Содержание

Предисловие

Это третье издание книги «Распределенные системы». Во многих отношениях она сильно отличается от предыдущих выпусков, и, возможно, самое важное состоит в том, что мы полностью обобщили «принципы» и «парадигмы», включив последние в соответствующие главы, где обсуждаются принципы распределенных систем.

Материал был существенно переработан и дополнен, и в то же время мы были заинтересованы в ограничении общего объема книги. Поэтому он был сокращен более чем на 10 % по сравнению со вторым изданием, в основном за счет удаления материала по парадигмам. Для лучшего понимания материала книги широким кругом читателей мы перенесли конкретные материалы в отдельные выделенные разделы. Эти разделы могут быть пропущены при первом чтении.

Еще одним важным отличием является использование кодов примеров, написанных на языке программирования Python с поддержкой коммуникаций посредством пакета Redis. Примеры в книге опускают много деталей для удобства чтения, но полные примеры доступны на веб-сайте www.distributed-systems.net. Рядом с кодами для запуска, тестирования и расширений алгоритмов сайт предоставляет доступ к слайдам, всем рисункам и упражнениям.

Новый материал был проверен в учебном процессе, за что мы выражаем особую благодарность Тилю Кильманну (Thilo Kielmann) из университета Амстердама. Его конструктивные и критические замечания помогли нам значительно улучшить книгу.

Наш издатель Pearson Education любезно вернул нам авторские права, и мы должны сказать большое спасибо Трейси Джонсон (Tracy Johnson) за возможность осуществить этот плавный переход. Возврат авторских прав позволил нам начать то, что мы оба хотели сделать: запустить эксперимент. Он заключался в том, чтобы найти способ, обеспечивающий доступность материала, сделать его относительно недорогим и упростить процедуру обновления.

Книга теперь может быть (свободно) загружена, что делает намного более простым использование гиперссылок, где это уместно. В то же время предлагается и печатная версия, доступная через Amazon.com по минимальной цене.

Книга полностью оцифрована, что позволяет нам включать обновления, когда это необходимо. Мы планируем выпускать обновления ежегодно, оставляя доступными предыдущие цифровые версии, а также с некоторой периодичностью публиковать печатные версии книги. Часто выпускать обновления не всегда правильно с точки зрения перспектив обучения, но ежегодные обновления и поддержка предыдущих версий кажется нам хорошим компромиссом.

Мартен ван Стин
Эндрю С. Таненбаум

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Springer очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Введение[1]

Темпы изменения компьютерных систем были, есть и продолжают оставаться большими. С 1945 года, когда началась современная компьютерная эра, до 1985 года компьютеры были большими и дорогими. Более того, из-за ограниченных возможностей их соединения они работали независимо друг от друга.

Однако начиная с середины 1980-х годов два технологических достижения начали изменять ситуацию. Первым достижением была разработка мощных микропроцессоров. Первоначально это были 8-битные машины, но вскоре общедоступными стали 16-, 32- и 64-битные процессоры. С многоядерными процессорами мы сейчас переходим к решению проблемы адаптации и разработки программ для использования параллелизма. Цена компьютеров нынешнего поколения в тысячу раз меньше, чем цена мейнфреймов 30- или 40-летней давности, обладающих такой же вычислительной мощностью.

Вторым достижением было изобретение и разработка высокоскоростных компьютерных сетей. **Локальные сети** (Local-area networks, LAN) позволяют тысячам компьютеров в пределах одного здания быть связанными таким образом, что небольшие объемы информации могут быть переданы в течение нескольких микросекунд. Большие объемы данных могут перемещаться между машинами со скоростью миллиардов *бит в секунду* (бит/с, bps). **Глобальные сети** (Wide-area networks, WAN) позволяют сотням миллионов компьютеров по всему миру быть подключенными друг к другу и обмениваться информацией на скоростях от десятков тысяч до сотен миллионов бит/с.

Параллельно с развитием все более мощных и объединенных сетью машин мы стали свидетелями миниатюризации компьютерных систем, и, возможно, смартфон – здесь самый впечатляющий результат. Снабженные датчиками, большой памятью и мощным процессором, эти устройства стали полноценными компьютерами, и они, конечно, обладают и сетевыми возможностями. Находят свой путь к рынку и так называемые **подключаемые компьютеры** (plug computers). Эти небольшие компьютеры, часто размером с адаптер питания, могут быть подключены непосредственно ко входу устройства и выполнять роль настольного компьютера.

В результате появления этих технологий теперь несложно собрать компьютерную систему, состоящую из большого числа сетевых компьютеров, будь

---

[1] Версия данной главы была опубликована как «Краткое введение в распределенные системы». Computing, vol. 98 (10): 967–1009, 2016.

они большие или маленькие. Компьютеры такой системы, как правило, географически разбросаны, и потому обычно говорят, что они образуют **распределенную систему** (distributed system). Распределенная система может состоять как из нескольких, так и из миллионов компьютеров. Соединяющая их сеть может быть проводной, беспроводной или сочетанием того и другого. Кроме того, распределенные системы часто очень динамичны, в том смысле, что компьютеры могут присоединяться и отсоединяться, так что топология базовой сети и ее производительность почти непрерывно меняются.

В этой главе мы представим начальное исследование распределенных систем и целей их построения, а затем обсудим некоторые известные типы систем.

1.1. Что такое распределенная система?

В литературе приводятся различные определения распределенных систем, ни одно из них не является удовлетворительным и плохо согласуется с другими. Для наших целей достаточно дать такое достаточно широкое определение:

> *Распределенная система представляет собой совокупность автономных вычислительных элементов и является для его пользователей единой связанной системой.*

В этом определении отмечается две характерные особенности распределенных систем. Во-первых, распределенная система представляет собой совокупность вычислительных элементов, каждый из которых в состоянии работать независимо от других. Вычислительный элемент, который мы обычно будем называть узлом, может быть аппаратным устройством или программным процессом. Вторая особенность заключается в том, что пользователи (будь то люди или приложения) считают, что они имеют дело с единой системой. Это означает, что в той или иной степени автономные узлы должны сотрудничать. Существо такого сотрудничества лежит в основе разработки распределенных систем. Обратите внимание, что мы не делаем никаких предположений относительно типа узлов. В принципе, даже в пределах одной системы они могут варьироваться от высокопроизводительных мейнфреймов до небольших устройств в сенсорных сетях. Кроме того, мы не делаем никаких предположений относительно того, как узлы взаимосвязаны.

Характеристика 1: совокупность автономных вычислительных элементов

Современные распределенные системы могут и часто состоят из всех видов узлов, начиная от очень больших высокопроизводительных компьютеров и заканчивая маленькими компьютерами или даже еще меньшими устройствами. Основополагающий принцип заключается в том, что узлы могут

действовать независимо друг от друга, хотя очевидно, что если они игнорируют друг друга, то нет смысла помещать их в одну и ту же распределенную систему. На практике узлы запрограммированы для достижения общих целей, которые реализуются путем обмена сообщениями между ними. Узел реагирует на входящие сообщения, которые затем обрабатываются и, в свою очередь, ведут к общению посредством дальнейшей передачи сообщений.

Важным является то, что временна́я последовательность взаимодействия независимых узлов определяется каждым из узлов самостоятельно. Другими словами, не всегда можно предположить, что есть что-то вроде **глобальных часов** (global clock). Этот недостаток приводит к необходимости решения фундаментальных вопросов, касающихся синхронизации и координации внутри распределенной системы, которые мы обсудим подробно в главе 6. Тот факт, что мы имеем дело с *совокупностью* узлов, подразумевает, что нам может понадобиться организация и управление этой совокупностью. Другими словами, нам может понадобиться регистрация узлов, входящих и не входящих в систему, а также обеспечение всех членов системы списком узлов, с которыми они могут общаться напрямую.

Управление **членством в группе** (group membership) может быть чрезвычайно сложным, хотя бы по причине контроля допуска. Проще говоря, мы различаем открытые и закрытые группы. В **открытой группе** любой узел может присоединиться к распределенной системе, что означает, что она может отправлять сообщения любому другом узлу в системе. В **закрытой группе** общаться друг с другом могут только члены этой группы. Необходим также отдельный механизм, позволяющий узлу присоединиться к группе или покинуть ее.

Нетрудно видеть, что контроль допуска может быть сложным. Во-первых, необходим механизм аутентификации узла, и, как мы увидим в главе 9, если он неправильно разработан, управление аутентификацией может легко создать узкое место масштабируемости. Во-вторых, каждый узел должен, в принципе, проверить, является ли его общение действительно общением с другим членом группы, а не, например, со злоумышленником, стремящимся создать хаос. Наконец, учитывая, что член группы может легко начать общаться в системе с участниками, не являющимися членами его группы, то при общении внутри распределенной системы должна обеспечиваться конфиденциальность, иначе можно столкнуться с проблемами доверия.

Что касается организации совокупности узлов, практика показывает, что распределенная система часто организуется в виде **оверлейной сети** (overlay network) [Tarkoma, 2010]. В этом случае узел обычно представляет собой программный процесс, снабженный списком других процессов, которым он может напрямую отправлять сообщения. Может также оказаться, что соседу нужно будет начать общение первым. Передача сообщений осуществляется через TCP/IP- или UDP-каналы, но, как мы увидим в главе 4, могут быть доступны также средства более высокого уровня. Существует два типа оверлейных сетей.

1. **Структурное наложение:** в этом случае каждый узел имеет четко определенный набор соседей, с которыми он может общаться. Например, узлы организованы в виде дерева или логического кольца.

2. **Неструктурированное наложение:** в этом случае каждый узел имеет несколько ссылок на случайно выбранные другие узлы.

В любом случае, оверлейная сеть должна быть в принципе всегда **соединена**, то есть между любыми двумя узлами всегда должен существовать канал связи, позволяющий этим узлам отправлять сообщения друг другу. Хорошо известный класс оверлейных сетей формируется **одноранговыми** (peer-to-peer, P2P) сетями. Примеры оверлейных сетей будут подробно обсуждаться в главе 2 и последующих главах. Важно понимать, что организация узлов требует особых усилий и что это иногда одна из самых сложных задач управления распределенными системами.

Характеристика 2: единая связанная система

Как уже упоминалось, распределенная система должна выглядеть как единая связанная согласованная система. В некоторых случаях исследователи даже зашли так далеко, что говорят, что должно быть единое системное представление, означающее, что конечные пользователи не должны даже замечать, что они имеют дело с процессами, данными и контролем, рассредоточенными по всей компьютерной сети. Достижение единства системы часто требует слишком многого, и по этой причине мы выбрали более слабое определение распределенной системы как *кажущейся* пользователям связанной. Грубо говоря, распределенная система является связанной, если она ведет себя в соответствии с ожиданиями ее пользователей. Более конкретно, в единой связанной системе совокупность узлов в целом работает одинаково, независимо от того, где, когда и как происходит взаимодействие между пользователем и системой.

Часто достаточно сложно представлять распределенную систему как единую связанную систему. Например, это требует, чтобы конечный пользователь не мог точно сказать, на каком компьютере в настоящее время выполняется процесс, или, возможно, эта часть задачи была вызвана другим процессом, выполняющимся где-то еще. Место, где хранятся данные, также не должно быть проблемой; не должно иметь значения и то, что система может реплицировать данные для повышения производительности. Эта так называемая **прозрачность распределения** (distributing transparancy), о которой мы поговорим подробнее в разделе 1.2, является важной целью проектирования распределенных систем. В некотором смысле это сродни подходу, принятому во многих Unix-подобных операционных системах, в которых доступ к ресурсам осуществляется через единый интерфейс файловой системы, эффективно скрывающий различия между файлами, устройствами хранения и основной памятью.

Однако стремление к единой согласованной связанной системе является важным компромиссом. Поскольку мы не можем игнорировать тот факт, что распределенная система состоит из нескольких сетевых узлов, неизбежно, что в любое время какая-то часть системы может отказать. Это означает, что неожиданное поведение системы, при котором, например, некоторые приложения могут продолжать успешно выполняться, в то время как другие

полностью останавливаются, – это реальность, с которой приходится иметь дело. Хотя частичные отказы присущи любой сложной системе, в распределенных системах с ними особенно трудно справляться. Это позволило обладателю премии Тьюринга Лесли Лэмпорт (Leslie Lamport) описать распределенную систему как «[...] такую, в которой сбой какого-то компьютера, о существовании которого вы даже не подозревали, может сделать ваш компьютер бесполезным».

Промежуточное программное обеспечение и распределенные системы

Чтобы помочь разработке распределенных приложений, распределенные системы часто организованы так, чтобы иметь отдельный слой программного обеспечения, который логически размещен поверх соответствующих операционных систем компьютеров, являющихся частью системы. Эта организация показана на рис. 1.1 и известна как **промежуточное программное обеспечение** (middleware) [Bernstein, 1996].

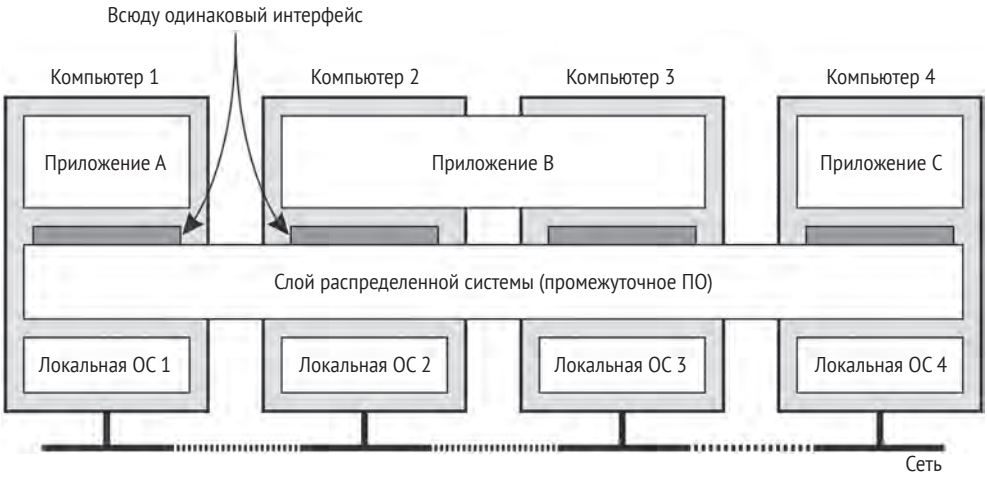

Рис. 1.1 ❖ Распределенная система, организованная на уровне промежуточного программного обеспечения, которая распространяется на несколько машин, предлагая каждому приложению один и тот же интерфейс

На рис. 1.1 показаны четыре сетевых компьютера и три приложения, причем приложение B распределено между компьютерами 2 и 3. Каждому приложению предлагается один и тот же интерфейс. Распределенная система предоставляет средства для связи компонентов одного распределенного приложения с каждым другим и позволяет различным приложениям общаться друг с другом. В то же время она скрывает, насколько это возможно и разумно, различия в оборудовании и операционных системах каждого приложения. В некотором смысле промежуточное ПО распределенной системы

подобно операционной системе компьютера – это менеджер ресурсов, предлагающий свои приложения для эффективного обмена и размещения этих ресурсов в сети. Помимо управления ресурсами, оно предлагает услуги, которые также можно найти в большинстве операционных систем, в том числе:

- средства для взаимодействия приложений;
- услуги безопасности;
- бухгалтерские услуги;
- маскировка и восстановление после сбоев.

Основное отличие от эквивалентов операционной системы заключается в том, что услуги промежуточного программного обеспечения предлагаются в сетевом окружении. Обратите внимание, что большинство услуг полезно для многих приложений. В этом смысле промежуточное программное обеспечение может также рассматриваться как контейнер часто используемых компонентов и функций, позволяющий реализовывать приложения отдельно. Чтобы проиллюстрировать это, давайте кратко рассмотрим несколько примеров типичных услуг промежуточного программного обеспечения.

- **Связь.** Общепринятой услугой связи является так называемый **удаленный вызов процедуры** (Remote Procedure Call, RPC). Сервис RPC, к которому мы вернемся в главе 4, позволяет приложению вызывать функцию, которая применяется и выполняется на удаленном компьютере, как если бы она была доступна локально. В конце разработчик должен просто указать заголовок функции, выраженный на специальном языке программирования, который позволяет подсистеме RPC генерировать затем необходимый код, который и устанавливает удаленные вызовы.
- **Транзакции.** Многие приложения используют несколько распределенных сервисов среди нескольких компьютеров. Промежуточное обеспечение обычно предлагает специальную поддержку для выполнения таких услуг, как «все или ничего», и обычно упоминается как атомарная **транзакция** (atomic transaction). В этом случае разработчик приложения должен только указать задействованные удаленные сервисы, и, следуя стандартизированному протоколу, промежуточное программное обеспечение гарантирует, что каждая служба вызывается или не вызывается вообще.
- **Состав услуг.** Становится все более распространенной разработка новых приложений путем выбора из существующих программ и склеивания их. Это особенно касается многих веб-приложений, в частности тех, которые известны как **веб-сервисы** [Alonso et al., 2004]. Промежуточное программное обеспечение на основе интернета может помочь стандартизировать способ доступа к веб-сервисам и обеспечить средства для генерации своих функций в определенном порядке. Простой пример того, как размещается состав службы, формируется с помощью гибридных веб-приложений **машапов** (mashup): веб-страницы, которые объединяют и группируют данные из разных источников. Известными машапами являются машапы на картах Google, которые дополнены такой информацией, как планировщики поездок или прогноз погоды в режиме реального времени.

- **Надежность.** В качестве последнего примера: было проведено множество исследований по расширенным функциям для создания надежных распределенных приложений. Инструментарий Horus [van Renesse et al., 1994] позволяет разработчику создавать приложение как группу процессов, такую что любое сообщение, отправленное одним процессом, гарантированно будет получено всеми или никакими другими процессами. Как оказалось, такие гарантии могут значительно упростить разработку распределенных приложений и, как правило, реализуются как часть промежуточного программного обеспечения.

Примечание 1.1 (историческая справка: термин «промежуточное программное обеспечение»)

Хотя термин «промежуточное программное обеспечение» стал популярным в середине 1990-х годов, он впервые, скорее всего, упоминается в сборнике докладов о разработке программного обеспечения конференции НАТО под редакцией Питера Наура (Peter Naur) и Брайана Рэнделла (Brian Randell) в октябре 1968 года [Peter Naur and Brian Randell, 1968]. В этих докладах промежуточное ПО действительно было размещено как раз между приложениями и подпрограммами обслуживания (эквивалент операционных систем).

1.2. Цели дизайна

Только то, что существует возможность создания распределенных систем, вовсе не обязательно означает, что это хорошая идея. В этом разделе мы обсудим четыре важные цели, которые должны иметь место, чтобы сделать построение распределенной системы стоящим усилий. Распределенная система должна сделать ресурсы легкодоступными; она должна скрывать тот факт, что ресурсы распределены по сети; она должна быть открытой; и она должна быть масштабируемой.

Поддержка совместного использования ресурсов

Важной целью распределенной системы является облегчение пользователям (и приложениям) доступа и совместного использования удаленных ресурсов. Ресурсы могут быть виртуально чем угодно, но типичные примеры включают в себя периферийные устройства, хранилища, данные, файлы, сервисы и сети, и это лишь некоторые из них. Есть много причин для желания поделиться ресурсами. Одна очевидная причина – это экономика. Например, дешевле иметь одно высококлассное надежное общее хранилище, чем покупать и поддерживать хранилище для каждого пользователя в отдельности.

Подключение пользователей и ресурсов также облегчает сотрудничество и обмен информацией, о чем свидетельствует успех интернета с его простыми протоколами для обмена файлами, почтой, документами, аудио и видео.

Подключение к интернету позволило географически широко рассредоточенным группам людей работать совместно с помощью всех видов **групповых программ** (groupware), то есть программного обеспечения для совместного редактирования, телеконференций и т. д., как показали многонациональные компании-разработчики программного обеспечения, которые передали многие работы по разработке кодов в Азию.

Однако совместное использование ресурсов в распределенных системах, возможно, лучше всего иллюстрируется успешным обменом файлами между одноранговыми сетями, такими как BitTorrent. Эти распределенные системы упрощают обмен файлами между пользователями интернета. Одноранговые сети часто связаны с распределением медиафайлов, таких как аудио и видео. В других случаях технология используется для распространения больших объемов данных, как в случае обновлений программного обеспечения, резервного копирования услуг и синхронизации данных на нескольких серверах.

Примечание 1.2 (дополнительная информация: общий доступ к папкам по всему миру)

Чтобы проиллюстрировать, где мы находимся в настоящее время, когда дело доходит до полной интеграции совместного использования ресурсов в сетевой среде, достаточно сказать, что сейчас развернуты веб-службы, которые позволяют группе пользователей размещать файлы в специальной общей папке, которая загружается и поддерживается третьей стороной где-то в интернете. Используя специальное программное обеспечение, общая папка почти не отличается от других папок на компьютере пользователя. По сути, эти службы заменяют использование общего каталога в локальной распределенной файловой системе, делая данные доступными для пользователей независимо от организации, которой они принадлежат, и вне зависимости от того, где они находятся. Эта услуга предлагается для разных операционных систем. Где именно хранятся данные, полностью скрыто от конечного пользователя.

Создание прозрачных распределений

Важной целью распределенной системы является скрытие того факта, что ее процессы и ресурсы физически распределены по нескольким компьютерам и, возможно, разделены большими расстояниями. Другими словами, она пытается сделать распределение процессов и ресурсов прозрачным, то есть невидимым для конечных пользователей и приложений.

Типы прозрачности распределений

Концепция прозрачности, то есть незаметности внутренней структуры для пользователя, может быть применена к нескольким аспектам распределенной системы, из которых наиболее важные перечислены на рис. 1.2. Мы используем термин «*объект*», имея в виду или процесс, или ресурс.

Прозрачность	Описание
Доступ	Скрыть различия в представлении данных и то, как получен доступ к объекту
Местоположение	Скрыть, где находится объект
Перемещение	Скрыть, что объект может быть перемещен в другое место, когда используется
Миграция	Скрыть, что объект может переместиться в другое место
Репликация	Скрыть, что объект тиражируется или дублируется
Параллельность	Скрыть, что объект может быть разделен между несколькими независимыми пользователями
Отказ	Скрыть сбой и восстановление объекта

Рис. 1.2 ❖ Различные формы прозрачности в распределенной системе (см. ISO[1995]). Объект может быть ресурсом или процессом

Прозрачность доступа (access transparency) связана с сокрытием различий в представлении данных и способа, который позволяет сделать объекты доступными. На базовом уровне мы хотим скрыть различия в архитектуре машин, но, что более важно, мы достигаем соглашения о том, как данные должны быть представлены различными машинами и операционными системами. Например, распределенная система может иметь в составе компьютеры, которые работают в разных операционных системах и каждая из которых имеет свои соглашения об именах файлов. Различия в соглашениях об именах, различия в файловых операциях или различия в том, какова низкоуровневая связь с другими процессами, являются примерами проблем доступа, которые желательно скрывать от пользователей и приложений.

Важной группой типов прозрачности являются местоположения процесса или ресурса. **Прозрачность местоположения** (location transparency) означает, что пользователи не могут сказать, где физически объект находится в системе. Наименование играет важную роль в достижении прозрачности местоположения. В частности, прозрачность местоположения часто может быть достигнута путем назначения ресурсам только логических имен, то есть имена, в которых местоположение ресурса не является тайным, не закодированы. Примером такого имени является **унифицированный указатель ресурса** (uniform resource locator, URL) http://www.prenhall.com/index.html, который не дает никакой информации о фактическом расположении основного веб-сервера Prentice Hall. URL также не дает подсказки о том, был ли файл index.html всегда в его текущем местоположении или недавно был перемещен туда. Например, весь сайт, возможно, был перемещен из одного центра данных в другой, но пользователи не должны этого замечать. **Прозрачность перемещения** (relocation transparency) становится все более важной в контексте облачных вычислений, к чему мы вернемся позже в этой главе.

Если прозрачность перемещения относится к перемещению самой распределенной системы, то **прозрачность миграции** (migration transparency) обеспечивается распределенной системой, когда она поддерживает мобильность процессов и ресурсов, инициируемых пользователями, без влияния на текущую связь и операции. Типичным примером этого является связь между

мобильными телефонами: независимо от того, как на самом деле движутся два человека, их мобильные телефоны позволят им продолжать разговор. Другие примеры, которые приходят на ум, включают онлайн-отслеживание местоположения и отслеживание товаров при их транспортировке из одного места в другое, а также телеконференции (частично) с использованием устройств, которые оснащены мобильным интернетом.

Как мы увидим, репликация (дублирование, тиражирование, replication) играет важную роль в распределенных системах. Например, ресурсы могут быть дублированы для повышения доступности или улучшения производительности путем размещения копии рядом с местом, где она доступна. **Прозрачность репликации** (replication transparency) связана с сокрытием того факта, что существует несколько копий ресурса или что несколько процессов работают в том или ином режиме перехвата, чтобы один мог работать, когда другой терпит неудачу. Чтобы скрыть репликацию от пользователей, необходимо, чтобы все реплики имели одинаковое имя. Следовательно, система, которая поддерживает прозрачность, как правило, должна поддерживать прозрачность местоположения, потому что иначе было бы невозможно обратиться к репликам в разных местах.

Мы уже упоминали, что важной целью распределенных систем является разрешение совместного использования ресурсов. Во многих случаях совместное использование ресурсов осуществляется путем кооперации, как и в случае каналов связи. Тем не менее также есть много примеров конкурентного совместного использования ресурсов. Например, каждый из двух независимых пользователей может хранить свои файлы на одном файловом сервере или иметь доступ к тем же таблицам в общей базе данных. В таких случаях важно, чтобы каждый пользователь не заметил, что другой использует тот же ресурс. Это явление называется **прозрачностью параллелизма** (concurrency transparency). Важной проблемой является то, что одновременный доступ к общему ресурсу оставляет ресурс в согласованном состоянии. Последовательность использования может быть достигнута с помощью блокировки механизма, который предоставляет пользователям эксклюзивный доступ к ресурсу по очереди. Более совершенный механизм заключается в использовании транзакций, но их, возможно, будет сложно реализовать в распределенной системе, особенно когда масштабируемость является проблемой.

Наконец, не менее важно, чтобы распределенная система обеспечивала **прозрачность отказов** (failure transparency). Это означает, что пользователь или приложение не замечает, что какая-то часть системы должным образом не работает, а система впоследствии (и автоматически) восстанавливается после этого сбоя. Маскировка отказов – это одна из самых сложных проблем в распределенных системах и даже невозможна, когда будут сделаны некоторые, по-видимому, реалистичные предположения, которые мы обсудим в главе 8. Основная трудность в маскировке и прозрачном восстановлении заключается в неспособности отличить полностью не работающий процесс и тот, который медленно, с трудом, но работает. Например, при контакте с перегруженным интернет-сервером браузер в конечном итоге берет таймаут и сообщает, что веб-страница недоступна. В этот момент пользователь

не может сказать, является сервер перегруженным на самом деле или сильно перегружена сеть.

Степень прозрачности распределения

Хотя прозрачность распределения обычно считается предпочтительной для любой распределенной системы, существуют ситуации, в которых пытаются слепо скрыть все аспекты распространения от пользователей, и это не очень хорошая идея. Простой пример – попросить вашу электронную газету появиться в вашем почтовом ящике до 7 утра по местному времени, в то время когда вы находитесь на другом конце света и в другом часовом поясе. Ваша утренняя газета не будет той утренней газетой, к которой вы привыкли.

Аналогично от глобальной распределенной системы, которая связывает процесс в Сан-Франциско, нельзя ожидать, что удастся скрыть факт связи с процессом в Амстердаме, что мать-природа не позволит ей отправить сообщение от одного процесса другому менее чем за 35 миллисекунд. Практика показывает, что это при использовании компьютерной сети передача на самом деле занимает несколько сотен миллисекунд. Передача сигнала ограничена не только скоростью света, но и ограниченной вычислительной мощностью и задержками в промежуточных переключениях. Существует также компромисс между высокой степенью прозрачности и производительностью системы. Например, многие приложения в интернете неоднократно попробуют связаться с сервером, прежде чем окончательно сдаться. Следовательно, пытаясь замаскировать временный сбой сервера, можно замедлить систему в целом. В таком случае, возможно, было бы лучше отказаться раньше или, по крайней мере, позволить пользователю отменить попытки установить контакт.

Другой пример – когда мы должны гарантировать, что несколько копий, расположенных на разных континентах, должны быть все время согласованными. Иными словами, если одна копия изменена, это изменение должно быть распространено на все копии до разрешения на любую другую операцию. Понятно, что одна только операция обновления может занять до нескольких секунд, что не может быть скрыто от пользователей.

Наконец, есть ситуации, в которых совершенно не очевидно, что скрытое распределение – это хорошая идея. Когда распределенные системы распространяются на устройства, которые люди носят с собой и где само понятие места в контексте осведомленности становится все более важным, может быть, лучше *раскрыть* распределение, а не пытаться скрыть его. Очевидный пример использования услуг на основе определения местоположения, которые часто можно найти на мобильных телефонах, – как найти ближайший китайский ресторан или проверить, далеко ли друзья.

Есть и другие аргументы против прозрачности распространения. Признаем также, что полная прозрачность распределения просто невозможна, и мы должны спросить самих себя, разумно ли делать вид, что мы можем этого достичь. Может быть, гораздо лучше сделать распределение явным, чтобы

пользователь и разработчик приложения никогда не обманывались, полагая, что есть такая вещь, как прозрачность. В результате пользователи будут намного лучше понимать (иногда неожиданно) поведение распределенной системы и поэтому лучше подготовятся к такому ее поведению.

Примечание 1.3 (обсуждение: против прозрачности распределения)

Некоторые исследователи утверждают, что скрытое распространение приведет только к дальнейшему усложнению разработки распределенных систем, именно по той причине, что полная прозрачность распределения никогда не может быть достигнута. Популярная техника для достижения прозрачности доступа заключается в расширении вызовов процедур для удаленных серверов. Однако Уолдо и др. [Waldo et al., 1997] уже показали, что попытка скрыть распространение с помощью процедур удаленных вызовов может привести к плохо понимаемой семантике, по той простой причине, что вызов процедуры *изменяется* при неисправности канала связи. В качестве альтернативы различные исследователи и практики в настоящее время выступают за меньшую прозрачность, например благодаря более явному использованию коммуникации в стиле сообщений или более явному размещению запросов и получению результатов от удаленной машины, как это делается в интернете при загрузке страниц. Такие решения будут подробно обсуждаться в следующей главе.

Несколько радикальной точки зрения придерживается Уэмс [Wams, 2011], утверждая, что частичные сбои не позволяют полагаться на успешное выполнение удаленного сервиса. Если такая надежность не может быть гарантирована, то всегда лучше положиться только на локальное исполнение, ведущее к принципу **копирования перед использованием** (copy-before-use). Согласно этому принципу, доступ к данным возможен только после их передачи на компьютер процесса, желающего получить эти данные. Кроме того, не должен быть измененен элемент данных. Вместо этого он может быть обновлен только до новой версии. Несложно представить, что появятся и другие проблемы. Тем не менее Уэмс показывает, что многие существующие приложения могут быть адаптированы к этому альтернативному подходу без ущерба для функциональности.

Вывод заключается в том, что стремление к прозрачности распределения может быть хорошей целью при разработке и реализации распределенных систем, но ее следует рассматривать совместно с другими вопросами, такими как производительность и понятность. Цена за достижение полной прозрачности может быть на удивление высокой.

Открытость

Другой важной целью распределенных систем является открытость. Открытая распределенная система, по сути, представляет собой систему, предлагающую компоненты, которые могут легко использоваться или интегрироваться в другие системы. В то же время сама открытая распределенная система часто состоит из компонентов, которые созданы в другом месте.

Функциональная совместимость, компоновка и расширяемость

Быть открытым означает, что компоненты должны придерживаться стандартных правил, которые описывают синтаксис и семантику того, что могут предложить эти компоненты (т. е. какую услугу они предоставляют). Общий подход заключается в определении услуг через интерфейсы, использующие **язык определения интерфейса** (Interface Definition Language, IDL). Определения интерфейса, записанные в IDL, почти всегда фиксируют только синтаксис сервисов. Другими словами, они точно определяют имена функций, которые доступны вместе с типами параметров, возвращаемыми значениями, возможными исключениями, которые можно применять, и т. д. Эта жесткая часть точно определяет, что эти сервисы делают, то есть семантику интерфейсов. На практике такие спецификации предоставляются в неформальной форме с помощью естественного языка.

При правильном указании определения интерфейса допускается произвольный процесс, который нуждается в определенном интерфейсе, чтобы обменяться с другим процессом, который обеспечивает этот интерфейс. Это также позволяет двум независимым сторонам строить совершенно разные реализации этих интерфейсов, что приводит к двум отдельным компонентам, которые действуют совершенно одинаково.

Правильные характеристики всегда полны и нейтральны. Полнота означает, что все, что необходимо для реализации, действительно указано. Однако многие определения интерфейсов не являются полными потому, что разработчику необходимо добавить детали реализации. Не менее важным является и то, что спецификации не предписывают, как реализация должна выглядеть; они должны быть нейтральными.

Как показывают Блэр и Стефани [Blair and Stefani, 1998], полнота и нейтральность важны для совместимости и мобильности. **Функциональная совместимость** (interoperability) характеризует степень, в которой две реализации систем или компонентов разных производителей могут сосуществовать и работать совместно, просто полагаясь на взаимные услуги в соответствии с общим стандартом. **Портативность** (portability) характеризует то, в какой степени приложение, разработанное для распределенной системы A, может работать без изменений в другой распределенной системе, в которой реализуются те же интерфейсы, что и в A.

Другой важной целью для открытой распределенной системы является то, что она должна легко конфигурировать систему из разных компонентов (возможно, и разных разработчиков). Кроме того, должна быть обеспечена простота добавления новых компонентов или замены существующих, не затрагивая остающихся компонентов. Иными словами, открытая распределенная система также должна быть **расширяемой** (extensible). Например, в расширяемую систему, видимо, относительно несложно добавить компоненты из другой операционной системы или даже заменить всю файловую систему.

Примечание 1.4 (обсуждение: открытые системы на практике)

Конечно, то, что мы только что описали, является идеальной ситуацией. Практика показывает, что многие распределенные системы не так открыты, как хотелось бы, и необходимо приложить еще много усилий и собрать разные кусочки, чтобы создать распределенную систему. Одним из решений вопроса открытости является раскрытие всех деталей компонента и предоставление разработчикам действительного исходного кода. Этот подход становится все более популярным и приводит к проектам с так называемым открытым исходным кодом, в которые многие вносят свой вклад по улучшению и отладке системы. По общему признанию, это наибольшая открытость, которую система может получить, но является ли это лучшим способом получить открытость, все еще под вопросом.

Отделение политики от механизма

Для достижения гибкости в открытых распределенных системах крайне важно, чтобы система была организована как набор относительно небольших и легко заменяемых или адаптируемых компонентов. Это подразумевает, что мы должны обеспечить определения не только интерфейсов самого высокого уровня, то есть тех, которые видят пользователи и приложения, но и определения интерфейсов для внутренних компонентов системы и описания, как эти компоненты взаимодействуют. Этот подход является относительно новым. Многие более старые и даже современные системы построены с использованием монолитного подхода, в котором компоненты только логически разделены, но реализованы как одна огромная программа. Такой подход затрудняет замену или адаптацию компонента, не оказывая влияния на всю систему. Таким образом, монолитные системы имеют тенденцию быть замкнутыми, а не открытыми.

Необходимость изменений в распределенной системе часто вызывается компонентом, который не обеспечивает оптимальную политику для конкретного пользователя или приложения. В качестве примера рассмотрим кеширование в веб-браузерах. Есть много разных параметров, которые необходимо учитывать.

- **Хранение:** где находятся данные для кеширования? Как правило, наряду с хранилищем на диске будет встроенная память – кеш. В этом случае должно быть рассмотрено ее точное положение в локальной файловой системе.
- **Исключение:** когда кеш заполняется, какие данные нужно удалить, чтобы вновь загруженные страницы могли быть сохранены?
- **Совместное использование:** каждый браузер использует отдельный кеш, или кеш делится между браузерами разных пользователей?
- **Обновление:** когда браузер проверяет актуальность кешированных данных? Кеши наиболее эффективны, когда браузер может возвращать страницы при отсутствии необходимости связи с оригинальным веб-сайтом. Это, однако, несет риск возвращения устаревших данных. Обратите внимание, что частота обновления сильно зависит от того, относительно каких данных фактически осуществляется кеширование:

в то время как расписание поездов вряд ли изменится, это совсем не так с веб-страницами, показывающими, скажем, погодные условия на трассе или, что еще хуже, цены на акции.

Необходимо разделение политики и механизма. В случае веб-кеширования, например, браузер в идеале должен обеспечить возможности только для хранения документов и в то же время позволять пользователям решать, какие документы хранятся и как долго. На практике это может быть реализовано обеспечением богатого набора параметров, которые пользователь может установить (динамически). Если сделать еще один шаг вперед, браузер даже может предложить возможности для подключения политики, которую пользователь реализовал как отдельный компонент.

Примечание 1.5 (обсуждение: действительно ли нам нужно строгое разделение?)

Теоретически следует строго отделять политику от механизма. Тем не менее есть важный компромисс, который необходимо учитывать: чем строже разделение, тем в большей степени нам нужно убедиться, что мы предлагаем соответствующий набор механизмов. На практике это означает, что предлагается богатый набор функций, что, в свою очередь, приводит ко многим параметрам конфигурации. Например, популярный браузер Firefox предлагает несколько сотен параметров конфигурации. Просто представьте, как пространство конфигурации буквально взрывается при рассмотрении больших распределенных систем, состоящих из многих компонентов. Другими словами, строгое разделение политики и механизмов может привести к очень сложным проблемам конфигурации.

Одним из вариантов решения этих проблем является предоставление разумных значений по умолчанию, и это то, что часто происходит на практике. Альтернативный подход заключается в том, что система наблюдает за своим использованием и динамически изменяет настройки параметров. Это приводит к системам, известным как самонастраиваемые системы. Тем не менее уже сам факт поддержки широкого ряда политик часто усложняет кодирование распределенных систем. Жесткое кодирование политик в распределенной системе может значительно снизить сложность, но ценой меньшей гибкости.

Поиск правильного баланса в отделении политики от механизмов является одной из причин, по которым проектирование распределенной системы часто является скорее искусством, чем наукой.

Масштабирование

Для многих из нас подключение через интернет столь же обычно, как возможность отправить открытку кому угодно в любую точку мира. Более того, там, где до недавнего времени для офисных приложений и хранения информации мы привыкли использовать настольные компьютеры, сейчас мы являемся свидетелями того, что подобные приложения и услуги размещаются в так называемом «облаке», а это, в свою очередь, приводит к увеличению количества гораздо меньших сетевых устройств, таких как планшетные компьютеры. Ввиду этого масштабируемость стала для разработчиков распределенных систем одной из наиболее важных целей проектирования.

Размерность масштабируемости

Масштабируемость системы может определяться как минимум по трем различным измерениям (см. [Neuman, 1994]):

- **масштабируемость размеров.** Система может быть масштабируема относительно ее размера, что означает, что мы можем легко добавить больше пользователей и ресурсов в систему без заметной потери ее производительности;
- **географическая масштабируемость.** Географически масштабируемая система – это система, в которой пользователи и ресурсы могут находиться далеко друг от друга, но тот факт, что задержки в линии связи могут быть значительными, не должен быть замечен системой;
- **административная масштабируемость.** Административно масштабируемая система – это система, которой все еще можно легко управлять, даже если она охватывает много независимых административных организаций.

Давайте подробнее рассмотрим каждое из этих трех измерений масштабируемости.

Масштабируемость размеров. Когда система нуждается в масштабировании, приходится решать очень разные типы проблем. Давайте сначала рассмотрим масштабирование относительно размера. Если необходимо поддерживать больше пользователей или ресурсов, мы часто сталкиваемся с ограничениями централизованных услуг, хотя и по очень разным причинам. Например, многие услуги централизованы в том смысле, что они реализуются с помощью одного сервера, работающего в распределенной системе на конкретной машине. В более современной обстановке мы можем иметь группу сотрудничающих серверов, расположенных в кластере тесно связанных машин, физически размещенных в одном и том же месте. Проблема с этой схемой очевидна: сервер или группа серверов может просто стать узким местом, когда необходимо обрабатывать все большее количество запросов. Чтобы проиллюстрировать, как это может случиться, предположим, что служба реализована на одной машине. В этом случае, по сути, существует три основные причины стать узким местом:

- вычислительная мощность, ограниченная процессорами;
- емкость памяти, включая скорость передачи ввода/вывода;
- сеть между пользователем и централизованным сервисом.

Давайте сначала рассмотрим вычислительную мощность. Просто представьте сервис для вычисления оптимальных маршрутов с учетом информации о движении в реальном времени. Такой сервис связан с вычислениями, требующими нескольких (десятков) секунд для выполнения запроса. Если есть и доступна только одна машина, то даже современная система высокого класса в конечном итоге столкнется с проблемами, если количество запросов возрастет выше определенного уровня.

Точно так же, но по разным причинам, мы столкнемся с проблемами, когда сервис в основном связан с вводом/выводом. Типичный пример – плохо спроектированный централизованный поисковик. Проблема с поисковыми запросами на основе контента – в том, что нам необходимо сопоставить за-

прос со всем набором данных. Даже с передовыми методами индексации мы все еще можем столкнуться с проблемой обработки огромного количества данных, превышающего емкость основной памяти машины, обслуживающей сервис. Как следствие большая часть времени обработки будет определяться относительно медленным доступом к диску и передачей данных между диском и основной памятью. Увеличение количества дисков или их скорости не даст устойчивого решения, если количество запросов продолжает расти.

Наконец, сеть между пользователем и службой сервиса также может быть причиной плохой масштабируемости. Просто представьте себе услугу «видео по запросу», которая должна выдавать потоковое видео высокого качества для нескольких пользователей. Видеопоток может легко потребовать полосу пропускания от 8 до 10 Мбит/с, что означает, что если служба устанавливает соединения со своими клиентами по принципу точка-точка, то это может скоро превзойти пределы пропускной способности сети ее собственных линий передачи.

Есть несколько решений проблемы размера масштабирования, которые мы обсудим далее, после рассмотрения географической и административной масштабируемостей.

Примечание 1.6 (дополнительно: анализ возможностей обслуживания)

Проблемы масштабируемости размера для централизованных сервисов могут быть формально проанализированы с использованием теории очередей и нескольких упрощающих предположений. На концептуальном уровне централизованный сервис может быть смоделирован как простая система очередей, показанная на рис. 1.3: запросы отправляются в службу, где они находятся в очереди до дальнейшего уведомления. Как только процесс может обработать следующий запрос, он извлекает его из очереди, выполняет обработку и выдает ответ. В объяснении работы централизованной службы мы в значительной степени следуем за [Menasce и Almeida, 2002].

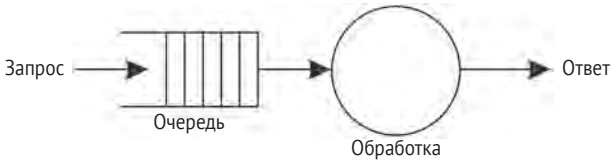

Рис. 1.3 ❖ Простая модель сервиса системы массового обслуживания

Во многих случаях мы можем предположить, что очередь имеет бесконечную емкость, что означает, что нет ограничений на количество запросов, которые могут быть приняты для дальнейшей обработки. Строго говоря, это означает, что скорость поступления запросов не зависит от того, что в данный момент находится в очереди или обрабатывается. Если предположить, что скорость поступления запросов равна λ запросов в секунду и что скорость обработки услуги составляет μ запросов в секунду, можно подсчитать временно́е отношение p_k, когда в системе находится k запросов:

$$p_k = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k.$$

Если мы определим использование U службы как время, которое она занята, то очевидно, что

$$U = \sum_{k>0} p_k = 1 - p_0 = \frac{\lambda}{\mu} \Rightarrow p_k = (1-U)U^k.$$

Затем мы можем вычислить $\bar{N}$ – среднее количество запросов в системе:

$$\bar{N} = \sum_{k\geq 0} k \cdot p_k = \sum_{k\geq 0} k \cdot (1-U)U^k = (1-U)\sum_{k\geq 0} k \cdot U^k = \frac{(1-U)U}{(1-U)^2} = \frac{U}{1-U}.$$

Что нас действительно интересует, так это время отклика R, то есть сколько времени это займет до того, как сервис обработает запрос, включая время, проведенное в очереди. Для этого нам нужна средняя пропускная способность X. Учитывая, что сервис «занят», когда обрабатывается хотя бы один запрос, и что это происходит с пропускной способностью μ запросов в секунду и в течение доли U от общего времени, получаем:

$$X = \underbrace{U \cdot \mu}_{\text{сервер работает}} + \underbrace{(1-U)\cdot 0}_{\text{сервер не работает}} = \frac{\lambda}{\mu} \cdot \mu = \lambda.$$

Используя формулу Литтла [Trivedi, 2002], мы можем вывести время отклика как

$$R = \frac{\bar{N}}{X} = \frac{S}{1-U} \Rightarrow \frac{R}{S} = \frac{1}{1-U},$$

где $S = \frac{1}{\mu}$ – фактическое время обслуживания. Обратите внимание, что если U очень мало, то отношение времени ответ–обслуживание близко к 1, что означает, что запрос практически мгновенно обработан, и на максимально возможной скорости. Однако как только использование становится ближе к 1, мы видим, что отношение времени ответ–обслуживание быстро увеличивается до очень высоких значений, что фактически означает, что система приближается к остановке. Здесь мы видим проблемы масштабируемости. Из этой простой модели заключаем, что единственным решением является сокращение времени обслуживания S. Мы предоставляем читателю в качестве упражнения определить, как можно уменьшить S.

Географическая масштабируемость. Географическая масштабируемость имеет свои проблемы. Одной из главных причин, почему все еще трудно масштабировать существующие распределенные системы, которые были разработаны для локальных сетей, является то, что многие из них основаны на **синхронной связи** (synchronous communication). В этой форме общения сторона запрашиваемого сервиса, обычно называемая **клиентом**, блокирует до тех пор, пока ответ не отправлен обратно с сервера, реализующего сервис. Более конкретно, мы часто видим подобную картину связи многих клиент-серверных взаимодействий, как это случается с транзакциями базы данных. Этот подход обычно работает нормально в локальных сетях, где связь между двумя машинами часто бывает в худшем случае несколько сотен микросекунд. Однако в глобальной системе необходимо принять во внимание, что межпроцессорное взаимодействие может составлять сотни миллисекунд, то

есть на три порядка медленнее. Создание приложений с использованием синхронной связи в глобальных системах требует большой осторожности (а не просто некоторого терпения), особенно с расширенной моделью взаимодействия между клиентом и сервером.

Другой проблемой, препятствующей географической масштабируемости, является то, что связь в глобальных сетях по своей природе гораздо менее надежна, чем связь в локальных сетях. Кроме того, необходимо также иметь дело с ограниченной пропускной способностью. В результате решения, разработанные для локальных сетей, не всегда легко переносятся на глобальную систему. Типичным примером является потоковое видео. В домашней сети это довольно просто, даже когда есть только беспроводные соединения, обеспечивающие стабильный, быстрый поток качественных видеокадров с медиасервера на дисплей. Но поместите тот же сервер подальше и используйте стандартное TCP-соединение с дисплеем, и связь обязательно ухудшится: немедленно проявятся ограничения пропускной способности, будет затруднено и поддержание того же уровня надежности.

Еще одна проблема, которая возникает, когда компоненты находятся далеко друг от друга, – это тот факт, что глобальные системы, как правило, имеют очень ограниченные возможности для многоточечной связи. Напротив, локальные сети часто поддерживают эффективные механизмы вещания. Такие механизмы оказались очень полезными для обнаружения компонентов и услуг, что очень важно с точки зрения управления. В глобальных системах необходимо разрабатывать отдельные службы, такие как службы имен и каталогов, к которым могут обращаться направляемые запросы. Эти службы поддержки, в свою очередь, также должны быть масштабированы, и во многих случаях для этого не существует очевидных решений, о чем мы поговорим в следующих главах.

Административная масштабируемость. Наконец, сложный и во многих случаях открытый вопрос – как распределить распределенную систему по нескольким независимым административным доменам. Основная проблема, которая должна быть решена, – это конфликт политики в отношении использования (и оплаты) ресурсов, управления и безопасности.

Иллюстрацией этого может служить то, что ученые в течение многих лет искали решения для получения возможности поделиться своим (часто дорогим) оборудованием и создать так называемую **вычислительную сетку** (computational grid). В этих сетках глобальная распределенная система строится как федерация локальных распределенных систем, что позволяет программе работать на компьютере в организации A с прямым доступом к ресурсам в организации B.

Например, многие компоненты распределенной системы, которые находятся в каком-то одном домене, часто могут пользоваться доверием компонентов, работающих в этом же домене. В таких случаях системное администрирование должно протестировать и сертифицировать приложения и, возможно, принять специальные меры для предотвращения появления поддельных компонентов. По сути, пользователи доверяют свои системы администратору. Однако это доверие не распространяется естественно за пределы домена.

Примечание 1.7 (пример: современный радиотелескоп)

В качестве примера рассмотрим разработку современного радиотелескопа, такого как обсерватория Pierre Auger [Abraham et al., 2004]. Полная система может рассматриваться как федеративная распределенная система:

- непосредственно радиотелескоп можно определить как беспроводную распределенную систему, разработанную в виде сетки из нескольких тысяч сенсорных узлов, каждый из которых собирает радиосигналы и взаимодействует с соседними узлами для фильтрации соответствующих событий. Узлы динамически поддерживают дерево приемников, по которому выбранные события отправляются к центральной точке для дальнейшего анализа;
- центральная точка должна быть достаточно мощной системой, способной хранить и обрабатывать события, отправленные ей сенсорными узлами. Эта система обязательно размещается в непосредственной близости от узлов датчика, но, с другой стороны, считается действующей независимо. В зависимости от ее функциональности она может работать как небольшая локальная распределенная система. В частности, она хранит все записанные события и предлагает доступ к удаленным системам, принадлежащим партнерам консорциума;
- большинство партнеров имеют локальные распределенные системы (часто в форме кластера компьютеров), которые они используют для дальнейшей обработки данных, собранных телескопом. В этом случае локальные системы имеют прямой доступ к центральной точке телескопа с использованием стандартного протокола связи. Естественно, многие результаты, полученные в рамках консорциума, предоставляются каждому партнеру.

Таким образом, видно, что вся система пересекает границы нескольких административных доменов и что необходимы специальные меры для обеспечения того, чтобы доступ к данным получали только (конкретные) партнеры по консорциуму и не разглашались неуполномоченным лицам. Как в этих условиях добиться административной масштабируемости, не очевидно.

Если распределенная система расширяется до другого домена, должны быть приняты два типа мер безопасности. Во-первых, распределенная система должна защищаться от вредоносных атак нового домена. Например, пользователи из нового домена могут иметь доступ только для чтения в файловой системе своего конкретного домена. Аналогично такие объекты, как дорогие сеттеры изображений или высокопроизводительные компьютеры, могут быть недоступны для посторонних пользователей. Во-вторых, новый домен должен защищать себя от злонамеренных атак распределенной системы. Типичный пример – загрузка таких программ, как апплеты в веб-браузерах. В принципе, новый домен не знает, чего ожидать от такого чужого кода. Проблема, как мы увидим в главе 9, в том, каким образом осуществлять эти ограничения.

В качестве контрпримеров распределенных систем, охватывающих несколько административных доменов и которые явно не затрагивают проблемы административной масштабируемости, рассмотрим современные файлообменные одноранговые сети. В этих случаях конечные пользователи просто устанавливают программу, реализующую распределенный поиск, загружают функции и в течение нескольких минут могут начать загрузку файлов. Другие примеры включают одноранговые приложения для телефонии

через интернет, такие как Skype [Baset and Schulzrinne, 2006] и потоковая передача таких аудиоприложений, как Spotify [Kreitz and Niemelä, 2010]. Что общее у таких распределенных систем, так это то, что для того, чтобы поддерживать систему в рабочем состоянии, сотрудничают конечные пользователи, а не административные организации. В лучшем случае административные организации, такие как **интернет-провайдеры** (Internet Service Providers, ISPs), могут контролировать сетевой трафик, вызываемый этими одноранговыми системами, но пока подобные усилия не были очень эффективными.

Техника масштабирования

Обсудив некоторые проблемы масштабируемости, мы подходим к вопросу о том, как эти проблемы могут быть решены в целом. В большинстве случаев проблемы масштабируемости в распределенных системах появляются в виде проблем с производительностью, вызванных ограничениями емкости серверов и сети. Просто улучшением своих возможностей (например, путем увеличения памяти, обновления процессоров или замены сетевых модулей) часто находится решение, которое называется **расширением масштаба** (scaling up). Когда дело касается масштабирования, такое расширение распределенной системы путем развертывания большего количества компьютеров мы можем, в принципе, осуществить только тремя методами: сокрытием коммуникационной задержки, распределением работы и репликацией (см. также [Neuman, 1994]).

Сокрытие коммуникационных задержек. Сокрытие коммуникационных задержек применимо в случае географической масштабируемости. Основная идея проста: попробовать в максимально возможной степени избегать ожидания ответов на запросы удаленного обслуживания. Например, когда служба была запрошена на удаленном компьютере, альтернативой ожидания ответа от сервера является выполнение другой полезной работы на стороне запрашивающего. По сути, это означает создание запрашивающего приложения таким образом, что оно использует только **асинхронную связь** (asynchronous communication). Когда приходит ответ, приложение прерывается, и вызывается специальный обработчик завершения ранее оформленного запроса. Асинхронная связь часто может использоваться в системах пакетной обработки и параллельных приложениях, в которых независимые задачи могут быть запланированы для выполнения, в то время как данная задача ожидает завершения связи. В качестве альтернативы может быть начат процесс выполнения нового запроса. Хотя это блокирует ожидание ответа, другие потоки в процессе могут продолжаться.

Однако есть много приложений, которые не могут эффективно использовать асинхронную связь. Например, в интерактивных приложениях, когда пользователь отправляет запрос, он, как правило, не имеет ничего лучшего, чем ожидание ответа.

В таких случаях гораздо лучшим решением является уменьшение общения, например путем перемещения части вычислений, которые обычно выполняются на сервере по клиентскому процессу, запрашивающему сервис. Типичный случай, когда этот подход работает, – это доступ к базам данных

с использованием форм. Заполнение форм может быть сделано путем отправки отдельного сообщения для каждого поля и ожидания подтверждения от сервера, как показано на рис. 1.4а. Например, сервер может проверить наличие синтаксических ошибок, прежде чем принять запись. Гораздо лучшее решение – отправить код для заполнения формы и, возможно, проверки записей клиенту и заставить клиента вернуть заполненную форму, как показано на рис. 1.4б. Такой подход кода доставки широко поддерживается в интернете с помощью Java-апплетов и Javascript.

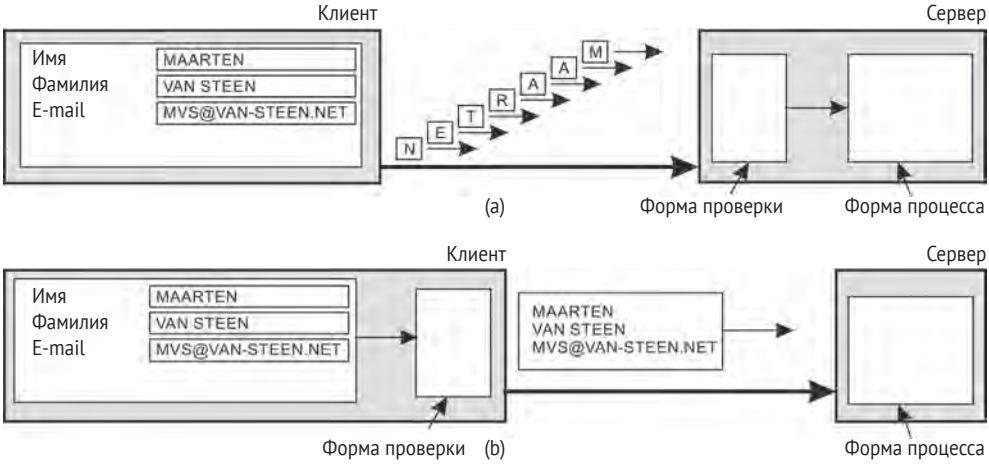

Рис. 1.4 ❖ Разница между разрешением (a) проверки сервера и (b) проверки клиента

Разбиение и распространение. Другой важный метод масштабирования – это **разбиение и распространение** (partitioning and distribution), которое включает в себя получение компонента, разделение его на более мелкие части, а затем распространение этих частей по всей системе. Хорошим примером разбиения и распространения является доменное имя в интернете (DNS). Пространство имен DNS иерархически организовано как дерево **доменов**, которые разделены на непересекающиеся **зоны**, как показано для оригинальной системы DNS на рис. 1.5. Имена в каждой зоне обрабатываются одним сервером имен. Не вдаваясь сейчас в подробности (подробно вернемся к DNS в главе 5), можно представить, что каждое имя пути является именем хоста в интернете и, следовательно, связано с сетевым адресом этого хоста. По сути, раскрытие имени означает возврат сетевого адреса связанному хосту. Рассмотрим, например, имя flits.cs.vu.nl. Чтобы раскрыть это имя, оно сначала передается на сервер зоны Z1 (см. рис. 1.5), который возвращает адрес сервера для зоны Z2, к которой относится остальное имя, flits.cs.vu, может быть переданным. Сервер для Z2 вернет адрес сервера для зоны Z3, который способен обрабатывать последнюю часть имени, и вернет адрес связанного хоста.

Этот пример иллюстрирует, как *сервис наименований* (naming service), обеспечиваемый DNS, распределен по нескольким машинам, избегая, та-

ким образом, того, чтобы один сервер имел дело со всеми запросами на разрешение имен.

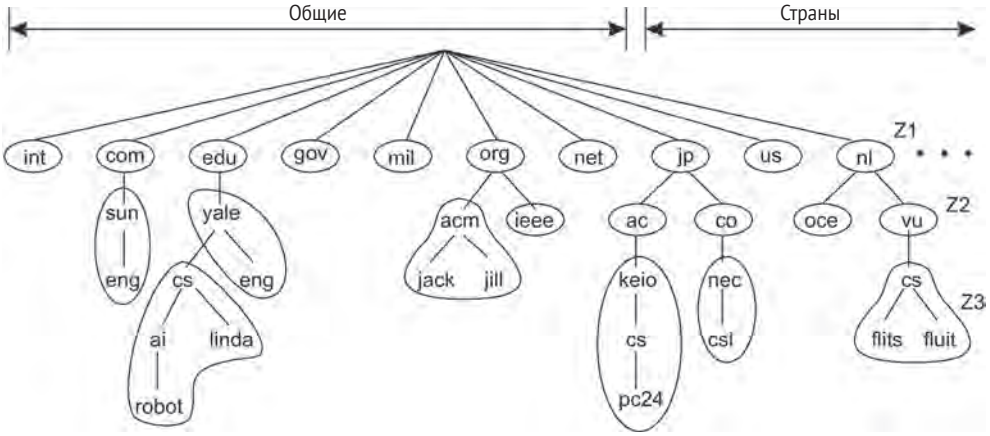

Рис. 1.5 ❖ Пример разделения (оригинального) пространства имен DNS на зоны

В качестве другого примера рассмотрим сеть интернета World Wide Web. Для большинства пользователей эта сеть представляется огромной информационной системой на основе документов, в которой каждый документ имеет свое уникальное имя в виде URL. Концептуально может даже показаться, что существует только один сервер. Тем не менее веб-сеть физически разделена и распределена по *нескольким сотням миллионов* серверов, каждый из которых обрабатывает несколько веб-документов. Имя сервера обработки документов закодировано в URL этого документа. Только благодаря такому распределению документов интернет смог быть масштабирован до существующего размера.

Репликация. Учитывая, что проблемы масштабируемости часто приводят к снижению производительности, как правило, хорошая идея – реплицировать (копировать, дублировать) компоненты в распределенной системе. Репликация не только увеличивает доступность, но и помогает сбалансировать нагрузку между компонентами, что позволяет повысить производительность. Кроме того, в широко географически рассредоточенных системах наличие поблизости копии может устранить большую часть проблем, связанных с задержкой связи, о чем упоминалось ранее.

Кеширование – это особая форма репликации, хотя найти различие между ними часто трудно или даже неестественно. Как и в случае репликации, кеширование (caching) приводит к созданию копии ресурса, как правило, в непосредственной близости от клиента, который обращается к этому ресурсу. Однако, в отличие от репликации, при кешировании решение принимается клиентом ресурса, а не его владельцем.

У кеширования и репликации существует один серьезный недостаток, который может влиять на масштабируемость. Поскольку существует несколько копий ресурса, изменение одной копии делает эту копию отличной от

других. Следовательно, кеширование и репликация приводят к проблемам согласованности. В какой степени такое несоответствие приемлемо, в значительной степени зависит от использования ресурса. Например, многие веб-пользователи считают приемлемым, если их браузер возвращает кешированный документ, когда его срок действия не был проверен в последние несколько минут. Вместе с тем существует также много вариантов, когда должны быть соблюдены строгие гарантии согласованности, например в случае электронных бирж и аукционов. Проблема со строгим соответствием заключается в том, что обновление должно быть немедленно распространено на все остальные копии. Более того, если два обновления происходят одновременно, часто также требуется, чтобы обновления везде обрабатывались в одном и том же порядке, приводя к дополнительной глобальной проблеме с заказом. Проблема усугубляется при необходимости сочетания соответствия с другими желательными свойствами, такими как доступность, и может стать просто нерешаемой, что мы обсудим в главе 8.

Поэтому для репликации часто требуется некоторый глобальный механизм синхронизации. К сожалению, такие механизмы чрезвычайно сложны, или их даже невозможно реализовать при масштабировании, хотя бы потому, что сетевые задержки имеют естественную нижнюю границу. Следовательно, масштабирование путем репликации может привести к другим, не относящимся к масштабируемости проблемам. Мы возвращаемся к репликации и согласованности в главе 7.

Обсуждение. При рассмотрении методов масштабирования можно отметить, что размеры масштабируемости наименее проблематичны с технической точки зрения. Во многих случаях увеличение мощности компьютера спасет положение, хотя, возможно, это и приведет к большим финансовым затратам. Географическая масштабируемость – гораздо более сложная проблема, поскольку сетевые задержки, естественно, имеют границы. Как следствие мы можем быть вынуждены копировать данные в места, близкие к клиентам, что, в свою очередь, приводит к проблемам поддержания копий согласованными. Практика показывает, что сочетание методов распространения, репликации и кеширования с различными формами согласованности обычно приводит к приемлемым решениям. Наконец, административная масштабируемость представляется наиболее сложной проблемой, отчасти потому, что нам нужно заниматься не техническими вопросами, а такими, как политика организаций и человеческое сотрудничество. Введено и в настоящее время широко распространено использование одноранговой технологии, которая успешно продемонстрировала, что может быть достигнуто, если конечные пользователи получают контроль [Lua et al., 2005; Орам, 2001]. Однако одноранговые сети, очевидно, не являются универсальным решением для всех проблем административной масштабируемости.

Ловушки

Теперь уже должно быть ясно, что разработка распределенной системы является трудной задачей. Как мы увидим в этой книге не однажды, вопро-

сов, которые следует одновременно учитывать, так много, что сложность их решения кажется закономерной. Вместе с тем, следуя ряду принципов проектирования, могут быть разработаны распределенные системы, в которых строго выполняются поставленные нами в этой главе цели.

Распределенные системы отличаются от традиционного программного обеспечения тем, что компоненты рассредоточены в сети. Не принимать во внимание при проектировании эту дисперсию – значит делать системы излишне сложными с недостатками, которые приходится потом исправлять. Питер Дойч (Peter Deutsch), работавший тогда в компании Sun Microsystems, перечислил следующие ложные предположения, которые делает каждый разрабатывающий впервые распределенное приложение:

- сеть надежна;
- сеть безопасна;
- сеть однородна;
- топология не меняется;
- задержка равна нулю;
- пропускная способность бесконечна;
- транспортные расходы равны нулю;
- есть один администратор.

Обратите внимание, как эти предположения относятся к свойствам, которые являются уникальными для распределенных систем: надежность, безопасность, неоднородность и топология сети; задержка и пропускная способность; транспортные расходы; и, наконец, административные домены. При разработке нераспределенных приложений многие из этих проблем, скорее всего, не появятся.

Большинство принципов, которые мы обсуждаем в этой книге, имеют непосредственное отношение к этим предположениям. Во всех случаях мы будем обсуждать решения проблем, которые вызваны тем, что одно или несколько предположений являются ложными. Например, надежные сети просто не существуют и приведут к невозможности достижения прозрачности отказа. Мы посвящаем целую главу тому, что сетевое общение по своей природе небезопасно. Мы уже обсуждали вопрос о том, что распределенные системы должны быть открытыми и учитывать гетерогенность. Аналогично при обсуждении репликации для решения проблем масштабируемости мы в основном решаем проблемы с временно́й задержкой и пропускной способностью. В этой книге в различных разделах мы также коснемся вопросов управления.

1.3. Типы распределенных систем

Прежде чем начать обсуждение принципов, посмотрим более внимательно на различные типы распределенных систем и разделим распределенные системы на информационные распределенные системы и распространенные (всеобъемлющие) системы (которые являются распределенными по своей природе).

Высокопроизводительные распределенные вычисления

Важным классом распределенных систем является класс систем, который используется для обеспечения высокой производительности вычислительных задач. Грубо можно различать две подгруппы. В **кластерных вычислениях** (cluster computing) базовое оборудование состоит из набора аналогичных рабочих станций или компьютеров, тесно связанных посредством высокоскоростной локальной сети. Кроме того, все узлы имеют одинаковые операционные системы. Ситуация становится совсем другой в случае **сетевых вычислений** (grid computing). Эта подгруппа состоит из распределенных систем, которые часто строятся как федерация компьютерных систем, где каждая система может подпадать под административный домен, и может сильно отличаться, когда речь идет об оборудовании, программном обеспечении и развернутой сетевой технологии.

С точки зрения сетевых вычислений, следующим логическим шагом будет просто *аутсорсинг* всей инфраструктуры, необходимой для приложений с интенсивными вычислениями. По сути, это и есть **облачные вычисления** (cloud computing): обеспечение средствами для динамичного построения инфраструктуры и объединения того, что необходимо для доступа к услугам. В отличие от сетевых вычислений, которые связаны с высокопроизводительными вычислениями, облачные вычисления гораздо больше, чем просто предоставление большого количества ресурсов. Мы кратко обсудим это здесь, но будем возвращаться к различным аспектам этого вопроса на протяжении всей книги.

Примечание 1.8 (дополнительная информация: параллельная обработка)

Высокопроизводительные вычисления в значительной мере обязаны появлению многопроцессорных машин. В них несколько процессоров организованы таким образом, что все они имеют доступ к одной и той же физической памяти, как показано на рис. 1.6a. Напротив, в мультикомпьютерной системе несколько компьютеров подключены через сеть и нет разделения основной памяти, как показано на рис. 1.6b. Модель общей памяти оказалась очень удобной для повышения производительности программ и относительно легко программировалась.

Суть в том, что одновременно выполняется несколько потоков управления и все потоки имеют доступ к общим данным. Доступ к этим данным контролируется через хорошо понятные механизмы синхронизации, такие как семафоры (для получения дополнительной информации о разработке параллельных программ см. [Ari, 2006] или [Herlihy and Shavit, 2008]). К сожалению, такая модель не столь легко масштабируется: до настоящего времени разрабатывались машины, в которых эффективный доступ к общей памяти имеют всего несколько десятков (иногда сотен) процессоров. Отметим, что в определенной степени те же ограничения характерны и для многоядерных процессоров.

Чтобы преодолеть ограничения систем с разделяемой памятью, высокопроизводительные вычисления стали осуществлять на системах с распределенной памятью. Этот сдвиг также означал, что многие программы должны были использовать передачу сообщений вместо изменения общих данных как средство общения

и синхронизации между потоками. К сожалению, модели передачи сообщений оказались гораздо сложнее и более подвержены ошибкам по сравнению с моделями программирования с общей памятью. По этой причине были проведены значительные исследования в попытке построить так называемую **распределенную общую память мультикомпьютеров** (distributed sharedmemory multicomputers, DSM) [Amza et al., 1996].

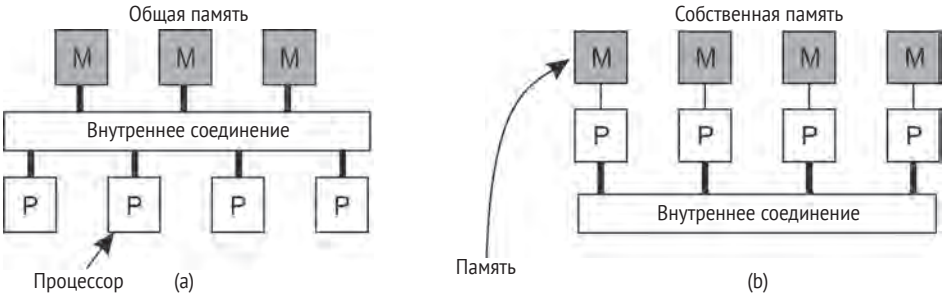

Рис. 1.6 ❖ Сравнение архитектур (a) мультипроцессора и (b) мультикомпьютера

По сути, система DSM позволяет процессору обращаться к области памяти на другом компьютере, как если бы это была локальная память. Это может быть достигнуто с помощью существующих методов, доступных операционной системе, например образованием из всех страниц основной памяти различных процессоров единого виртуального адресного пространства. Если процессор A обращается к странице, расположенной на другом процессоре B, страница с ошибкой, появляющаяся на A, позволяет операционной системе в точке A получать содержимое страницы B так же, как она обычно получает содержимое локально из диска, а процессор B будет проинформирован о том, что эта страница в настоящее время недоступна. От этой элегантной идеи имитации систем с разделяемой памятью и использованием мультикомпьютеров пришлось, в конце концов, отказаться по той простой причине, что производительность не соответствовала ожиданиям программистов, которые предпочли бы более сложные, но предсказуемо выполняемые модели программ передачи сообщений. Важным побочным эффектом изучения аппаратно-программных границ параллельной обработки стало глубокое понимание согласованности моделей, к которой мы вернемся в главе 7.

Кластерные вычисления

Кластерные вычислительные системы стали популярны, когда для персональных компьютеров и рабочих станций улучшилось соотношение цена/производительность. С некоторых пор стало и финансово, и технически привлекательным создание суперкомпьютера с использованием готовой технологии, просто соединив набор относительно простых компьютеров высокоскоростной сетью. Практически кластерные вычисления используются во всех случаях параллельного программирования, в котором одна (интенсивная вычислительная) программа выполняется параллельно на нескольких машинах.

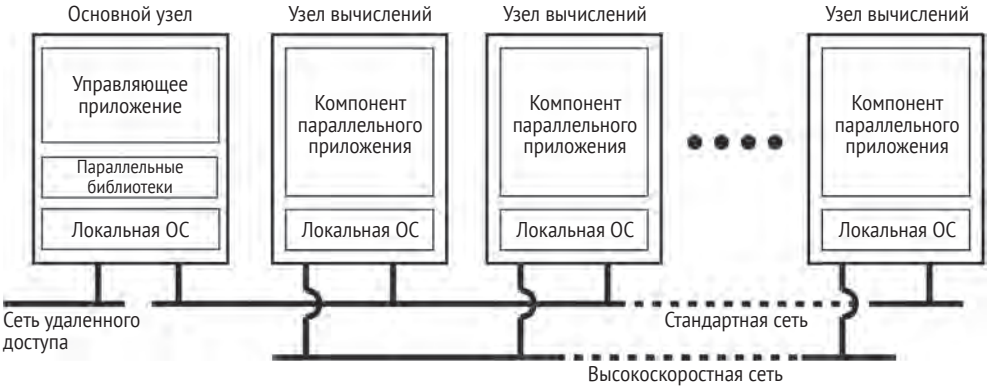

Рис. 1.7 ❖ Пример кластерной вычислительной системы

Примером широко применяемого кластерного компьютера является компьютер, созданный на основе ОС Linux с кластерами Беовульфа, общая конфигурация которых показана на рис. 1.7. Каждый кластер состоит из набора вычислительных узлов, которые управляются и доступны с помощью одного основного узла. Основной узел типично обрабатывает распределение узлов для конкретной параллельной программы, поддерживает пакетную очередь отправленных заданий и предоставляет интерфейс для пользователей системы. Таким образом, основной узел фактически запускает промежуточное программное обеспечение, необходимое для выполнения программ и управления кластером, а вычислительные узлы оснащены стандартной операционной системой с расширенной функцией промежуточного программного обеспечения для связи, хранения, отказоустойчивости и т. д. Таким образом, кроме основного узла, все вычислительные узлы идентичны.

Еще более симметричный подход используется в системе MOSIX [Amar et al., 2004]. Система MOSIX пытается предоставить **односистемный образ** (single-system image) кластера, что означает предоставление кластерным компьютером в высшей степени прозрачного распределения для обработки, представившись одним компьютером. Как мы уже упоминали, предоставление такого образа при любых обстоятельствах невозможно. В случае же MOSIX высокая степень прозрачности обеспечивается за счет разрешения процессов динамически и превентивной миграции между узлами, которые составляют кластер. Процесс миграции позволяет пользователю запускать приложение на любом узле (называемом домашним узлом), после которого он может прозрачно перейти к другим узлам, например с целью эффективного использования ресурсов. Мы вернемся к процессу миграции в главе 3. Аналогичные подходы с попыткой обеспечить односистемные образы сравниваются в [Lottiaux et al., 2005]. Однако в нескольких современных кластерных компьютерах отказались от таких симметричных архитектур в пользу более гибридных решений, в которых промежуточное программное обеспечение функционально разделено между различными узлами [Engelmannt al., 2007]. Преимущество такого разделения очевидно: наличие вычислительных узлов с выделенными, облегченными операционными системами обеспечит

более оптимальную производительность для приложений с интенсивными вычислениями. Точно так же, скорее всего, функциональность хранилища может быть оптимально обработана другими сконфигурированными узлами, такими как файловый сервер и сервер каталогов. То же самое относится и к иным специализированным сервисам промежуточного программного обеспечения, включая управление заданиями, услуги баз данных и, возможно, общий доступ в интернете к внешним услугам.

Сетевые вычисления

Характерной особенностью традиционных кластерных вычислений является их однородность. В большинстве случаев компьютеры в кластере в основном одинаковы, имеют одну и ту же операционную систему и все подключены к одной сети. Однако, как мы только что обсуждали, наметилась тенденция к архитектурам, в которых узлы специально настроены для определенных задач. Такое разнообразие в большой степени присуще сетевым вычислительным системам, в которых не делается никаких предположений относительно одинаковости аппаратных средств, операционных систем, сетей, административных доменов, политики безопасности и т. д.

Ключевой вопрос в сетевых вычислительных системах заключается в том, что собранные вместе ресурсы разных организаций и обеспечивающие сотрудничество группы людей из разных учреждений должны действительно образовывать федерацию систем. Такое сотрудничество осуществляется в форме **виртуальной организации** (virtual organization). Процессы, принадлежащие одной виртуальной организации, имеют право доступа к предоставляемым этой организации ресурсам, которые, как правило, состоят из вычислительных серверов (включая суперкомпьютеры, возможно, реализованные в виде кластера компьютеров), хранилищам и базам данных. Кроме того, в кластере также могут быть предоставлены и специальные сетевые устройства, такие как телескопы, датчики и т. д.

Учитывая существо сетевых вычислений, большая часть программного обеспечения для их реализации сводится к предоставлению доступа к ресурсам из разных административных доменов и только тем пользователям и приложениям, которые входят в конкретную виртуальную организацию. По этой причине основное внимание чаще всего уделяется архитектурным вопросам. Архитектура, первоначально предложенная в [Foster et al., 2001] и которая до сих пор является основой для многих вычислительных систем, показана на рис. 1.8.

Архитектура состоит из четырех слоев. Самый нижний *уровень сборки* обеспечивает интерфейсы к локальным ресурсам на конкретном сайте. Обратите внимание, что эти интерфейсы адаптированы для совместного использования ресурсов внутри виртуальной организации. Как правило, они будут предоставлять функции для запроса состояния и возможностей ресурса наряду с функциями для фактического управления ресурсами (например, блокировки ресурсов).

Уровень связи состоит из протоколов связи для поддержки сетевых транзакций, которые охватывают использование нескольких ресурсов. Например,

необходимы протоколы для обмена данными между ресурсами или просто для доступа ресурса из удаленного места. Кроме того, уровень связи будет содержать протоколы безопасности для аутентификации пользователей и ресурсов. Обратите внимание, что во многих случаях пользователи не проходят проверку подлинности. Вместо этого проходят аутентификацию программы, действующие от имени пользователей. В этом смысле делегирование прав от пользователя программе является важной функцией, которая должна поддерживаться на уровне связи. Мы вернемся к делегированию прав при обсуждении вопросов безопасности в распределенных системах в главе 9.

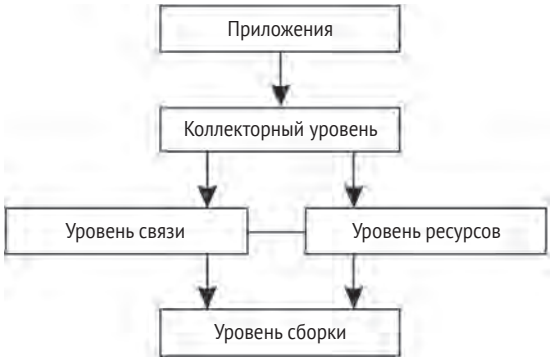

Рис. 1.8 ❖ Многоуровневая архитектура сетевых вычислительных систем

Уровень ресурсов отвечает за управление одним ресурсом. Он использует функции, предоставляемые уровнем связи, и вызывает напрямую интерфейсы, делая их доступными для уровня сборки. Например, этот уровень будет предлагать функции для получения информации о конфигурации конкретного ресурса или, вообще, выполнять определенные операции, такие как создание процесса или чтение данных. Уровень ресурсов, таким образом, считается ответственным за контроль доступа и, следовательно, будет полагаться на аутентификацию, выполняемую как часть уровня связи.

Следующий уровень в иерархии – это *коллекторный уровень*. Он управляет доступом к нескольким ресурсам и обычно состоит из сервисов для обнаружения ресурсов, распределения и планирования задач на нескольких ресурсах, данных тиражирования и т. д. В отличие от уровней связи и ресурсов, состоящих каждый из относительно небольшого, стандартного набора протоколов, коллективный уровень может состоять из множества различных протоколов, отражающих широкий спектр услуг, которые он может предложить виртуальной организации.

Наконец, *уровень приложений* состоит из приложений, которые работают в виртуальной организации и обеспечивают использование сетки вычислительной среды.

Обычно коллекторный уровень, уровень связи и уровень ресурсов формируют то, что можно назвать уровнем сетевого промежуточного программ-

ного обеспечения. Эти уровни совместно обеспечивают доступ к ресурсам, которые потенциально могут быть распределены по нескольким сайтам.

Важным наблюдением с точки зрения промежуточного программного обеспечения является то, что при сетевом вычислении распространенным является понятие сайта (или административной единицы). Эта распространенность подчеркивается постепенным сдвигом в сторону **сервис-ориентированной архитектуры** (service-oriented architecture), в которой сайты предлагают доступ к различным уровням через коллекцию веб-сервисов [Joseph et al., 2004]. Это к настоящему времени привело к определению альтернативной архитектуры, известной как архитектура **открытого сетевого сервиса** (Open Grid Services, OGSA) [Foster et al., 2006]. Архитектура OGSA основана на оригинальных идеях, сформулированных в [Foster et al., 2001], но прохождение ею процесса стандартизации делает ее довольно сложной. Пользователи OGSA обычно следуют стандартам веб-сервиса.

Облачные вычисления

Пока исследователи размышляли о том, как организовать вычислительные сети, которые были бы легкодоступны, организации, отвечающие за эксплуатацию центров обработки данных, столкнулись с проблемой открытия своих ресурсов для клиентов. В конце концов, это привело к концепции утилитарных вычислений, с помощью которых клиент может загружать задачи в центр обработки данных и платить за каждый ресурс. Утилиты вычисления составили основу того, что сейчас называется **облачными вычислениями** (cloud computing). Следуя [Vaquero et al., 2008], облачные вычисления характеризуются легко используемым и доступным пулом *виртуальных ресурсов*, которые, как и используемые ресурсы, могут быть настроены динамически, обеспечивая основу для масштабируемости: если требуется выполнить больше работы, клиент может просто приобрести больше ресурсов. Ссылка на утилиту вычислений формируется тем фактом, что облачные вычисления обычно основаны на модели оплаты за использование, в которой гарантии предлагаются посредством специальных *соглашений об уровне обслуживания* (*service level agreements*, SLA).

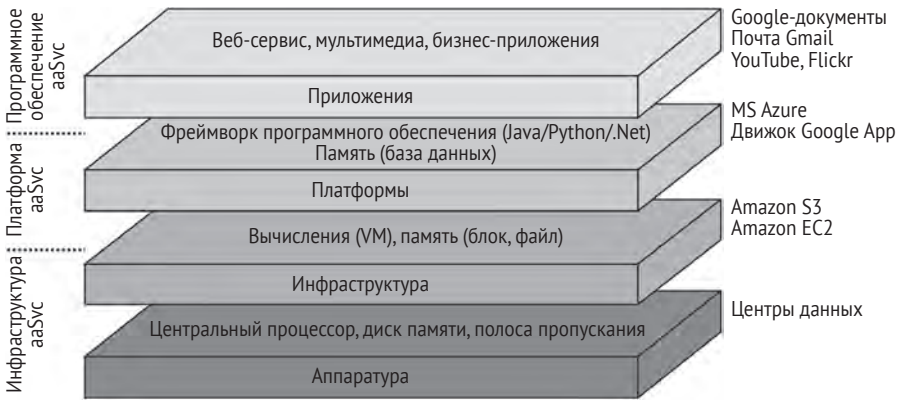

Рис. 1.9 ❖ Организация облаков (адаптировано из [Zhang et al., 2010])

На практике облака организованы в четыре слоя, как показано на рис. 1.9 (см. также [Zhang et al., 2010]):

- **аппаратное обеспечение.** Самый нижний уровень формируется средствами управления необходимым аппаратным обеспечением: процессорами, маршрутизаторами, а также системами питания и охлаждения. Обычно применяется в центрах обработки данных и содержит ресурсы, которые клиенты напрямую никогда не видят;
- **инфраструктура.** Это важный уровень, формирующий основу для большинства платформ облачных вычислений. Он использует методы виртуализации (обсуждается в разделе 3.2), обеспечивающие клиентов инфраструктурой, состоящей из виртуальной памяти и вычислительных ресурсов. На самом деле все это не то, чем кажется: облачные вычисления развиваются вокруг распределения и управления виртуальными устройствами хранения и виртуальными серверами;
- **платформа.** Можно утверждать, что уровень платформы обеспечивает клиенту облачных вычислений то, что операционная система обеспечивает приложению, а именно средства, облегчающие разработку и внедрение разработчиками приложений, которые нужно запустить в облаке. На практике разработчику прилагается специфичный для поставщика интерфейс (API), который включает в себя вызовы для загрузки и выполнения программы в облаке этого поставщика. В некотором смысле это сопоставимо с семейством системных вызовов Unix exec, которые принимают исполняемый файл как параметр и передают его операционной системе для выполнения.

  Как и операционные системы, уровень платформы обеспечивает более высокий уровень абстракции для хранения и других процедур. Например (мы будем обсуждать это более подробно позже), система хранения Amazon S3 [Murty, 2008] предлагается разработчику приложения в виде API, разрешающего (локально созданные) файлы организовать и хранить в пакетах. Пакет можно сравнить с директорией. Сохраненный в пакете файл автоматически загружается в облако Amazon;
- **приложение.** На этом уровне запускаются и предлагаются пользователям для дальнейшей настройки актуальные приложения. Хорошо известные примеры включают приложения, имеющиеся в офисных наборах (текстовые процессоры, электронные таблицы, приложения презентации и т. д.). Важно понимать, что эти приложения выполняются в облаке поставщика. Как и раньше, их можно сравнить с традиционным набором приложений, которые поставляются для установки вместе с операционной системой.

Поставщики облачных вычислений предлагают эти уровни своим клиентам через различные интерфейсы (включая инструменты командной строки, программные интерфейсы и веб-интерфейсы), что приводит к трем различным типам услуг:

- **инфраструктура как услуга** (Infrastructure-as-a-Service IaaS), охватывающая оборудование и уровень инфраструктуры;
- **платформа как услуга** (Platform-as-a-Service, PaaS), охватывающая уровень платформы;

- **программное обеспечение как услуга** (Software-as-a-Service, SaaS), в которое включаются приложения поставщика.

Использовать облака в настоящее время относительно легко, и мы обсудим более конкретные примеры интерфейсов для облачных провайдеров в последующих главах. Как следствие облачные вычисления, будучи средством для инфраструктуры аутсорсинга локальных вычислений, стали хорошим выбором для многих предприятий. Однако есть еще ряд серьезных препятствий, включая блокировку провайдера, безопасность, вопросы конфиденциальности, зависимость от доступности услуг и ряд других (см. также [Armbrust et al., 2010]). Кроме того, в связи с тем, что, как правило, детали того, как фактически выполняются конкретные облачные вычисления, скрыты и даже, возможно, неизвестны или непредсказуемы, выполнение требований по производительности оговорить заранее может стать невозможным. Кроме того, в [Li et al. 2010] показано, что разные провайдеры могут легко указывать очень разные профили производительности. Облачные вычисления больше не ажиотаж и, безусловно, являются серьезной альтернативой для поддержания огромной местной инфраструктуры, но еще многое может быть улучшено.

Примечание 1.9 (дополнительно: облачные вычисления дешевле?)

Одна из важных причин перехода в облачную среду заключается в том, что это намного дешевле по сравнению с обслуживанием локальной вычислительной инфраструктуры. Есть много способов подсчитать экономию, но, как оказалось, реальную оценку можно получить только для простых и очевидных случаев. В работе [Hajjat and et. al, 2010] предлагается более тщательный подход: переносить в облако только часть набора приложений, а другую часть оставлять работающей в локальной инфраструктуре. Суть этого метода заключается в обеспечении правильной модели набора корпоративных приложений.

Основу такого подхода составляет потенциально большой набор *программного обеспечения* компонентов. Предполагается, что каждое корпоративное приложение состоит из компонентов. Кроме того, считается, что каждый компонент C_i работает на N_i-сервере. Простым примером является компонент базы данных, который будет выполняться одним сервером. Более сложным примером является веб-приложение для расчета велосипедных маршрутов, состоящее из интерфейса веб-сервера для рендеринга HTML-страниц и принятия пользовательского ввода компонентов для вычисления кратчайших путей (возможно, при различных ограничениях) и компонентов базы данных, содержащих различные карты.

Каждое приложение моделируется как ориентированный граф, в котором вершина представляет компонент, а дуга $arc\langle i, \overrightarrow{j}\rangle$ – тот факт, что данные передаются от компоненты C_i к компоненте C_j. Каждая дуга имеет два связанных веса: $T_{i,j}$ представляет число транзакций за единицу времени, приводящих к потокам данных от C_i до C_j, а $S_{i,j}$ – среднее значение размера этих транзакций (то есть средний объем данных за транзакцию). Предполагается, что $T_{i,j}$ и $S_{i,j}$ известны и, как правило, получаются простым измерением.

Миграция набора приложений из локальной инфраструктуры в облако затем сводится к поиску оптимального плана миграции M: для каждого компонента C_i выясняется, сколько n_i его серверов N_i следует перенести в облако, так чтобы денежные расходы, получаемые от M, максимально уменьшались на дополнительную стоимость общения по интернету. План M также должен соответствовать следующим ограничениям.

Ограничения политики выполнены. Например, могут быть данные, которые юридически должны быть расположены в локальной инфраструктуре организации.

Поскольку связь в настоящее время частично осуществляется через интернет-каналы большой дальности, определенные транзакции между компонентами могут выполняться намного медленнее. План M может быть приемлем только в том случае, если какие-либо дополнительные задержки не нарушают конкретные ограничения.

Должны соблюдаться уравнения баланса потоков: транзакции продолжают работать правильно, и запросы или данные во время транзакции не теряются.

Экономический эффект. Для каждого плана миграции M можно ожидать экономии денежных средств, выражающейся как экономия (M), поскольку нужно поддерживать меньше машин или сетевых подключений. Во многих организациях такие затраты известны, так что экономия может быть относительно легко подсчитана. С другой стороны, существуют расходы на использование облака. В работе [Hajjat et al., 2010] сделано упрощающее различие между экономией B_c миграции компонента, интенсивно использующего вычислительные ресурсы, и экономией B_s от миграции компонента, интенсивно использующего память. Если обозначить компоненты через M_c с вычислительной интенсивностью и M_s – с интенсивным хранением, у нас есть *экономия*(M) = $B_c M_c + B_s \cdot M_s$. Очевидно, что могут быть развернуты и более сложные модели.

Расходы на интернет. Чтобы рассчитать увеличение затрат на связь, потому что компоненты распределены по облаку и по локальной инфраструктуре, нам нужно принимать во внимание пользовательские запросы. Чтобы упростить дело, мы не делаем различия между внутренними пользователями (то есть членами предприятия) и внешними пользователи (как можно видеть в случае веб-приложений). Трафик от пользователей до миграции может быть выражен как

$$Tr_{local,inet} = \sum_{C_i} (T_{user,i} S_{user,i} + T_{i,user} S_{i,user}),$$

где $T_{user,i}$ обозначает количество транзакций в единицу времени, ведущих к данным, исходящим от пользователей к C_i. У нас есть аналогичные интерпретации для $T_{i,user}$, S_{user_i} и $S_{i,user}$.

Для каждого компонента C_i пусть $C_{i,local}$ обозначает серверы, которые продолжают работать в локальной инфраструктуре, и $C_{i,cloud}$ – размещенные в облаке серверы. Заметим, что $|C_{i,cloud}| = n_i$. Для простоты предположим, что сервер из $C_{i,local}$ раздает трафик в тех же пропорциях, что и сервер от $C_{i,cloud}$. Мы заинтересованы в скорости транзакций между локальными серверами, облачными серверами и между локальными и облачными серверами после миграции. Пусть s_k будет сервером для компонента C_k, и обозначим через f_k дробь n_k/N_k. Тогда мы имеем для скорости транзакций $T^*_{i,j}$ после миграции:

$$T^*_{i,j} = \begin{cases} (1-f_i)\cdot(1-f_j)\cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \\ (1-f_i)\cdot f_j \cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \\ f_i \cdot (1-f_j)\cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \\ f_i \cdot f_j \cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \end{cases}.$$

$S_{i,j}$ – количество данных, связанных с $T^*_{i,j}$. Обратите внимание, что f_k обозначает долю сервера компоненты C_k, которая перемещается в облако. Другими словами, $(1 - f_k)$ – часть, которая остается в местной инфраструктуре. Мы оставляем читателю получение выражения для $T^*_{i,user}$.

Наконец, обозначим через $cost_{local,inet}$ и $cost_{cloud,inet}$ стоимость трафика через интернет в расчете на единицу для в и из локальной инфраструктуры и облака соответственно. Игнорируя несколько тонкостей, объясненных в [Hajjat et al., 2010], мы можем затем вычислить локальный интернет-трафик после миграции как:

$$Tr^*_{local,inet} = \sum_{C_{i,local}, C_{j,local}} (T^*_{i,j} S^*_{i,j} + T^*_{j,i} S^*_{j,i}) + \sum_{C_{j,local}} (T^*_{user,j} S^*_{user,j} + T^*_{j,user} S^*_{j,user}).$$

И таким же образом для трафика облака интернета после миграции:

$$Tr^*_{cloud,inet} = \sum_{C_{i,cloud}, C_{i,cloud}} (T^*_{i,j} S^*_{i,j} + T^*_{j,i} S^*_{j,i}) + \sum_{C_{j,cloud}} (T^*_{user,j} S^*_{user,j} + T^*_{j,user} S^*_{j,user}).$$

Объединив, мы получим модель для увеличения стоимости связи:

$$cost_{local,inet}(Tr^*_{local,inet} - Tr_{local,inet}) + cost_{cloud,inet} Tr^*_{cloud,inet}.$$

Очевидно, что для ответа на вопрос, дешевле ли переход в облако, нужно иметь много подробной информации и провести тщательное планирование того, что именно нужно для миграции. Статья [Hajjat et al., 2010] показывает первый шаг к принятию обоснованного решения. Их модель более подробна, чем мы готовы объяснить ее здесь. Важный аспект, который мы не затронули, – это то, что перенос компонентов требует также внимания к безопасности миграции компонентов. Заинтересованный читатель отсылается к их статье.

Распределенные информационные системы

Другой важный класс распределенных систем приходится на организации, которые сталкиваются с применением многих сетевых приложений и для которых совместимость оказывается болезненным опытом. Многие из существующих решений промежуточного программного обеспечения являются результатом работы с инфраструктурой, в которой проще интегрировать приложения в корпоративную информационную систему [Alonso et al., 2004; Bernstein, 1996; Hohpe and Woolf, 2004].

Мы можем выделить несколько уровней, на которых может происходить интеграция. Во многих случаях сетевое приложение просто состоит из сервера, на котором стоит приложение (и часто включает в себя базу данных) и который делает его доступным для удаленных программ, называемых клиентами. Такие клиенты отправляют запрос на сервер для выполнения конкретной операции, после которой ответ отправляется обратно. Интеграция на самом низком уровне позволяет клиентам упаковать несколько запросов, возможно для разных серверов, в один большой запрос и выполнить его как **распределенную транзакцию** (distributed transaction). Основная идея заключается в том, что выполняются или все, или ни один из запросов. По мере того как приложения становились все более изощренными и постепенно разделялись на независимые компоненты (особенно выделяя компоненты базы данных из компонентов обработки), стало ясно, что, обеспечивая общение напрямую, интеграция приложений также имеет право на существование. Эта привело в настоящее время к огромной отрасли, которая сосредоточена на **корпоративных приложениях** (Enterprise Application Integration, EAI).

Распределенная обработка транзакций

Чтобы конкретизировать наше обсуждение, мы сосредоточимся на приложениях базы данных. На практике операции с базой данных осуществляются

в форме **транзакций**. Программирование с использованием транзакций требует специальных примитивов, которые должны либо предоставляться базовой распределенной системой, либо языковой системой поддержки выполнения программы. Типичные примеры примитивов транзакций показаны на рис. 1.10. Точный список примитивов зависит от того, какие объекты используются в транзакции [Gray and Reuter, 1993; Bernstein and Newcomer, 2009]. В почтовой системе могут быть примитивы для отправки, получения и пересылки почты. В системе бухгалтерского учета они могут быть совершенно разными. Типичными примерами являются `READ` и `WRITE`. Внутри транзакции также допускаются обычные операторы, вызовы процедур и т. д. В частности, **удаленные вызовы процедур** (remote procedure calls, RPC), то есть вызовы процедур к удаленным серверам, также часто инкапсулируются в транзакции, что приводит к тому, что известно как **транзакционный RPC**. Мы обсуждаем **RPC** подробно в разделе 4.2.

Примитив	Описание
`BEGIN_TRANSACTION`	Отметить начало транзакции
`END_TRANSACTION`	Завершить транзакцию и восстановить старые значения
`ABORT_TRANSACTION`	Завершить транзакцию и попытаться зафиксировать
`READ`	Чтение данных из файла, таблицы или иным образом
`WRITE`	Записать данные в файл, таблицу или другое

Рис. 1.10 ❖ Пример примитивов для транзакций

Примитивы `BEGIN_TRANSACTION` и `END_TRANSACTION` используются для разграничения объема транзакции. Операции между ними образуют тело транзакции. Характерной особенностью транзакции является то, что или выполняются все эти операции, или не выполняется никакая. Ими могут быть системные вызовы, библиотечные процедуры или операторы в квадратных скобках на языке, в зависимости от применения.

Это свойство транзакций «все или ничего» является одним из четырех свойств, которые имеют транзакции. В частности, транзакции придерживаются так называемого свойства ACID (Atomic, Consistent, Isolated, Durable):

- **атомарная:** для внешнего мира транзакция осуществляется неделимой;
- **согласованная:** транзакция не нарушает системные инварианты;
- **изолированная:** параллельные транзакции не мешают друг другу;
- **продолжающаяся:** после совершения транзакции изменения являются постоянными.

В распределенных системах транзакции часто строятся как ряд субтранзакций, совместно формирующих **вложенную транзакцию** (nested transaction), как показано на рис. 1.11. Транзакция верхнего уровня может разветвиться на потомков, которые работают параллельно один с другим на разных машинах, для повышения производительности или упрощения программирования. Каждый из этих потомков может также выполнить одну или несколько субтранзакций или разветвлять своих детей.

Рис. 1.11 ❖ Вложенная транзакция

Субтранзакции вызывают тонкую и важную проблему. Представьте, что транзакция запускает несколько субтранзакций параллельно, и одна из них осуществляется, делая результаты видимыми для родительской транзакции. После дальнейшего вычисления родитель отменяет работу, восстанавливая всю систему до ее состояния, которое было до начала транзакции верхнего уровня. Следовательно, результаты выполненной субтранзакции тем не менее должны быть отменены. Таким образом, постоянство, упомянутое выше, относится только к транзакциям верхнего уровня.

Поскольку транзакции могут быть вложены сколь угодно глубоко, администрация транзакции требует значительных усилий, нужно, чтобы все было сделано правильно. Однако семантика ясна. Когда любая транзакция или субтранзакция начинается, ей концептуально предоставляется личная копия всех данных во всей системе, чтобы она могла манипулировать ею по своему усмотрению. Если она прерывается, ее личная вселенная просто исчезает, как будто ее никогда не существовало. Если это происходит, ее личная вселенная заменяет родительскую вселенную. Таким образом, если субтранзакция осуществляется, а затем запускается новая субтранзакция, вторая видит результаты, полученные первой. Аналогично, если вложенная (более высокий уровень) транзакция прерывается, все ее субтранзакции должны быть также прерваны. Если несколько транзакций запускаются одновременно, результат таков, как будто они запускались последовательно в неустановленном порядке.

Вложенные транзакции важны в распределенных системах, поскольку они обеспечивают естественный способ распределения транзакций по нескольким машинам. Они следуют *логическому* разделению работы исходной транзакции. Например, транзакция для планирования поездки, по которой должно быть зарезервировано три разных рейса, может быть логически разделена на три субтранзакции. Каждая из этих субтранзакций может управляться отдельно и независимо от других двух.

В начале появления корпоративных систем промежуточного программного обеспечения компонент, который обрабатывал распределенные (или вложенные) транзакции, формировал ядро для интеграции приложения на уровне сервера или базы данных. Этот компонент был назван **монитор обработки транзакций** (transaction processing monitor, TP monitor), или для

краткости TP-монитор. Его главной задачей было разрешить приложению доступ к нескольким серверам/базам данных, предлагая ему модель транзакционного программирования, как показано на рис. 1.12. По сути, TP-монитор координирует осуществление субтранзакций в соответствии со стандартным протоколом, известным как **распределенная фиксация** (distributed commit), который мы обсудим в разделе 8.5.

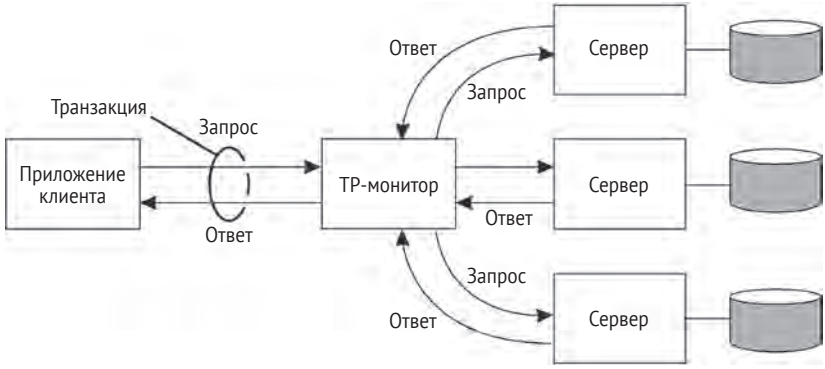

Рис. 1.12 ❖ Роль TP-монитора в распределенной системе

Важным обстоятельством является то, что приложения, желающие координировать несколько субтранзакций в единую транзакцию, не должны реализовывать эту координацию сами. Просто эту координацию для них делает TP-монитор. Именно здесь вступает в игру промежуточное ПО: оно реализует сервисы, которые полезны для многих приложений, и, таким образом, позволяет разработчикам приложений избегать повторения услуги снова и снова.

Интеграция корпоративных приложений

Как уже упоминалось, чем больше приложений отделяется от баз данных, в которых они были построены, тем становится все более очевидным, что необходимы средства для интеграции приложений независимо от их баз данных. В частности, приложения компонентов должны иметь возможность напрямую общаться друг с другом, и не просто с помощью средств реализации запроса/ответа системами обработки транзакций.

Эта потребность во взаимодействии приложений привела ко многим различным моделям коммуникации, основная идея которых заключалась в том, что существующие приложения могут напрямую обмениваться информацией, как показано на рис. 1.13.

Существует несколько типов связующего промежуточного программного обеспечения. Компонент приложения с **удаленной процедурой вызова** (remote procedure calls, RPC) может эффективно отправить запрос другому компоненту приложения, выполняя локальный вызов процедуры, что приводит к упаковке запроса как сообщения и отправке вызываемому абоненту.

Точно так же результат будет отправлен обратно и возвращен в приложение в результате вызова процедуры.

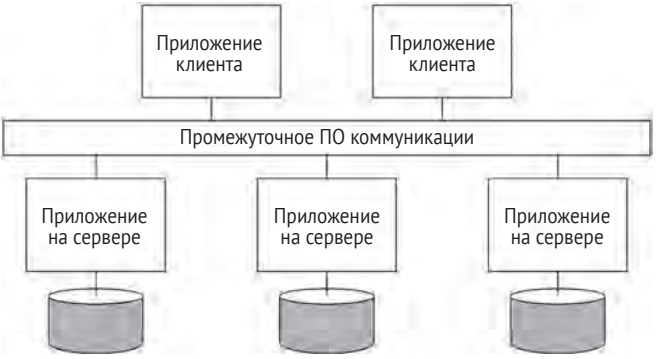

Рис. 1.13 ❖ Промежуточное программное обеспечение как средство коммуникации при объединении корпоративных приложений

Поскольку популярность объектной технологии возросла, были разработаны методы разрешения запросов к удаленным объектам, что привело к тому, что известно как **метод объединения удаленных вызовов** (remote method invocations, RMI). RMI – по сути такой же метод, как RPC, за исключением того, что он воздействует на объекты вместо функций. Методы RPC и RMI имеют тот недостаток, что как вызывающий, так и вызываемый абоненты во время общения должны быть в рабочем состоянии. Кроме того, им нужно точно знать, как обращаться друг к другу. Такое тесное соединение часто является серьезным недостатком и привело к так называемому **промежуточному программному обеспечению, ориентируемому на сообщения** (message-oriented middleware, MOM). В этом случае приложения отправляют сообщения к логическим точкам, часто описываемым средствами субъекта. Точно так же приложения могут указать свой интерес к определенному типу сообщения, после чего ПО коммуникации позаботится о том, чтобы эти сообщения были доставлены в эти приложения. Эти так называемые **системы публикации/подписки** (publish/subscribe systems) формируют важный и расширяющийся класс распределенных систем.

Примечание 1.10 (дополнительная информация: об интеграции приложений)

Поддержка интеграции корпоративных приложений является важной целью для многих промежуточных программ. В общем, есть четыре способа интеграции приложений [Hohpe и Вульф, 2004].

Передача файлов. Суть интеграции с помощью передачи файлов заключается в том, что приложение создает файл, содержащий общие данные, которые впоследствии читаются другими приложениями. Подход технически очень прост, что делает его привлекательным. Недостатком, однако, является то, что есть многое, что должно быть согласовано:

- формат и размещение файла: текст, двоичный файл, его структура и т. д. В настоящее время XML стал популярным, так как его файлы, в принципе, описывают себя сами;

- управление файлами: где они хранятся, как их называют, кто ответствен за удаление файлов;
- обновление распространения: когда приложение создает файл, может быть несколько приложений, которые должны прочитать этот файл, чтобы обеспечить единство согласованной системы, говорилось в разделе 1.1. Как следствие иногда необходимо реализовать отдельные программы, которые уведомляют приложения об обновлениях файлов.

Общая база данных. Многие проблемы, связанные с интеграцией через файлы, облегчаются при использовании общей базы данных. Все приложения будут иметь доступ к тем же данным, и часто с помощью языка высокого уровня, такого как SQL. Кроме того, приложения легко уведомлять о возникновении изменений, поскольку переключатели являются частью современных баз данных. Однако есть два основных недостатка. Во-первых, все еще существует необходимость в разработке общей схемы данных, что может оказаться далеко не тривиальным, если набор приложений, которые необходимо интегрировать, не полностью известен заранее. Во-вторых, когда есть много чтений и обновлений, общая база данных может легко стать узким местом в части производительности.

Удаленный вызов процедуры. Интеграция через файлы или базу данных неявно предполагает, что изменения, внесенные одним приложением, могут легко вызвать участие других приложений. Однако практика показывает, что иногда небольшие изменения действительно должны запускать многие приложения. В таких случаях на самом деле важно не изменение данных, а выполнение серии действий. Последовательность действий лучше всего фиксируется путем выполнения процедуры (что, в свою очередь, может привести к всевозможным изменениям в общих данных). Чтобы предотвратить то, что каждое приложение должно знать все внутренние действия (как это реализовано в другом приложении), могут быть использованы методы стандартной инкапсуляции как развернутые с традиционными вызовами процедур или как обращение к объекту. Для таких ситуаций приложению лучше всего предложить процедуру другим приложениям в форме удаленного вызова процедуры, или RPC. По сути, RPC позволяет приложению A использовать информацию, доступную только приложению B, без предоставления прямого доступа A к этой информации. Есть много преимуществ и недостатков в процедуре удаленных вызовов, которые подробно обсуждаются в главе 4.

Обмен сообщениями. Главный недостаток RPC заключается в том, что вызывающий и вызываемый абоненты для успешного обмена должны работать одновременно. Однако во многих сценариях эту одновременную деятельность часто бывает трудно или невозможно гарантировать. В таких случаях все, что нужно, – чтобы система, предлагающая обмен сообщениями, передающая запросы приложения A, выполнила действие в приложении. Система обмена сообщениями гарантирует, что в конечном итоге запрос будет доставлен, и если необходимо, то ответ также будет возвращен. Очевидно, что обмен сообщениями не является панацеей для интеграции приложений: он также вносит проблемы, касающиеся форматирования данных и размещения, требует, чтобы приложение знало, куда отправить сообщение, должны быть разработаны сценарии поиска потерянных сообщений и т. д. Как и RPC, мы будем обсуждать эти вопросы подробно в главе 4.

Эти четыре подхода говорят нам о том, что интеграция приложений обычно не бывает простой. Промежуточное программное обеспечение (в форме распределенной системы) может, однако, значительно помочь в интеграции, предоставляя необходимые средства, такие как поддержка для RPC или обмена сообщениями. Как уже говорилось, интеграция корпоративных приложений является важным целевым полем приложения сил для разработчиков промежуточного программного обеспечения.