартами описания интерфейсов, использование гибких транспортных протоколов для обмена информацией между сервисными компонентами позволяет повысить программную совместимость сервис-ориентированной системы [50].

Механизмы обнаружения сервисов (или регистры сервисов) используются для поиска сервисных компонентов, обеспечивающих требуемую функциональность. Среди всего множества различных систем, обеспечивающих обнаружения сервисов [7], можно выделить две основных категории: системы динамического и статического обнаружения. Статические системы обнаружения сервисов (например, UDDI) ориентированы на хранение информации о сервисах в редко изменяющихся системах. Динамические системы обнаружения сервисов [10, 47, 54] ориентированы на системы, в которых допустимо частое появление и исчезновение сервисных компонентов.

На сегодняшний день, веб-сервисы – это наиболее часто встречающейся реализация сервис-ориентированной архитектуры. Веб-сервисы – это технология, разработанная для поддержки взаимодействия между распределенными системами посредством вычислительной сети [72]. В [14] дается следующее определение *веб-сервисов* – это слабосвязанные программные компоненты, поддерживающие многократное использование, которые семантически инкапсулируют отдельные функциональные возможности и программным образом доступны по стандартным протоколам Интернета. Веб-сервисы – это независимая от платформы и языка программирования среда, так как базируются на стандарте XML.

Большинство веб-сервисов используют HTTP для передачи сообщений. Это дает значительное преимущество при разработке распределенных приложений в масштабе Интернет, так как обычно брандмауэры и прокси-серверы без проблем пропускают HTTP-трафик, и в процессе взаимодействия не возникает неожиданных трудностей (которые могут возникнуть, например, при использовании технологии CORBA).

8.2 Связанность программных систем

Связанностью называют степень знания и зависимости одного объекта от внутреннего содержания другого. В соответствии с данным определением, программные системы можно разделить на 2 типа:

- *сильносвязанные системы* (Strong coupling): Java RMI, .NET Remoting и т.п;
- *слабосвязанные системы* (Loose coupling): COA.

Сильная связанность возникает, когда зависимый класс содержит ссылку непосредственно *на определенный класс*, предоставляющий некоторые возможности. В противоположность этому, слабая связанность возникает, когда зависимый класс содержит *ссылку на интерфейс* который может быть реализован одним или несколькими конкретными классами.

Основная цель использования концепции слабосвязанных программных систем – это уменьшение количества зависимостей между компонентами. При уменьшении количества связей, уменьшается объем возможных последствий, возникающих в связи со сбоями или системными изменениями. Наиболее яркие характеристики слабосвязанных распределенных систем приведены в таблице 1 [41].

Таблица 1. Сравнение слабосвязанных и сильносвязанных систем

	Сильносвязанные системы	**Слабосвязанные системы**
Физические соединения	Точка-точка	Через посредника
Стиль взаимодействий	Синхронные	Асинхронные
Модель данных	Общие сложные типы	Простые типы
Связывание	Статическое	Динамическое
Платформа	Сильная зависимость от базовой платформы	Независимость от платформы
Развертывание	Одновременное	Постепенное

Традиционный подход разработки распределенных приложений, поддерживаемый технологиями распределенных объектов, основывается на тесной связи между всеми программными компонентами. Слабосвязанность программных компонентов, поддерживаемая технологией веб-сервисов, позволяет значительно упростить координацию распределенных систем и их реконфигурацию [14].

8.3 Принципы построения COA

Способность двух или более информационных систем (или их компонентов) к взаимодействию, с целью решения определенной задачи и получения определенной информации, называют *интероперабельностью*. Это определение объединяет в себе два понятия:

– *техническая интероперабельность* означает совместимость систем на техническом уровне, включая протоколы передачи данных и форматы их представления;
– *семантическая интероперабельность* — свойство информационных систем, обеспечивающее взаимную употребимость полученной информации на основе общего понимания системами ее значения.

Примером семантической интероперабельности программных систем может служить процесс передачи определенных данных в текстовом виде по каналам связи. Например, если системы семантически не интероперабельны, то получатель не сможет однозначно интерпретировать полученную строку «1.23»: это может быть число с плавающей запятой записанное в десятичной или шестнадцатеричной системе счисления, а может быть дата, которую надо интерпретировать «23 января».

СОА не предписывает жесткой вертикальной («сверху вниз») методологии проектирования, внедрения или управления ИТ-инфраструктурой. Вместо этого, СОА ограничивается лишь рядом принципов, характеризующих каждый из этих процессов; поэтому ее иногда называют не архитектурой, а архитектурным стилем. Отметим некоторые из этих принципов.

– *Распределенное проектирование.* Решения относительно внутренних особенностей информационных систем принимаются различными группами людей, имеющими собственные организационные, политические и экономические мотивы.
– *Постоянство изменений.* Отдельные участки архитектуры могут претерпевать изменения в любой момент времени.
– *Последовательное совершенствование.* Локальное улучшение компонентов архитектуры должно приводить к совершенствованию всей архитектуры в целом – к росту суммарной полезности компонентов того же уровня, что и изменяемый, равно как и компонентов более низкого и более высокого уровня. Например, известный веб-сервис Google Translate постоянно претерпевает изменения. Изначально, он обеспечивал только веб-интерфейс для перевода и ограниченный набор языков. Постепенно увеличивались функциональные возможности сервиса: расширялся набор языков, появилась возможность голосового воспроизведения перевода, при переводе отдельного слова начали выдаваться словарные статьи с несколькими результатами перевода и т.п. При этом API и интерфейс менялся настолько незначительно, что клиенты могли использовать новые возмож-

ности посредством старого API (например, получение словарных статей) или же не менять используемый интерфейс до тех пор, пока не потребуется использование новых возможностей.

– *Рекурсивность.* Однотипные решения имеют место на различных уровнях архитектуры.

8.4 Подход СОА

С точки зрения информационных технологий, логика предприятия может быть разделена на две важных половины: бизнес-логика и логика приложения (рис. 23). Каждый слой существует в своем собственном «мире».

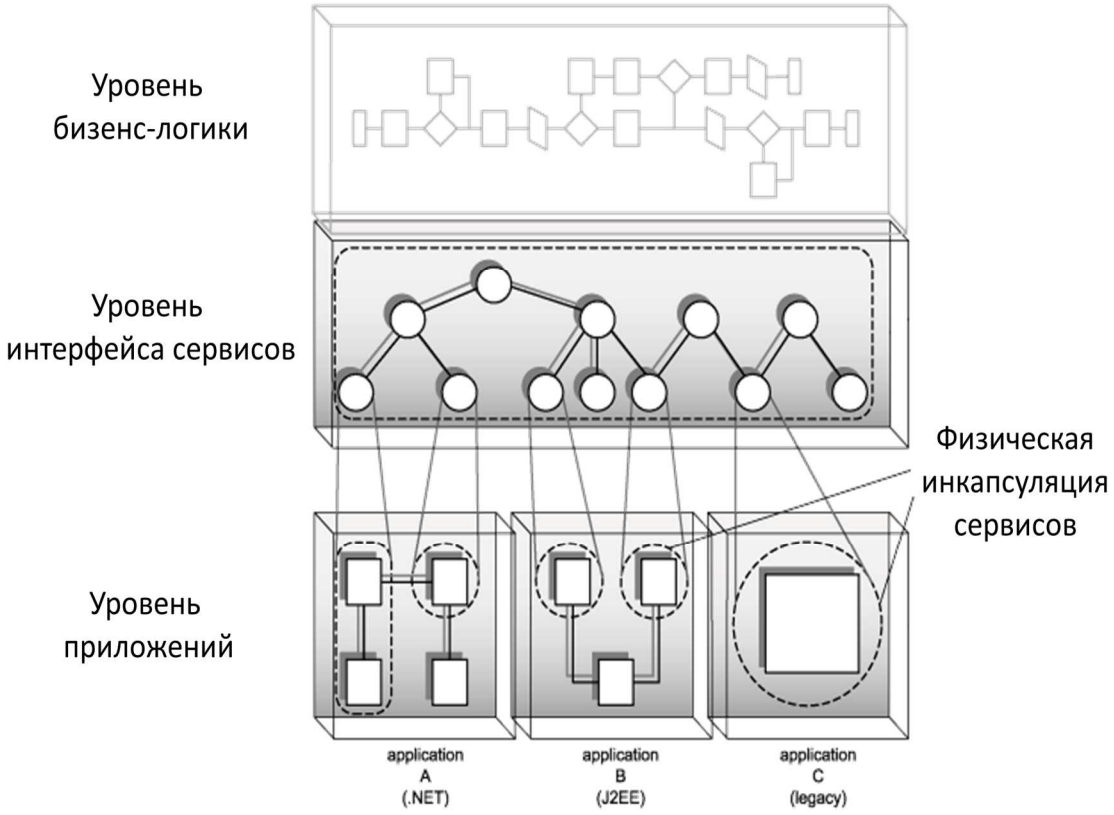


Рис. 23. Уровни логики предприятия

Бизнес-логика является документальной реализацией бизнес-требований, которые исходят из проблемной области, в которой работает предприятие. Бизнес-логика, как правило, структурирована в процессах, которые выражают эти требования, а также ограничениях и зависимости от внешних влияний.

Логика приложения – это реализация бизнес-логики, организованная на основе различных технологических решений. Логика приложения выражает процессы бизнес-логики посредством приобретенных или специально разработанных программных систем в условиях ограниченных технических возможностей и зависимостей от поставщика решения.

Процесс преобразования бизнес-логики в логику приложений и реализация сервисов на основе данных требований является процессом создания сервисно-ориентированной инфраструктуры для задач предприятия. Не существует «догматических» принципов построения СОА, но при реализации собственной инфраструктуры желательно придерживаться некоторых основных подходов, описанных ниже.

1. Сервисы должны поддерживать повторное использование. СОА-системы должны поддерживать повторное использование всех сервисов, независимо от сиюминутных требований к их функциональным особенностям. Если при разработке системы постараться максимально учесть это требование, то повышаются шансы значительно упростить процесс решения задач, которые непременно появятся в будущем, при развитии системы. Также изначально ориентированный на повторное использование сервис позволяет избежать разработки «обертки», которая бы подстраивала старый сервис для решения новых задач.

Так как сервис – это не что иное, как просто набор связных операций, логика каждой индивидуальной операции, предоставляемой сервисом, должна поддерживать повторное использование.

Например, в компании A-comp для отправки счета в систему биллинга «B-comp Account Payable Service» используется метод `SubmitInvoice`. Для того чтобы проверить состояние счета, необходимо постоянно проверять метаданные отправленного счета. Для этого используется метод `GetBcompMetadata`.

Т.к. эти операции ориентированы на решение *сиюминутных*, и вполне конкретных задач, они *не имеют потенциала повторного использования*. Метод `SubmitInvoice` разработан таким образом, чтобы транслировать сообщение в специфичном XML-формате, соответствующем системе B-comp, то есть «заточен» на конкретную версию протокола обмена информацией и конкретные данные. Метод `GetBcompMetadata` (как мы можем понять из имени метода) изначально ориентирован на использование исключительно с компанией B-comp, и не допускает повторного использования с любой другой системой биллинге.

Если бы компания A-comp задумалась о правильной сервисно-ориентированной инфраструктуре, то были бы разработан универсальный интерфейс с методами типа «`SubmitInvoice`» и «`CheckInvoice`», реализацию которых можно было бы использовать для любой внешней системы биллинга.

2. Сервисы должны обеспечивать формальный контракт использования. Контракт сервиса предоставляет следующую информацию:

- конечную точку (service endpoint): адрес, по которому можно обратиться к данному сервису;
- все операции, предоставляемые сервисом;
- все сообщения, поддерживаемые каждой операцией;
- правила и характеристики сервиса и его операций.

3. Сервисы должны быть слабосвязаны. Никто не может предугадать, в какую сторону будет развиваться IT-инфраструктура. Решения могут развиваться, взаимодействовать, заменять друг друга. В связи с этим основной задачей является сохранение целостности системы в рамках такого развития, независимо от происходящих изменений.

Система сервисов является слабосвязанной, если сервис может приобретать знания о другом сервисе, оставаясь независимым от внутренней реализации логики данного сервиса. Это достигается посредством использования контрактов сервисов.

Слабосвязанность программных компонентов, лежащая в основе СОА, позволяет значительно упростить координацию распределенных систем и их реконфигурацию. Основная цель использования концепции слабосвязанных программных систем – это уменьшение количества зависимостей между компонентами. При уменьшении количества связей, уменьшается объем возможных последствий, возникающих в связи со сбоями или системными изменениями.

Слабосвязанность – это не синоним «инкапсуляции» объектно-ориентированной концепции построения программных систем. Программная система может полностью соответствовать требованиям инкапсуляции, но быть сильносвязанной на семантическом уровне.

Например, существует сервис, который должен передавать большой объем текстовой информации в открытом виде другому сервису. При этом символ новой строки передавать нельзя, так как это неадекватно обрабатывается XML-парсером, и вся информация после знака новой строки теряется. Что же делать в этом случае? Первое решение, которое приходит в голову разработчику – это замена всех вхождений символов переноса строки на любой другой редко встречающийся в передаваемых сообщениях символ (или последовательность символов). В этом случае единственная деталь, которую необходимо учесть –

это описать в приемнике простейшую обратную процедуру, которая будет делать обратную замену, и вопрос решен. Поступая так, разработчик игнорирует принцип слабосвязанности программных систем, так как *в этом случае клиент и сервер будут взаимодействовать между собой не на основе открытого интерфейса, а на основе информации о внутренних процессах работы друг друга.*

Разработчик не задумывается о последствиях, которые может повлечь за собой в дальнейшем такое решение. Если через несколько месяцев (когда этот «финт» забудется) сторонний разработчик захочет создать собственного клиента для данного сервиса он будет неприятно удивлен, увидев вместо ожидаемого форматированного текста странные символы. Еще большая проблема возникнет, если вдруг в самом тексте будет появляться последовательность символов, на которую разработчик заменил символ новой строки. Поддержка такой системы превратится в постоянную пытку.

Существует несколько вариантов решения данной задачи посредством перехода к концепции слабосвязанных систем. Одним из оптимальных решений является переход от передачи данных в виде строки к передаче данных в виде массива строк. Это будет явно отражено в интерфейсе системы, соответственно, любой клиент будет знать, что данные к нему будут переданы в виде массива строк, каждая из которых заканчивается символом перехода на новую строку.

4. Сервисы должны абстрагировать внутреннюю логику. Каждый сервис должен действовать как «черный ящик», скрывающий свои детали от окружающего мира. Нет четкого определения, какой объем логики должен помещаться в отдельном сервисе. Взаимодействие на уровне интерфейсов является одним из требований для обеспечения слабой связанности.

5. Сервисы должны быть совместимы. Сервис может как самостоятельно реализовывать логику, так и применять другие сервисы для ее реализации. Сервисы должны быть спроектированы таким образом, чтобы поддерживать возможность их использования в качестве элементов другого сервиса. Принцип совместимости не зависит от того, использует ли сервис для выполнения своей работы другие сервисы.

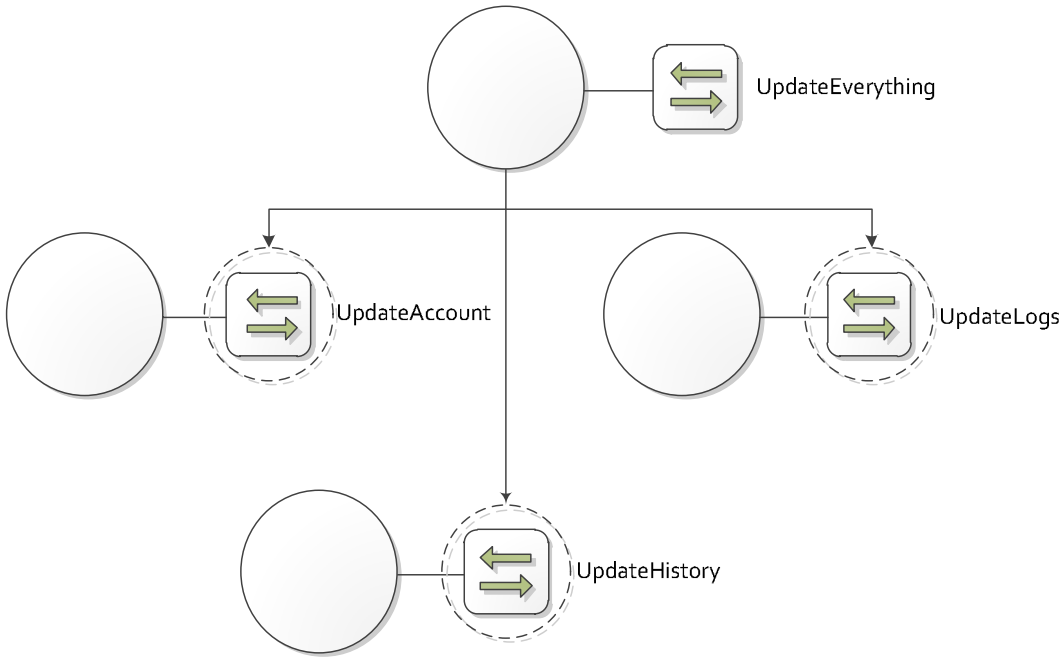


Рис. 24. Сервисы, используемые в качестве элементов другого сервиса

В качестве примера такого взаимодействия сервисов выступает концепция оркестрации сервисов. В этом случае, сервис-ориентированный процесс (который в принципе может быть определен как композиция сервисов) управляется сервисом родительского процесса, который включает в себя другие сервисы, являющиеся участниками данного процесса.

Кроме того, принцип совместимости также определяет вид сервисных операций. Совместимость – это, по сути, просто другая форма повторного использования, и поэтому операции должны быть стандартными, а для наибольшей совместимости должны обладать необходимым уровнем детализации.

6. Сервисы должны быть автономными. Свойство автономности требует, чтобы область бизнес-логики и ресурсов, используемых сервисом были ограничены явными пределами. Это позволяет сервису самому управлять всеми своими процессами (рис. 25). Также это устраняет зависимость от других сервисов, что освобождает сервис от связей, которые могут препятствовать его применению и развитию. Вопрос автономности – наиболее важный аргумент при распределении бизнес-логики на отдельные сервисы.

Обратите внимание, что автономность не обязательно предоставляет сервису исключительное право собственности на бизнес-логику, которую он инкапсулирует. Есть только гарантия того, что во время исполнения сервис контролирует любую логику, которую он реализует. Поэтому мы можем выделить два типами автономности.

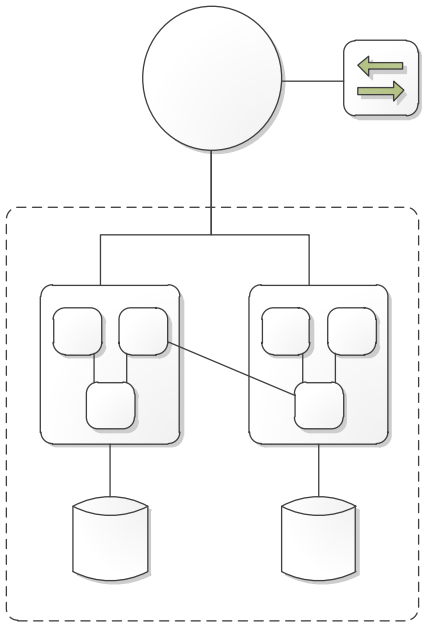


Рис. 25. Автономность сервиса: во время выполнения сервис управляет логикой нижнего уровня

– *Автономность на уровне сервиса*: границы ответственности сервисов отделены, но они могут использовать общие ресурсы. Например, сервис обертки, который инкапсулирует унаследованную программную систему, которая также кем-то используется (независимо от данного сервиса), обладает автономностью данного типа. Он управляет унаследованной системой, но также совместно использует ресурсы с другими существующими клиентами.
– *Чистая автономность:* бизнес-логика и ресурсы находятся под полным контролем сервиса. Как правило, такой вид автономности используется, когда для реализации сервиса бизнес-логика создается с нуля.

7. Сервисы не должны использовать информацию о состоянии. Сервисы должны сводить к минимуму объем информации о состоянии, и время, в течение которого они ею обладают. Информация о состоянии – это определенные данные, характеризующие текущую деятельность. Например, пока сервис обрабатывает сообщение, он временно зависит от состояния (stateful). Если сервис несет ответственность за сохранение состояния в течение более длительного времени, его способность оставаться доступным для других клиентов будет затруднена.

Независимость от состояния (statelessness) позволяет повысить возможности масштабируемости и повторного использования сервисов. Чтобы сервис

как можно меньше зависел от состояния, его операции должны быть разработаны с учетом соображений обработки информации без данных о состоянии.

Основной особенностью COA, поддерживающей независимость от состояния, является использование сообщений-документов. Чем выше сложность сообщения, тем более независимым и самодостаточным оно остается.

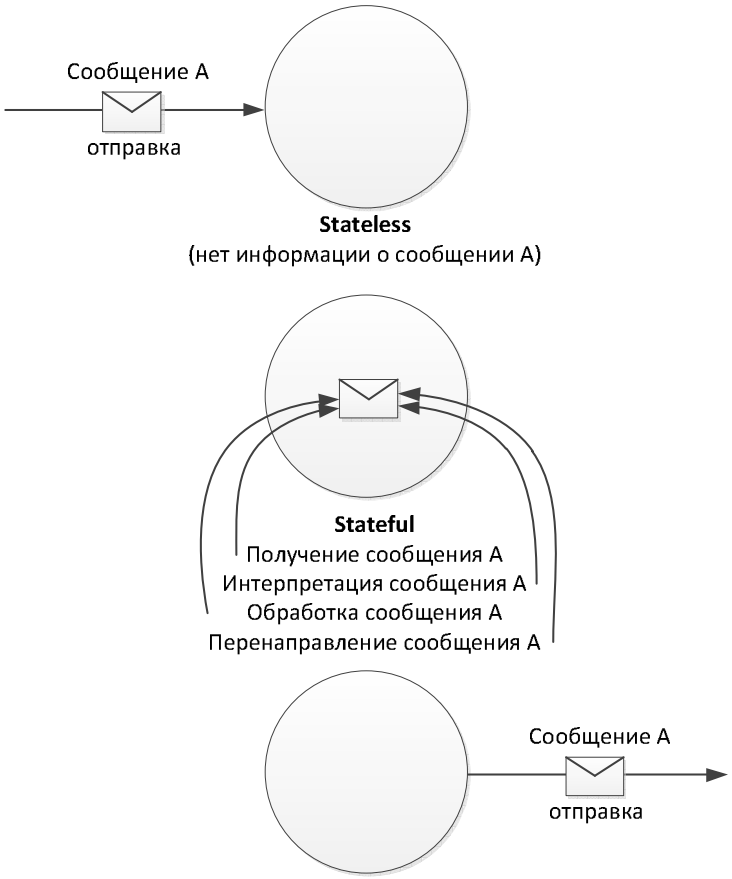


Рис. 26. Временная зависимость от состояния во время обработки сообщения

8. Сервисы должны поддерживать обнаружение. Обнаружение сервисов позволяет избежать случайного создания избыточного сервиса, обеспечивающего избыточную логику. Метаданные сервиса должны подробно описать не только общую цель сервиса, но и функциональность, реализуемую его операциями.

На уровне COA, обнаружение характеризует способность архитектуры обеспечить механизмы поиска, такие как реестр или каталог. На уровне сервиса, принцип обнаружения относится к процессу проектирования отдельного сервиса, так чтобы данный сервис настолько подавался обнаружению, насколько это возможно.

Рассмотрим пример. Компания RailCo не обеспечивает никаких средств обнаружения своих сервисов, как внутри своей компании, так и для внешних

пользователей. Хотя каждый сервис и обладает собственным WSDL-документом и в полной мере способен выступать в качестве поставщика услуг, сервис подачи счета в первую очередь используется в качестве элемента другого сервиса и не ожидает никаких сообщений от других сервисов, кроме сервиса оплаты счетов.

Сервис выполнения заказа (RailCo) был вручную зарегистрирован с помощью B2B-решения компании TLS, так что он будет помещен в список доступных сервисов. Этот сервис не предоставляет функциональность многоразового использования, и поэтому считается, что он не обладает способностью к обнаружению.

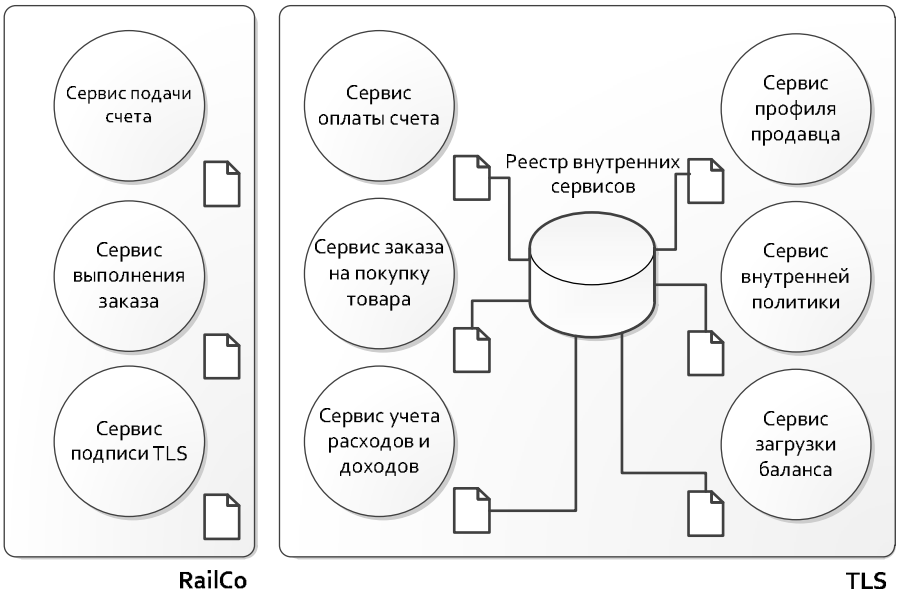


Рис. 27. Обнаружение на уровне логики предприятия

Из-за многоразового характера использования сервисов компании TLS и из-за количества сервисов, которые, как ожидалось, будут реализованы в компании TLS, был создан реестр внутренних сервисов (рис. 27). Эта часть инфраструктуры TLS способствует обнаружению сервисов и предотвращению случайной избыточности.

9. Веб-сервисы

9.1 Веб-сервисы первого поколения

XML Веб-сервисы (Веб-службы) – это программные компоненты, с помощью которых можно создавать независимые масштабируемые слабосвязанные приложения.

В основе технологии веб-сервисов лежит процесс обмена сообщениями в формате XML-документов. Передача сообщений происходит с использованием протоколов *HTTP, XML, XSD, SOAP, WSDL, UDDI.*

Спецификация определяет три основных стандарта, используемых для поддержки представления, поиска и обмена информацией между веб-сервисами – это WSDL, UDDI и SOAP, образующие так называемый «треугольник СОА». Процесс взаимодействия между клиентом и поставщиком веб-сервиса представлен на рисунке 28.

Рис. 28. Процесс взаимодействия между клиентом и поставщиком веб-сервиса

Протокол *SOAP* ***(уже не*** *«Simple Object Access Protocol», см. ниже)* предназначен для организации взаимодействия удаленных систем при помощи асинхронного обмена XML-отформатированными документами, состоящими из трех частей: конверта (обертки), заголовка и тела. SOAP формирует базовый слой стека протоколов веб-сервисов, обеспечивая инфраструктуру обмена сообщениями между ними. Хотя изначально название протокола SOAP и расшифровывалось как «Простой Протокол Доступа к Объектам» («Simple Object Access Protocol»), в версии 1.2 стандарта от этого определения отошли, и теперь этот акроним ничего не означает.

Язык *WSDL (Web Services Description Language)* описывает сервисы в виде неких абстрактных ресурсов, способных принимать на вход документы определенных типов и инициировать отправление документов других типов. WSDL

используется для описания веб-сервисов и для определения их расположения. WSDL написан на XML и является XML-документом.

WSDL определяет сервис с двух точек зрения: *абстрактной* и *конкретной*. *На абстрактном уровне* сервис задается в терминах посылаемых и принимаемых им сообщений, которые описываются средствами XML Schema в виде, независимом от конкретного транспортного протокола. *На конкретном уровне* определяются привязки к транспортным форматам и точкам физического размещения.

Группа спецификаций WSDL 2.0 состоит из трех основных документов – WSDL Part 1: Core language («Основной язык»), WSDL Part 2: Message exchange patterns («Шаблоны обмена сообщениями»), WSDL Part 2: Bindings («Привязки»).

Протокол *UDDI (Universal Description Discovery & Integration)* представляет собой стандарт на внутреннее устройство и внешние интерфейсы базы данных (репозитория), хранящей описания сервисов. Все описания в БД хранятся в виде XML-записей. Последняя версия UDDI (3.01) обеспечивает репликацию репозиториев со сложными моделями их подчиненности друг другу, построение репозитория из нескольких узлов (и репликацию данных между ними), глобальную уникальность записей и ключей, API публикации описаний и подписки на изменения, средства обеспечения целостности данных, интернационализации записей, шифрования содержимого. В то время как версия UDDI 2.0 предназначалась для поддержки каталогов электронного бизнеса, версия 3.0 ориентирована и на внутрикорпоративное использование для построения корпоративных систем в рамках идеологии сервис-ориентированной архитектуры. Поэтому она допускает создание реестров нескольких типов (общедоступный, частный и с разделяемым доступом).

Архитектуру веб-сервисов составляет масса протоколов и спецификаций. Их можно разбить на четыре части (процесс, описание, сообщения, связь), образующие стек протоколов, в котором каждый верхний уровень опирается на нижний уровень.

Рис. 29. Стек протоколов веб-сервисов

9.2 Стандарт WSDL

Рассмотрим пример простейшего веб-сервиса. Данный сервис будет обеспечивать возведение в квадрат переданного числа. На языке Java этот сервис можно было бы описать в виде следующего исходного кода.

```
public class MyMath
{
     public int squared(int x)
     {
          return x * x;
     }
}
```

Рис. 30. Пример исходного кода для тестового веб-сервиса

Предположим, что данный веб-сервис мы будем размещать в интернете по адресу http://supercomputer.susu.ru/MyMath.

Для того чтобы любой клиент мог получить информацию о том, какие методы предоставляет веб-сервис и как к ним необходимо обращаться, используется WSDL-документ , описывающий полную спецификацию методов взаимодействия с указанным сервисом. Соответственно, нам необходимо сформировать (вручную или автоматически) WSDL-документ и поместить его по адресу http://supercomputer.susu.ru/MyMath.wsdl.

Как уже было сказано в начале главы, стандарт WSDL обеспечивает описание веб-сервиса в виде сообщений, которые может отправить или же принять веб-сервис. Также, он отвечает за методы связывания данных сообщений с ба-

зовой средой передачи данных (чаще всего используется протокол HTTP). В связи с этим выделяют следующие *элементы WSDL-документа:*

- *Блок types* – типы данных, используемые веб-сервисом;
- *Блок message* – сообщения, используемые веб-сервисом;
- *Блок portType* – методы, предоставляемые веб-сервисом;
- *Блок binding* – протоколы связи, используемые веб-сервисом.

9.2.1 Порты WSDL <portType>

Элемент `<portType>` является наиболее важным элементом WSDL. Он определяет сам веб-сервис, предоставляемые им операции и используемые сообщения. Этот элемент можно сравнить с библиотекой функций, в которой указаны входные параметры и результаты работы функции. Рассмотрим, каким образом будет описываться наш сервис “`MyMath`”:

```
<wsdl:portType name="MyMath">
     <wsdl:operation name="squared" parameterOrder="in0">
          <wsdl:input message="impl:squaredRequest“
        name="squaredRequest"/>
          <wsdl:output message="impl:squaredResponse“
       name="squaredResponse"/>
     </wsdl:operation>
</wsdl:portType>
```

Рис. 31. Пример описания блока portType

Как видно из описания, в блоке `portType` описан интерфейс нашего сервиса `MyMath`, состоящий из одной операции `squared`. Также из описания видно, что данная операция должна выполняться с одним параметром `in0` (его описание будет дано в следующем разделе WSDL-документа, блоке `<message>`). Наша операция состоит из двух сообщений: входного (`wsdl:input message="impl:squaredRequest"`) и выходного (`wsdl:output message= "impl:squaredResponse"`). Входное сообщение `"squaredRequest"` – это то сообщение, которое должен передать пользователь веб-сервиса в наш сервис для того, чтобы запустить операцию возведения в квадрат. Выходное сообщение `"squaredResponse"` – это сообщение, которое будет возвращено пользователю после того, как наша операция возведения в квадрат успешно завершится. Интерфейс этих сообщений будет описан в блоке `<message>` WSDL-документа.

9.2.2 Сообщения WSDL <message>

Элемент `<message>` определяет элементы данных операции. Каждое сообщение может содержать одну или несколько частей. Эти части можно сравнить с параметрами вызываемых функций в традиционных языках программирования.

```
    <wsdl:message name="squaredRequest">
          <wsdl:part name="in0" type="xsd:int"/>
     </wsdl:message>
     <wsdl:message name="squaredResponse">
          <wsdl:part name="squaredReturn" type="xsd:int"/>
     </wsdl:message>
```

Рис. 32. Пример описания блока message

Как можно заметить, в блоке `<message>` приводится описание всех частей сообщений `"squaredRequest"` и `"squaredResponse"`, интерфейс которых мы описали ранее в блоке `<portType>`. Каждая часть сообщения – это параметр вызова метода нашего сервиса. В соответствии с этим легко расшифровать, что для проведения нашей операции возведения в квадрат, пользователь должен передать нам 1 входной параметр `"in0"` в виде целого числа, что определяется атрибутом `type="xsd:int"`. Результатом же операции также будет целое число, что описано в блоке `<wsdl:part name="squaredReturn" type="xsd:int"/>`.

9.2.3 Связи WSDL <binding>

Элемент `<binding>` определяет формат сообщения и детали протокола для каждого порта. Он отвечает за то, каким образом элементы абстрактного интерфейса в блоке `<portType>` преобразуются в массивы информации в формате протоколов взаимодействия, например SOAP.

```
    <wsdl:binding name="MyMathSoapBinding" type="impl:MyMath">
      <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="squared">
          <wsdlsoap:operation soapAction=""/>
          <wsdl:input name="squaredRequest">
               <wsdlsoap:body  encod-
ingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://DefaultNamespace" use="encoded"/>
               </wsdl:input>
          <wsdl:output name="squaredResponse">
               <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://DefaultNamespace" use="encoded"/>
          </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>
```

Рис. 33. Пример описания блока binding

Основной частью блока `<binding>` является элемент `<soap:binding>`, определяющий конкретный протокол передачи данных. Атрибут `style` определяет тип запроса и может иметь два значения: `"rpc"` и `"document"`. Также, в блоке `<soap:binding>` является `transport`, определяющий протокол, на основе которого будет производиться взаимодействие (обычно HTTP). Внутри

<soap:operation> находится элемент, который описывает значение поля soapAction HTTP-запроса (если вы знакомы с SOAP). Элементы input и output определяют как будут декодироваться входные и выходные сообщения этой операции.

9.2.4 Блоки <port> и <service>

В данных блоках происходит определение, где находится сервис: port – описывает расположение и способ доступа к конечной точке, service – именованная коллекция портов.

```
<wsdl:service name="MyMathService">
      <wsdl:port binding="impl:MyMathSoapBinding“ name="MyMath">
            <wsdlsoap:address
            location="http://supercomputer.susu.ru/MyMath"/>
      </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Рис. 34. Пример блока service

Как можно видеть из примера, в блоке <port> не происходит непосредственного описания методов взаимодействия с веб-сервисом. Они описываются ранее, в блоке <binding>, а в блоке <port> только дается ссылка на описанный метод связи binding.

9.2.5 Стандарт WSDL 2.0

В 2004 году был принят черновой вариант стандарта WSDL 2.0. WSDL 1.2 была переименована WSDL 2.0 из-за существенных отличий от WSDL 1.1. Не смотря на то, что до сих пор нет финальной спецификации данного стандарта, консорциум W3C в 2007 году рекомендовал использовать именно тот формат WSDL документов, который описан в стандарте WSDL 2.0.

В качестве основных изменений в стандарте WSDL 2.0 можно отметить следующее:

- многие блоки WSDL-документа были переименованы в соответствии с устоявшейся терминологией разработчиков распределенных приложений: Port превратился в Endpoint, PortType – в Interface;
- исчез блок Message, слившись с блоком Types (который стал обязательным);
- при описании методов связывания в блоке binding появилась возможность указывать любой метод HTTP запросов (GET, POST, PUT, DELETE) посредством атрибута whttp:method. Это сделало стандарт WSDL гораздо ближе к идеологии взаимодействия посредством веб.

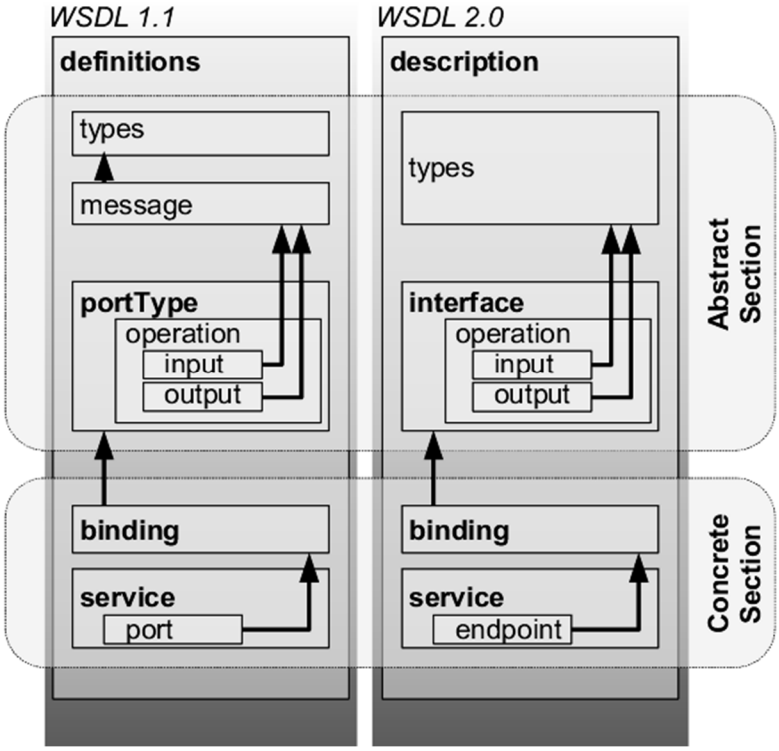


Рис. 35. Сравнение блоков WSDL 1.1 и WSDL 2.0

Однако поддержка WSDL 2.0 до сих пор является достаточно бедной в смысле программного обеспечения для разработки веб-сервисов.

9.3 Стандарт SOAP

Стандарт SOAP обеспечивает взаимодействие между веб-сервисами. Сообщением SOAP называют одностороннюю передачу информации от источника к приемнику между узлами SOAP. Сообщения SOAP являются основным строительным блоком, обеспечивающим возможности более сложных шаблонов взаимодействия: запрос/ответ, «диалоговый» режим и т.п.

Сообщение SOAP состоит из 3-х частей: *конверта*, содержащего *заголовок* и тело *сообщения*. В *теле* содержится XML-блок с информацией, которая должна быть доставлена конечному адресату. *Заголовок* – это не обязательный элемент SOAP-сообщения, при помощи которого можно передавать данные, не являющиеся собственно основной рабочей нагрузкой (к примеру: директивы и/или информацию о контексте, необходимые для обработки сообщения). SOAP-сообщение способно следовать по маршруту, содержащему несколько узлов, каждый из которых может вносить в него изменения или как-то еще его обрабатывать. Статус этих изменений отражается в блоках заголовка сообщения. Оба этих раздела содержатся внутри *конверта*.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    ...
    ...
  </soap:Header>
  <soap:Body>
    ...
    ...
  </soap:Body>
</soap:Envelope>
```

Рис. 36. Шаблон SOAP-сообщения

В заголовке SOAP-сообщения мы имеем возможность ввести новые элементы сообщения, не предусмотренные стандартом SOAP. Эти элементы играют утилитарную роль по отношению к основному сообщению, содержащемуся в теле SOAP. Например, мы можем передать номер транзакции, в рамках которой пришло то или иное сообщения, передать информацию, необходимую для авторизации пользователя и др.

```
<soap:Header>
  <trans:Transaction
  xmlns:trans="http://www.host.com/namespaces/space/"
  soap:mustUnderstand="1">
  12
  </trans:Transaction>
</soap:Header>
```

Рис. 37. Пример заголовка SOAP-сообщения

У каждого элемента, введенного в заголовок SOAP-сообщения, мы можем указать значения следующих атрибутов:

– *mustUnderstand* – если значение данного атрибута равно 1, получатель обязан обрабатывать этот элемент заголовка. Если он не умеет этого делать, то он обязан отбросить полученное сообщение;
– *actor* – указывает название конкретного приложения-получателя если SOAP-сообщение проходит цепочку приложений при обработке.

В теле SOAP-сообщения производится передача сообщения по формату, определенному в блоках `<porttype>` и `<message>` WSDL-документа. Имя основного блока, находящегося в теле SOAP-сообщения соответствует имени сообщения, которое определено в интерфейсе веб-сервиса.

Запрос

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetails xmlns="http://warehouse.example.com/ws">
       <productID>12345</productID>
     </getProductDetails>
   </soap:Body>
 </soap:Envelope>
```

Ответ

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetailsResponse
xmlns="http://warehouse.example.com/ws">
       <getProductDetailsResult>
         <productID>12345</productID>
         <productName>Роутер NoName</productName>
         <description>Роутер NoName, WiFi, LAN</description>
         <price>550</price>
         <inStock>true</inStock>
       </getProductDetailsResult>
     </getProductDetailsResponse>
   </soap:Body>
 </soap:Envelope>
```

Рис. 38. Пример запроса и ответа посредством SOAP-сообщений

Как уже было сказано, стандарт SOAP обеспечивает одностороннюю передачу сообщений. Соответственно, для того чтобы получить ответ от сервера, которому было передано SOAP сообщение, может потребоваться чтобы он инициировал процесс передачи сообщение и установил соединение с клиентом. Это может вызвать значительные затруднения в современных условиях организации связи в сети Интернет, т.к. большинство клиентов не имеют выделенных статических IP-адресов. Даже наоборот, чаще всего входящие соединения блокируются сетевыми брандмауэрами.

Данная проблема решена в рамках стандартного связывания протокола SOAP и протокола HTTP, реализующего паттерн поведения «запрос-ответ» (анг. *SOAP HTTP Binding*).

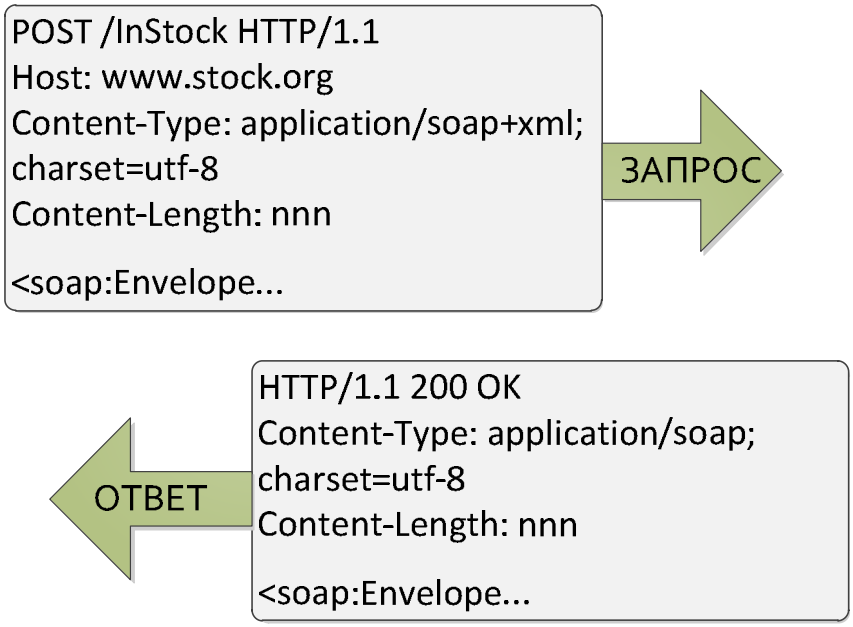


Рис. 39. Пример реализации паттерна взаимодействия «запрос-ответ» посредством HTTP-связывания

Заголовок «Content-Type» для сообщений HTTP-запроса и HTTP-ответа request устанавливается в значение text/xml (application/soap+xml в спецификации SOAP 1.2). HTTP-запрос должен использовать POST (начиная со спецификации SOAP 1.2 можно использовать GET). HTTP-ответ должен использовать статусный код 200 если обработка SOAP-сообщения прошла нормально, или же 500 если в теле сообщения содержится ошибка SOAP.

9.4 Второе поколение стандартов веб-сервисов

После появления технологии веб-сервисов в начале 2000-х годов, многие разработчики ощутили, что отсутствие стандартизации в наиболее важных областях построения распределенных вычислительных систем приводят к несовместимости разрабатываемых решений. Например, отсутствие единых протоколов авторизации и аутентификации пользователей приводило к тому, что каждому разработчику приходилось самостоятельно «придумывать велосипед» в этой области. Поэтому, когда возникала необходимость взаимодействия с внешней сервис-ориентированной системой приходилось тратить большое количество усилий на «скрещивание велосипедов», чтобы обеспечить стабильную работу объединенного решения.

Для решения этих проблем, множество коммерческих и некоммерческих организаций объединили свои усилия в рамках различных консорциумов для разработки нового поколения стандартов веб-сервисов (WS-* стандартов). К сожалению, это дало не совсем тот результат, на который рассчитывали с само-

го начала: каждый консорциум норовил разработать свой пакет стандартов, что привело к большому количеству различных стандартов, направленных на решение одних и тех же задач.

В рамках данной книги мы рассмотрим следующие, наиболее признанные и распространенные стандарты веб-сервисов:

– WS-Security – обеспечение безопасности веб-сервисов;
– WS-Addressing – маршрутизация и адресация SOAP-сообщений;
– WSRF, WS-Notification – работа с состоянием веб-сервисов.

9.4.1 Безопасность веб-сервисов и WS-Security

Стандарты веб-сервисов первого поколения не подразумевали обеспечения какой-либо безопасности при взаимодействии веб-сервисов. Разработчикам приходилось самостоятельно решать проблемы аутентификации и авторизации пользователей, разграничения прав доступа, передачи конфиденциальной информации и подписи сообщений. Это привело к тому, что практическое применение веб-сервисов в сфере бизнеса было ограничено.

Корпорации IBM и Microsoft совместно разработали семейство стандартов GXA – Global XML Web Services Architecture. В рамках данной архитектуры был предложен стандарт обеспечения безопасность веб-сервисов WS-Security. На схеме изображен стек протоколов обеспечения безопасности в GXA.

Рис. 40. Стек протоколов безопасности веб-сервисов на основе WS-Security

WS-Security является базой для других технологий в области безопасности веб-сервисов. В рамках данного стандарта описываются процессы обеспечения целостности (неизменности), конфиденциальности и неподдельности авторства (аутентичность) SOAP-сообщения, передаваемого в рамках уже установленных сессий, контекста и политики безопасности. В спецификации не говорится о

том, как формировать защищенное соединение, производить аутентификацию сторон, обмен ключами и обеспечивать другие аспекты безопасного общения. Это задачи других спецификаций GXA. Кроме того, в ней задается *только* рамочная *архитектура* – для производства реальных операций она полагается на уже имеющиеся технологии вроде PKI, Kerberos и SSL.

Таким образом, стандарт WS-Security ориентирован на комплексное решение задач безопасности при взаимодействии веб-сервисов, обеспечивая определение основных методов идентификации пользователя, цифровые подписи, шифрование.

WS-Security обеспечивает перемещение задач идентификации и авторизации в область обмена SOAP сообщениями. Чтобы обеспечить необходимый уровень безопасности SOAP-сообщения, информация, содержащаяся в SOAP-сообщении должна обеспечивать следующее:

- идентификацию категорий пользователей, связанных с сообщением;
- доказательство того, что категории пользователей имеют правильный набор прав доступа;
- доказательство того, что сообщение не изменялось.

WS-Security дает возможность применять маркеры безопасности (Security Tokens) при работе с SOAP сообщениями. Этот подход очень сильно напоминает «скидочные карточки», которые обеспечивают возможность получать товары и услуги и получать скидки в магазинах. Используя такие маркеры безопасности, SOAP сообщение может переправить следующую информацию [55]:

- *идентификацию вызывающего*: Я User Vasya Pupkin;
- *принадлежность к группе*: Я разработчик PupkinSite.com;
- *подтверждение прав*: поскольку я разработчик PupkinSite.com, я могу создавать базы данных и добавлять веб-приложения в сервера PupkinSite.com.

Чтобы создать сообщение, которое может создать новую базу данных на серверах *PupkinSite.com*, приложению придется запрашивать некоторые маркеры доступа. Эти маркеры доступа могут быть предоставлены в виде пары «имя пользователя/пароль», при помощи специальной смарт-карты, содержащей закрытый ключ шифрования или какого-либо другого метода. Когда *Vasya Pupkin* принимает решение создавать новую базу данных в *PupkinSite.com*, среда идет в Сервис присваивания удостоверений (Ticket Granting Service) и запрашивает Удостоверение сервиса (Service Ticket), который показывает, что *Vasya Pupkin* имеет право на создание новой базы данных на *PupkinSite.com*. Среда берет это

Удостоверение сервиса и представляет его серверу базы данных в *PupkinSite.com*. Этот сервер проверяет достоверность удостоверения и затем позволяет *Vasya Pupkin* создавать новый процесс.

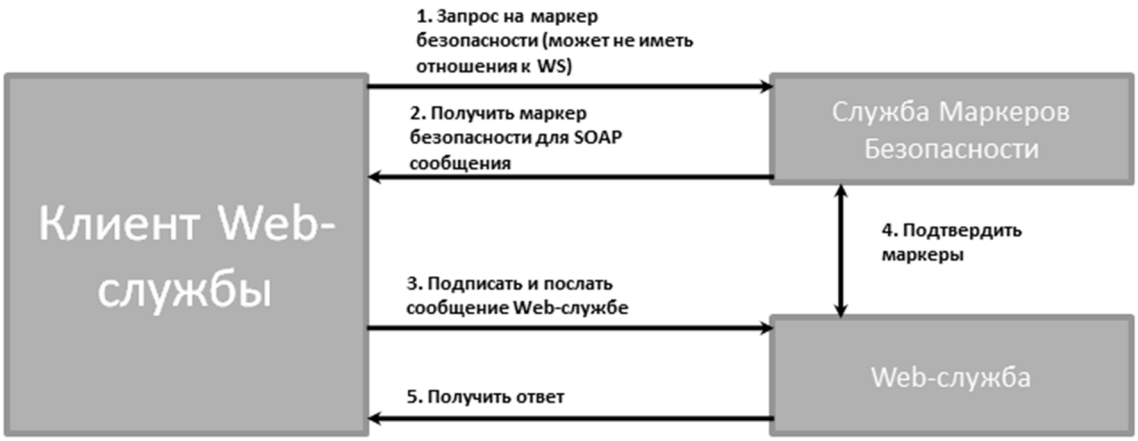


Рис. 41. Процедура авторизации пользователя на основе WS-Security

9.4.2 Аутентификация пользователя посредством WS-Security

Стандарт WS-Security предусматривает множество различных способов проверки достоверности пользователя. Из этого бесчисленного множества спецификация выделяет три метода:

– <*wsse:UsernameToken*> – аутентификация пользователя посредством пары «Имя пользователя/пароль»;
– <*wsse:X509v3*> – аутентификация посредством сертификата X.509v3;
– *Kerberos* – аутентификация посредством протокола Kerberos (Kerberos Domain Controller) (используется в Windows2000, Red Hat Linux и т.п.).

```
<soap:Envelope>
 <soap:Header>
    <wsse:Security soap:mustUnderstand="1">
       <wsse:UsernameToken>
          <wsse:Username>myName</wsse:Username>
          <wsse:Password Type="wsse:PasswordText">myPassword
           </wsse:Password>
       </wsse:UsernameToken>
   </wsse:Security>
 </soap:Header>
 ...
</soap:Envelope>
```

Рис. 42. Внедрение имени пользователя и пароля в заголовок SOAP-сообщения в виде открытого текста

В приведенном примере вы можете заметить префикс «wsse» у некоторых блоков, входящих в заголовок или тело SOAP-сообщения. Этот префикс означает, что формат данного XML-элемента определен в спецификации WS-Security.

Один из наиболее распространенных способов передачи удостоверения пользователя – использование комбинации имени пользователя и пароля. Для передачи удостоверения пользователя таким способом, в WS-Security определен элемент UsernameToken. Этот фрагмент использует два других типа: Username и Password. Эти два типа, представляют собой строки, которые, если необходимо, могут содержать дополнительные атрибуты.

Пароль может передаваться как простой текст или в цифровом формате. Передача имени пользователя и пароля в открытом формате обычно применяется в том случае, если обмен SOAP-сообщениями ведется поверх установленного защищенного соединения. При передаче пароля в цифровом виде производится его хеширование на основе случайного ключа и информации о времени формирования сообщения. Такой алгоритм обеспечивает возможность производить авторизацию пользователя по имени и паролю даже в том случае, если обмен сообщениями происходит по открытому каналу.

```
<wsse:UsernameToken>
    <wsse:Username>scott</wsse:Username>
    <wsse:Password Type="wsse:PasswordDigest">
        KE6QugOpkPyT3Eo0SEgT30W4Keg=</wsse:Password>
    <wsse:Nonce>5uW4ABku/m6/S5rnE+L7vg==</wsse:Nonce>
    <wsu:Created xmlns:wsu=
        "http://schemas.xmlsoap.org/ws/2002/07/utility">
            2002-08-19T00:44:02Z
    </wsu:Created>
</wsse:UsernameToken>
```

Рис. 43. Передача пароля в виде цифрового хэша

Другим вариантом авторизации пользователей является аутентификация на основе сертификата X.509. Сертификат X.509 позволяет однозначно определить, кем конкретно является пользователь системы. Использование сертификата по-своему может способствовать осуществлению очень простых атак воспроизведения. В результате, нелишним будет заставлять отправителя сообщения также подписывать его с помощью секретного ключа. Таким образом, когда сообщение получает ключ для дешифровки, вы будете знать, что это действительно пользователь.

Когда сообщение посылает сертификат X.509, оно передаст открытую версию сертификата в маркер WS-Security BinarySecurityToken. Сам сертификат получает отправленное сообщение как шифрованные base64 данные. Для обеспечения безопасности при использовании сертификата надо прибегнуть к дополнительным средствам обеспечения безопасности: подпись сообщения секретным ключом сертификата и добавлению `wsu:Timestamp` для определения времени жизни сообщения.

В заголовке WS-Security аутентификация пользователя посредством сертификата X.509 будет выглядеть примерно так:

```
<wsse:BinarySecurityToken
ValueType="wsse:X509v3"
EncodingType="wsse:Base64Binary"
Id="SecurityToken-f49bd662-59a0-401a-ab23-1aa12764184f">
   MIIHdjCCBCCAWqgAwIBAgIBGzANBgkqh-
kiG9w0BAQQFADBHMQ0wCwYDVQQKEwRMRwwGgYDV
   QQLExNDYWViZW-
FucyBkZXZlbG9wZXJzMRgwFgYDVQQLEw9jYWViZWFucy5uZXQucnUwHhcN
   MDcxMjAxMDAwMDAwWhcNMDgwMTMxMjM1OTU5Wj...
</wsse:BinarySecurityToken>
```

Рис. 44. Аутентификация на основе сертификата X.509

9.4.3 Подпись сообщения

Подпись сообщения позволяет получателю SOAP-сообщения удостовериться в том, что подписанные элементы не были изменены по пути передачи сообщения. При этом не надо забывать, что подпись сама по себе не может защитить сообщение от просмотра его содержимого третьими лицами.

Непосредственно за подпись сообщения отвечает спецификация XML Signature. WS-Security просто описывает, как использовать данный формат для подтверждения того, что SOAP-сообщение не было изменено по пути следования.

В зависимости от выбранного метода аутентификации, при подписи сообщения может быть использована следующая информация:

– UsernameToken – *пароль пользователя*;
– X.509 – *секретный ключ*;
– Kerberos – *сеансовый ключ*.

```
<soap:Envelope>
 <soap:Header>
    <wsse:Security soap:mustUnderstand="1">
    ...
      <ds:Signature>
        <ds:Signedlnfo>
                  ...
        </ds:Signedlnfo>
        <ds:SignatureValue>
            Hp1ZkmFZ/2kQLXDJbchm5gK...
        </ds:SignatureValue>
     <ds:Keylnfo>
        <wsse:SecurityTokenReference>
           <wsse:Reference URI=" #X509Token"/>
         </wsse:SecurityTokenReference>
     </ds:Keylnfo>
   </ds:Signature>
    </wsse:Security>
 </soap:Header>
 ...
</soap:Envelope>
```

Рис. 45. Подпись сообщения на основе сертификата X.509

9.4.4 Шифрование

Аутентификация и подпись сообщения – это не всегда достаточная мера обеспечения безопасности, особенно при передаче конфиденциальной информации. Как и с подписью сообщений, разработчики спецификации WS-Security приняли уже существующий стандарт, который хорошо выполняет работу, связанную с шифрованием. За шифрование отвечает стандарт XML Encryption.

```
<soap:Envelope>
...
<soap:Body>
      <xenc:EncryptedData
      Id="EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51"
      Type="http://www.w3.org/2001/04/xmlenc#Content">
         <xenc:EncryptionMethod Algorithm=
         "http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
         <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
            <KeyName>Symmetric Key</KeyName>
         </KeyInfo>
         <xenc:CipherData>
            <xenc:CipherValue>
          InmSSXQcBV5UiT...  Y7RVZQqnPpZYMg==
          </xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedData>
...
</soap:Envelope>
```

Рис. 46. Шифрование сообщения на основе XML Encryption

Как можно увидеть из представленного примера, все содержимое тела SOAP-сообщения шифруется и заменяется на блок `<xenc:EncryptedData>`, кото-

рый может быть прочитан только при наличии секретной авторизационной информации.

9.5 Адресация и WS-Addressing

В стандартах первого поколения полный адрес веб-сервиса содержался в WSDL-описании, в блоке `<port>`. Это может доставлять значительные неудобства, т.к. при изменении адреса сервиса приходится редактировать WSDL-файл целиком. А при обмене сообщениями SOAP, адресация возложена на транспортный протокол (при связывании с HTTP) и не может быть изменена непосредственно в SOAP-сообщении.

В стандарте WS-Addressing предусматривается введение полей `<wsa:To>` и `<wsa:Action>` в заголовок SOAP-сообщения, определяющих URI приемника сообщения и соответствующее действие:

В стандарте первого поколения подразумевается, что ответное сообщение передается по уже открытому HTTP каналу в соответствии с правилами HTTP-связывания. При этом нет стандартной поддержки асинхронной коммуникации между веб-сервисами. Стандарт WS-Addressing вводит следующие поля: `<MessageID>`, `<From>` (адрес отправителя), `<ReplyTo>` (адрес получения ответа), `<FaultTo>` (адрес извещения об ошибках), `<RelatedTo>` (позволяет выстраивать сообщения в цепочки).

Рис. 47. Адресация посредством WS-Addressing

Стандарт WS-Addressing обеспечивает расширенную адресацию конечных точек в WSDL-файле. Так как WSDL не поддерживает расширение элемента <Service>, в WS-Addressing определен элемент <EndpointReference>, который может быть использован в WSDL.

<EndpointReference> расширяет элемент <Service> добавляя поля ReferenceProperties и Policy. Поля Address, ServiceName и PortType уже включены в элемент <Service>.

EndpointReference:

```
<wsa:EndpointReference>
     <wsa:Address />
     <wsa:ReferenceProperties />
     <wsa:ServiceName PortName=””/>
     <wsa:PortType />
     <wsa:Policy />
</wsa:EndpointReference>
```

WSDL Service:

```
<wsdl:service name="MyMathService">
      <wsdl:port binding="impl:MyMathSoapBinding“ name="MyMath">
           <wsdlsoap:address
           location="http://supercomputer.susu.ru/MyMath"/>
      </wsdl:port>
 </wsdl:service>
```

Рис. 48. Сравнение EndpointReference и блока WSDL Service

Рассмотрим основные поля, входящие в состав адреса конечной точки по стандарту WS-Addressing:

- <wsa:Address /> – URI, который идентифицирует конечную точку;
- <wsa:ReferenceProperties /> – может содержать отдельные свойства, необходимые для идентификации лица или ресурса (например, имя файла, с которым должно производиться определенное действие или идентификатор корзины покупателя;
- <wsa:PortType />; <wsa:ServiceName /> – используются по аналогии с блоками PortType и ServiceName WSDL;

Также существует ряд необязательных полей (policy, reference parameters, selected port type, service-port).

Рассмотрим пример, каким образом может быть сформировано SOAP-сообщение на основе блока <EndpointReference> на примере веб-сервиса интернет-магазина. На рисунке представлен пример информации, хранящейся в ссылке на конечную точку. Приложение, работающее с интернет магазином, сохранило не только адрес веб-сервиса, который необходимо вызывать для по-

лучения доступа к магазину, но также и уникальную идентификационную информацию, которая позволяет узнать, от какого покупателя поступил запрос (блок <fabrikam:CustomerKey>) и уникальный идентификатор его корзины (<fabrikam:ShoppingCart>). Эти данные хранятся в блоке дополнительных свойств ссылки <wsa:ReferenceProperties />.

Рис. 49. Формирование SOAP-сообщения на основе <EndpointReference>

При формировании SOAP-сообщения, данные свойства переходят в его заголовок и передаются веб-сервису, который знает, каким образом обрабатывается эта информация. Таким образом, стандарт WS-Addressing не только обеспечивает независимые методы адресации веб-сервисов, но и возможность передачи и хранения дополнительной информации в заголовках SOAP-сообщений.

9.6 Состояние веб-сервисов и WSRF

Изначально, использование веб-сервисов не подразумевало существования «состояния» (этот вопрос был подробно рассмотрен в разделе 8.4 «Подход СОА»). Типовой сценарий использования веб-сервиса подразумевал шаблон «запрос – ответ – отключение». При этом каждый следующий запрос недолжен зависеть от результатов предыдущего запроса.

Но для многих приложений, как коммерческих, так и научных, сохранение информации о состоянии было существенным требованием успешного функционирования. Например, для разработки грид-систем не получилось применить «чистые» веб-сервисы, т.к. они не обладали возможностью работа с состоянием. В течение довольно продолжительного времени, отсутствие стандартных технологий обработки состояния привело к появлению множества несовмести-

мых друг с другом вариантов «симуляции» работы с состоянием посредством веб-сервисов.

Спецификация Web Services Resource Framework (WFRF) является попыткой решить указанную архитектурную проблему с помощью введения понятия «состояние» в веб-сервисы, превратив их в веб-ресурсы, и указав механизмы использования этого понятия.

Спецификация WSRF была предложена в 2004-м году, утверждена в качестве стандарта OASIS в 2006-м году. WSRF – это ряд спецификаций, которые определяют стандартные способы запроса значений свойств или способы указания того, что эти свойства должны быть изменены. Также WSRF определяет стандартные способы разрешения всех различных проблемных вопросов работы с WS-ресурсами.

Спецификация WSRF включает следующие стандарты:

– *WS-Resource specification* – описание WS-ресурсов;
– *WS-ResourceProperties (WSRF-RP)* – описание свойств WS-ресурсов;
– *WS-ResourceLifetime (WSRF-RL)* – описание управления временем жизни (создание и уничтожение) WS-ресурсов;
– *WS-ServiceGroup (WSRF-SG)* – работа с группами ресурсов;
– *WS-BaseFaults (WSRF-BF)* – описание основных ошибок, которые могут возникнуть при работе с WS-ресурсами.

В спецификации *WS-ResourceLifetime (WSRF-RL)* определяется, когда WS-ресурс должен прекратить свою активность или быть явно уничтожен, когда пропадает необходимость его существования. Спецификация *WS-ServiceGroup (WSRF-SG)* определяет способ создания набора веб-сервисов, такого, например, как регистр имеющихся сервисов. Спецификация *WS-Base Faults (WSRF-BF)* определяет стандартный способ фиксации ошибок в WSRF-приложении. В этом списке нет спецификации, определяющей как все это должно работать совместно. Эту функцию выполняет (почти полностью) документ Modeling stateful resources with Web services (Моделирование ресурсов с состоянием посредством веб-сервисов) [22], в котором объясняется общая концепция этих спецификаций и их связь друг с другом.

В соответствии со спецификацией, *WS-ресурс* является объединением веб-сервиса и ресурса с состоянием, на который веб-сервис может воздействовать.

9.6.1 Процесс создания WS-ресурсов

Состояние объекта можно определить через значения его различных свойств. Мы можем представить наш ресурс с состоянием в виде XML-

документа, в котором содержатся его свойства. Такой документ называется *Resource properties document*. В качестве примера, сформируем список свойств такого ресурса, как космический спутник. В качестве свойств ресурса определим такие его характеристики как широта, долгота, углы наклона, высота над земной поверхностью, а также текущий вид из объектива фотокамеры, установленной на данном спутнике.

```
<satProp:GenericSatelliteProperties
xmlns:satProp="http://example.com/satellite">
 <satProp:latitude>30.3</satProp:latitude>
 <satProp:longitude>223.2</satProp:latitude>
 <satProp:altitude>47700</satProp:altitude>
 <satProp:pitch>49</satProp:pitch>
 <satProp:yaw>0</satProp:yaw>
 <satProp:roll>32</satProp:roll>
 <satProp:focalLength>21999992</satProp:focalLength>
 <satProp:currentView>
       http://example.com/satellite/2239992333.zip
 </satProp:currentView>
</satProp:GenericSatelliteProperties>
```

Рис. 50. Свойства WS-ресурса «космический спутник»

К этому моменту мы создали модель нашего ресурса с состоянием (спутника), но чтобы завершить создание WS-ресурса, мы должны связать его с нашим сервисом, используя WSDL-файл. Для добавления информации о свойствах нашего ресурса, сформируем типы данных, которые будут представлять наш WS-ресурс в WSDL-файле.

```
<definitions name="Satellite" ...>
 ...
 <types>
       <xsd:schema targetNamespace="http://example.com/satellite"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
             ...
             <xsd:element name="latitude" type="xsd:float" />
             <xsd:element name="longitude" type="xsd:float" />
             <xsd:element name="altitude" type="xsd:float" />
             ...
             <xsd:element name="GenericSatelliteProperties">
             <xsd:complexType>
                   <xsd:sequence>
                   <xsd:element ref="latitude" minOccurs="1"
                   maxOccurs="1"/>
                   <xsd:element ref="longitude" minOccurs="1"
                   maxOccurs="1"/>
                   ...
                   </xsd:sequence>
             </xsd:complexType>
             </xsd:element>
       </xsd:schema>
 </types>
       <portType name="SatellitePortType"
             wsrp:ResourceProperties=
             "tns:GenericSatelliteProperties">
       </portType>
</definitions>
```

Рис. 51. Свойства WS-ресурса в WSDL-файле

Мы начали с добавления базовых элементов веб-сервиса, элемента `service` и элемента `binding`, который ассоциирует `service` с элементом `portType`. Сам

элемент `portType` пока что не содержит в себе никаких операций, самой главной составляющей в нем является атрибут `wsrp:ResourceProperties`. Этот атрибут говорит о том, что любая операция, которую выполняет наш веб-сервис, совершается над определенным типом ресурса с состоянием, так, как это определено элементом `GenericSatelliteProperties`. Элемент `GenericSatelliteProperties` определен в элементе `schema`. Объединение этого ресурса с состоянием и нашего веб-сервиса и является требуемым WS-ресурсом.

Теперь добавим несколько операций по работе с ресурсом в базовые элементы WSDL-описания нашего веб-сервиса.

```
...
 <types>
        ...
        <xsd:element name="createSatellite">
              <xsd:complexType/>
        </xsd:element>
        <xsd:element name="createSatelliteResponse"> <xsd:complexType>
              <xsd:sequence>
              <xsd:element ref="wsa:EndpointReference"/>
              </xsd:sequence>
        </xsd:complexType> </xsd:element>
        ...
</types>
<message name="CreateSatelliteRequest">
 <part name="request" element="tns:createSatellite">
</message>
<message name="CreateSatelliteResponse">
 <part name="response" element="tns:createSatelliteResponse"/>
</message>
<portType name="SatellitePortType" wsrp:ResourceProperties=
              "tns:GenericSatelliteProperties">
 <operation name="createSatellite">
        <input message="tns:CreateSatelliteRequest"
              wsa:Action="http://example.com/CreateSatellite" />
        <output message="tns:CreateSatelliteResponse"
              wsa:Action="http://example.com/CreateSatelliteResponse"/>
 </operation>
</portType>
```

Рис. 52. Описание операций по работе с WS-ресурсом

Нужно обратить внимание на одну вещь. Вместо того чтобы возвращать простое значение, сервис возвращает *EndpointReference, где содержится ссылка на вновь созданный WS-ресурс*. Посмотрим, как это используется в SOAP-сообщении.

Запрос:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Header/>
 <SOAP-ENV:Body>
        <createSatellite xmlns="http://example.com/satellite"/>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Ответ:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.
org/soap/envelope/">
 <SOAP-ENV:Header/>
 <SOAP-ENV:Body>
      <wsa:EndpointReference
      xmlns:wsa="http://www.w3.org/2005/02/addressing"
      xmlns:sat="http://example.org/satelliteSystem">
           <wsa:Address>http://example.com/satellite</wsa:Address>
           <wsa:ReferenceProperties>
                <sat:SatelliteId>SAT9928</sat:SatelliteId>
           </wsa:ReferenceProperties>
      </wsa:EndpointReference>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Рис. 53. Запрос на создание ресурса

У нас еще нет фактического объекта, поэтому сам запрос направляется согласно URI, записанному в WSDL-файле, и мы определили содержание запроса как простой элемент `createSatellite`. После того, как мы послали запрос на создание нового спутника, сервер создает ссылку на новый WS-ресурс и отсылает ее обратно в форме *EndpointReference*.

Заметим, что элемент *Address* в *EndpointReference* указывает на тот же самый URI, который мы поместили в WSDL-файл, поэтому информация отправляется, как и раньше, по прежнему адресу; только информации стало больше. Необходимо также отметить, что мы имеем дело не с обычной конечной ссылкой. В элементе *EndpointReference* указан идентификатор, который обязательно должен использоваться при идентификации WS-ресурса. В связи с этим, этот элемент можно рассматривать как действительно квалифицированную конечную ссылку на WS-ресурс.

Элемент wsa:Action не является частью созданной ссылки на конечную точку; он изменяется в зависимости от того, что мы пытаемся сделать. В рассматриваемом случае мы используем действие GetResourceProperty. Элемент wsa:To выбирает значение по адресу wsa:Address (из ссылки на крайнюю точку), и любые значения wsa:ReferencsProperty непосредственно переносятся в заголовок (Header).

Вы заметили, что мы не обсуждали значение SatelliteId. Это было сделано специально. Любая, содержащаяся в ссылке на крайнюю точку информация, относящаяся к идентификации конкретного WS-ресурса, должна игнорироваться вашим приложением, вы просто передаете ее при отправлении сообщений. В соответствии со спецификацией, считается некорректным даже попытка интерпретации значения SatelliteId. Безусловным является предположение, что оно

передается как «черный ящик», ведущий независимую и недоступную для наблюдений жизнь.

```
    <SOAP-ENV:Envelope>
    <SOAP-ENV:Header>
     <wsa:Action>
          http://docs.oasis-open.org/wsrf/2004/06/WS-
ResourceProperties/GetResourceProperty
     </wsa:Action>
     <wsa:To SOAP-ENV:mustUnderstand="1">
          http://example.com/satellite
     </wsa:To>
     <sat:SatelliteId>SAT9928</sat:SatelliteId>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
     <wsrp:GetResourceProperty
          xmlns:satProp="http://example.com/satellite">
          satProp:altitude
     </wsrp:GetResourceProperty>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

Рис. 54. Запрос на получение информации о состоянии ресурса

Таким образом, мы рассмотрели основные, на сегодняшний день, стандарты технологии веб-сервисов, которые применяются как в коммерческих так и в научных распределенных вычислительных системах. Одной из крупнейших областей, где стандарты веб-сервисов второго поколения нашли свое применение, стала область грид-технологий, о которой мы поговорим в главе 11.

10. Технологии одноранговых сетей

10.1 Основы технологии одноранговых сетей

Технология одноранговых сетей (или P2P-сетей от анг. peer-to-peer – равный-к-равному) обеспечивает формирование РВС на базе принципа децентрализации [67], где разделение вычислительных ресурсов и сервисов производится напрямую посредством прямого взаимодействия между участниками сети друг с другом, без участия центрального сервера [52]. Одноранговые вычислительные сети, в какой-то степени, являются противоположностью клиент-серверным архитектурам (таким как CORBA, RMI). Можно сказать, что основным принципом P2P-сетей является воплощение идеи коммунизма – «Каждый – по способностям, каждому – по потребностям!».

10.1.1 Сравнение P2P и клиент-серверной технологий

В отличие от традиционной клиент-серверной архитектуры в P2P-сетях каждый узел, входящий в вычислительную сеть, может являться как клиентом, так и сервером, предоставляя или используя ресурсы сети. На рис. 55 представлены связи в сетях с P2P и с централизованной архитектурой [49].

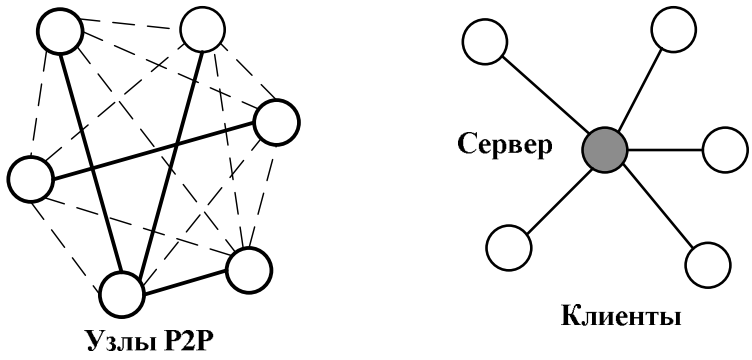


Рис. 55. Сравнение вида связей P2P и централизованной (клиент-серверной) архитектур

Можно выделить следующие проблемы клиент-серверной архитектуры, связанные с наличием централизованного сервера, обеспечивающего обработку запросов от множества клиентов:

– *Проблемы масштабируемости*. При увеличении количества клиентов растут требования к мощности сервера и пропускной способности канала. Единственным вариантом решения данной задачи является наращивание пропускной способности канала до сервера и использование более высокопроизводительных решений для аппаратной платформы сервера;

– *Зависимость*. Стабильная работа всех клиентов зависит от загруженности и функционирования одного сервера. При выходе из строя или отключении сервера, клиенты не смогут выполнять функциональные обязанности.

Этим проблемам можно противопоставить следующие преимущества P2P:

– отсутствие зависимости от централизованных сервисов и ресурсов;
– система может пережить серьезное изменение в структуре сети;
– высокая масштабируемость модели одноранговых вычислений.

На рис. 56 представлена диаграмма, позволяющая сравнить принципы этих архитектур. На основе этой диаграммы можно сделать предположение, что нет четкой границы между архитектурой P2P и клиент-серверной архитектурой. Обе модели могут быть построены с реализацией в различной степени каких-либо характеристик (например, управляемость), функциональности, структур (например, иерархии и сети) и др. Они могут выполняться на различных платформах (Интернет, Интранет и др.), и обе могут служить в качестве базы для приложений. Таким образом, понятие P2P чрезвычайно тесно переплетено с другими существующими технологиями.

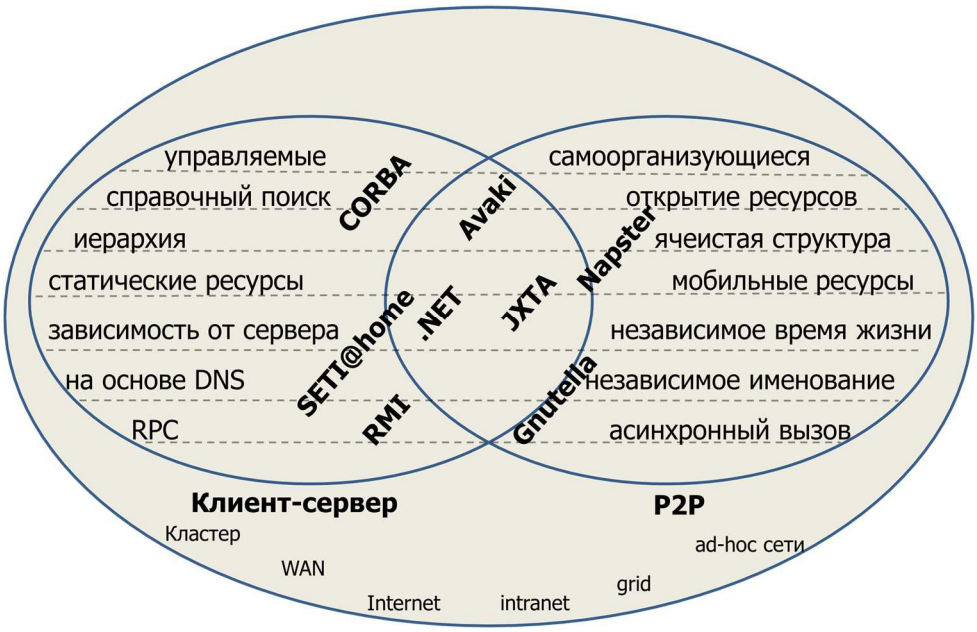


Рис. 56. Сравнение P2P и централизованной (клиент-серверной) архитектур

10.1.2 Задачи P2P сетей

Можно выделить следующие основные задачи, которые с легкостью решают P2P сети:

1. *Уменьшение/распределение затрат*. Серверы централизованных систем, которые обслуживают большое количество клиентов, обычно несут на себе основной объем затрат ресурсов (денежных, вычислительных и др.) на

поддержание вычислительной системы. P2P архитектура может помочь распределить эти затраты между узлами сети. Так как узлы, как правило, автономны, важно, чтобы затраты были распределены справедливо.

2. *Объединение ресурсов.* Каждый узел в P2P-системе обладает определенными ресурсами (вычислительные мощности, объем памяти). Приложения, которым необходимо большое количество ресурсов, например ресурсозатратные задачи моделирования или распределенные файловые системы, используют возможность объединения ресурсов всей сети для решения своей задачи. При этом важны как объем дискового пространства для хранения данных, так и пропускная способность сети.
3. *Повышенная масштабируемость.* Поскольку в сетях P2P отсутствует сильный центральный механизм, важной задачей является повышение масштабируемости и надежности системы. Масштабируемость определяет количество систем, которые могут быть достигнуты из одного узла, сколько систем могут функционировать одновременно, сколько пользователей может пользоваться сетью, сколько памяти может быть использовано. Надежность сети определяется такими параметрами как количество сбоев в работе сети, отношение времени простоя к общему времени работы, доступностью ресурсов и т.д. Таким образом, основной проблемой становится разработка новых алгоритмов обнаружения ресурсов, на которых базируются новые P2P платформы.
4. *Анонимность.* Бывает, пользователь не желает, чтобы другие пользователи или поставщики услуг знали о его нахождении в сети. При использовании центрального сервера трудно обеспечить анонимность, так как серверу, как правило, необходимо идентифицировать клиента, по крайней мере через интернет адрес. При использования P2P-сети пользователи могут избежать предоставления любой информацию о себе. FreeNet является ярким примером того, как механизмы анонимности могут быть встроены в P2P-приложения. В системе FreeNet используется схема переадресации сообщений при которой невозможно отследить первого отправителя. Также степень анонимности увеличивается за счет использования вероятностных алгоритмов.

10.1.3 Основные элементы P2P сетей

Пир (Peer) – это фундаментальный составляющий блок любой одноранговой сети:

– каждый пир имеет уникальный идентификатор;

– каждый пир принадлежит одной или нескольким группам;
– каждый пир может взаимодействовать с другими пирами, как в своей так и в других группах.

Можно выделить следующие виды пиров:

– *Простой пир:* обеспечивает работу конечного пользователя, предоставляя ему сервисы других пиров и обеспечивая предоставление ресурсов пользовательского компьютера другим участникам сети;
– *Роутер:* обеспечивает механизм взаимодействия между пирами, отделенными от сети брандмауэрами или NAT-системами.

Группа пиров – это набор пиров, сформированный для решения общей задачи или достижения общей цели. Группы пиров могут предоставлять членам своей группы такие наборы сервисов, которые недоступны пирам, входящим в другие группы.

Сервисы – это функциональные возможности, которые может привлекать отдельный пир для полноценной работы с удаленными пирами. В качестве примера сервисов, которые может предоставлять отдельный пир можно указать сервисы передачи файлов, предоставления информации о статусе, проведения вычислений и др. *Сервисы пира* – это такие сервисы, которые может предоставить конкретный узел P2P. Каждый узел в сети P2P предоставляет определенные функциональные возможности, которыми могут воспользоваться другие узлы. Эти возможности зависят от конкретного узла и доступны только тогда, когда узел подключен к сети. Как только узел отключается, его сервисы становятся недоступны. *Сервисы группы* – это функциональные возможности, предоставляемые группой входящим в нее узлам. Возможности могут предоставляться несколькими узлами в группе, для обеспечения избыточного доступа к этим возможностям. Как только к группе подключается узел, обеспечивающий необходимый сервис, он становиться доступной для всей группы.

P2P — это не только сети, но еще и *сетевой протокол*, обеспечивающий возможность создания и функционирования сети равноправных узлов, их взаимодействия. Множество узлов, объединенных в единую систему и взаимодействующих в соответствии с протоколом P2P, образуют пиринговую сеть. P2P относятся к прикладному уровню сетевых протоколов и являются *наложенной сетью*, использующей существующие транспортные протоколы стека TCP/IP – TCP или UDP. *Протоколы* сети P2P обеспечивают:

– поиск узлов в сети;
– получение списка служб отдельного узла;

– получение информации о статусе узла;
– использование службы на отдельном узле;
– создание, объединение и выход из групп;
– создание соединений с узлами;
– маршрутизацию сообщений другим узлам.

Одну из удачных попыток стандартизации протоколов P2P предприняла компания Sun Microsystems в рамках проекта JXTA. *Платформа JXTA* позиционируется как базовая платформа для организации P2P сетей на основе гетерогенных вычислительных сетей.

10.2 Алгоритмы работы P2P сетей

10.2.1 Структура P2P сети

Структура P2P сети определяет принципы поиска новых узлов и замены узлов вышедших из состава сети новыми. Можно выделить два основных типа P2P сетей: централизованные и децентрализованные [46].

Централизованная структура P2P сети подразумевает наличие выделенного индексного сервера (*трекера*) собирающего информацию об узлах, входящих в P2P-сеть и обеспечивающего поиск и предоставление необходимых сервисов одних узлов другим. Первой P2P сетью с централизованной структурой была сеть Napster, центральный узел которой отвечал за хранение идентификаторов всех узлов в сети и списков файлов, доступных на каждом из узлов. Еще одним примером сети с централизованной структурой является сеть BitTorrent. Центральным узлом данной сети является *трекер* – сервер, содержащий информацию о списке узлов, подключенных к сети, и сервисах, предоставляемых каждым узлом (например, список файлов, доступных для загрузки с данного узла). Для получения необходимого файла, узел посылает трекеру запрос, содержащий уникальный идентификатор необходимого файла. На данный запрос трекер возвращает список узлов, на которых доступен требуемый файл. Естественно, степень централизованности системы BitTorrent значительно меньше, чем была у системы Napster, т.к. BitTorrent позволяет работать сразу с большим количеством трекеров, в то время как Napster предполагал наличие только одного центрального сервера.

Децентрализованная структура P2P сети предполагает отсутствие выделенного сервера. Поиск и предоставление сервисов производится путем процедуры пошагового поиска, в которой могут участвовать все узлы, входящие в сеть. Типичным примером одноранговой сети с децентрализованной структу-

рой является система Gnutella. В данной сети, обнаружение и подключение к узлам сети происходит посредством процедуры случайного обхода. Каждый узел содержит таблицу соседей, содержащую IP адрес и порт известного узла Gnutella. При запуске новый узел Gnutella переходит в режим начальной загрузки, в котором посредством одного из доступных источников (список узлов на одном из узлов интернет; внутренний предустановленный список узлов и др.) формирует начальный список соседей. После чего, соседям высылается сообщение обнаружения, пересылаемое далее по цепочке систем. Таким образом обеспечивается обнаружение ресурсов, предоставляемых всеми узлами подключенными к сети.

10.2.2 Алгоритмы работы P2P сетей

Поскольку сегодня в перенасыщенном информацией мире задача полноты поиска отводится на второй план, то главная задача поиска в пиринговых сетях сводится к быстрому и эффективному нахождению наиболее релевантных откликов на запрос, передаваемый от узла всей сети. В частности, актуальна задача — уменьшение сетевого трафика, порождаемого запросом (например, пересылки запроса по многочисленным узлам), и в то же время получение наилучших характеристик выдаваемых документов, т.е. наиболее качественного результата. Для решения данной задачи применяют несколько подходов.

Централизованный индекс: узлы публикуют информацию о своих сервисах в центральном индексе. При подключении к сети, пир формирует список документов и сервисов, доступных на узле. Данный список передается на центральный сервер, хранящий полную информацию о текущем состоянии вычислительной сети. Когда любой узел P2P-сети хочет получить информацию о том, какие ресурсы доступны, он отправляет поисковый запрос на центральный сервер. В ответ на этот запрос, центральный сервер выдает список доступных ресурсов и адреса узлов, на которых они располагаются. Недостатком является малая масштабируемость сети (в связи с возрастающей нагрузкой на центральный сервер) и высокая зависимость от центрального сервера. Данные недостатки решаются посредством масштабирования центрального сервера (например, использование нескольких независимых центральных серверов). *Пример*: Napster, BitTorrent, eDonkey.

Широковещательные запросы (Breadth Search): процесс поиска информации производится посредством отправки поискового запроса всем подключенным узлам, известным отправителю запроса. Данный запрос транслируется всем дальнейшим узлам, пока не получен ответ или не достигнут предел коли-

чества пересылок. Недостатками данного метода является большая нагрузка на сеть, генерируемая поисковыми запросами, а также негарантируемая достижимость результата. Однако есть метод, позволяющий избежать перегрузки всей сети сообщениями. Он заключается в приписывании каждому запросу параметра времени жизни (time-to-live, TTL). Параметр TTL определяет максимальное число переходов, по которому можно пересылать запрос. *Пример*: Gnutella

Маршрутизация документов. Данный метод обеспечивает поиск документов без участия центрального индекса, средствами самой вычислительной сети. В основе данного метода лежит принцип присвоения уникальных идентификаторов каждому узлу вычислительной сети, а также каждому ресурсу (документу, сервису и др.), который данная сеть может предоставлять.

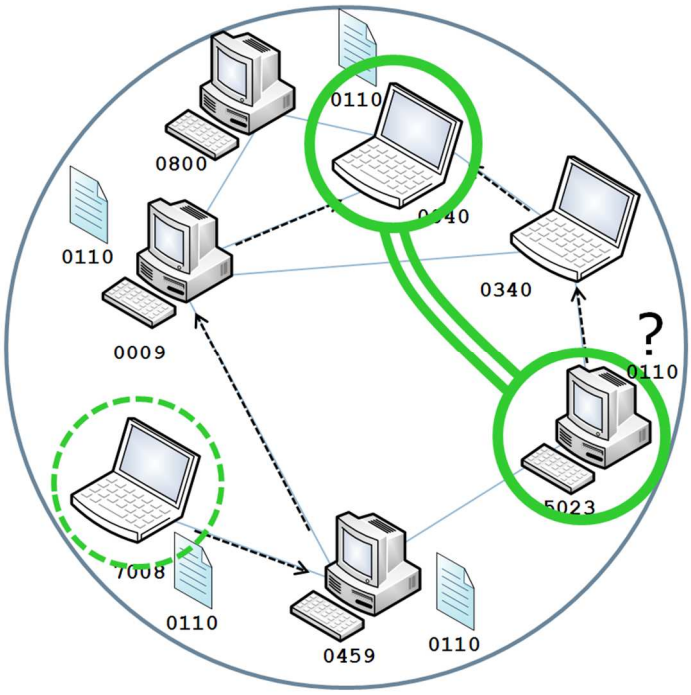


Рис. 57. Трассировка и поиск ресурса посредством алгоритма маршрутизации документов

На рисунке 57 представлен пример работы алгоритма маршрутизации документа. Когда какой-либо пир (на рисунке – узел 7008) производит публикацию ресурса в сети, каким-либо образом вычисляется идентификатор данного ресурса (на рисунке – документ 0110). При этом, идентификаторы узлов сети и ресурсов имеют единую область возможных значений. Далее, копия данного ресурса (или ссылка на него) трассируется к узлу, который имеет наиболее похожий идентификатор (на рисунке показана трасса документа 0110: узлы 7008 – 0459 – 0009 – 0040). Данная процедура производится следующим образом:

1. Производится сравнение идентификатора ресурса с идентификаторами всех соседних узлов.

2. Если идентификатор *текущего* узла ближе всего (по некоторой метрике) к идентификатору документа, то процесс трассировки завершается.
3. Если один из идентификаторов соседних узлов ближе к идентификатору документа, чем идентификатор текущего узла, то ссылка на данный ресурс *копируется* на узел с более близким идентификатором и процесс повторяется с п. 1.

В результате данной процедуры ссылка на документ отправляется на вычислительный узел, идентификатор которого больше всего соответствует идентификатору документа, среди соседей начального узла (узел 0040 на рисунке).

Поиск ресурса производится по аналогичному алгоритму, но вместо копирования документа происходит трансляция запроса (в запросе содержится идентификатор запрашиваемого ресурса, например 0110 на рисунке 57) от узла, инициировавшего запрос (на рисунке – узел 5203) к узлу, идентификатор которого ближе всего соответствует идентификатору запрашиваемого документа. Соответственно, в процессе трансляции данного запроса должен появиться такой узел, который хранит информацию об интересуемом документе, если данный документ находится в соответствующей части сети.

К недостаткам такого подхода можно отнести необходимость знания точного идентификатора документа, который необходимо найти в сети (невозможность поиска по отдельным атрибутам документа), а также возможность образования «островов», затрудняющих алгоритм поиска. *Примеры*: FreeNet, протокол DHT.

Отдельно стоит отметить *алгоритм обхода брандмауэров и NAT*, применяющийся при работе с конечными узлами, установка внешнего соединения с которыми затруднена или невозможна. Для обеспечения обмена информацией, необходимо использовать возможности узла-роутера для трансляции сообщений во внешнюю сеть и получения информации из внешней сети. Узел-роутер буферизует всю информацию, предназначенную конечному узлу до момента, пока не получит от него запрос на загрузку. Как только получен запрос на загрузку, узел-роутер отвечает, выгружая всю накопившуюся информацию на конечный узел.

10.3 Применение технологий P2P

Наибольшее распространение одноранговых сетей наблюдается в системах, обрабатывающих большие объемы данных и обеспечивающих индивиду-

альный обмен информацией между пользователями. В настоящий момент, технологии P2P наиболее ярко представлены в 3-х направлениях:

- *Распределенные вычисления*: разбиение общей задачи на большое число независимых в обработке подзадач (проекты на платформе BOINC [9]);
- *Файлообменные сети*: P2P выступают альтернативой FTP-архивам, которые утрачивают перспективу ввиду значительных информационных перегрузок (однако требуются эффективные механизмы поиска) (Gnutella [33], eDonkey, BitTorrent [8]);
- *Приложения для совместной работы*: требуют обеспечения прозрачных механизмов для совместной работы. (Skype [36,59], Groove [17]).

10.3.1 Распределенные вычисления

В основном, к данному типу проектов относят системы типа проекта SETI@home (распределенный поиск внеземных цивилизаций), который продемонстрировал огромный вычислительный потенциал для распараллеливаемых задач. В настоящий момент в нем принимают участие свыше трех миллионов пользователей на бесплатной основе. Данная система основана на платформе *BOINC*.

BOINC (англ. Berkeley Open Infrastructure for Network Computing — открытая программная платформа Беркли для распределённых вычислений) — некоммерческое межплатформенное ПО для организации распределённых вычислений. Система состоит из двух основных частей:

- сервер BOINC – это набор PHP-сценариев для организации и управления проектом: регистрация участников, распределение заданий, получение результатов;
- клиент BOINC – это пользовательское приложение, позволяющее участвовать в одном или нескольких проектах. Обычно представляет собой хранитель экрана, который производит вычисления в моменты простоя компьютера.

Наиболее популярные проекты, реализованные на основе BOINC:

- SETI@home — анализ радиосигналов с радиотелескопа Аресибо для поиска инопланетных цивилизаций.
- Einstein@Home — проверка гипотезы Альберта Эйнштейна о гравитационных волнах с помощью анализа гравитационных полей пульсаров или нейтронных звёзд.

- Climate Prediction — построение модели климата Земли для предсказания его изменений на 50 лет вперёд.
- World Community Grid — Различные проекты. Организатор — IBM.
- Malaria Control Project — Контроль распространения Малярии в Африке (AFRICA@home).
- Predictor@home — моделирование 3-хмерной структуры белка из последовательностей аминокислот.
- LHC@home — расчёты для ускорителя заряженных частиц в CERN (Centre Europeen de Recherche Nucleaire).

10.3.2 Файлообменные сети

По статистическим данным на конец 2006 года объем трафика, генерируемого файлообменными сетями на базе P2P-сетей, составил более 70% всего сетевого трафика. На сегодняшний день существует большое число P2P-сетей, ориентированных на обмен файлами между пользователями. Они могут развиваться и функционировать как в глобальном сетевом пространстве, так и в отдельных подсетях.

Самым ярким примером таких сетей, является система BitTorrent. Протокол BitTorrent был разработан в 2001 Брэмом Коэном. В соответствии с протоколом BitTorrent файлы передаются не целиком, а частями, причем каждый клиент, закачивая эти части, в это же время отдает их другим клиентам, что снижает нагрузку и зависимость от каждого клиента-источника и обеспечивает избыточность данных.

Если узел хочет опубликовать файл или набор файлов, то программа-клиент BitTorrent сети разделяет передаваемые файлы на части и создает *файл метаданных* (идентификатор раздачи), который содержит следующую информацию:

- URL трекера;
- Общая информация о файлах (имя, длина и пр.);
- Хеш-суммы SHA1 сегментов раздаваемых файлов;
- Passkey пользователя – ключ, который однозначно определяет пользователя загрузившего файл;
- Хеш-суммы файлов целиком (не обязательно);
- Альтернативные источники – адреса альтернативных трекеров, на которых можно найти информацию по данному файлу (не обязательно).

Алгоритм загрузки документа производится следующим образом:

- клиент подключается к трекеру по URL из файла метаданных;
- сообщает хеш-идентификатор требуемого файла;
- получает адреса пиров скачивающих и раздающих данный файл;
- клиенты соединяются между собой и обмениваются информацией без участия трекера.

В последнее время стала распространяться альтернативная технология поиска и загрузки документов на основе «магнитных ссылок» (magnet links) и подхода распределенных хеш-таблиц (Distributed Hash Table — DHT) по сути дела представляющих собой реализацию алгоритма маршрутизации документов, описанную ранее. Причина возникновения этой технологии – дальнейшее развитие деперсонализации и попытка торрент-трекеров защититься от юридического преследования правообладателей. Торрент-файл для такой раздачи создаётся без адреса трекера и клиенты находят друг друга через распределенные хеш-таблицы.

DHT – это система распределенного хранения данных о скачиваемых файлах. Все клиенты, подключенные к DHT-сети и сами становятся «узлами», чем-то вроде мини-трекеров. Каждый узел имеет уникальный идентификатор – «node ID». Все узлы хранят информацию об узлах, «близких к ним», кроме того узел должен хранить информацию о пирах в раздачах, чей хеш напоминает «node ID».

При этом торренты представляют собой Magnet-ссылки, которые в основном идентифицируют файлы не по их расположению или имени, а по содержанию, точнее — по хеш-коду.

Одно из преимуществ magnet-ссылок — их открытость и независимость от платформы: ссылка может быть использована для загрузки файла при помощи разнообразных приложений на практически всех операционных системах. Благодаря тому, что magnet-ссылка представляет собой короткую строку текста, пользователи могут использовать обычные операции копирования-вставки и отправить ее по электронной почте или программе мгновенного обмена сообщениями.

10.3.3 Приложения для совместной работы

Приложения для совместной работы, это такие приложения, которые обеспечивают возможность общения, совместной работы и т. п. различных географически-распределенных пользователей.

- Jabber: мгновенный обмен сообщениями

- Skype: голосовое общение, видеоконференции, приложения для совместной работы
- Groove: система для совместной работы

Самой популярной на сегодняшний день службой Интернет-телефонии является Skype, созданная в 2003 году шведом Никласом Зеннстромом и датчанином Янусом Фриисом, авторами известной пиринговой сети KaZaA. В настоящее время Skype принадлежит корпорации Microsoft, которая приобрела ее за $8,5 млрд. в мае 2011 года.

Структура Skype-сети состоит из обычных узлов (normal/ordinal node/host/nest), обычно обозначаемых аббревиатурой SC (Skype Client), и super-узлов (super node/host/nest), которым соответствует аббревиатура SN. Любой узел, имеющий публичный IP-адрес (т.е. тот, который маршрутизируется в Интернет) и обладающий достаточно широким каналом, автоматически становится super-узлом и пропускает через себя трафик обычных узлов, помогая им преодолеть защиты типа брандмауэров или трансляторов сетевых адресов (NAT) и равномерно распределяя нагрузку между хостами. Единственным централизованным элементом является Skype-login сервер, отвечающий за процедуру авторизации Skype-клиентов и гарантирующий уникальность «позывных» для всей распределенной сети.

Такая архитектура позволяет установить и использовать прямое соединение между любыми вычислительными узлами в одной подсети, увеличивая скорость и уменьшая накладные расходы. Связь между узлами через интернет осуществляется не напрямую, а через цепочку super-узлов. «Серверов» в общепринятом смысле этого слова (таких, например, как в сети eDonkey) в Skype-сети нет и любой узел с установленным Skype-клиентом является потенциальным сервером, которым он автоматически становится при наличии достаточных системных ресурсов (объема оперативной памяти, быстродействия процессора и пропускной способности сетевого канала, не защищенного никакими средствами защиты).

Каждый узел Skype-сети хранит перечень IP-адресов и портов известным ему super-узлов в динамически обновляемых кэш-таблицах (Host Cache Tables, HC-tables). Начиная с версии Skype 1.0, кэш-таблицы представляют собой простой XML-файл, в незашифрованном виде записанный на диске в домашней директории пользователя.

Таким образом, в состав системы входят следующие элементы:

- *Skype-login сервер* – единственный централизованный элемент Skype-сети, обеспечивающий авторизацию Skype-клиентов.
- *Обычный узел (Skype Client)* – обычный конечный узел в сети.
- *Супер-узел (Super node)* – узлы, играющие роль роутеров в сети Skype. Любой узел, обладающий публичным IP и обладающий широким каналом становится супер-узлом.
- *Выделенные узлы* для установки связи со стационарными телефонными линиями.

10.4 Достоинства и недостатки P2P

Можно выделить следующие основные преимущества одноранговых сетей:

- высокая масштабируемость, связанная с равномерным распределением вычислительной нагрузки на всех участников сети;
- стабильность работы сети, обусловленная отсутствием «узкого места» – выделенного сервера, обрабатывающего все сетевые запросы;
- возможность объединения ресурсов отдельных участников сети, и их предоставление другим участникам;
- распределение совокупных затрат на предоставление ресурсов между участниками сети.

С другой стороны, отдельно стоит упомянуть о следующих недостатках и особенностях функционирования P2P-сетей:

- в одноранговых сетях не может быть обеспечено гарантированное качество обслуживания: любой узел, предоставляющий те или иные сервисы, может быть отключен от сети в любой момент;
- индивидуальные технические характеристики узла могут не позволить полностью использовать ресурсы P2P сети (каждый из узлов обладает индивидуальными техническими характеристиками что, возможно, будет ограничивать его роль в P2P-сети и не позволят полностью использовать ее ресурсы: низкий рейтинг в torrent-сетях, LowID в eDonkey могут значительно ограничить ресурсы сети, доступные пользователю);
- при работе того или иного узла через брандмауэр может быть значительно снижена пропускная способность передачи данных в связи с необходимостью использования специальных механизмов обхода;

- участниками одноранговых сетей в основном являются индивидуальные пользователи, а не организации, в связи с чем возникают вопросы безопасности предоставления ресурсов: владельцы узлов P2P-сети, скорее всего, не знакомы друг с другом лично, предоставление ресурсов происходит без предварительной договоренности;
- при увеличении числа участников P2P сети может возникнуть ситуация значительного возрастания нагрузки на сеть (как с централизованной, так и с децентрализованной структурой);
- в случае применения сети типа P2P приходится направлять значительные усилия на поддержку стабильного уровня ее производительности, резервное копирование данных, антивирусную защиту, защиту от информационного шума и других злонамеренных действий пользователей.

11. Технологии Грид

Термин «грид» был введен в обращение Яном Фостером в начале 1998 года публикацией книги «Грид. Новая инфраструктура вычислений» [27]:

> ***Грид*** *– это система, которая координирует распределенные ресурсы посредством стандартных, открытых, универсальных протоколов и интерфейсов для обеспечения нетривиального качества обслуживания (QoS – Quality of Service).*

Хотя в последнее десятилетие базовая идея грид не претерпела существенных изменений, всеобъемлющего определения грид не существует до сих пор [62].

11.1 Архитектура Грид

Основной идеей, заложенной в концепции грид-вычислений, является централизованное удаленное предоставление ресурсов, необходимых для решения различного рода вычислительных задач. В каком-то смысле, концепция грид-вычислений идея рифмуется с концепцией электросети (англ. Power Grid): нам не важно, откуда к нам в розетку приходит электричество. Независимо от этого мы можем подключить к электросети утюг, компьютер или стиральную машину. Также и в идеологии грид: мы можем запустить любую задачу с любого компьютера или мобильного устройств на вычисление, ресурсы же для этого вычисления должны быть автоматически предоставлены на удаленных высокопроизводительных серверах, независимо от типа нашей задачи.

С более практической точки зрения, основная задача, лежащая в основе концепции грид, это согласованное распределение ресурсов и решение задач в условиях динамических, многопрофильных *виртуальных организаций*. Распределение ресурсов, в котором заинтересованы разработчики грид, это не обмен файлами, а прямой доступ к компьютерам, программному обеспечению, данным и другим ресурсам, которые требуются для совместного решения задач и стратегий управления ресурсами, возникающих в промышленности, науке и технике. *Виртуальной организацией (ВО)* называют ряд отдельных людей или учреждений, объединенных едиными правилами коллективного доступа к распределенным вычислительным ресурсам [31]. Для организации работы в рамках таких ВО возникает необходимость в следующем:

1. в гибких механизмах разделения ресурсов, начиная от клиент-серверных заканчивая одноранговыми;
2. в развитой системе контроля используемых ресурсов, включая контроль над мелкомодульными и другими методами доступа и использование локальных и глобальных подходов;
3. в распределенном доступе к различным ресурсам, начиная от программ, файлов и данных заканчивая компьютерами, сенсорами и сетями;
4. в различных моделях использования ресурсов (от однопользовательских до многопользовательских, от высокопроизводительных до мало затратных) и, следовательно, включающих регулирование качества предоставляемого обслуживания, планирование, перераспределение и ведение учета ресурсов.

Анализ альтернативных технологий построения распределенных вычислительных систем, проведенный в [31], показал, что их применение не позволяет в полной мере достичь исполнения всех требований, указанных выше. В соответствии с этим была предложена альтернативная архитектура грид. Исследования и разработки в сообществе грид привели к разработке протоколов, сервисов и инструментария, направленного именно на те проблемы, которые возникают при попытке создания масштабируемых ВО. Эти технологии включают в себя:

1. решения по безопасности, поддерживающие управление сертификацией и политиками безопасности, когда вычисления производятся несколькими организациями;
2. протоколы управления ресурсами и сервисами, поддерживающие безопасный удаленный доступ к вычислительным ресурсам и ресурсам данных, а также перераспределение различных ресурсов;
3. протоколы запроса информации и сервисы, обеспечивающие настройку и мониторинг состояния ресурсов, организаций и сервисов;
4. сервисы обработки данных, обеспечивающие поиск и передачу наборов данных между системами хранения данных и приложениями.

Выделяют следующие уровни архитектуры грид:

1. *Базовый уровень (Fabric)* – содержит различные ресурсы, такие как компьютеры, устройства хранения, сети, сенсоры и др.
2. *Связывающий уровень (Connectivity)* – определяет коммуникационные протоколы и протоколы аутентификации.
3. *Ресурсный уровень (Resource)* – реализует протоколы взаимодействия с ресурсами РВС и их управления.

4. *Коллективный уровень (Collective)* – управление каталогами ресурсов, диагностика, мониторинг;
5. *Прикладной уровень (Applications)* – инструментарий для работы с грид и пользовательские приложения.

На *базовом уровне* определяются службы, обеспечивающие непосредственный доступ к ресурсам, использование которых распределено посредством протоколов Грид.

1. Вычислительные ресурсы предоставляют пользователю Грид-системы (точнее говоря, задаче пользователя) процессорные мощности. Вычислительными ресурсами могут быть как кластеры, так и отдельные рабочие станции. При всем разнообразии архитектур любая вычислительная система может рассматриваться как потенциальный вычислительный ресурс Грид-системы.
2. Ресурсы памяти представляют собой пространство для хранения данных. Для доступа к ресурсам памяти также используется программное обеспечение промежуточного уровня, реализующее унифицированный интерфейс управления и передачи данных.
3. Информационные ресурсы и каталоги являются особым видом ресурсов памяти. Они служат для хранения и предоставления метаданных и информации о других ресурсах Грид-системы.
4. Сетевой ресурс является связующим звеном между распределенными ресурсами Грид-системы. Основной характеристикой сетевого ресурса является скорость передачи данных.

Связывающий уровень определяет коммуникационные протоколы и протоколы аутентификации, обеспечивая передачу данных между ресурсами базового уровня. Связывающий уровень грид основан на стеке протоколов TCP/IP:

1. Интернет (IP, ICMP);
2. Транспортные протоколы (TCP, UDP);
3. Прикладные протоколы (DNS, OSRF…).

Ресурсный уровень реализует протоколы, обеспечивающие выполнение следующих функций:

- согласование политик безопасности использования ресурса;
- процедура инициации ресурса;
- мониторинг состояния ресурса;
- контроль над ресурсом;
- учет использования ресурса.

Отдельно выделяются 2 типа протоколов ресурсного уровня:

1. *Информационные протоколы* – используются для получения информации о структуре и состоянии ресурса.
2. *Протоколы управления* – используются для согласования доступа к разделяемым ресурсам, определяя требований и допустимых действий по отношению к ресурсу (например, поддержка резервирования, возможность создания процессов, доступ к данным).

Коллективный уровень отвечает за глобальную интеграцию различных наборов ресурсов и может включать в себя службы каталогов; службы совместного выделения, планирования и распределения ресурсов; службы мониторинга и диагностики ресурсов; службы репликации данных.

На *прикладном уровне* располагаются пользовательские приложения, исполняемые в среде ВО. Они могут использовать ресурсы, находящиеся на любых нижних слоях архитектуры Грид.

11.2 Стандарты Грид

Ключевым моментом в разработке грид приложений является *стандартизация*, позволяющая организовать поиск, использование, размещение и мониторинг различных компонентов, составляющих единую виртуальную систему, даже если они предоставляются различными поставщиками услуг или управляются различными организациями [28]. К началу 2001 года в различных проектах были представлены различные методы реализации грид-вычислений. Но все они сходились в одном: для гибкого, прозрачного и надежного предоставления доступа к вычислительным ресурсам была предложена сервисно-ориентированная модель.

В 2001 году в качестве базы для создания стандарта архитектуры грид приложений была выбрана технология веб-сервисов. Данный выбор был обусловлен двумя основными достоинствами данной технологии. Во-первых, язык описания интерфейсов веб-сервисов WSDL (Web Service Definition Language) поддерживает стандартные механизмы для определения интерфейсов отдельно от их реализации, что в совокупности со специальными механизмами связывания (транспортным протоколом и форматом кодирования данных) обеспечивает возможность динамического поиска и компоновки сервисов в гетерогенных средах. Во-вторых, широко распространенная адаптация механизмов веб-сервисов означает, что инфраструктура, построенная на базе веб-сервисов, может использовать различные утилиты и другие существующие сервисы, такие

как различные процессоры WSDL, системы планирования потоков задач и среды для размещения веб-сервисов [30].

Разработанный стандарт архитектуры грид получил название *OGSA* (Open Grid Services Architecture – Открытая архитектура грид-сервисов) [18]. Он основывается на понятии грид-сервиса. *Грид-сервисом* называется сервис, поддерживающий предоставление полной информации о текущем состоянии (потенциально временного) экземпляра сервиса, а также поддерживающий возможность надежного и безопасного исполнения, управления временем жизни, рассылки уведомлений об изменении состояния экземпляра сервиса, управления политикой доступа к ресурсам, управления сертификатами доступа и виртуализации [30]. Грид-сервис поддерживает следующие стандартные интерфейсы.

1. *Поиск*. Грид приложениям необходимы механизмы для поиска доступных сервисов и определения их характеристик.
2. *Динамическое создание сервисов*. Возможность динамического создания и управления сервисами – это один из базовых принципов OGSA, требующий наличия сервисов создания новых сервисов.
3. *Управление временем жизни*. Распределенная система должна обеспечивать возможность уничтожения экземпляра грид-сервиса.
4. *Уведомление*. Для обеспечения работы грид приложения наборы грид сервисов должны иметь возможность асинхронно уведомлять друг друга о изменениях в их состоянии.

Первая реализация модели OGSA, разработанная в 2003 г., называлась *OGSI* (Open Grid Service Infrastructure). В связи с тем, что существовавшие тогда стандарты веб-сервисов (к которым относились WSDL, SOAP, UDDI) не могли обеспечить всех требований, предъявляемых разработчиками к функциональным возможностям грид-сервисов, при создании OGSI потребовалось модифицировать и расширить соответствующие стандарты [21]. Это привело к тому, что совместное использование веб-сервисов и грид-сервисов в одной среде стало невозможным, из-за несовместимости базовых стандартов [4].

Дальнейшие совместные усилия сообщества грид и организаций по разработке стандартов веб-сервисов привело к определению стандартов, соответствующих требованиям грид. В предложенном стандарте *WSRF* (*Web Service Resource Framework*) [13, 19, 22, 66] специфицированы универсальные механизмы для определения, просмотра и управления состоянием удаленного ресурса, что является критически-важным с точки зрения грид [25]. На сегодняшний день реализация модели OGSA посредством стандарта WSRF (и сопут-

ствующих стандартов, таких как WS-Notification и WS-Addressing) является наиболее распространенной в среде грид.

В настоящее время, существуют две системы, обеспечивающие инфраструктуру разработки грид-систем в соответствии со стандартами OGSA, реализованными посредством WSRF: Globus Toolkit [20] и UNICORE [48].

11.3 Система Globus

Globus – это проект по разработке и предоставлению инфраструктуры для грид-вычислений [20]. Становление данного проекта приходится на 1997 год, а его разработка продолжается и сегодня. Когда как первоначально Globus был развитием проекта I-WAY, в процессе развития, основной акцент был перенесен с поддержки высокопроизводительных вычислений в сторону сервисов поддержки виртуальных организаций.

Цель его создания – предоставление возможности приложениям работать с распределенными разнородными вычислительными ресурсами как с единой виртуальной машиной. Основная направленность данного проекта – вычислительные грид-системы. Под вычислительной грид-системой подразумевается инфраструктура аппаратных и программных ресурсов, реализующая надежный и полномасштабный доступ к высокопроизводительным вычислительным системам, независимо от географического расположения пользователей или ресурсов.

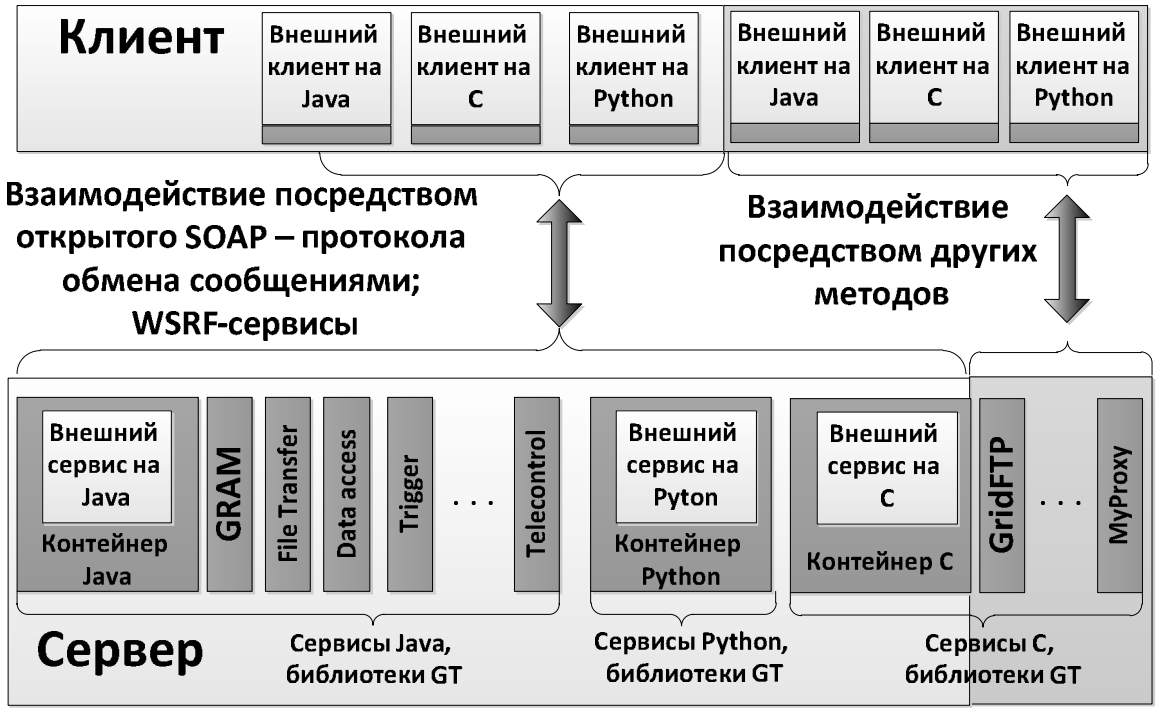


Рис. 58. Общая схема взаимодействия компонентов Globus Toolkit 4.0

Базовым элементом системы Globus выступает Globus Toolkit (инструментарий Globus), описывающий базовые сервисы и возможности, необходимые для создания вычислительных грид-систем [20]. Система Globus предоставляет

высокоуровневым приложениям доступ к сервисам, каждыйt из которых приложение или разработчик может использовать для достижения собственных целей. Такой метод работы может быть реализован только при высокой степени изолированности отдельных сервисов и четко определенном программном интерфейсе каждого предоставляемого сервиса.

Рассмотрим базовые сервисы, предоставляемые системой Globus на сегодняшний день (см. рис. 58).

1. Протокол GRAM (“Globus Toolkit Resource Allocation Manager” – Менеджер Распределения Ресурсов Globus Toolkit) используется для распределения вычислительных ресурсов и для контроля вычислений, с использованием данных ресурсов.
2. Расширенная версия протокола передачи файлов GridFTP используется для организации доступа к данным, включая вопросы безопасности и параллелизма высокоскоростной передачи данных.
3. Контейнеры для пользовательских сервисов, поддерживающие аутентификацию, управление состоянием, поиск и т.п. обеспечивающие поддержку стандартов WSRF, WS-Security, WS-Notification.
4. Сервисы аутентификации и безопасности соединений GSI (“Grid Security Infrastructure” – Инфраструктура Безопасности Грид).
5. Распределенный доступ к информации о структуре и состоянии системы распределенных вычислений.
6. Удаленный доступ к данным посредством последовательных и параллельных интерфейсов.
7. Создание, кэширование и поиск исполняемых ресурсов.
8. Библиотеки, для обеспечения взаимодействия сторонних приложений с GTK 4.0 и/или пользовательскими сервисами.

11.4 Система UNICORE

Проект UNICORE (Uniform Interface to Computing Resources – единый интерфейс к вычислительным ресурсам) зародился в 1997 году, и к настоящему моменту представляет собой комплексное решение, ориентированное на обеспечение прозрачного безопасно5го доступа к ресурсам грид [65].

Архитектура UNICORE 6 [64] формируется из клиентского, сервисного и системного слоев (см. рис. 59). Верхним слоем в архитектуре является клиентский слой. В нем располагаются различные клиенты, обеспечивающие взаимодействие пользователей с грид средой:

– UCC (Unicore Command Line Client – клиент командной строки для UNICORE): клиент, обеспечивающий интерфейс командной строки для постановки задач и получения результатов;
– URC (Unicore Rich Client – многофункциональный клиент UNICORE): клиент, основанный на базе интерфейса среды Eclipce, предоставляет в графическом виде полный набор всех функциональных возможностей системы UNICORE;
– HiLA (High Level API for Grid Applications – высокоуровневый программный интерфейс для приложений грид): обеспечивает разработку клиентов к системе UNICORE;
– Порталы: доступ пользователей к грид-ресурсам через интернет, посредством интеграции UNICORE и систем интернет-порталов.

Промежуточный сервисный слой содержит все сервисы и компоненты системы UNICORE, основанные на стандартах WSRF и SOAP. Шлюз – это компонент, обеспечивающий доступ к узлу UNICORE посредством аутентификации всех входящих сообщений [48]. Компонент XNJS обеспечивает управление задачами и исполнение ядра UNICORE 6. Регистр сервисов обеспечивает регистрацию и поиск ресурсов, доступных в грид-среде. Также, на уровне сервисного слоя обеспечивается поддержка безопасных соединений, авторизации и аутентификации пользователей.

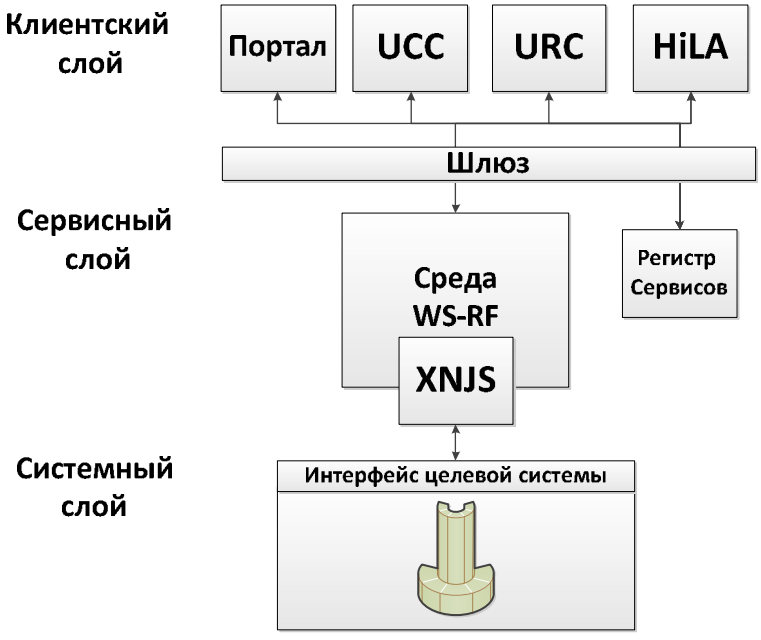


Рис. 59. Архитектура UNICORE 6

В основании архитектуры UNICORE лежит системный слой. Интерфейс целевой системы (TSI – Target System Interface) обеспечивает взаимодействие между UNICORE и отдельным ресурсом грид-сети. Он обеспечивает трансляцию команд, поступающих из грид-среды локальной системе.

Основным достоинством использования системы UNICORE 6 для разработки распределенных вычислительных систем можно считать наличие богатого арсенала различных клиентов, обеспечивающих взаимодействие пользователя с ресурсами вычислительной сети, а также развитых средств обеспечения безопасности при разработке грид-приложений.

11.5 Параметрические модели производительности Грид

Принцип работы и функциональность грид приложений значительно отличаются от обычных последовательных и параллельных систем. Основное отличие – это возможность агрегирования и совместного использования больших наборов гетерогенных ресурсов, распределенных между географически-разделенными областями. Во многих случаях это приносит большие выгоды, например, когда приложение требует ресурсов, недоступных в рамках одного узла, оно может затребовать ресурсы у других узлов, подключенных к грид [15].

Но такое сложное поведение несет в себе и определенные проблемы. К высоко-гетерогенной, динамически-формируемой распределенной среде очень трудно напрямую применить такие традиционные метрики производительности, как скорость вычислений, пропускная способность канала и др. В связи с этим, для оценки качества предоставляемого сервиса требуется использование специализированных метрик.

Предположим, что в грид-среде доступно m ресурсов и существует система распределения заданий τ, обеспечивающая распределение поставленных задач $j \in \tau$ на доступные ресурсы. В рамках данной системы, каждое задание может быть разбито на действия $k \in j$. Количество заданий в системе $|\tau|$; количество действий в задаче $|j|$. При постановке задачи, указывается время d_j, до которого пользователь желает получить результаты решения.

Каждая задача j и все ее действия $k \in j$ поступают в грид в момент времени r_j. В связи с тем, что грид работает в “online”-режиме, значение r_j не известно заранее для большинства задач. Как только появляется определенная задача, производится планирование ее работы, после чего производится поиск и выделение ресурсов необходимых для запуска.

11.5.1 Метрики, зависящие от времени

Предположим, что в результате финального распределения S, каждое действие $k \in j$ будет исполнено за время $C_k(S)$. Таким образом, задача j может быть решена не раньше, чем за время

$$C_j(S) = \max_{k \in j} C_k(S).$$

Определим время реализации действия $k \in j$ как p_k. Таким образом, время решения p_j задачи j может быть вычислено следующим образом:

$$p_j = C_j(S) - \min_{k \in j}(C_k(S) - p_k).$$

Полученные величины позволяют оценить интегральные свойства грид-среды. Для анализа качества сервиса, предоставляемого грид-средой можно использовать показатель максимального опоздания задач:

$$L_{max} = \max_{j \in \tau}\big(C_j(S) - d_j\big).$$

При оптимизации работы распределенной среды необходимо стремиться к минимизации значения данного показателя. Также, можно использовать показатель TJ, определяемый как количество опоздавших задач: $j \in \tau \wedge C_j > d_j$. Такой показатель предоставляет информацию о количестве невыполненных пользовательских запросов.

Потребление ресурсов RC_k определенной подзадачей определим как произведение соответствующего времени решения на количество используемых ресурсов:

$$RC_k = p_k \cdot m_k.$$

Следовательно, мы можем определить потребление ресурсов определенным заданием (1), и всего перечня задач планировщика соответственно (2).

$$RC_k = \sum_{k \in j} RC_k\,; \tag{1}$$

$$RC(S) = \sum_{j \in r} RC_j. \tag{2}$$

Используя определение суммарного потребления ресурсов, можно определить величину использования U (3) доступных ресурсов.

$$U = \frac{RC(S)}{m \cdot \left(\max\limits_{j \in \tau} C_j(S) - \min\limits_{j \in \tau}\big(C_j(S) - p_j\big)\right)}. \tag{3}$$

Данная величина характеризует насколько оптимально используются ресурсы, доступные в распределенной сети.

В процессе работы грид, очень часто возникают ситуации, когда в процессе исполнения задачи происходит сбой. Тогда задание должно быть запущено несколько раз для того, чтобы успешно выполниться. Вследствие этого, мы можем определить полное потребление ресурсов $RC\{k, j\}^{true}$ и полную величину использования ресурсов U^{true}, как соответствующую величину плюс затра-

ты на исполнение сбойных заданий. В связи с этим, можно определить метрику растрат:

$$WASTE = U^{true} - U,$$

которая определяет динамическую величину ошибок грид системы и должна быть минимизирована владельцем вычислительных ресурсов.

Так как пользователи и администраторы часто выдвигают различные (и даже конфликтующие) требования к грид системе, очень трудно подобрать метрику, которая удовлетворяла бы всех. С точки зрения пользователя, возможно выделить метрики среднего времени ответа (4) (Average Response Time – ART) и среднего времени ожидания (5) (Average Wait Time – AWT) [57]:

$$ART = \frac{1}{|\tau|}\sum_{j\in\tau}\big(C_j(S)\big), \quad (4)$$

$$AWT = \frac{1}{|\tau|}\sum_{j\in\tau}\big(C_j(S) - p_j\big). \quad (5)$$

Значение параметра ART характеризует, насколько быстро происходит решение задач пользователей. С другой стороны, значение параметра AWT интересно пользователям, которые производят постановку относительно небольших заданий.

Относительно хороший и простой метод измерения справедливости использования ресурсов это расчет девиации среднего времени ожидания:

$$AWTD = \frac{1}{|\tau|}\sqrt{\sum_{j\in\tau}\big(WT_j\big)^2 - \left(\sum_{j\in\tau}\frac{WT_j}{|\tau|}\right)^2},$$

где $WT_j = (\mathrm{C}_j(S) - p_j)$.

Для достижения оптимальных результатов работы грид, необходимо добиться минимизации параметра $AWTD$ каждым владельцем ресурсов.

Также, важна обработка результатов мониторинга работы грид системы. В этом случае, можно использовать метрику эффективности грид (Grid Efficiency – GE) [57]:

$$GE = \frac{\sum_{j\in\tau}((EndTime_j - StartTime_j)\times CPUs_j \times CPUSpeed_j)}{(EndTime_{lastJob} - SubmitTime_{firstJob})\times\sum_{m\in M}(CPUs_m \times CPUSpeed_m)}\times 100\%, \quad (6)$$

где $(EndTime_{lastJob} - SubmitTime_{firstJob})$ – это время работы системы;

$CPUs_j$ и $CPUSpeed_j$ – это количество процессоров использованных задачей j и их производительность;

$CPUs_m$ и $CPUSpeed_m$ – это количество процессоров в машине m и их производительность.

11.5.2 Метрики, зависящие от объема работы

В современных грид системах, возможность завершить исполнение данного объема работы может быть даже более важным, чем ускорение, полученное посредством такого исполнения (требуется отметить, что задачи, исполняемые в грид средах, могут быть значительно сложнее тех заданий, которые исполняются в традиционных параллельных системах, например потоки заданий обладают значительно более сложной логической структурой, чем пакеты задач). Грид требует переопределения понятия ошибки приложения: грид приложение, которое не смогло успешно выполниться в рамках отведенного ей бюджета, генерирует сообщение об ошибке, как только обнаружится невозможность успешного исполнения. Например, сбой может произойти вследствие того, что не могут быть найдены ресурсы для выполнения вычислений или в связи с наступлением крайнего срока работы приложения. Используя это понятие, отказоустойчивость можно определить как возможность на как можно больший срок переносить время появления ошибки, пока есть хоть какие-то шансы того, что приложение завершится успешно [38].

Определим метрику *завершенного объема работы* (Workload Completion) как отношение успешно завершенных задач к объему всех задач, поставленных планировщику грид-среды.

$$WC = \frac{\sum_{j \in \tau \wedge (j\ complited)} 1}{|\tau|}.$$

Данная метрика позволяет определить ограничения грид системы, и ее максимизация может быть основной целью как пользователей, так и владельцев ресурсов. С другой стороны, WC имеет некоторые ограничения с токи зрения владельцев ресурсов, так как задачи с меньшим количеством действий имеют большее влияние на данную величину.

В качестве дополнительной метрики предлагается ввести метрику *завершения действий* (Task Completion), которую можно определить как количество завершенных действий к общему количеству действий, исполненных в рамках системы распределения заданий:

$$TC = \frac{\sum_{j \in \tau \wedge k \in j \wedge (k\ complited)} 1}{\sum_{j \in \tau} |j|}.$$

Также, можно ввести понятие завершения разблокированных действий (Enabled Task Completion), где под понятием разблокированного действия мы

будем понимать те действия, которые могут быть выполнены только после того, как все зависимости для данного действия будут выполнены:

$$ETC = \frac{\sum_{j \in \tau \wedge k \in j \wedge (k\ complited)} 1}{\sum_{j \in \tau \wedge k \in j \wedge (k\ enabled)} 1}.$$

Таким образом, владельцы ресурсов должны стремиться к максимизации ETC. Если метрики TC и ETC значительно разнятся, то необходимо принять специальные меры для обеспечения выполнения критичных действий (таких действий, от которых зависит исполнение большого количества других действий).

12. Облачные вычисления

Облачные вычисления привлекают много внимания в последнее время. В средствах массовой информации, в интернете, даже на телевидении можно встретить восторженные материалы, описывающие все прекрасные возможности, которые может предоставить данная технология.

Не смотря на то, что метафора «облако» уже давно используется специалистами в области сетевых технологий для изображения на сетевых диаграммах сложной вычислительной инфраструктуры (или же Интернета как такового), скрывающей свою внутреннюю организацию за определенным интерфейсом, термин «Облачные вычисления» появился на свет совсем недавно. Согласно результатам анализа поисковой системы Google, термин «Облачные вычисления» («Cloud Computing») начал набирать вес в конце 2007 – начале 2008 года, постепенно вытесняя популярное в то время словосочетание «Грид-вычисления» («Grid Computing»). Судя по заголовкам новостей того времени, одной из первых компаний, давших миру данный термин, стала компания IBM, развернувшая в начале 2008 года проект «Blue Cloud» и спонсировавшая Европейский проект «Joint Research Initiative for Cloud Computing».

На сегодняшний день уже можно говорить о том, что облачные вычисления прочно вошли в повседневную жизнь каждого пользователя Интернета (хотя многие об этом и не подозревают). По некоторым экспертным оценкам, технология облачных вычислений может в три-пять раз сократить стоимость бизнес-приложений и более чем в пять раз стоимость приложений для конечных потребителей, но, несмотря на общее радужное чувство по отношению к облачным технологиям, до сих пор нет единого мнения о том, что такое «Облачные Вычисления» и каким образом они соотносятся с парадигмой «Грид-вычислений». Для того, чтобы разобраться в этом, взглянем сначала на несколько существующих определений облачных вычислений, выясним основные характеристики облаков и рассмотрим, какие общие аспекты можно выявить в архитектуре облачных решений. Рассмотрим достоинства и недостатки, а также попробуем классифицировать платформы облачных вычислений. В конце главы мы попытаемся сделать сравнение двух концепций: грид-вычислений и облачных вычислений.

12.1 Определение облачных вычислений

С одной стороны, у термина «Облачные вычисления» нет устоявшегося стандартного определения. С другой стороны множество различных корпораций, ученых и аналитиков дают собственные определения этому термину. Определение облачных вычислений вызвало дебаты и в научном сообществе. В отличие от определений, которые можно найти в коммерческих изданиях, научные определения ориентируются не только на то, что будет предоставлено пользователю, но и на архитектурные особенности предлагаемой технологии. Например, в лаборатории Беркли дают следующее определение облачных вычислений:

«Облачные вычисления – это не только приложения, поставляемые в качестве услуг через Интернет, но и аппаратные средства и программные системы в центрах обработки данных, которые обеспечивают предоставление этих услуг. Услуги сами по себе уже давно называют «предоставление программного обеспечения как услуги» (Software-as-a-Service или SaaS). Облаком называется аппаратное и программное обеспечение центра обработки данных. Общественное облако предоставляет ресурсы облака широкому кругу пользователей по принципу «*оплата по мере использования*» (*pay-as-you-go* – принцип предоставления услуг, при котором пользователь оплачивает только те ресурсы, которые были по факту затрачены на решение поставленной задачи). Частное облако – это внутренние центры обработки данных, в коммерческой или иной организации, которые не доступны широкому кругу пользователей. Таким образом, облачные вычисления являются суммой SaaS и «*коммунальных вычислений*» (*Utility Computing* – модель вычислительных систем, в которой предоставление данных и процессорных мощностей организовано по принципам коммунальных услуг). Люди могут быть пользователями или провайдерами SaaS, либо пользователями или поставщиками коммунальных вычислений».

Данное сложное и пространное определение выделяет другую сторону облачных вычислений: с точки зрения провайдера, важнейшей составляющей облака является центр обработки данных (ЦОД). ЦОД содержит вычислительные ресурсы и хранилища информации, которые вместе с программным обеспечением предоставляются пользователю по принципу «оплата по мере использования».

Ян Фостер определяет облачные вычисления как «парадигму крупномасштабных распределенных вычислений, основанную на эффекте масштаба, в рамках которой пул абстрактных, виртуализованных, динамически-

масштабируемых вычислительных ресурсов, ресурсов хранения, платформ и сервисов предоставляется по запросу внешним пользователям через Интернет».

Данное определение добавляет два важнейших аспекта в определение облачных вычислений: *виртуализацию* и *масштабируемость*. Облачные вычисления абстрагируются от базовой аппаратной и программной инфраструктуры посредством виртуализации. Виртуализованные ресурсы предоставляются посредством определенных абстрактных интерфейсов (программных интерфейсов API или сервисов). Такая архитектура обеспечивает масштабируемость и гибкость физического уровня облака без последствий для интерфейса конечного пользователя.

Наконец в работе [70], после обработки более 20 определений облачных вычислений, было дано следующее определение облачных вычислений (возможно, чуть более целостное, чем остальные определения приведенные в данной главе):

«*Облако* – это большой пул легко используемых и легкодоступных виртуализованных ресурсов (таких как аппаратные комплексы, сервисы и др.). Эти ресурсы могут быть динамически перераспределены (масштабированы) для подстройки под динамически изменяющуюся нагрузку, обеспечивая оптимальное использование ресурсов. Этот пул ресурсов обычно предоставляется по принципу «оплата по мере использования». При этом владелец облака гарантирует качество обслуживания на основе определенных соглашений с пользователем».

Все определения иллюстрируют одну простую мысль: феномен облачных вычислений объединяет несколько различных концепций информационных технологий и представляет собой новую парадигму предоставления информационных ресурсов (аппаратных и программных комплексов). Со стороны владельца вычислительных ресурсов облачные вычисления ориентированы на предоставление информационных ресурсов внешним пользователям. Со стороны пользователя, облачные вычисления – это получение информационных ресурсов в виде услуги у внешнего поставщика, оплата за которую производится в зависимости от объема потребленных ресурсов согласно установленному тарифу. Ключевыми характеристиками облачных вычислений являются масштабируемость и виртуализация.

Масштабируемость представляет собой возможность динамической настройки информационных ресурсов к изменяющейся нагрузке, например к увеличению или уменьшению количества пользователей, изменению необходимой емкости хранилищ данных или вычислительной мощности. *Виртуализация*, которая также рассматривается как важнейшая технология всех облачных

систем, в основном используется для обеспечения абстракции и инкапсуляции. Абстракция позволяет унифицировать «сырые» вычислительные, коммуникационные ресурсы и хранилища информации в виде пула ресурсов и выстроить унифицированный слой ресурсов, который содержит те же ресурсы, но в абстрагированном виде. Они представляются пользователям и верхним слоям облачных систем как виртуализованные серверы, кластеры серверов, файловые системы и СУБД. Инкапсуляция приложений повышает безопасность, управляемость и изолированность. Еще одной важной особенностью облачных платформ является интеграция аппаратных ресурсов и системного ПО с приложениями, которые предоставляются конечному пользователю в виде сервисов.

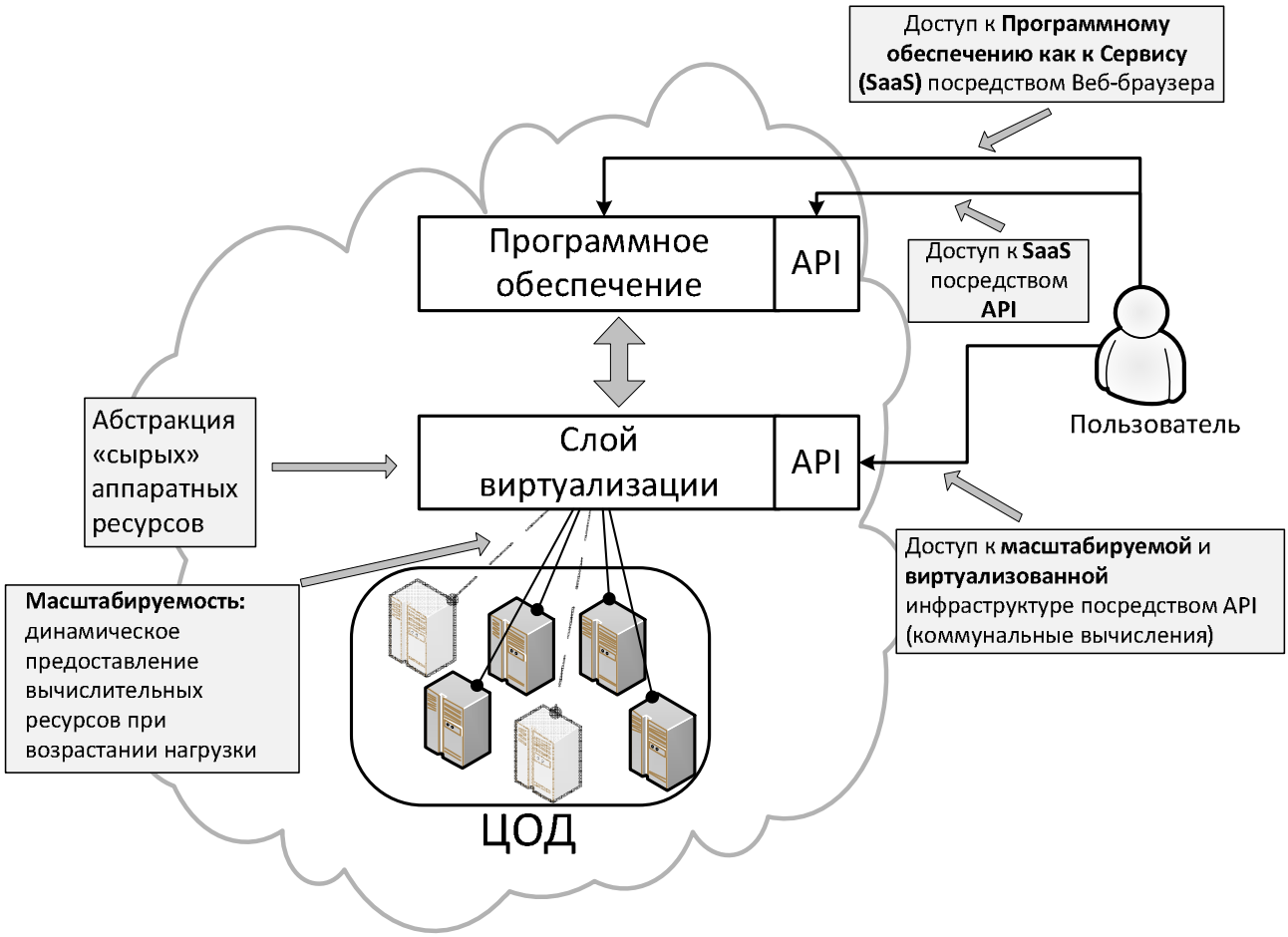


Рис. 60. Основные особенности облачных вычислений

В соответствии со всем вышесказанным, можно выделить следующие основные черты облачных вычислений:

– облачные вычисления представляют собой новую парадигму предоставления вычислительных ресурсов;
– базовые инфраструктурные ресурсы (аппаратные ресурсы, системы хранения данных, системное ПО) и приложения предоставляются в виде серви-

сов. Данные сервисы могут предоставляться независимым поставщиком для внешних пользователей по принципу «оплата по мере использования»;
- основными особенностями облачных вычислений являются виртуализация и динамическая масштабируемость;
- облачные сервисы могут предоставляться конечному пользователю через веб-браузер или посредством определенного программного интерфейса.

12.2 Многослойная архитектура облачных приложений

Все возможные методы классификации облаков можно свести к трехслойной архитектуре облачных систем, состоящей из следующих уровней:
- Инфраструктура как сервис (Infrastructure as a Service: IaaS);
- Платформа как сервис (Platform as a Service: PaaS);
- Программное обеспечение как сервис (Software as a Service: SaaS).

Рассмотрим более подробно, что собой представляет каждый из указанных уровней, и каким образом они взаимодействуют друг с другом.

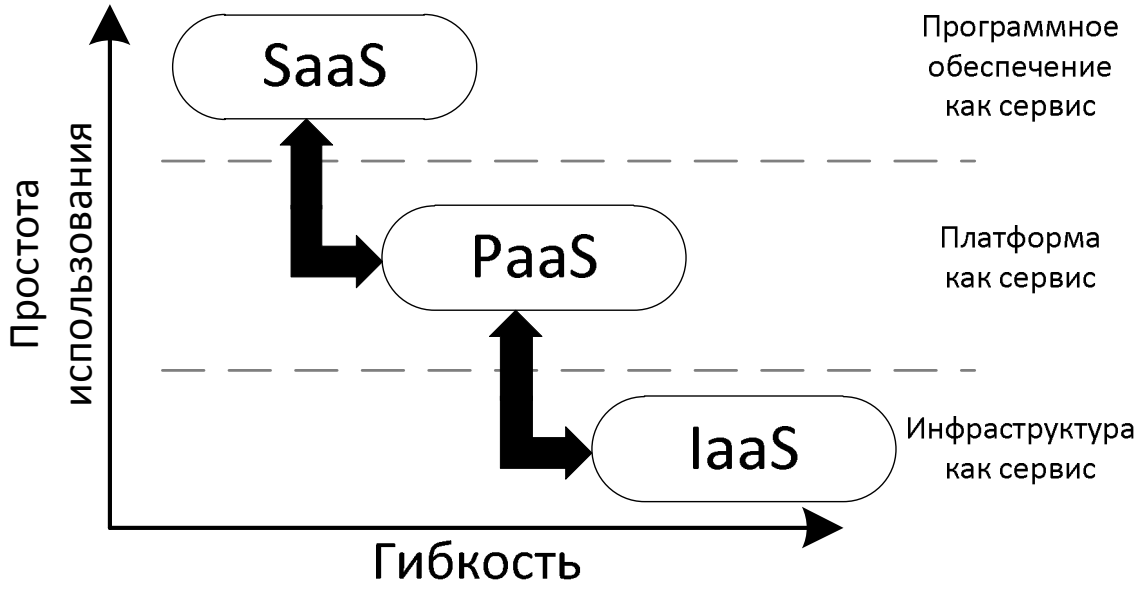


Рис. 61. Три слоя облачных вычислений

12.2.1 Инфраструктура как сервис (IaaS)

IaaS предлагает информационные ресурсы, такие как вычислительные циклы или ресурсы хранения информации, в виде сервиса. Ярким примером такого подхода является облако компании Amazon – Amazon Web Services, состоящее из Elastic Compute Cloud (EC2), предоставляющего информационные ресурсы в виде сервисов и Simple Storage Service (S3) для хранения информации. Другим примером такого подхода может являться сервис Joynet обеспечиваю-

щий хостинг высоко-масштабируемых веб-сайтов и веб-приложений. Вместо предоставления доступа к «сырым» вычислительным устройствам и системам хранения, поставщики IaaS обычно предоставляют виртуализованную инфраструктуру в виде сервиса. Обычно «сырые» ресурсы (процессорные циклы, сетевое оборудование, системы хранения) располагают на базовом уровне, над которым посредством виртуализации надстраивают слои сервисов, которые и предоставляются конечным пользователям в виде IaaS.

Надо сказать, что еще задолго до появления облачных вычислений инфраструктура была доступна как сервис. Такой подход назывался «коммунальные вычисления», и это словосочетание часто применяется некоторыми авторами при описании инфраструктурного уровня облачных систем. Например, в марте 2006 года компания Sun запустила систему Sun Grid Compute Utility. Эта система предоставляла пользователям вычислительные ресурсы по цене 1$ за 1 процессоро-час, то есть работала по принципу «оплата по мере использования». По мере развития этой системы, Sun открыла каталог приложений Application Catalog, который позволял разработчикам с легкостью запускать их приложения online, а двумя годами позже Sun анонсировала Open Cloud Platform, которая должна была стать конкурентом Amazon Web Services. К сожалению, проект Sun Cloud был закрыт сразу же после поглощения Sun компанией Oracle в 2010 году.

В сравнении с ранними попытками организации коммунальных вычислений, подход IaaS предоставляет разработчикам понятный интерфейс, к которому легко получить доступ и использовать в собственных приложениях. Данный интерфейс должен легко интегрироваться с инфраструктурой потенциальных пользователей и разработчиков решений SaaS. Давно замечено, что ресурсы поставщиков коммунальных вычислений могут быть эффективно использованы только в том случае, если они используются большим числом потребителей, а этого можно добиться путем организации хорошего программного интерфейса к своим ресурсам.

12.2.2 Платформа как сервис (PaaS)

Платформа – это слой абстракции между программными приложениями (SaaS) и виртуализованной инфраструктурой (IaaS). Основной целевой аудиторией PaaS являются разработчики приложений. Разработчики могут писать собственные приложения на основе спецификаций определенной платформы, не заботясь о том, каким образом организовать взаимодействие с нижележащей инфраструктурой (IaaS). Разработчики загружают свой программный код на

платформу, которая обеспечивает автоматическое масштабирование приложения в зависимости от нагрузки. Ярким примером реализации подхода PaaS является платформа Google App Enginge, обеспечивающая исполнение пользовательских приложений на инфраструктуре Google. Слой PaaS основывается на стандартизованном интерфейсе, предоставляемом слоем IaaS, который виртуализует базовые вычислительные ресурсы и предоставляет стандартный интерфейс для разработки приложений, функционирующих на слое SaaS.

12.2.3 Программное обеспечение как сервис (SaaS)

SaaS – это программное обеспечение, которое предоставляется по принципу «оплата по мере использования» и управляется удаленно одним или несколькими поставщиками. SaaS – это наиболее заметный слой облачных вычислений, так как именно он представляет реальную ценность для конечного пользователя и обеспечивает решение его задач.

С точки зрения пользователя, основным достоинством SaaS является ценовое преимущество перед «классическим» ПО. Оплата SaaS осуществляется по модели «оплата по мере использования», что означает отсутствие необходимости инвестиций в собственную аппаратную и программную инфраструктуру. Ярким примером SaaS является комплекс Google Apps, включающий в себя такие системы как Google Mail и Google Docs.

Типичный пользователь SaaS не может контролировать базовую инфраструктуру, будь это программная платформа, на которой SaaS основано (PaaS) или же непосредственно аппаратная инфраструктура (IaaS). Однако поставщик SaaS обязан проработать взаимодействие данных слоев, потому что они необходимы для работы системы. Например, SaaS-приложение может быть разработано на базе существующей платформы и исполняться на инфраструктуре, предоставленной сторонней компанией. Работа с платформами и/или инфраструктурой как с сервисом является очень привлекательной с точки зрения поставщиков SaaS, так как это может уменьшить в разы отчисления на используемые лицензии или затраты на инфраструктуру. Это также позволяет им сосредоточиться на тех областях, в которых они по-настоящему компетентны. Как можно заметить, это очень похоже на те преимущества, которые мотивируют конечных пользователей ПО переходить к использованию решений SaaS. По данным рыночных аналитиков, высокое давление рынка приводит компании к необходимости сокращения расходов на ИТ, что приводит к высокому спросу и росту решений SaaS, и, тем самым, росту облачных вычислений в целом.

12.3 Компоненты облачных приложений

На сегодняшний день не существует единой компонентной архитектуры облачных приложений. Это вызвано высокой закрытостью различных аспектов реализации наиболее распространенных облачных систем. Но, не смотря на это, можно выделить основные наиболее важные компоненты, присущие практически всем существующим облачным платформам.

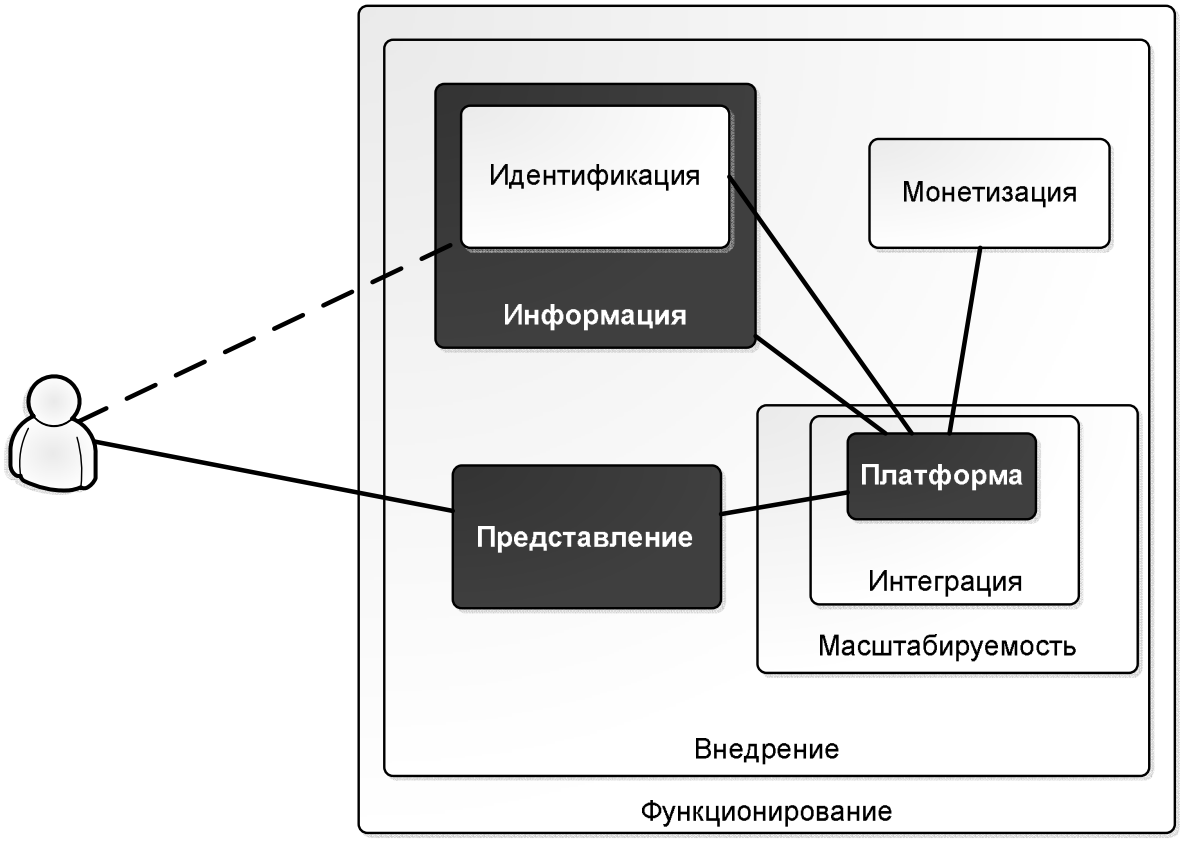


Рис. 62. Компонентная модель облачного решения

Облако можно разбить на основные компоненты, отражающие следующие ключевые особенности облачных решений.

1 **Платформа**: среда и набор утилит, обеспечивающих разработку, интеграцию и предоставление облачных сервисов. Платформа является центральным компонентом модели облака. Платформа с одной стороны, предоставляет набор базовых сервисов, доступных разработчику облачного приложения, а с другой накладывает определенные ограничения на методы разработки и предоставления приложения. При выборе платформы можно основываться как на уже готовых решениях (Google App Engine или Microsoft Azure), так и самостоятельно разработать масштабируемую платформу на базе готовой облачной инфраструктуры. При выборе базовой платформы необходимо исходить из критериев стоимости законченного

решения, производительности и необходимой масштабируемости. Также необходимо помнить, что любая выбранная платформа потребует использования определенных языков программирования и программных фреймворков для реализации приложения.

2 **Представление:** интерфейс, через который пользователь производит взаимодействие с облаком. Этот компонент обеспечивает получение входных данных и предоставление информации конечному пользователю. Наиболее типичным методом реализации представления является веб-приложение, обеспечивающее взаимодействие с пользователем посредством веб-браузера. В последнее время все чаще используются все возможности предоставления пользователю удобного интерфейса работы независимо от устройства, с которого он выходит в Интернет. Это влечет за собой разработку отдельных пользовательских интерфейсов для мобильных устройств (смартфонов, планшетов) которые могут представлять собой как отдельные интернет-страницы, так и полноценные мобильные приложения, взаимодействующие с облаком посредством API.

3 **Информация:** источники данных, обеспечивающие распределенное хранение структурированных или неструктурированных, статических или динамически-изменяющихся данных. Пользовательская информация в облачных системах может достигать огромных объемов. Например, в системе Gmail на начало 2010 года было более 170 миллионов активных пользователей. Даже если предположить, что в среднем используется не более 5% от доступных пользователю 7 Гб, получается что Google необходимо обеспечивать хранение более чем 60 000 Тб данных почтовой переписки. И этот объем экспоненциально увеличивается с каждым месяцем. На таких объемах данных классические SQL базы данных уже не дают удовлетворительных результатов по скорости обработки. Более того, облачным платформам часто приходится обрабатывать связанные структуры данных (графы, деревья) что становится очень тяжело при использовании SQL-подхода и реляционных баз данных. В связи с этим, в последние несколько лет стали активно развиваться альтернативные «NoSQL» системы управления базами данных (колоночные СУБД, такие как Hadoop; документно-ориентированные СУБД, такие как CouchDB и MongoDB) и альтернативные подходы к обработке сверхбольших объемов информации. Наиболее известной реализацией такого подхода является фреймворк MapReduce, представленный компанией Google. Он используется для параллельных

вычислений над очень большими (несколько петабайт) наборами данных в компьютерных кластерах.

4 **Идентификация:** информация об основных потребителях облачных ресурсов, используется для оптимизации и подстройки облака под их задачи. Большинству приложений требуется уметь отличать пользователей друг от друга для предоставления релевантной информации (например, почтовому клиенту необходимо обеспечить аутентификацию и авторизацию пользователей, чтобы представить каждому из них возможность чтения их личной почтовой корреспонденции). Такая информация имеет первостепенную значимость для облачной платформы, так как на нее завязаны большие объемы пользовательских данных, и при этом необходимо обеспечить ее максимальную доступность в рамках системы, т.к. процедура авторизации должна проходить максимально быстро. Также, необходимо обеспечить прозрачную авторизацию пользователя во всех сервисах одной облачной платформы, чтобы не требовалось каждый раз вводить имя пользователя и пароль заново. В связи с распределенной природой облачных сервисов необходимо обеспечить высочайший уровень безопасности при работе с пользовательской информацией.

5 **Интеграция:** инфраструктура, упрощающая обмен информацией и исполнение задач в распределенной вычислительной среде. Высокая степень декомпозиции сервисов позволяет достичь максимальной эффективности и гибкости выполнения облачных приложений, так как появляется возможность загрузки сразу нескольких вычислительных машин при исполнении одной пользовательской задачи. В связи с этим появляется необходимость организации обмена информацией и исполнения задач в распределенных вычислительных средах. В рамках этого компонента необходимо обеспечить максимальную производительность и безопасность процесса обмена данными между сервисами. Далее, необходимо обеспечить совместимость форматов данных и разработать механизмы синхронного и асинхронного взаимодействия с унаследованным ПО. На более высоком уровне необходимо обеспечить слабосвязанность программных компонентов и убедиться в отсутствии *узких мест* в программной архитектуре системы.

6 **Масштабируемость:** гибкость методов предоставления ресурсов, обеспечивающая поддержку выделения дополнительных информационных ресурсов при возрастании нагрузки на приложение. При этом необходимо учитывать не только возможность кратковременного увеличения нагрузки на приложение (например, в результате наплыва посетителей после появ-

ления рекламной статьи на одном из популярных Интернет-ресурсов) но и планировать долгосрочное увеличение производительности системы в результате постоянного прироста аудитории. В обоих случаях, необходимо обеспечить декомпозицию облачного приложения на отдельные модульные компоненты, которые могут быть распределены на несколько вычислительных устройств.

7 **Монетизация:** учет и биллинг ресурсов, затраченных на исполнение пользовательских задач. Это ключевой компонент множества коммерческих приложений. Для организации качественного биллинга облачных платформ необходимо организовать сбор и предоставление полноценной информации о всевозможных ресурсах, затрачиваемых на решение пользовательских задач. Также, необходимо обеспечить пользователю возможность удобной и быстрой оплаты затраченных ресурсов.

8 **Внедрение:** процесс разработки нового облачного приложения, который включает в себя разработку, тестирования и внедрение в эксплуатацию. На этапе разработки облачного приложения требуется совсем небольшой объем вычислительных ресурсов, который значительно увеличивается при переходе к этапу нагрузочного тестирования и внедрения в эксплуатацию. При этом очевидно, что применение готовой облачной инфраструктуры позволяет значительно сократить издержки на разработку и внедрение высокомасштабируемого приложения, так как оплата использованных информационных ресурсов производится на основе модели коммунальных вычислений и не требует значительных инвестиций в собственную инфраструктуру. Это позволяет минимизировать начальные затраты и сконцентрировать финансирование на всестороннем тестировании приложения. Но, не смотря на все перечисленные достоинства, необходимо оценить все возможные недостатки модели облачных вычислений, как то сложности организации репликации данных между сервисами, сложность отката на предыдущие версии при появлении неожиданных ошибок в процессе внедрения, необходимость аккуратного и всестороннего тестирования разработанных сервисов на совместимость данных и слаженность работы приложения.

9 **Функционирование:** мониторинг и поддержка приложений, находящихся в стадии эксплуатации. Приложение, которое запущено в эксплуатацию, необходимо администрировать, что может оказаться чрезвычайно сложной задачей, если учесть большое число отдельных сервисов, составляющих облачное приложение. В связи с этим необходимо обеспечить интеграцию

процессов администрирования и управления сервисами в виде единого «центра управления сервисами». Параллельно, в него можно включить мониторинг нагрузки приложения, панель управления пользовательскими задачами и т.п.

Все вышеперечисленные компоненты облачного приложения должны быть запланированы с самого начала разработки для обеспечения высокого уровня масштабируемости и автоматизации.

12.4 Достоинства и недостатки облачных вычислений

Как ранее было описано, облачные вычисления ориентированы на предоставление информационных ресурсов на трех уровнях: уровне инфраструктур (IaaS), уровне платформ (PaaS) и уровне программного обеспечения (SaaS). Предоставляя интерфейсы ко всем трем различным уровням, облака взаимодействуют с тремя различными типами потребителей.

1. *Конечные пользователи*, которые используют SaaS-решения через веб-браузер, или же какие-либо базовые ресурсы инфраструктурного слоя которые предоставляются посредством слоя SaaS (например, облачные ресурсы хранения посредством Dropbox.com).
2. *Корпоративные потребители*, которые могут использовать все три слоя: IaaS – для того, чтобы расширить собственную программно-аппаратную инфраструктуру или получить дополнительные вычислительные ресурсы по требованию; PaaS – для того, чтобы иметь возможность запуска собственных приложений в облаке; SaaS – для получения возможностей тех приложений, которые уже доступны в облаке.
3. *Разработчики и независимые поставщики программного обеспечения*, разрабатывающие приложения, которые предоставляются в виде облачных SaaS-решений. Обычно, эта категория пользователей напрямую взаимодействует со слоем PaaS, и уже через него, опосредованно, со слоем IaaS.

С точки зрения пользователя, основное достоинство облачных вычислений – это модель оплаты ресурсов по мере их использования. Отсутствует необходимость предварительных инвестиций в инфраструктуру: нет необходимости инвестиций в лицензионное ПО, отсутствует необходимость инвестиции в аппаратную инфраструктуру и связанные с этим затраты на обслуживание и персонал. Таким образом, капитальные затраты превращаются в текущие расходы. Покупатели облачных сервисов используют только тот объем информационных ресурсов, который им на самом деле необходим, и оплачивают только тот объ-

ем информационных ресурсов, которыми они реально воспользовались. В то же время, они пользуются такими достоинствами облачных вычислений как масштабируемость и гибкость. Облачные вычисления позволяет легко и быстро предоставить необходимые вычислительные ресурсы по требованию.

Тем не менее, существуют некоторые недостатки модели облачных вычислений, которые вытекают из одной очевидной особенности – облака обслуживают сразу множество различных клиентов. Пользователь облачной платформы не знает, задачи каких пользователей будут исполняться на том или ином сервере, входящем в облачную инфраструктуру. Типичное облако является внешним по отношению к внутренней инфраструктуре любой компании, то есть находится вне зоны ответственности администраторов и служб безопасности компании-потребителя. В то время как для отдельного потребителя это может быть несущественным, крупные компании уделяют этому вопросу очень большое значение

Пользователю приходится полагаться на обещания поставщика облачных решений в вопросах надежности, производительности и качества обслуживания инфраструктуры облака. Использование облаков также сопряжено с высокими рисками безопасности и защиты конфиденциальной информации. Это связано с двумя факторами: 1) необходимость загрузки и получения данных из облака; 2) хранение данных производится на базе удаленных хранилищ, в связи с чем владельцу информации приходится полагаться на гарантии поставщика облачных решений, что несанкционированного доступа к данным не произойдет. Более того, использование облаков требует определенных инвестиций в интеграцию собственной инфраструктуры и приложений в облако. В настоящее время нет стандартов для интерфейсов IaaS, PaaS и SaaS. В связи с этим выбор поставщика облачных решений становится довольно рискованным занятием (вспомним про то, как, на первый взгляд крупнейшая и старейшая облачная платформа Sun Cloud была закрыта и забыта в течение 2-х месяцев после поглощения компанией Oracle).

В связи с этим, необходимо всегда учитывать риски, связанные с использованием облаков. В каждом отдельном случае необходимо внимательно взвесить все потенциальные выгоды и угрозы при переходе на облачную платформу. Также, необходимо решить какие данные и какая обработка может производиться на базе «облачного аутсорсинга», а какие данные лучше никогда не выводить за рамки локальной сети организации.

12.5 Классификация облаков

В общем случае, облака можно классифицировать в соответствии с тем, кто владеет центрами обработки данных, формирующими облако. Далее будет представлена классификация единых облачных систем в соответствии с тем, кто владеет облачной инфраструктурой и классификация распределенных облачных сред в соответствии с тем, какие типы облаков объединяются.

12.5.1 Общественные и частные облака

Как было сказано ранее, облачные вычисления, с точки зрения поставщика ресурсов, можно определить как предоставление информационных ресурсов для внешних клиентов. При этом с точки зрения конечного пользователя, облако обеспечивает получение информационных ресурсов от внешнего поставщика, в виде сервиса доступного через Интерет за определенную плату. Кроме того, мы определили, что ключевыми характеристиками облачных вычислений являются масштабируемость и виртуализация.

Не смотря на это, виртуализация «сырых» аппаратных ресурсов и предоставление их в виде сервисов, не всегда связана с их предоставлением внешним пользователям. Организации часто используют виртуализацию и сервис-ориентированные вычисления для повышения коэффициента использования собственных информационных инфраструктур. Если «классические» методы использования серверного оборудования могут обеспечить 5-15% загрузки, то применение виртуализации может легко вывести этот показатель на уровень от 18% до 38% (у таких провайдеров как Google). Это приводит к уменьшению затрат на поддержку оборудования, затрат на помещения и охлаждение и в целом снижает стоимость владения вычислительными серверами.

Для того чтобы различать облака, которые изначально разрабатывались для использования внешними пользователями и те облака, которые разрабатываются для поддержки внутренней инфраструктуры предприятия часто используют термины «Общественное облако» и «Частное облако».

Общественным облаком называют центры обработки данных, предоставляющие свои ресурсы третьим лицам через Интернет (например, Google или Amazon). Общественное облако не ограничивает базу пользователей, каждый может подключиться к нему и получить ресурсы за определенную плату. Таким образом, общественные облака могут предоставлять свои ресурсы как частным лицам, так и сторонним организациям.

С другой стороны, любая организация может с опаской относиться к идее предоставления собственных внутренних данных через интернет. Тогда на по-

мощь приходит концепция *частного облака*, которое может быть развернуто в рамках внутренней сети любой организации. Частное облако полностью контролируется той организацией, на базе которой оно развернуто, включая управление доступными приложениями, инфраструктурой, физическим расположением вычислительных узлов и заканчивая управлением пользователями облака. Основным достоинством такого подхода, по сравнению с классическим методом построения центров обработки данных является увеличение коэффициента использования вычислительных ресурсов за счет применения технологии виртуализации.

Не смотря на то, что разница между частными и общественными облаками, по большому счету, состоит только в масштабе, многие исследователи и разработчики считают, что частные облака не могут быть отнесены к «истинной» облачной концепции, так как они не предоставляют информационные ресурсы внешним пользователям. Также, частные облака не обладают «потенциально бесконечной масштабируемостью и гибкостью настоящих облачных платформ».

12.5.2 Гибридные облака и федерации облачных платформ

Отдельные облака могут быть объединены в многооблачные вычислительные среды. В зависимости от того, какие базовые платформы используются в таких облачных объединениях, их можно разделить на следующие типы:

– гибридные облака;
– федерации облаков.

Гибридные облака объединяют общественные и частные облака, позволяя организациям запускать часть приложений внутри частного облака, а другую часть данных передавать на обработку в общественное облако. Такой подход позволяет распределить данные организации: с одной стороны, воспользоваться возможностями общественного облака для быстрой обработки больших объемов данных, а с другой стороны сохранить частную информацию в рамках корпоративной сети. Но такое решение может повлечь значительное усложнение логики распределения приложений и данных в рамках организации. Появляется необходимость мониторинга внутренней и внешней вычислительной инфраструктуры, усложнение политик безопасности и т.п. В связи с этим, такое решение не подходит для задач, связанных со сложным обменом данных.

Федерации облаков (federated clouds) представляют собой объединения нескольких общественных облачных платформ (хотя частные облака также могут входить в такие федерации). Не смотря на то, что с точки зрения пользователя

общественное облако предоставляет практически бесконечную масштабируемость и может справиться практически с любой нагрузкой, поставщики общественных облаков могут столкнуться с проблемой недостатка того или иного информационного ресурса (канала данных, вычислительной мощности и др.) в связи с большим количеством пользователей. Следовательно, в таком случае поставщикам облачных услуг приходится объединять свои инфраструктуры чтобы удовлетворить спрос потребителей на необходимые информационные ресурсы.

Федерации облаков представляют собой коллекцию отдельных облачных платформ, которые могут обмениваться между собой данными и вычислительными ресурсами посредством определенных интерфейсов. В соответствии с принципами федерации, каждое облако остается независимым, но может взаимодействовать с другими облаками посредством стандартного интерфейса. В настоящее время, федерация облаков – это скорее теоретическая концепция, в связи с тем что не существует общепринятого стандарта меж-облачного взаимодействия. Хотя сегодня предпринимаются первые шаги в этом направлении: организация Open Grid Forum, в настоящий момент, разрабатывает такой стандарт под названием Open Cloud Computing Interface. Целью данной работы является разработка стандартизованного API, который позволил бы обеспечить коммуникацию между различными поставщиками облачных услуг, а также обеспечил бы появление новых моделей предоставления ресурсов:

- интеграторов, которые предоставляли бы услуги по распределению работ по различным облакам;
- агрегаторов, которые предлагали бы единый интерфейс ко множеству различных облачных платформ.

Многие разработчики считают, что появление открытых стандартов меж-облачной коммуникации может очень серьезно повлиять на дальнейшее развитие всей технологии облачных вычислений.

12.6 Наиболее распространенные облачные платформы

Рассмотрим возможности и организацию наиболее распространенных на сегодняшний день платформ облачных вычислений: Amazon Web Services, Google App Engine и Microsoft Windows Azure (таблица 2).

Таблица 2. Сравнение платформ облачных вычислений

Платформы / Характеристики	Amazon Web Services	Google App Engine	Microsoft Windows Azure
Тип	IaaS	PaaS	PaaS
Разрабатываемые сервисы	Вычислительные сервисы, сервисы хранения	Web-приложения	Как Web-приложения, так и не Web-приложения
Виртуализация	Уровня ОС, с запущенным гипервизором Xen	Контейнер приложений	Уровня ОС
Интерфейс доступа пользователя	Утилиты консоли Amazon EC2	Web-консоль администрирования	Портал Microsoft Windows Azure
Web APIs	Да	Да	Да
Среда разработки	Отсутствует	Python, Java	Microsoft .NET

12.6.1 Amazon Web Services

Решение, предлагаемое компанией Amazon, стало стандартом «de facto» в области облачной инфраструктуры. Не смотря на то, что решение, предлагаемое компанией Amazon, не является уникальным, все остальные решения рассматриваются либо как дополнения к тому, что предлагает Amazon Web Services (AWS), либо как конкуренты (хотя может быть и не явные) данной платформы. Основным отличием AWS от всех остальных платформ является то, что каждый разработчик может использовать собственную среду исполнения приложений. При этом можно выбрать один из множества образов виртуальных машин предлагаемых компанией Amazon, так и использовать собственное решение.

На рисунке представлены наиболее существенные элементы решения, предлагаемого в рамках системы AWS. Как можно заметить, Amazon предлагает очень широкий набор возможностей организации вычислений, хранения информации и предоставления информации. При этом множество сервисов по управлению и биллингу, предлагаемые компанией, просто не поместились на диаграмму.

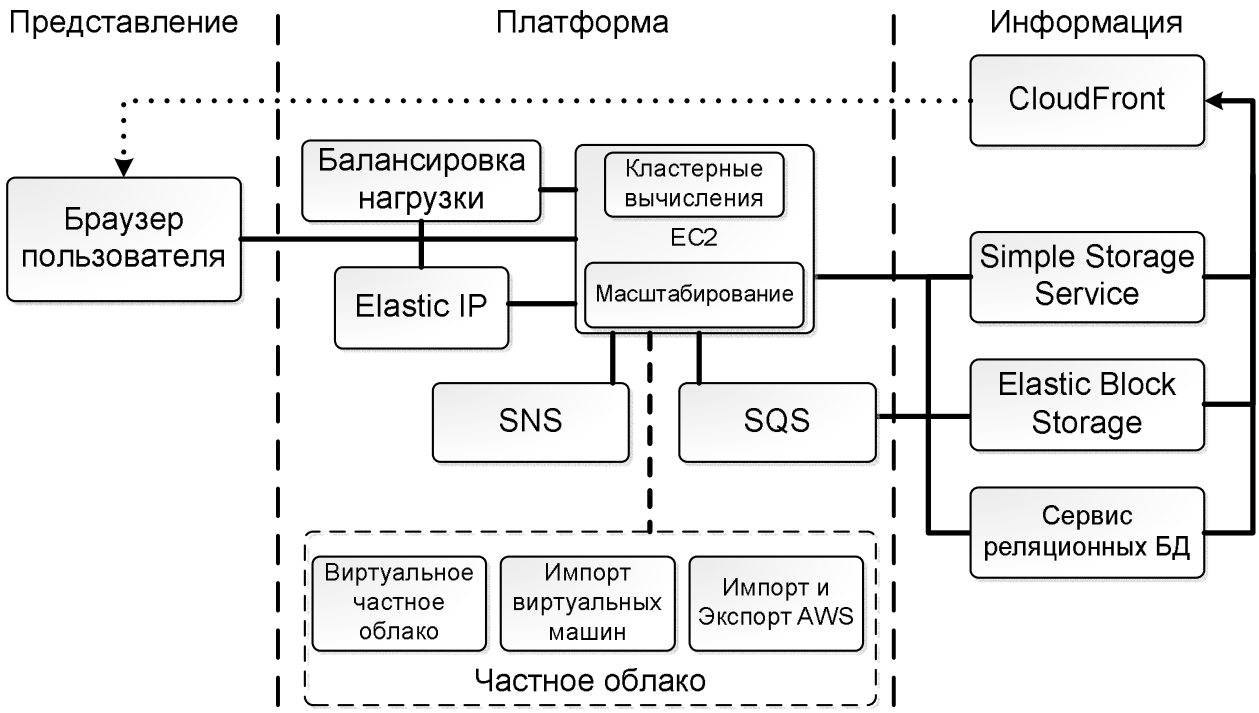


Рис. 63. Архитектура Amazon Web Services

Ядром Amazon Web Services служит система Amazon Elastic Compute Cloud (Amazon EC2) совместно с сервисами хранения. EC2 предоставляет пользователю выбор виртуальных машин, которые можно запустить в распределенной вычислительной среде. Виртуальная машина (называется Amazon Machine Image – AMI) может быть основана практически на любой операционной системе (различные версии Windows и дистрибутивы Linux) и обеспечивать работу с любой программной инфраструктурой (MySQL, Oracle и др.).

Наряду с виртуальными машинами, Amazon обеспечивает несколько механизмов хранения данных:

– *Simple Storage Service (S3)* представляет собой сервис хранения данных, предоставляющий доступ на основе REST и SOAP. Система S3 распределена по всему миру: сервера расположены в Европе, Азии и Соединенных Штатах Америки. При этом обеспечивается возможность работы с данными размером от одного байте до 5TB.
– *Система Elastic Block Storage (EBS)* представляет собой высокопроизводительный виртуальный жесткий диск размером от 1Гб до 1Тб. Он может быть подключен к любой виртуальной машине работающей в рамках EC2 или же может быть помещен на длительное хранение в систему S3.
– *Сервисы реляционных баз данных* – это веб-сервисы, обеспечивающие установку, управление и масштабирование реляционных баз данных в об-

лаке. В настоящий момент предоставляется поддержка баз данных на основе MySQL (в дальнейшем планируется внедрение и баз данных Oracle).

– *Amazon CloudFront* является веб-сервисом для предоставления контента. Обеспечивается статическая и потоковая передача данных. В связи с распределенной инфраструктурой системы, CloudFront может обеспечить минимальную латентность предоставления информации, выбирая наиболее географически-близкий к пользователю сервер.

Одной из отличительных особенностей платформы, предоставляемой Amazon, является предоставление сервисов интеграции приложений, обеспечивающих прозрачную адресацию и доступность:

– *Elastic IP* обеспечивает привязку статического IP адреса к аккаунту пользователя. При этом, Elastic IP может обеспечить обход ошибок и сбоев в работе отдельных виртуальных машин, обеспечивая автоматическое переназначение IP адреса другой виртуальной машине.
– *Simple Queue Service (SQS)* обеспечивает разработчиков практически бесконечным количеством очередей. Каждое авторизованное приложение может регистрироваться в очереди, отправлять, получать или удалять сообщения. При этом не принятые сообщения могут оставаться в системе вплоть до 4-х дней.
– *Simple Notification Service (SNS)* – это сервис, обеспечивающий оповещение об изменении состояния по подписке. Пользователи системы, облачные приложения и устройства могут отправлять и получать оповещения из облака.
– *Сервисы реляционных баз данных* – это веб-сервисы, обеспечивающие установку, управление и масштабирование реляционных баз данных в облаке. В настоящий момент предоставляется поддержка баз данных на основе MySQL (в дальнейшем планируется внедрение и баз данных Oracle).

Также, в настоящий момент Amazon выходит на корпоративный рынок, предоставляя услуги создания виртуальных частных облаков, расширяющих возможности корпоративных вычислительных инфраструктур. По сути, предлагается организация виртуальных частных сетей (VPN), обеспечивающих безопасную и прозрачную связь между внутренней корпоративной сетью и EC2. Обеспечивается импорт корпоративных виртуальных машин, а также возможность пересылки виртуальных машин на физических носителях, минуя интернет (повышается надежность и безопасность передачи данных).

Решение, предлагаемое Amazon Web Services, обеспечивает максимально возможную гибкость при разработке и внедрении облачных решений. Наиболее активными пользователями данной платформы являются крупные организации, или проекты, которым необходима максимальная масштабируемость и гибкость разработки. При этом такое решение может оказаться чрезмерно сложным и нагруженным для отдельных разработчиков или приложений, которым не требуется настолько мощные механизмы масштабирования.

12.6.2 Google App Engine

Платформа Google App Engine является одним из наиболее известных облачных сервисов, ориентированных на предоставление платформы (PaaS) для облачных приложений. Наряду с готовой средой исполнения облачных приложений, предоставляются расширенные средства для администрирования и разработки приложений. На рисунке представлена принципиальная схема архитектуры системы Google App Engine.

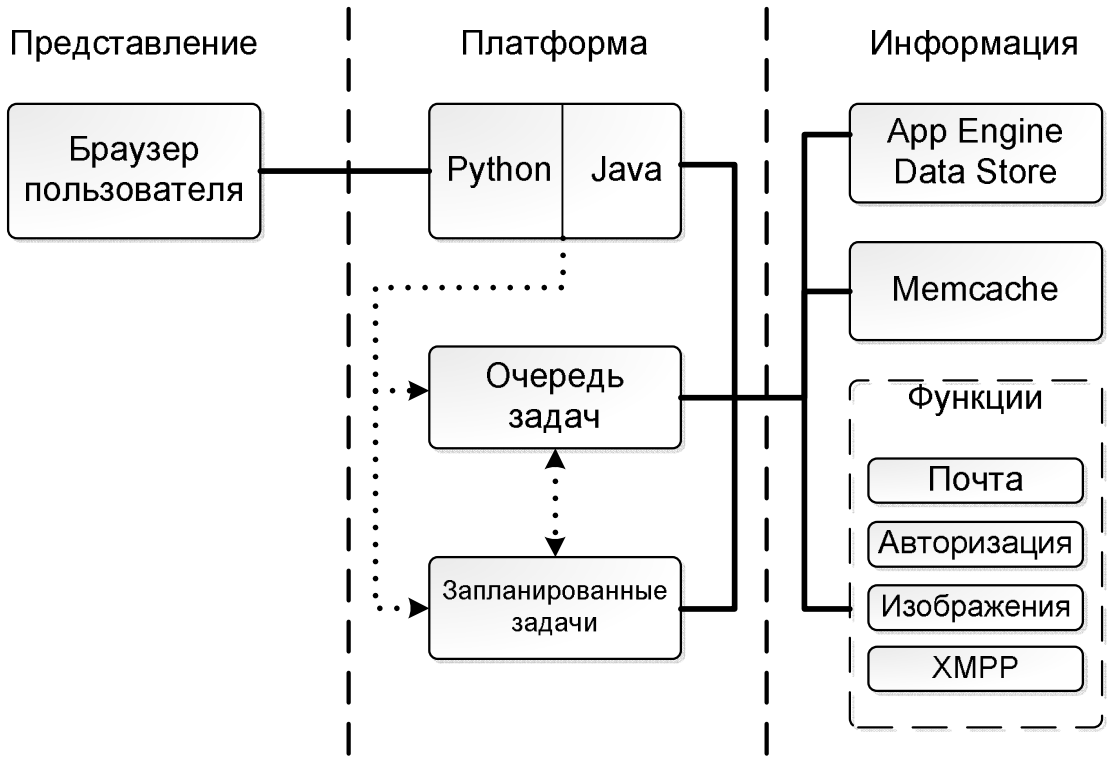


Рис. 64. Архитектура системы Google App Engine

На рисунке представлена упрощенная схема архитектуры системы Google App Engine. Пользователь получает доступ к облачному приложению посредством веб-браузера. До недавнего времени Google App Engine обеспечивало возможность разработки только на базе языка Python, но недавно была представлена поддержка разработки на базе любого языка, который может исполняться внутри виртуальной машины Java. Таким образом, в настоящий момент,

Google App Engine может запускать приложения написанные на Java, Jython, Scala и т.п. Разработчикам Google App Engine предоставляется полноценный комплект средств разработки (SDK), включающий в себя полноценную симуляцию работы Google App Engine на рабочей машине.

Исполнение в облаке накладывает определенные ограничения на набор библиотек, доступных разработчику. Например, в настоящее время не поддерживаются модули Python, написанные с использованием языка C. Аналогично Java-приложения могут использовать только ограниченный подкласс библиотек JRE SE, а также не могут работать с потоками. В связи с этим практически невозможно просто взять и скопировать готовое Python или Java приложение в Google App Engine и надеяться, что оно запустится в облаке. Основные трудности, с которыми придется столкнуться при переносе приложения в облако Google App Engine – это работа с хранилищем App Engine Datastore и ограничения доступа к локальным ресурсам в связи с работой в «песочнице» виртуальной машины.

Хранилище *App Engine Datastore* – это высоко распределенная система хранения данных, основывающаяся на проприетарной базе данных BigTable, разработанной компанией Google. Методы работы с хранилищем очень сильно отличаются от методов работы с реляционными базами данных. Основным отличием является то, что в App Engine Datastore отсутствуют схемы данных. Все данные хранятся в виде блоков «ключ»-«значение»-«метка времени». Соответственно, вся ответственность на соответствие сохраненных данных бизнес-логике приложения ложится на разработчика. Естественно, в SDK предоставляются специальные библиотеки, которые позволяют обеспечить согласованность данных. Также, Google разработали собственный язык запросов (язык GQL), который похож на выражения SELECT в языке SQL, но с большим количеством ограничений (например, отсутствует оператор JOIN).

Вторым важным ограничением, накладываемым на разработчиков приложений в рамках Google App Engine, является необходимость работы в рамках «песочницы» (виртуальной машины с ограниченным доступом к локальным ресурсам). Например, значительно ограничен набор протоколов и портов, по которым приложение может связываться со внешним миром (практически оставлена возможность связи только посредством HTTP и HTTPS). Также, приложению запрещены операции работы с локальной файловой системой (разрешено только чтение файлов, которые были загружены вместе с кодом приложения). С другой стороны, Google App Engine предоставляет большой набор библиотечных функций для выполнения стандартных операций:

– работы с почтовыми сообщениями;
– авторизации и аутентификации пользователей;
– обработки изображений;
– загрузки и обработки веб-страниц;
– планирования задач;
– обработки данных посредством MapReduce;
– хранения больших (до 2 GB) объемов информации;
– обмена сообщениями на основе протокола XMPP (Jabber).

Таким образом, не смотря на то, что Google App Engine не позволит вам развернуть облачное приложение, просто скопировав ваш старый код на Java или Python, он может обеспечить возможность предоставления высокопроизводительного сервиса без существенных затрат на инфраструктуру. Но процесс разработки облачного приложения может потребовать больших затрат, связанных с использованием новых моделей работы с данными и ограничениями виртуальной машины.

12.6.3 Microsoft Windows Azure

Платформа Windows Azure это PaaS от корпорации Microsoft, позволяющая запускать приложения, разработанные на базе платформы Microsoft .NET на серверах компании Microsoft [3]. Обеспечивается автоматическое управление вычислительными ресурсами, балансировка нагрузки и репликация данных.

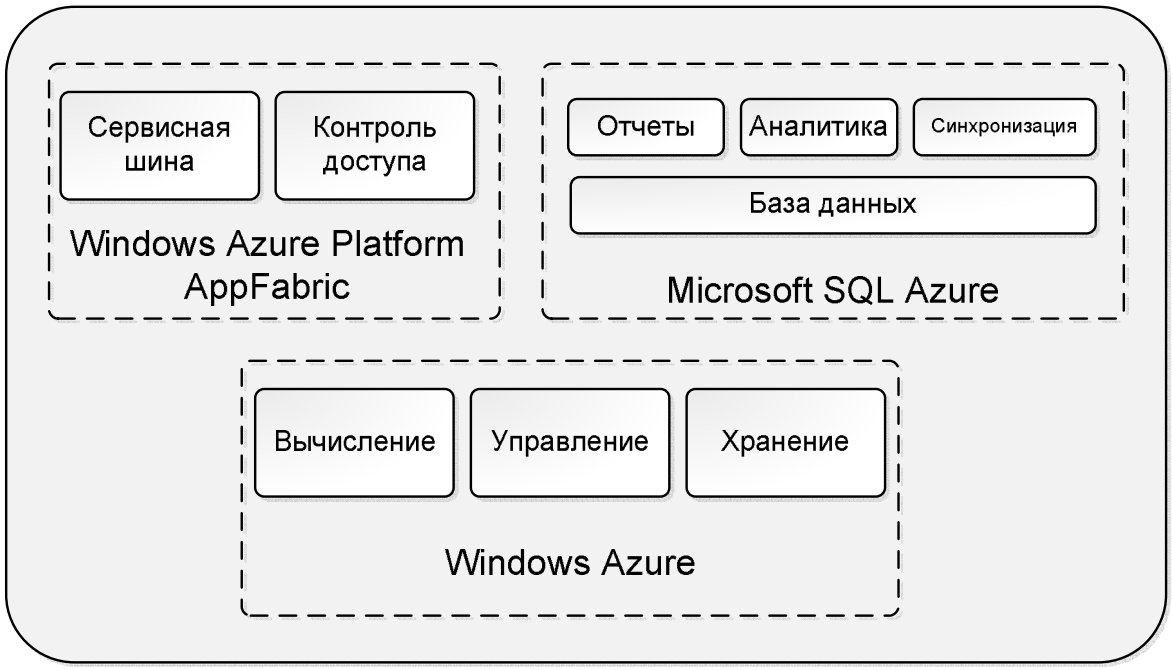


Рис. 65. Платформа Windows Azure

Доступ разработчика к платформе осуществляется посредством инструментария, интегрированного в последние версии Visual Studio (Windows Azure SDK, Windows Azure Tools for MVS). При этом поддерживается возможность локального тестирования приложений до их публикации на сервисе Azure.

Windows Azure создана на основе технологий виртуализации, схожих с технологией Windows Server Hyper-V и управляется с помощью специального инфраструктурного слоя, называемого Windows Azure Fabric Controller. Задача данного слоя – организация массива виртуальных машин на основе Windows Server в виде единого виртуального сервера, обеспечивающей автоматическое управление ресурсами, балансировкой нагрузки и др.

Платформа Windows Azure состоит из трех основных компонент: вычислительных сущностей, сущностей хранения и фабрик.

Вычислительные сущности – это контейнеры для приложений с поддержкой современных технологий разработки, включая .NET, Java, PHP, Python, Ruby on Rails и нативный код. Существует 2 основных роли, которые могут исполнять вычислительные сущности. *Веб-роль (Web Role)* – это веб-приложение, доступное пользователю через интернет, доступ к которой можно получить посредством веб-браузера. *Прикладная роль (Worker Role)* – это приложение, которое выполняет некоторую вычислительную нагрузку в фоновом режиме (что-то на подобие службы в Microsoft Windows).

Сущности хранения – это набор сервисов, обеспечивающих хранение данных. Каждый сервис обеспечивает свой тип хранения данных.

- *Бинарные объекты* – обеспечивается хранение больших наборов неструктурированных байтов (файлов). Обеспечивается возможность именования файлов и работы с метаданными. Максимальный объем хранимого файла – не более 1 Тб.
- *Таблицы* – используются для хранения структурированных данных. Таблицы представляют собой набор однородных строк (называемых сущностями) структура которых определяется набором колонок (называемых свойствами). Один объект может иметь до 256 свойств. Таблицы распределены таким образом, чтобы максимально поддерживать балансировку нагрузок.
- *Очереди* – механизм обеспечивающий асинхронное взаимодействие приложений посредством хранения и передачи сообщений. Допускается использование неограниченного числа очередей, а очереди могут содержать неограниченное число сообщений.
- *Диски* – аналогично AWS, Windows Azure позволяет управлять дисками в рамках своих виртуальных машин. Azure позволяет работать с томами

NTFS, обеспечивая их доступность для приложений. Это может позволить упростить процесс миграции обычных приложений в облако, т.к. появляется возможность сохранения состояния в файловой системе.

Также платформа Windows Azure обеспечивает ряд сервисов, доступных разработчикам как интернет-, так и классических десктопных приложений. Наиболее важным является компонент *SQL Azure*, обеспечивающий предоставление реляционной базы данных Microsoft SQL Server как сервиса. Так как данная технология основана на классическом подходе SQL Server, она не обеспечивает такой массивной масштабируемости как сущности хранения (максимальный объем хранимой информации составляет 10 Гб на одну базу данных). Но, не смотря на это, нельзя недооценивать все те возможности, которые могут быть предоставлены классической архитектурой, включая транзакционную целостность и возможность анализа данных. Также, предоставляются службы построения отчетов.

Очевидно, что решение Microsoft будет пользоваться популярностью на рынке облачных сервисов. Прежде всего, это связано с распространенностью решений Microsoft на рынке корпоративных систем, а технология Windows Azure предоставляет множество сервисов, призванных облегчить перенос таких приложений в облачную платформу. При этом естественно, что вся инфраструктура Azure заточена на программные продукты и технологии, предоставляемые компанией Microsoft. Соответственно, если ваше приложение разрабатывается на основе альтернативной идеологии, весьма возможно стоит поискать альтернативные облачные платформы.

12.7 Сравнение Грид и Облачных вычислений

Описание концепции грид-вычислений в предыдущей главе и концепции облачных вычислений, которая была представлена в этой, показывают, что между ними много общего. Это повлекло за собой множество дискуссий, как в коммерческой, так и в научной среде. Основной вопрос этих дискуссий заключался в следующем: чем облачные и грид вычисления отличаются друг от друга? Действительно ли термин «облачные вычисления» – это просто новая маркетинговая уловка, под соусом которой публике предоставляются услуги грид-сред?

Некоторые исследователи считают, что основным отличием облачных вычислений от грид является виртуализация: «Облачные вычисления, в отличие от грид, применяют виртуализацию для максимизации вычислительной мощно-

сти. Виртуализация, посредством отделения логического уровня от физического, решает множество проблем, с которыми сталкиваются грид-решения».

В то время как грид-системы обеспечивают высокую загрузку вычислительных ресурсов посредством распределения одной сложной задачи на несколько вычислительных узлов, облачные вычисления идут по пути исполнения нескольких задач на одном сервере в виде виртуальных машин. Кроме того, есть особенности в основных вариантах использования грид и облачных вычислений. Тогда как грид, в основном, используется для решения задач за определенный (ограниченный) промежуток времени, облачные вычисления в основном ориентированы на предоставление «долгоживущих» сервисов.

Мнения сходятся в одном – облачные вычисления выросли из концепции грид. Фостер определяет взаимодействие грид и облачных вычислений следующим образом:

«Мы считаем, что облачные вычисления не просто пересекаются с концепцией грид. На самом деле облака выросли из грид-вычислений и основываются на концепции инфраструктуры грид. Эволюция подхода заключается в том, что вместо предоставления «сырых» вычислительных ресурсов и ресурсов хранения обеспечивается предоставление более абстрактных ресурсов в виде сервисов».

Таким образом, можно считать что грид и облачные вычисления дополняют друг друга. Интерфейсы и протоколы грид могут обеспечить взаимодействие между облачными ресурсами или же обеспечить объединение облачных платформ. Также, более высокий уровень абстракции, предоставляемый облачными платформами, может помочь пользователям грид-систем в организации прозрачного и удобного предоставления ресурсов грид-платформ и привлечь новые группы пользователей к использованию таких ресурсов.

С точки зрения пользователя, разница между облачными вычислениями и грид вычислениями будет состоять в следующем:

– *Облачные платформы фокусируются на подходе «всё как сервис»*. Грид вычисления фокусируются на промежуточном программном обеспечении, которое предоставляется в виде открытых исходных кодов или же в виде готовых пакетов. При этом коммунальные вычисления являются всего лишь одной из форм предоставления грид. По сравнению с этим, облачные вычисления фокусируются исключительно на платном предоставлении информационных ресурсов конечному пользователю. При этом промежуточное программное обеспечение, которое позволило бы обеспечить разработку собственного облака, пока не очень распространено.

– *Грид и облачные вычисления фокусируются на различные типы вычислений*. Изначально, грид вычисления были ориентированы на решение научных задач посредством суперкомпьютерных систем. В настоящее время грид применяется для научно-исследовательских задач, решение которых требует объединение нескольких суперкомпьютерных платформ. С другой стороны, облачные вычисления ориентированы не на решение отдельных задач, а на перманентное предоставление определенных сервисов конечным пользователям. Они обеспечивают динамическое распределение физических ресурсов для удовлетворения переменной загрузки таких сервисов.
– *Различное взаимоотношение с поставщиками ресурсов*. Грид вычисления основываются на понятии виртуальных организаций, включающих в себя несколько различных отдельных организаций с четкими правилами взаимодействия между ними и четкими политиками предоставления программно-аппаратных ресурсов. Концепция облачных вычислений обеспечивает возможность любой компании использовать облачные сервисы для решения собственных задач, оплачивая только те ресурсы, которые необходимы.
– *Различные области применения*. Грид-платформы предоставляют базу для развертывания вычислительной инфраструктуры. Облачные вычисления предоставляют интегрированный подход на всех уровнях предоставления информационных ресурсов: IaaS, PaaS, SaaS.
– *Расширение количества пользовательских интерфейсов*. Грид вычисления ориентированы на представление различных вычислительных ресурсов в гетерогенных вычислительных средах для решения конкретных задач. Таким образом, интерфейсы грид ориентированы на взаимодействие вычислительных инфраструктур на физическом уровне посредством API, которым может воспользоваться только профессиональный программист. Облачные вычисления разрабатываются таким образом, чтобы предоставлять интерфейсы конечным пользователям через веб-доступ или посредством API. На каждом слое (IaaS, PaaS, SaaS) предоставляется свой собственный интерфейс. Повышение уровня абстракции позволяет обеспечить применение облачных вычислений как на уровне отдельных пользователей, так и на уровне корпоративных клиентов.

  В общем и целом, грид вычисления обеспечивают объединение гетерогенных вычислительных ресурсов в единую вычислительную среду. Это то, с чего начинаются и на чем основываются облачные вычисления. Облачные

вычисления обеспечивают более высокий уровень абстракции, предоставляя вычислительные ресурсы конечным пользователям (будь то частные клиенты или организации) в виде сервисов.

Список литературы

1. Миков А.И., Замятина Е.Б. Распределенные системы и алгоритмы. Интуит.ру, 2008. 370 с.
2. Таненбаум Э., Ван-Стеен М. Распределенные системы. Принципы и парадигмы. Спб.: Питер, 2003. 877 с.
3. Федоров А., Мартынов Д. Windows Azure. Облачная платформа Microsoft. Microsoft, 2010. 96 с.
4. Черняк Л. Web-сервисы, grid-сервисы и другие // Открытые системы. СУБД. №12. 2004. С. 20-27.
5. Alhaisoni M., Liotta A. Characterization of signaling and traffic in Joost // Peer-to-Peer Networking and Applications. 2008. Vol. 2, No. 1. P. 75-83.
6. Berners-Lee T. Universal Resource Identifiers – Axioms of Web Architecture. URL: http://www.w3.org/DesignIssues/Axioms.html (дата обращения: 19.05.2011)
7. Bettstetter C., Renner C. A comparison of service discovery protocols and implementation of the service location protocol // Proceedings of Sixth EUNICE Open European Summer School – EUNICE 2000. 2000. URL: http://citeseer.ist.psu.edu/334042.html (дата обращения: 19.05.2011).
8. BitTorrent. URL: http://www.bittorrent.com/ (дата обращения: 19.05.2011).
9. BOINC – Berkeley Open Infrastructure for Network Computing. URL: http://boinc.berkeley.edu/ (дата обращения: 19.05.2011).
10. Brogi A., Corfini S. SAM: A Semantic Web Service Discovery System // Knowledge-Based Intelligent Information and Engineering Systems. Vol. 4694. 2008. P. 703-710.
11. Clark D. Face-to-Face with Peer-to-Peer Networking. Computer, Vol. 34, No. 1, January 2001, pp. 18-21.
12. Cougaar – Cognitive Agent Architecture. URL: http://cougaar.org/ (дата обращения: 19.05.2011).
13. Czajkowski K., Ferguson D., Foster I. et al. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. – The Globus Project Whitepaper. 2004. URL: http://www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf (дата обращения: 02.06.2009).

14. Defining Web Services / The Stencil Group, 2001. URL: http://www.site.uottawa.ca/~stan/csi5389/readings/wsdefined.pdf (дата обращения: 03.06.2009).
15. Ernemann C. et al. On advantages of grid computing for parallel job scheduling. // Proc. of the 2nd IEEE Int'l. Symp. on Cluster Computing and the Grid (CCGrid). Washington, DC: IEEE Computer Society, 2002. P 39-49.
16. Ferguson D. Trends and statistics in peer-to-peer. Workshop on technical and legal aspects of peer-to-peer television. Amsterdam, Netherlands, 2006, Mar. 17. 2006. URL: http://www2.noticiasdot.com/publicaciones/2006/0406/1804 /noticias/images/CacheLogic_AmsterdamWorkshop_Presentation_v1.0.ppt (дата обращения: 02.06.2009).
17. Finlayson N., Morrison J. P2P and Client-Server Hybrids: Groove-enabling a J2EE portal using web services // Networking and Electronic Commerce Research Conference (NAEC 2005), October 6-9, 2005, Riva Del Garda, Italy. URL: http://www.atlanticshack.com/resources/P2P_and_Client-Server_NAEC2005_final.pdf (дата обращения: 15.05.2009).
18. Foster I. et al. Grid Services for Distributed System Integration // Computer. Vol. 35, Issue 6. 2002. P. 37-46.
19. Foster I. et al. How Do I Model State? Let Me Count the Ways // Queue. Vol. 7, Issue 2. 2009. P. 54-64.
20. Foster I. Globus Toolkit Version 4: Software for Service-Oriented Systems // IFIP International Conference on Network and Parallel Computing. Springer, 2005. P. 2-13.
21. Foster I. Service-Oriented Science // Science. 2005. Vol. 308, No. 5723. P. 814–817.
22. Foster I., Frey J., Graham S. et al. Modeling Stateful Resources with Web Services / The Globus Project Whitepaper, 2004 URL: http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf (дата обращения: 16.05.2009).
23. Foster I., Geisler J., Nickless W. Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment // Proc. 5th IEEE Symposium on High Performance Distributed Computing. pp. 562-571, 1997.
24. Foster I., Iamnitchi A. On death, taxes, and the convergence of peer-to-peer and grid computing // In 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03). Springer, 2003. P. 118-128.
25. Foster I., Jennings N.R., Kesselman C. Brain Meets Brawn: Why Grid and Agents Need Each Other // Proceedings of the Third International Joint

Conference on Autonomous Agents and Multiagent Systems. Washington, DC: IEEE Computer Society, 2004. Vol. 1. P. 8-15.

26. Foster I., Kesselman C. Globus: A Metacomputing Infrastructure Toolkit // International Journal of Supercomputer Applications Vol. 11, Issue 2. 1997. P. 115-128.
27. Foster I., Kesselman C. The Grid. Blueprint for a new computing infrastructure. San Francisco: Morgan Kaufman, 1999. 677 p.
28. Foster I., Kishimoto H., Savva A. et al. The Open Grid Services Architecture. 2005. 62 p. URL: http://forge.gridforum.org/projects/ogsa-wg pdf (дата обращения: 16.05.2009).
29. Foster I., Zhao Y., Raicu I, Lu S. Cloud computing and grid computing 360-degree compared // Grid Computing Environments Workshop, 2008. GCE'08. pp. 1-10, 2008.
30. Foster I., Kesselman C., Nick J., Tuecke S. The Physiology of the Grid: An Open Grid Service Architecture for Distributed Systems Integration. / Global Grid Forum. 2002. URL: http://www.globus.org/ogsa/ (дата обращения: 14.05.2009).
31. Foster I., Kesselman C., Tuecke S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations // International Journal of Supercomputer Applications and High Performance Computing. 2001. Vol. 15, No 3. P. 200-222.
32. Franklin S., Graesser C. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents // Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop (ATAL), Budapest, Hungary, August 12-13, 1996, Proceedings, Lecture Notes in Computer Science. 1997. Vol. 1193. pp. 22-35.
33. Gnutella. URL: http://www.gnutella.com/ (дата обращения: 25.05.2009)
34. Gray P. et al. Advances in heterogeneous network computing // Lecture Notes in Computer Science. 1998. Vol. 1497. pp. 83-92(дата обращения: 12.11.2011).
35. Grimshaw A., Wulf W. et al. The Legion Vision of a Worldwide Virtual Computer // Communications of the ACM, vol. 40(1), January 1997.
36. Guha S., Daswani N., Jain R. An experimental study of the Skype peer-to-peer VoIP system. In: Proceedings of the IPTPS'06. Santa Barbara, CA, Feb. URL: http://iptps06.cs.ucsb.edu/talks/guha-skype-talk.pdf (дата обращения: 25.05.2009).
37. Henning M. The Rise and Fall of CORBA // ACM Queue. Vol. 4, Num. 5. 2006. P. 28-34.

38. Iosup A., Epema D. et al. On Grid Performance Evaluation using Synthetic Workloads // CoreGRID Technical Report Number TR-0039. 2006. 18 p.
39. JADE (Java Agent DEvelopment Framework) URL: http://jade.tilab.com (дата обращения: 27.01.2012)
40. Jennings N. R. An agent-based approach for building complex software systems // Comms. of the ACM, 44 (4) 35-41, 2001.
41. Josuttis N. SOA in Practice. The Art of Distributed System Design. Sebastopol, CA: O'Reilly Media Inc. 2007. 342 p.
42. Kamel M., Leue S. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN // International Journal on Software Tools for Technology Transfer (STTT). Vol. 2, No. 4. 2000. P. 394-409.
43. King J.L. Centralized versus decentralized computing: organizational considerations and management options // ACM Computing Surveys. Vol. 15, Issue 4. 1983. P. 319-349.
44. Lamport L. Distributin. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt (дата обращения: 06.11.2011).
45. Li J. On peer-to-peer (P2P) content delivery // Peer-to-Peer Networking and Applications. 2008. Vol. 1, No. 1. P. 45-63.
46. Li J. On peer-to-peer (P2P) content delivery // Peer-to-Peer Networking and Applications. 2008. Vol. 1, No. 1. P. 45-63.
47. Li S. et al. ABSDM: Agent Based Service Discovery Mechanism in Internet // Computational Science – ICCS 2004. Vol. 3036/2004. 2004. P. 441-444.
48. Memon M. S. et al. Enhanced resource management capabilities using standardized job management and data access interfaces within UNICORE Grids // 13th International Conference on Parallel and Distributed Systems. Julich: Central Inst. of Appl. Math, 2007. Vol. 2. P. 1-6.
49. Milojicic D. S. et al. Peer-to-Peer Computing, Hewlett-Packard, Tech. Rep. HPL-2002-57R1. 2003. URL: http:// www.hpl.hp.com/ techreports/ 2002/HPL-2002-57R1.html (дата обращения: 19.05.2009).
50. Pallos M. S. Service-oriented architecture: A primer // eAI Journal. 2001. P 32-35.
51. Papazoglou M.P., Georgakopoulos D. Service-Oriented Computing // Communications of the ACM. Vol. 46, No. 10. 2003. P. 25-28.
52. Peer-to-peer Working Group. URL: http://www.p2pwg.org (дата обращения: 19.05.2009).

53. Rehrl K. et al. Towards a Service-Oriented Architecture for Mobile Information Systems // Mobile Information Systems. Boston: Springer. 2005. P. 37-50.
54. Schmidt C., Parashar M. A Peer-to-Peer Approach to Web Service Discovery // World Wide Web. Vol. 7, No. 2. 2004. P. 211-229.
55. Seely S. Understanding WS-Security. Microsoft corp. URL: http://msdn.microsoft.com/en-us/library/ms977327.aspx (дата обращения: 13.11.2011).
56. Service Oriented Architecture (SOA) Reference Model Public Review Draft 1.0 (Feb) / Organization for the Advancement of Structured Information Standards (OASIS), 2006. URL: http://www.oasisopen.org/committees /download.php/16587/wdsoa-cd1ED.pdf (дата обращения: 4.06.2009).
57. Shan H., Oliker L., Biswas R. Job superscheduler architecture and performance in computational grid environments // Proceedings of the ACM/IEEE conference on Supercomputing. Washington, DC: IEEE Computer Society, 2003. P. 44–54.
58. Siegel J. OMG overview: CORBA and the OMG in enterprise computing. Communications of the ACM, vol. 41, no. 10, 1998. pp. 37-43.
59. Skype. URL: http://www.skype.com (дата обращения: 02.06.2009).
60. Sotomayor B. The Globus Toolkit 4 Programmer's Tutorial / University of Chicago, Department of Computer Science, 2005. URL: http://gdp.globus.org/gt4-tutorial/download/progtutorial-pdf_0.2.1.tar.gz (дата обращения: 02.06.2009).
61. Stevens R. et. al. From the I-WAY to the National Technology Grid // Communications of the ACM. Vol. 40, Issue 11. 1997. P. 50-60.
62. Stockinger H. Defining the Grid: A Snapshot on the Current View // The Journal of Super-computing. 2007. № 42(1). P. 3-17.
63. Strang T. Towards autonomous context-aware services for smart mobile devices // MDM 2003 (4th International Conference on Mobile Data Management). Vol. LNCS 2574. 2003. P. 279-292.
64. Streit A. UNICORE – What lies beneath Grid functionality? // eStrategies. Vol. 7. 2008. P. 38-39.
65. Streit A. UNICORE: Getting to the heart of Grid technologies // eStrategies. Vol. 3. 2009. P. 8-9.
66. Sundaram B. Understanding WSRF, Part 1: Using WS-ResourceProperties. 2005. URL: http://www.ibm.com/developerworks/edu/gr-dw-gr-wsrf1-i.html (дата обращения: 02.06.2009).
67. Taylor I., Harrison A. From P2P and Grids to Services on the Web. Springer. 2008. 462 p.

68. The Distributed Component Object Model (DCOM). See: http://www.microsoft.com/com/tech/DCOM.asp (дата обращения: 06.11.2011).
69. Touch J. Overlay Networks // Computer Networks, vol. 36, pp. 115-116. 2001.
70. Vaquero L. M. et al. A break in the clouds: towards a cloud definition // ACM SIGCOMM Computer Communication Review, 2009. Vol. 39. P. 50-55.
71. Vogels W. Web Services Are Not Distributed Objects // IEEE Internet Computing. Vol. 7, No. 6. 2003. P. 59-66.
72. W3C Working Group Note. Web Services Architecture, 11 February 2004. URL: http:// www.w3.org/TR/2004/NOTE-ws-arch-20040211/ (дата обращения: 03.06.2009).

Предметный указатель

A

Amazon Elastic Compute Cloud (Amazon EC2), 162
Amazon Web Services (AWS), 161

B

Berkeley Open Infrastructure for Network Computing (BOINC), 125
BitTorrent, 121, 123, 126

C

Common Object Request Broker Architecture (CORBA), 20, 57

D

Distributed Hash Table (DHT), 124, 127
Domain Logic. *См.* Бизнес-логика
Domain Name System (DNS), 12

E

eDonkey, 123
Enterprise JavaBeans (EJB), 75
  Домашний интерфейс, 79
  Сессионные компоненты, 78
  Сущностные компоненты, 78
  Удаленный интерфейс, 79

F

Factoring via Network-Enabled Recursion (FAFNER), 17

G

General Inter-ORB Protocol (GIOP), 58
Globus, 19, 136
Gnutella, 12, 22, 122, 123
Google App Engine, 164

H

Hypertext Transfer Protocol (HTTP). *См.* Протокол передачи гипертекста

I

Information Wide Area Year (I-WAY), 17
Infrastructure as a Service (IaaS). *См.* Инфраструктура как сервис
Interface Definition Language (IDL), 57
Internet Inter-ORB Protocol (IIOP), 58

J

Java RMI, 56
JavaBeans, 74

L

Legion, 19
Loose coupling. *См.* Слабосвязанность

M

Magnet-ссылки, 127
Microsoft Windows Azure, 166

N

Napster, 21, 121, 123

O

Object Management Architecture (OMA), 57
Object Management Group (OMG), 57
Object Resource Broker (ORB), 57
Open Grid Service Infrastructure (OGSI), 135
Open Grid Services Architecture (OGSA). *См.* Открытая архитектура грид-сервисов

P

Pay-as-you-go. *См.* Оплата по мере использования
Peer-to-peer (P2P). *См.* Одноранговые сети
Platform as a Service (PaaS). *См.* Платформа как сервис
Proxy. *См.* Посредник

R

Remote Method Invocation (RMI). *См.* Вызов удаленных методов

Remote Procedure Call (RPC). *См.* Вызов удаленных процедур

S

Service-Oriented Architecture (SOA). *См.* Сервис-ориентированная архитектура
SETI@home, 23, 125
Skeleton. *См.* Каркас
Skype, 128
SOAP, 93, 99
SOAP HTTP Binding, 101
Software as a Service (SaaS). *См.* Программное обеспечение как сервис
Software-as-a-Service (SaaS), 146
Stateless сервисы, 91

T

Target System Interface (TSI), 139

U

Uniform Interface to Computing Resources (UNICORE), 138
Uniform Resource Identifier (URI). *См.* Унифицированный идентификатор ресурса
Uniform Resource Locator (URL). *См.* Унифицированный локатор ресурса
Uniform Resource Name (URN). *См.* Унифицированное имя ресурса
Universal Description Discovery & Integration (UDDI), 94
Utility Computing. *См.* Коммунальные вычисления

W

Web Service Resource Framework (WSRF), 136
Web Services Description Language (WSDL), 93, 95
Web Services Resource Framework (WFRF), 112
WS-Addressing, 109
  EndpointReference, 110
WS-Security, 103
WS-ресурс, 112

X

XML Encryption, 108
XML Signature, 107

А

Автономность сервисов, 89
Агент
  автономный, 64
  интеллектуальный, 66
  программный, 24
  программный, 63
Агентная платформа, 68
Агентная система, 68

Б

Бизнес-логика, 43

В

Веб-сервис, 24, 82, 93
Веб-служба. *См.* Веб-сервис
Виртуализация, 148
Виртуальная организация (ВО), 131
Вызов удаленных методов, 54
Вызов удаленных процедур, 51

Г

Гетерогенные вычислительные среды, 9
Гибридные облака, 159
Грид, 15, 18, 131
Грид-сервис, 135

З

Заголовки HTTP, 35
  DELETE, 35
  GET, 35
  HEAD, 35
  OPTIONS, 36
  POST, 35
  PUT, 35

И

Идемпотентный метод, 38
Интероперабельность, 84
  семантическая, 84
  техническая, 84
Инфраструктура как сервис, 150

К

Каркас, 55
Клиент, 10, 41
Клиент-серверная архитектура, 41
  вертикальное распределение, 48
  горизонтальное распределение, 48
Коммунальные вычисления, 146
Контейнер EJB, 77
Контракт сервиса, 81

Л

Логика предметной области. *См.* Бизнес-логика

М

Маршрутизация документов, 123
Масштабируемость, 148
Метрика
  девиации среднего времени ожидания (AWTD), 142
  завершения действий (Task Completion – TC), 143
  завершения разблокированных действий (Enabled Task Completion – ETC), 143
  завершенного объема работы (Workload Completion – WC), 143
  среднего времени ожидания (Average Wait Time – AWT), 141
  среднего времени ответа (Average Response Time – ART)), 141
  эффективности грид (Grid Efficiency – GE), 142
Мультиагентная система, 67

О

Облачные вычисления, 25, 146, 147
Общественное облако, 159
Одноранговые сети, 21, 117
Оплата по мере использования, 146
Открытая архитектура грид-сервисов, 135

П

Пир, 10, 120
Платформа как сервис, 151
Посредник, 55
Программное обеспечение как сервис, 151
Программный компонент, 73
Промежуточное программное обеспечение (ППО), 9
Протокол, 13
Протокол P2P, 121
Протокол передачи гипертекста, 35

Р

Распределенная вычислительная система (РВС), 8–9
Регистры сервисов, 82
Ресурс, 10

С

Связанность, 83
Сервер, 10, 41
Сервис, 10
Сервисные компоненты, 81
Сервис-ориентированная архитектура, 23
Сервис-ориентированная архитектура (СОА), 81
Сервисы группы, 120
Сервисы пира, 120
Сериализация, 55
Сертификат X.509, 106
Слабосвязанность, 83, 87
Соединитель сервисов, 82
Стаб, 52

Т

Трекер, 121

У

Узел, 10
Унифицированное имя ресурса, 29
Унифицированный идентификатор ресурса, 29
Унифицированный локатор ресурса, 29

Ф

Федерации облаков, 160

Ч

Частное облако, 159

Ш

Шаблон URI, 33
Широковещательные запросы, 123

Подписано в печать 14.09.11.,
Формат 60х84 1/16. Бумага офсетная.
Усл.печ. л. 11,25. Тираж 50 экз. Заказ 0112.
Отпечатано в типографии «Фотохудожник»
454091, г. Челябинск, ул. Свободы, 155/1, тел. (351) 237-17-43