С. Л. Бабичев, К. А. Коньков

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

УЧЕБНОЕ ПОСОБИЕ ДЛЯ ВУЗОВ

Рекомендовано Учебно-методическим отделом высшего образования в качестве учебного пособия для студентов высших учебных заведений, обучающихся по ИТ-направлениям

**Книга доступна в электронной библиотеке biblio-online.ru,
а также в мобильном приложении «Юрайт.Библиотека»**

Москва ▪ Юрайт ▪ 2019

УДК 004(075.8)
ББК 22.18я73
Б12

***Авторы*:**

Бабичев Сергей Леонидович — кандидат физико-математических наук, доцент кафедры информатики и вычислительной математики Московского физико-технического института (г. Долгопрудный);

Коньков Константин Алексеевич — кандидат физико-математических наук, доцент кафедры информатики и вычислительной математики Московского физико-технического института (г. Долгопрудный).

***Рецензент*:**

Тормасов А. Г. — доктор физико-математических наук, профессор, ректор АНО «Университет Иннополис».

Бабичев, С. Л.

Б12 Распределенные системы : учебное пособие для вузов / С. Л. Бабичев, К. А. Коньков. — Москва : Издательство Юрайт, 2019. — 507 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-11380-8

Книга представляет собой систематизированный учебный курс по теории распределенных систем. В ней рассмотрены фундаментальные принципы, концепции, технологии и особенности функционирования современных распределенных систем. Теоретический материал дополнен разнообразными практическими примерами. Книга предназначена студентам и преподавателям, а также может представлять интерес для специалистов в области информационных технологий. Для ее понимания достаточно иметь начальные сведения из курсов по теории алгоритмов, операционных систем, сетевых технологий и программированию.

Содержание учебника соответствует актуальным требованиям Федерального государственного образовательного стандарта высшего образования.

Для студентов высших учебных заведений (бакалавриат и магистратура), обучающихся по ИТ-направлениям.

УДК 004(075.8)
ББК 22.18я73

ISBN 978-5-534-11380-8

Оглавление

Предисловие

Изобретение высокоскоростных сетей позволило собирать компьютерные системы большой размерности, называемые распределенными системами (РС), и привело к быстрому росту числа приложений, требующих распределенной обработки. Снижение цен на системы хранения данных, рост пропускной способности сетей, а также квалификации пользователей позволили сделать распределенные системы производительными и отказоустойчивыми. Эти факторы, а также возможности беспроводных и мобильных технологий обусловили быстрое развитие распределенных приложений и интерес к сфере распределенных систем со стороны компьютерного сообщества, научных, коммерческих и правительственных организаций. Таким образом, построение распределенных вычислительных систем — важная область теоретической и практической информатики, которая затрагивает все аспекты распределенных вычислений и информационного доступа и нуждается в хорошо продуманном теоретическом обосновании.

По распределенным системам имеется обширная библиография, например, монографии [3, 6, 8, 17, 18] и ряд других, систематизирующая отдельные положения в этой области. В этих книгах всесторонне рассмотрены наиболее важные структурные и алгоритмические проблемы функционирования распределенных систем. В настоящей книге предпринята попытка синтеза теории и практики — наряду с традиционным анализом теоретических тем большое внимание будет уделено созданию условий для практической реализации наиболее интересных алгоритмов и разработке отдельных базовых подсистем в составе распределенной системы.

Книга состоит из 14 глав. Главы 1—7 посвящены теоретическим основам функционирования отказоустойчивых распределенных систем. В главах 8—14 рассмотрены практические вопросы реализации распределенных алгоритмов.

Книга написана на основе учебного курса по распределенным системам, читаемого авторами в течение ряда лет в Московском физико-техническом институте.

Книга может быть интересна студентам и аспирантам технических специальностей учебных заведений, программистам и дизайнерам распределенных масштабируемых систем. Для ее понимания достаточно иметь начальные сведения из курсов по теории алгоритмов, операционных систем, сетевых технологий и программированию.

В результате освоения дисциплины «Теория и практика распределенных систем» обучающийся должен:

знать

- фундаментальные понятия, законы, теории операционных систем, дискретной математики, теории информации, теории построения алгоритмов, криптографии (РС);
- современные проблемы соответствующих разделов теории информации, дискретной математики, теории построения алгоритмов, криптографии (РС);
- основы проектирования, построения и функционирования распределенных систем;
- основные свойства соответствующих объектов;

уметь

- понять поставленную задачу;
- использовать свои знания для решения задач проектирования, построения и использования РС;
- оценивать корректность постановок задач;
- строго доказывать или опровергать утверждения;
- самостоятельно находить алгоритмы решения задач, требующихся для проектирования, построения и использования РС, в том числе и нестандартных, и проводить их анализ;
- самостоятельно видеть следствия полученных результатов;
- точно представить математические знания в области РС в устной и письменной форме;

владеть

- навыками освоения большого объема информации и решения задач РС (в том числе сложных);
- навыками самостоятельной работы и освоения новых дисциплин;
- культурой постановки, анализа и решения математических и прикладных задач, требующих для своего решения в том числе и использования математических подходов, лежащих в основе РС;
- предметным языком теории информации, дискретной математики, криптографии, теории операционных систем и навыками грамотного описания решения задач и представления полученных результатов.

Благодарности

Авторы хотели бы поблагодарить сотрудников кафедры информатики МФТИ за помощь в создании книги.

Часть 1
ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Глава 1
ВВОДНАЯ ГЛАВА

1.1. Введение. Понятие распределенной системы

Под распределенной системой обычно понимают систему, в которой аппаратные и программные компоненты, расположенные в сетевых компьютерах, взаимодействуют и координируют свои действия только путем посылки сообщений. Это простейшее определение ведет к следующим основным важным особенностям распределенных систем: наличие в них параллельных процессов, зачастую разделяющих ресурсы, отсутствие глобальных часов и независимость отказов отдельных компонентов.

На рис. 1.1 показана типичная распределенная система, представляющая собой набор автономных узлов, состоящих из памяти и процессора, соединенных коммуникационной сетью.

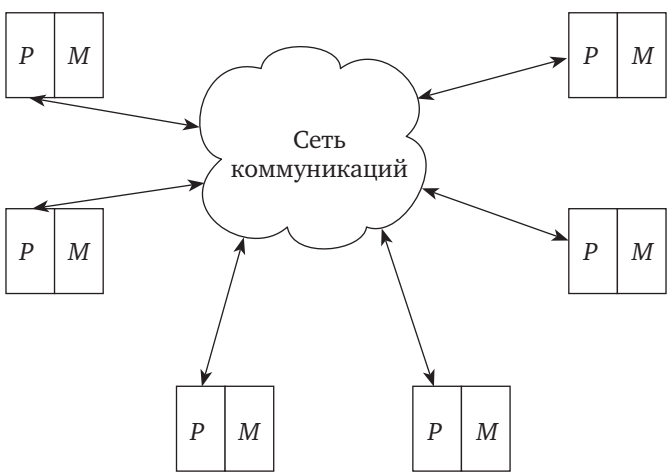

Рис. 1.1. **Распределенная система процессоров, связанных коммуникационной сетью**
P — процессоры; *M* — устройства памяти

1.1.1. Определение

В литературе отсутствует каноническое определение того, что является распределенной системой. Определения разных авторов не согласуются друг с другом. Например, в книге [18] сказано, что «под распределенной системой мы понимаем всякую вычислительную систему, в которой несколько компьютеров или процессоров так или иначе всту-

пают во взаимодействие». Под это определение наряду с глобальными коммуникационными вычислительными сетями попадают локальные сети, многопроцессорные компьютеры, в которых каждый компьютер наделен собственным устройством управления, а также системы взаимодействующих процессов.

Можно также считать, что распределенная система есть набор независимых узлов, взаимодействующих для решения проблемы, которая не может быть решена индивидуально.

При этом важно, что распределенная система представляется пользователю единой объединенной системой.

Моделью распределенной системы может быть набор программных средств, представляющий собой совокупность взаимосвязанных процессов, выполняемых на одном и том же вычислительном устройстве. Такое представление очень удобно для исследования свойств распределенных систем.

Однако в большинстве случаев в составе распределенных систем имеется несколько удаленных узлов, связанных друг с другом средствами коммуникации.

Независимость или автономность узлов является важной особенностью распределенных систем, и именно эта особенность позволяет им легко масштабироваться. Объединение системы обычно достигается за счет введения промежуточного или связующего программного слоя, в англоязычной литературе известного как *middleware*, который добавляет к лежащей в основе сети общую концепцию, обеспечивающую однородный вид системы. Промежуточный слой включает элементы коммуникационной подсистемы и, будучи распределенным между узлами, маскирует гетерогенность отдельных узлов, создавая платформу для выполнения распределенных приложений (рис. 1.2).

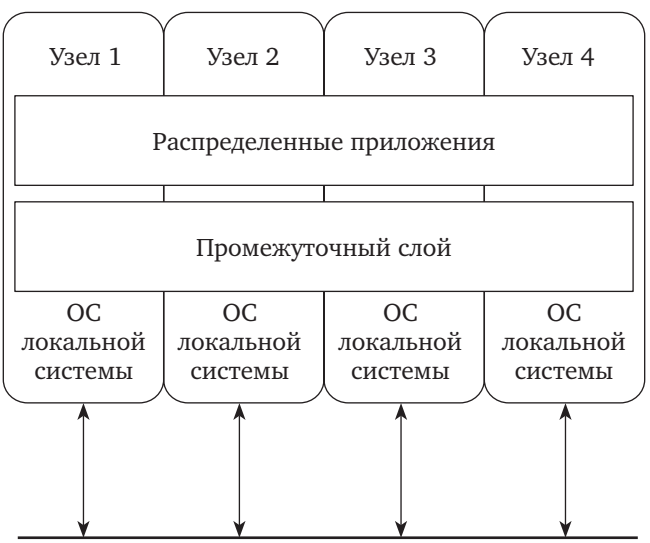

Рис. 1.2. **Распределенная система организована в виде службы промежуточного уровня**

1.1.2. Особенности распределенных систем

Проблемы кооперации взаимодействующих процессов характерны для самого широкого класса систем, от сильно связанных (симметричные мультипроцессорные системы) до слабосвязанных (географически удаленных автономных узлов). Специфику собственно распределенных систем можно определить следующими важными признаками.

- **Отсутствие общей памяти** и необходимость организовать взаимодействие путем посылки сообщений.
- **Отсутствие общих физических часов**. Разная скорость течения времени на узлах (*рассинхронизация*) приводит к необходимости прибегать к специальным алгоритмам упорядочивания событий в системе.
- **Асинхронная связь и асинхронное исполнение** в качестве базовых моделей.
- **Географическая удаленность**. Удаленность процессов есть отличительный признак распределенности. Впрочем, локальные сети и кластеры рабочих станций также могут считаться малыми распределенными системами.
- **Автономность и гетерогенность**. Процессы слабо связаны, исполняются на вычислительных узлах разной производительности, работают под управлением разных операционных систем.
- **Отказоустойчивость**. Сбой отдельных узлов или коммуникационных каналов не приводит к прекращению работы системы.
- **Недетерминизм**, который является следствием асинхронности. Задержки, возникающие при передаче сообщений, точно не известны, поэтому запросы могут поступать в разном порядке.

1.1.3. Целесообразность построения распределенных систем

Существует много важных задач, решение которых делает существование распределенных систем осмысленным. Одной из наиболее очевидных причин для использования распределенных систем является возможность разделения ресурсов, таких как периферийные устройства, базы данных, суперкомпьютеры и др., которые нецелесообразно или невозможно иметь на каждом отдельном узле.

Другой причиной может быть выполнение вычислений, имеющих «врожденную распределенность», например сбор информации с разных сайтов, банковские переводы, достижение консенсуса между субъектами.

Свойственные распределенным системам репликация данных и резервирование исполнительных модулей и технологии построения отказоустойчивых алгоритмов могут обеспечить повышенную надежность системы.

Наконец, можно добиться хороших показателей цена-производительность за счет гибкого наращивания мощности. Для этого важно уметь строить масштабируемые системы.

1.1.4. Примеры и применения

Распределенные системы широко используются в различных сферах деятельности человека. Ниже приведен далеко не полный перечень примеров применения распределенных систем, см. также [8, 17, 3].

- *Финансы и коммерция*. Рост электронной коммерции, например такой как Amazon или eBay, и связанные с ними такие нижележащие платежные технологии как PayPal; связанное возникновение банковских и торговых технологий, а также сложное распространение информации для финансовых рынков.
- *Информационные сообщества*. Рост WWW как репозитория информации; развитие поисковых Web-машин, таких как Google и Yandex, для поиска в этом обширном репозитории; возникновение цифровых библиотек и широкомасштабное преобразование в цифровой вид таких традиционных источников информации как книги (например, Google Books); увеличивающееся значение контента, сгенерированного пользователями через сайты, такие как YouTube, Wikipedia и Flickr; возникновение социальных сетей на основе таких сервисов как Facebook или MySpace.
- *Креативная индустрия и мероприятия*. Возникновение онлайн-игр и высокоинтерактивных форм мероприятий, доступность музыки и фильмов дома с помощью сетевых медиацентров или через загрузку потокового контента; роль контента, сгенерированного пользователем (как упоминалось выше), как новая форма креативности, например через сервисы, такие как YouTube; создание новых форм искусства и мероприятий, возможных благодаря возникновению новых (включая сетевые) технологий.
- *Здравоохранение*. Рост объема информации о здоровье, ведение онлайн электронных записей пациентов и связанные с этим проблемы приватности; увеличение роли телемедицины в поддержке удаленной диагностики и более продвинутые сервисы, такие как удаленная хирургия (включая совместную работу команд здравоохранения); увеличение влияния приложений сетевых и встроенных системных технологий в ассистировании жизни, например, мониторинг пожилых людей в их доме.
- *Образование*. Возникновение электронного обучения через, например, инструменты, базирующиеся на Web, такие как виртуальное обучающее окружение; ассоциированная поддержка удаленного обучения; поддержка совместной деятельности или обучение, основанное на создании сообщества.
- *Транспорт и логистика*. Использование технологий локализации, таких как GPS, в маршрутизации и более общем управлении трафиком; системы управления современными транспортными средствами; развитие сервисов карт, основанных на Web, таких как MapQuest, Google Maps и Яндекс-карты.
- *Наука*. Возникновение GRID как фундаментальной технологии поддержки современной науки, включая использование сложных се-

тей компьютеров для хранения, анализа и обработки научных данных и организации совместной работы между группами ученых.

- *Взаимодействие с окружающей средой.* Использование сенсорных технологий для мониторинга природной среды; организация ранних предупреждений о природных бедствиях, таких как землетрясения, затопления, цунами; анализ параметров глобального окружения для лучшего понимания сложных природных явлений, таких как изменение климата.

1.2. Параллельные и распределенные системы

Выше распределенная система охарактеризована как система, в которой вычисления и данные распределены, а автономные узлы взаимодействуют путем посылки сообщений. Следует отметить, что проблемы разработки приложений и решения возникающих при этом задач остаются актуальными и для других классов систем, в частности для параллельных систем, которые, как и распределенные системы, еще предстоит четко определить. Иногда говорят, что параллельная система — это система, в которой события могут быть частично или полностью упорядочены. В соответствии с этой точкой зрения каждая распределенная система есть подкласс параллельной системы, в котором процессы автономны и их адресные состояния не перекрываются. Говорят также, что параллельные системы работают в синхронном окружении и фокусируются на увеличении производительности, тогда как распределенные системы обычно функционируют в асинхронном окружении и фокусируются на допустимости отдельных отказов. При этом многие проблемы, например проблемы синхронизации в параллельных и распределенных системах, решаются сходным образом. Поэтому различные параллельные системы представляют большой интерес в контексте изучения распределенных систем, хотя и не обладают всеми особенностями, присущими распределенным системам. Вследствие этого анализ программно-аппаратного обеспечения распределенных систем может быть проведен с учетом структурно-функциональной систематизации компьютеров, а также известных таксономий параллельных мультипроцессорных/мультикомпьютерных систем.

Общая память и распределенная память

Основным параметром классификации мультипроцессорных вычислительных систем является способ организации памяти.

Параллельная система с общей памятью. В системах с общей памятью все процессоры «видят» единое адресное пространстве и имеют к нему равноправный доступ при помощи команд LOAD и STORE. Для обмена информацией один процессор записывает данные в общую память, а другой их считывает. Для реализации такой модели в реальной распределенной системе используются различные технические решения, обычно связанные с коммутацией шин.

Эта модель наиболее прозрачна для программистов, поскольку программы могут получать доступ к любому месту в памяти, не требуя никакой информации о схеме реализации (можно написать процедуру прямого доступа к памяти как обычный оператор языка). Прозрачность модели делает ее популярной, к тому же данная модель позволяет решать широкий круг задач.

Параллельная система с распределенной памятью. Каждый процессор имеет собственную, доступную только ему память, к которой может обращаться при помощи команд LOAD и STORE. Другие процессоры при этом не могут получить доступ к локальной памяти данного процессора. Таким образом, формируются отдельные физические адресные пространства для каждого процессора.

Не имея доступа к общей памяти, взаимодействующие процессы вынуждены прибегать к обмену сообщениями с помощью коммуникационной подсистемы. При этом используются примитивы обмена сообщениями SEND и RECEIVE.

Модель взаимодействия становится более сложной, и это накладывает свой отпечаток на разрабатываемое в рамках этой модели программное обеспечение.

Некоторые системы с распределенной памятью позволяют пользовательским программам обращаться к памяти других процессоров с помощью команд LOAD и STORE. Но эта возможность должна быть подкреплена аппаратно (например, как это сделано в архитектуре NUMA) или программно-распределенная разделяемая память (когда речь заходит о взаимодействии приложений, общую память принято называть разделяемой). При этом доступ к удаленной памяти выполняется медленнее, чем доступ к локальной памяти.

Возможность создания распределенных систем с разделяемой памятью базируется на хорошо известном факте: система с обменом сообщениями может быть смоделирована в рамках системы с разделяемой памятью, и наоборот, т. е. две парадигмы эквивалентны. Стремление сохранить привычную модель программирования стимулировало создание многих систем с разделяемой памятью. Однако не надо забывать, что в среде с распределенной памятью это всего лишь абстракция. Задержки при выполнении операций чтения и записи велики, так как базируются на сетевых коммуникациях.

Сильносвязанные и слабосвязанные системы. Другим параметром классификации может служить «сила связи», в соответствии с которым системы могут делиться на сильносвязанные и слабосвязанные. *Связанностью* называют степень знания и зависимости одного объекта от внутреннего содержания другого. Главное отличие — скорость обмена информацией. Обычно системы с разделяемой памятью связаны более сильно, чем системы с распределенной памятью, так как они могут обмениваться данными со скоростью копирования и перемещения байтов в памяти. Очевидно, что когда два процессора находятся рядом и обмениваются большими объемами данных с небольшими задержка-

ми, они связаны сильно. По мере удаления процессоров друг от друга связь ослабевает. Примерный спектр систем с разной силой связи иллюстрируют рис. 1.3, см. также [16].

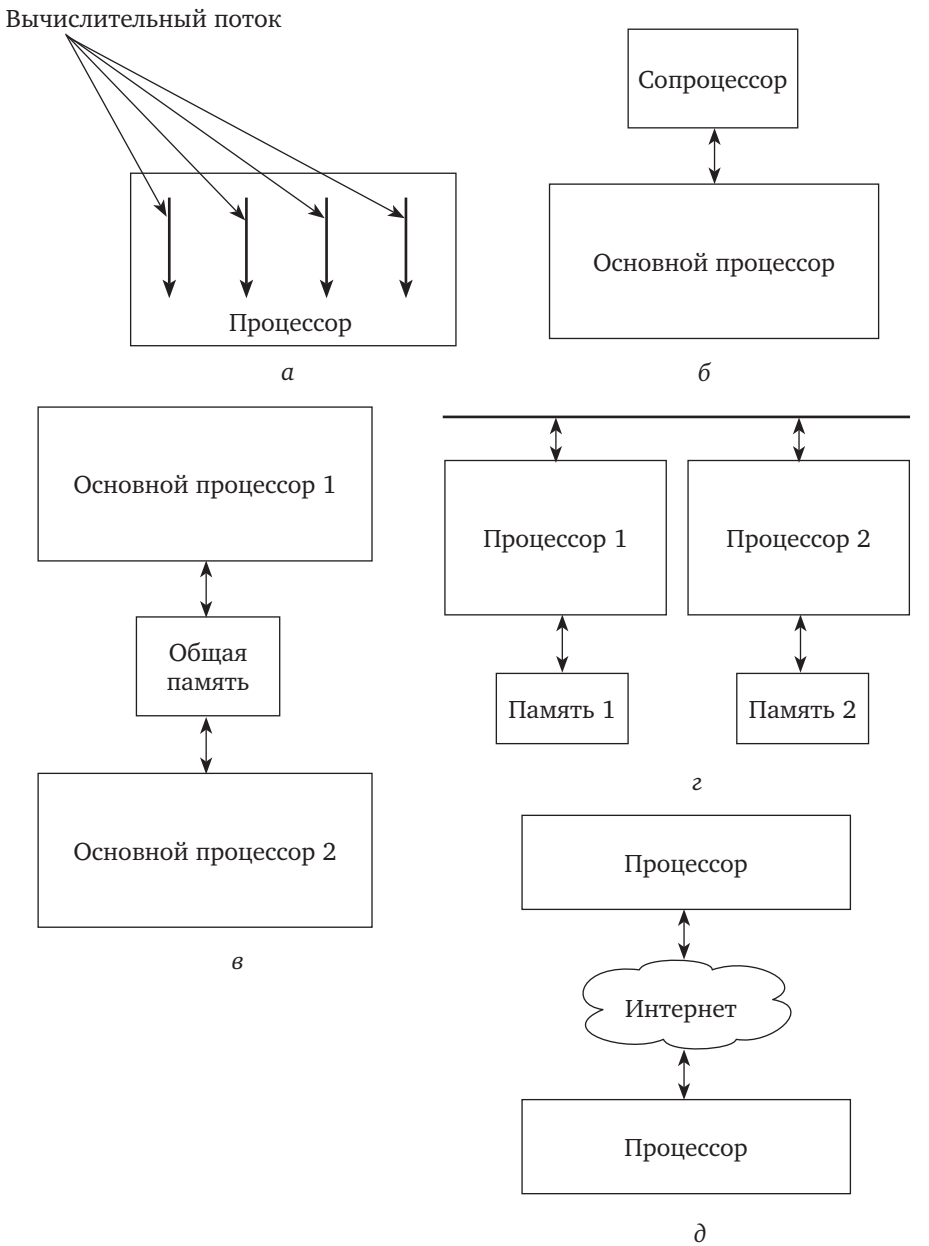

Рис. 1.3. **Внутрипроцессорный параллелизм (*а*); сопроцессор (*б*); мультипроцессор (*в*); мультикомпьютер (*г*); слабо связанная распределенная вычислительная система (*д*)**

Использование слабой связи представляет значительный интерес в связи с построением распределенных систем, так как это делает си-

стему более гибкой и толерантной по отношению к возможным сбоям и модификациям. В качестве примера можно привести технологию веб-сервисов, где принято минимизировать зависимости между сервисами для того, чтобы обеспечивать гибкость нижележащей архитектуры (снижающую риски влияния изменений в одном сервисе на другие сервисы). Далее в книге будут рассмотрены примеры распространенных технологий построения распределенных систем, базирующихся на отсутствии непосредственной связи компонентов системы во времени и пространстве.

Классификация параллельных систем. Таксономия Флинна

С момента появления компьютеров параллельного действия неоднократно предпринимались попытки их классификации (см., например, классификации Хокни, Фенга, Хендлера, Шнайдера, Скиликорна и др. [11]). Одна из наиболее популярных классификаций принадлежит Майклу Флинну, предложена она в 1966 г.

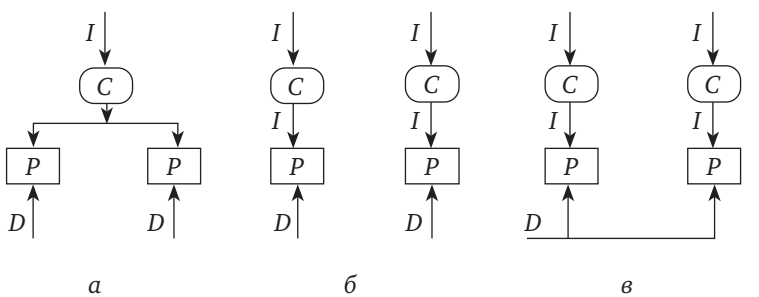

Рис. 1.4. **Классификация Флинна:** ***а*) SIMD; *б*) MIMD; *в*) MISD;**
C — устройство управления;
P — исполнительное устройство;
I — поток инструкций;
D — поток данных

Основой этой классификации служат понятия *поток команд* и *поток данных*. К системам SISD (*Single Instruction*, *Single Data*) относят ныне устаревшие компьютерные архитектуры, например Intel 80386. Распределенные системы классифицируются как MIMD, т. е. системы с наличием нескольких потоков данных и нескольких потоков команд. Впрочем, даже современные микропроцессорные архитектуры принято относить к MIMD-системам, так как большинство из них имеет несколько вычислительных ядер. Некоторые распределенные системы, особенно предназначенные для распределенных вычислений, могут подпадать под категорию SIMD, т. е. системы с одном потоком команд и несколькими потоками данных.

Отнесение конкретной системы к той или иной категории по классификации Флинна не является самоцелью, этой классификации не хватает необходимой для разработки распределенных систем детализации,

в частности, в рамках этой классификации нельзя определить степень *масштабируемости* системы, т. е. возможности наращивания производительности системы при увеличении числа ее компонентов.

Мультипроцессоры, мультикомпьютеры, кластеры

Параллельные вычислительные системы с общей памятью принято называть *мультипроцессорами*, а с распределенной памятью — *мультикомпьютерами*. Процессоры в мультипроцессоре взаимодействуют путем обращений к общей памяти, а в мультикомпьютере обмениваются сообщениями через коммуникационную сеть, что относит их к классу более сложных систем, чем мультипроцессоры.

Мультипроцессоры, в особенности мультипроцессоры с симметричной архитектурой, широко используются в настольных офисных системах, однако они плохо масштабируются. Поэтому высокопроизводительные вычисления выполняются на суперкомпьютерах, имеющих распределенную память. Более того, разделение суперкомпьютеров на комплексы с общей памятью и комплексы с распределенной памятью потеряло актуальность, поскольку объединить сотни тысяч ядер в систему с общей памятью технически невозможно.

Мультикомпьютеры весьма разнообразны по форме и размерам и плохо поддаются систематизации см., например, [12]. Можно выделить два основных варианта: массово-параллельные (MPP) системы и кластеры. К категории массово-параллельных систем относят дорогостоящие суперкомпьютеры, где компоненты тесно интегрированы за счет специализированного коммуникационного оборудования. Более популярны кластеры, основу которых составляют общедоступные вычислительные и коммуникационные компоненты. В отличие от массово-параллельных систем, кластеры можно сформировать, используя серийное оборудование — серийные сервера, установленные в стандартные стойки, серийное (возможно, дорогое) коммутационное оборудование. Массовость использованного оборудования позволяет достичь демократичной цены. Важным является и то, что отсутствие закрытых технологий позволяет использовать стандартные операционные системы со стандартными средствами разработки.

Кластеры можно отнести к распределенным системам, особенно так называемые децентрализованные кластеры, которые состоят из автономных вычислительных узлов, связанных локальной сетью.

Гетерогенные мультикомпьютерные системы. Распределенные системы

Большинство распределенных систем являются мультикомпьютерными гетерогенными и отличаются разнообразием аппаратных и программных компонентов. Процессы выполняются на машинах, связанных локальной или глобальной сетью, и взаимодействуют, обмениваясь сообщениями. Отдельные компьютеры, называемые *узлами*, могут функционировать в произвольные моменты времени, автоном-

ны, т. е. имеют собственное оборудование, свою операционную систему, и могут решать свои собственные задачи. Тем не менее они могут обмениваться информацией и взаимодействовать с общими ресурсами для решения общей проблемы и при необходимости координировать свою работу.

Иногда узлы работают корректно, иногда — отказывают и даже проявляют намеренно некорректное поведение (такое поведение принято называть *византийским*). В отличие от изолированной системы, распределенная система может работать правильно, несмотря на отказы отдельных узлов, во всяком случае, это одна из целей создания распределенных систем. Распределенное приложение не вправе рассчитывать на то, что ему в любой момент будут доступны ресурсы или службы какой-либо подсистемы.

Задача распределенной системы — превратить совокупность автономных гетерогенных узлов в основанную на общей концепции однородную систему. Таким образом, распределенная система — совокупность независимых компьютеров, которая представляется пользователю единой системой, ориентированной на решение задач, которые не могут быть выполнены каждым компьютером в отдельности.

Объединение узлов распределенных систем обеспечивается наличием промежуточного слоя, который маскирует гетерогенность нижележащих сетей, оборудования, операционных систем, языков программирования и создает виртуальную среду для выполнения распределенных приложений.

Иногда распределенные системы называют системой систем (зеркально отображая термин Интернет — сеть сетей). Такая система может быть определена как сложная система, состоящая из множества подсистем, имеющих свои права и объединенных для решения частных задач.

1.3. Архитектурные особенности

Архитектурные модели дают формальное описание системы в терминах вычислительных и коммуникационных задач, выполняемых вычислительными элементами; вычислительные элементы могут быть отдельными компьютерами или их совокупностями, поддерживаемыми соответствующими сетевыми взаимосвязями.

Особенности архитектурного построения распределенных систем обычно определяются набором служб и компонентов промежуточного слоя. Напомним, что промежуточное программное обеспечение объединяет независимые, зачастую слабосвязанные, узлы в единую систему, создавая среду для выполнения распределенных приложений. Прикладная программа осуществляет вызовы функций, определенных в библиотеках промежуточного уровня. Данную ситуацию можно отобразить в виде последовательности слоев (рис. 1.5), причем в реальных

системах промежуточный уровень может быть интегрирован с уровнем приложений или уровнем операционной системы.

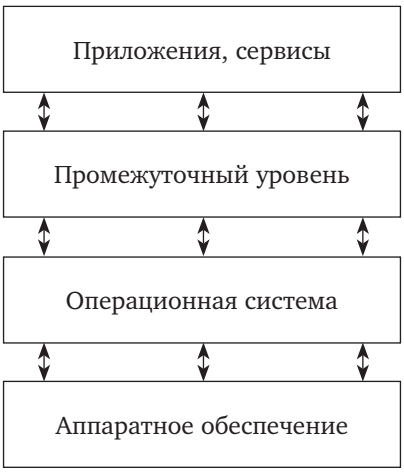

Рис. 1.5. **Программные и аппаратные слои в распределенных системах**

Для понимания строительных блоков распределенных систем нужно ответить на четыре ключевых вопроса.

1. Какие сущности взаимодействуют в распределенных системах?
2. Каким образом они взаимодействуют, какая коммуникационная парадигма при этом используется?
3. Каковы их роли и ответственность в архитектуре распределенных систем; как и когда они изменяются?
4. Как эти сущности отображаются на физическую распределенную архитектуру?

Напомним, что под *взаимодействующими сущностями* в распределенных системах обычно подразумевают процессы, но в некоторых примитивных системах коммуникационные сущности — отдельные узлы.

С точки зрения прикладных программ этого зачастую оказывается недостаточно, поэтому иногда в качестве элементов взаимодействия в распределенных системах рассматривают проблемно-ориентированные абстракции, например объекты, компоненты, веб-сервисы и др.

Взаимодействие процессов (межпроцессный обмен) может быть организовано путем непосредственного обмена сообщениями, например с помощью API, низкоуровневых интернет-протоколов. Широко применяются более универсальные коммуникационные механизмы, которые базируются на удаленных вызовах процедур и объектов. Асинхронные взаимосвязи обычно организованы с использованием техники *косвенного* (*indirect*) взаимодействия, например, с помощью очередей сообщений, распределенной разделяемой памяти или группового взаимодействия. Поэтому в состав программного обеспечения промежуточного уровня могут быть включены возможности для надежного и упорядоченного

обмена сообщениями, организации удаленного вызова процедур (*Remote Procedure Call*, RPC) и обращения к удаленным объектам (*Remote Method Invocation*, RMI), технологии *Message Passing Interface* (MPI), а также службы именования, распределенных транзакций, защиты и ряда других.

1.3.1. Сервисы, роли и архитектурные стили

В распределенных системах процессы (или объекты и компоненты) взаимодействуют друг с другом для выполнения некоторой полезной деятельности, например для поддержки чат-сессии. Каждый из процессов в таком взаимодействии играет определенную роль. Ниже описаны три архитектурных стиля: клиент-сервер, пиринговые системы и сервисно-ориентированная архитектура.

Сервис — термин, использующийся для обозначения отдельной части компьютерной системы, управляющей набором связанных ресурсов и представляющей их функциональность пользователям и приложениям. Например, доступ к разделяемому файлу осуществляется через файловый сервис, а покупка товаров производится через электронный платежный сервис. Доступ к сервису возможен только через экспортируемый набор операций. Например, для файлового сервиса такими операциями могли бы быть *read*, *write* и *delete*.

Тот факт, что сервис обеспечивает доступ к ресурсу путем предоставления строго определенного набора операций, есть часть стандартной программной инженерной практики. Но это также отражает и физическую организацию распределенной системы. Ресурсы в распределенной системе физически инкапсулированы и могут быть доступны другим узлам только посредством коммуникаций. Для эффективного разделения каждый ресурс должен управляться специальной программой, которая предоставляет интерфейс к ресурсу, делая ресурс доступным, а его модификацию безопасной и *консистентной*.

1.3.2. Клиент-сервер

В основе коллективного взаимодействия всегда лежит двухточечная связь. Технология связи двух узлов может служить структурным элементом масштабируемой архитектуры, а также решать частные задачи. Если связь выходит за рамки однократной посылки сообщения (например, при удаленной аутентификации или RPC), то принято обсуждать ее в понятиях клиентов, запрашивающих службы у серверов. *Сервер* — процесс, реализующий некоторую службу.

Клиент — процесс, запрашивающий службу у сервера путем посылки запроса, ожидания и получения ответа.

Термин «сервер», вероятно, хорошо знаком большинству читателей. Он относится к исполняемой программе (процессу) на сетевом компьютере, который принимает запросы от программ, работающих на других компьютерах, чтобы выполнять обслуживание этих запросов и отвечать подходящим образом. Запрашивающие процессы называются *клиентами*, а подход в целом называется *клиент-серверным взаимодействием*.

При данном подходе запросы и отклики посылаются в виде сообщений. Когда клиент посылает запрос на операцию, которую нужно выполнить, мы говорим, что клиент делает *вызов* (*invoke*) операции на сервере. Полное взаимодействие между клиентами и серверами с точки зрения клиента называется *удаленным вызовом* (*remote invocation*).

Процесс, аналогичный описанному, может происходить и на сервере, и на клиенте, так как сервер иногда вызывает операции на других серверах. Термины «клиент» и «сервер» применяются только к ролям, исполняемым в ходе отдельного запроса. Клиенты активны (делают запрос) и серверы пассивны (просыпаются при поступлении запроса); серверы работают непрерывно, тогда как клиенты — нет.

Заметим, что по умолчанию термины «клиент» и «сервер» относятся к процессу, а не к узлу, на котором они работают, но в повседневном общении эти термины относят также к узлам, компьютерам. Другое уточнение, которое мы обсудим в главе 3, связано с тем, что ресурсы могут быть инкапсулированы в виде серверных объектов, а доступ к ним происходит со стороны клиентских объектов. В этом случае мы говорим о том, что *клиентский объект* вызывает метод *серверного объекта*.

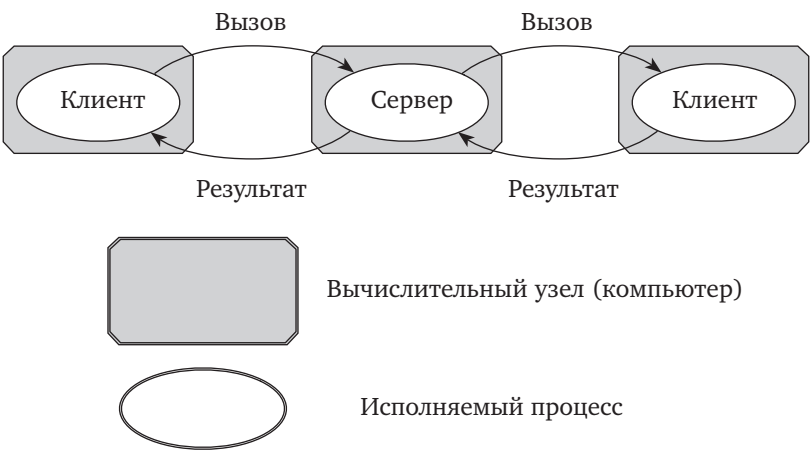

Рис. 1.6. **Клиенты вызывают отдельные серверы**

Хотя модель клиент-сервер является двухточечной и не учитывает потенциального параллелизма приложения, она весьма популярна, особенно в коммерческих приложениях.

Ее принципиальная ценность заключается в простоте реализации и в возможности декомпозиции большой задачи на ряд подзадач, которые могут быть выполнены параллельно и независимо друг от друга. Разбиение на подзадачи требует аккуратности и зависит от специфики приложения. Так, например, в системах управления базами данных выделяют три функциональных слоя:

- пользовательский интерфейс;
- слой обработки данных;
- слой хранения данных.

Эти слои могут быть разнесены по разным узлам, что порождает множество вариантов реализации. При этом могут возникать «тонкие» и «толстые» в зависимости от объема функционирующего ПО клиенты и серверы.

В тех случаях, когда серверы, в свою очередь, выступают в роли клиентов, возникают так называемые многозвенные архитектуры, где цепочка вовлеченных в обработку транзакции серверов располагается на разных узлах. Подобная организация распределенной обработки носит название *вертикального распределения*. Альтернативный способ организации, когда клиент и сервер содержат физически разделенные части одного модуля, носит название *горизонтального распределения*.

1.3.3. Одноранговые сети

Очень часто распределенные приложения могут функционировать без выделенного сервера. Для этого достаточно на каждом узле запустить несложное приложение, после чего они могут обмениваться сообщениями. В таких случаях говорят об *одноранговой* или *пиринговой* (*peer-to-peer*, P2P) архитектуре. При этом все узлы равны и играют симметричную роль в вычислениях, в отличие от модели клиент-сервер, где роли узлов существенно асимметричны, а серверы являются централизованными компонентами. Подобная сеть может легко самоорганизовываться и может иметь регулярную структуру, но может и не иметь таковой. Важной особенностью пиринговых сетей является их способность масштабироваться, т. е. создавать большие комбинированные хранилища данных и мощные вычислительные системы.

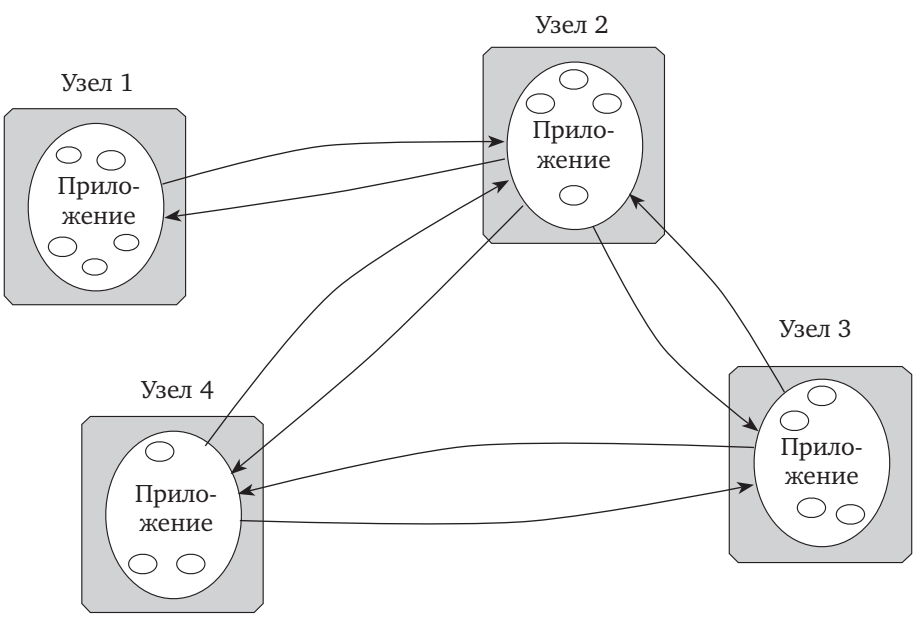

Рис. 1.7. **Пиринговая архитектура**
◯ — Разделяемые объекты

Примеры таких систем — eMule, BitTorrent, популярные файлообменные системы, позволяющие распределять информацию между большими группами компьютеров.

1.3.4. Сервисно-ориентированная архитектура

Модель клиент-сервер перестает быть адекватной, когда надо организовать доступ к глобальным ресурсам масштабируемой системы. В условиях слабой связи между узлами предпочтительным подходом к построению масштабируемой системы считается *сервисно-ориентированная архитектура*[1], когда компоненты системы реализованы в виде модульных сервисов (служб) с четко определенным интерфейсом, которые могут быть найдены и использованы клиентами.

Абстракция изолированного сервиса помогает установить правильные отношения между самой службой, ее средой и ее потребителями. Особенно важно выделять в отдельные службы операции чтения и записи и масштабировать каждую из них по отдельности. Для каждой службы могут существовать свои приемы оптимизации производительности.

На клиентской стороне устанавливается свой набор служб, которые позволяют, например, связаться с сервисами регистрации и найти сервис, который удовлетворяет потребностям клиента.

Взаимодействующие программные компоненты имеют минимальные знания друг о друге: они находят информацию, которая им нужна для взаимодействия, непосредственно перед взаимодействием, вызывая методы обнаружения сервисов, после чего обращаются к самим сервисам. Сервис может базироваться на любом сервере, а при необходимости — перемещен. Главное, что должно соблюдаться при его перемещении — это то, что запись о новом месторасположении должна соответствовать его фактическому новому местопребыванию. Пока ссылка на этот сервис есть в службе регистрации, предполагаемые клиенты будут в состоянии найти и использовать его.

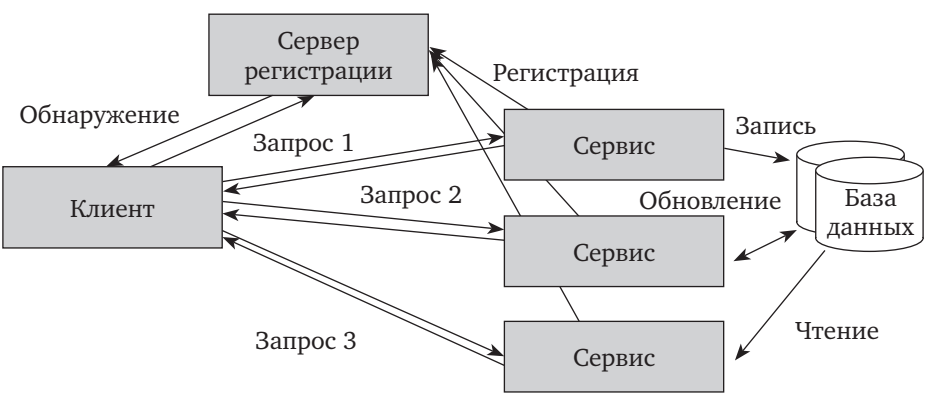

Рис. 1.8. **Многошаговое взаимодействие клиента и сервисов**

[1] *Демичев А. П., Ильин В. А., Крюков А. П.* Введение в грид-технологии // Препринт НИИЯФ МГУ, 11/832. М., 2007.

Сервисы могут быть реализованы как несколько серверных процессов на отдельных узлах, взаимодействующих с целью предоставления необходимого сервиса клиенту.

Сервисно-ориентированная архитектура — абстрактная концепция, которая может быть реализована с использованием различных технологий, включая распределенные объекты и компоненты, а также веб-сервисы. Ее главным достоинством является гибкость, возможность адаптации к изменяющимся условиям, что обусловлено наличием слабой связи между сервисами.

1.4. Дизайн масштабируемых распределенных систем

В настоящем разделе будет дан краткий перечень ключевых аспектов дизайна распределенных систем, которые более подробно будут рассмотрены позже. На заре развития распределенных систем наиболее важными проблемами считались:

- обеспечение надежного доступа к удаленным ресурсам;
- структура распределенной файловой системы.

Широкое распространение распределенных систем способствовало появлению новых приоритетных направлений в их проектировании. Заметно расширился класс распределенных алгоритмов. В сфере внимания научного сообщества находятся вопросы создания алгоритмов для масштабируемых распределенных систем. Бизнес-приложения при этом зачастую довольствуются более простыми моделями, в числе которых модель клиент-сервер, что определяется стандартами и потребностями индустрии.

Принято считать, что распределенные системы должны быть открытыми. *Открытость* компьютерных систем — особенность, позволяющая легко вносить в них различные улучшения и модификации, обеспечивать доступность новых сервисов разным клиентским программам. Для достижения открытости важно сделать спецификации ключевых интерфейсов компонентов системы доступными разработчикам программного обеспечения.

Сокрытие от пользователей и приложений деталей разбиения системы на компоненты и распределения ресурсов среди множества компьютеров известно как свойство *прозрачности*. При этом система воспринимается как целое, а не как набор независимых компонентов. Концепция прозрачности применима к различным аспектам дизайна распределенных систем. Например, прозрачность доступа позволяет скрыть разницу в представлении данных и способах доступа к ресурсам, а прозрачность репликации скрывает факт наличия нескольких копий ресурса. Основополагающие документы ISO регламентируют и другие аспекты прозрачности: прозрачность локализации за счет надлежащего именования ресурсов, прозрачность отказов, мобильности и т. д.

Распределенные системы являются также и *параллельными*. Как сервисы, так и приложения могут разделять ресурсы. Любой объект, который инкапсулирует разделяемый ресурс в распределенной системе, должен гарантировать корректную работу с ним в параллельном окружении. Все данные должны оставаться консистентными, причем применение стандартной техники, например семафоров, для синхронизации одновременного доступа к данным нескольких процессов здесь невозможно.

Отказы отдельных узлов и связей становятся естественной частью современных распределенных систем, что обусловлено масштабами сетей и хранимых данных. Но при этом отказы носят частичный характер, т. е. в то время, как некоторые компоненты отказывают, другие продолжают работать. Поэтому обработка отказов очень трудна. Обсуждаемая на протяжении всей книги техника работы с отказами включает в себя моделирование, обнаружение, маскировку отказов, восстановление после отказов и многое другое.

1.4.1. Масштабируемость

Распределенные системы проектируются, чтобы объединять большое число компьютеров. Построение крупномасштабных систем нуждается в правильной методологии, поэтому придание распределенной системе свойства масштабируемости есть одна из наиглавнейших задач при ее проектировании. Масштабируемая система должна гибко адаптироваться к добавлению ресурсов. Коммуникационная подсистема должна справляться с увеличением числа узлов системы, а производительность распределенной системы должна увеличиваться пропорционально их количеству. Масштабирование по размеру предполагает легкость подключения дополнительных пользователей и ресурсов. Масштабируемая система должна быть управляемой, давать возможность обновления и модификации.

Различают два способа масштабирования: вертикальное и горизонтальное. *Вертикальное масштабирование* — увеличение производительности каждого компонента системы с целью повышения общей производительности. Например, для больших наборов данных это означает добавление большого числа жестких дисков к файловым серверам системы. Это самый простой способ масштабирования, здесь не требуется никаких изменений в прикладных программах, однако система быстро достигает своего предела при наращивании мощности узлов.

Горизонтальное масштабирование подразумевает рост производительности приложения за счет увеличения числа узлов. Этот способ масштабирования может требовать внесения изменений в программы, разбиения служб на сегменты и модули, чтобы программы смогли воспользоваться дополнительным количеством ресурсов. Дизайн может потребовать распределения и репликации данных, а также наличия служб индексирования и именования для связи имен ресурсов и их ло-

кализации. Таким образом, горизонтальное масштабирование должно быть внутренним принципом разработки системы.

Масштабируемая система должна быть *децентрализованной*: узлы принимают решения на основе локальной информации и не обладают полной информацией о состоянии системы. Централизованные компоненты, такие как серверы, службы, алгоритмы, легко могут стать узким местом системы.

Другим важным решением, особенно при наличии географически удаленных узлов и ненадежных соединений, является использование асинхронной связи, когда клиент, не будучи блокируемым в ожидании ответа, может заняться альтернативной работой. Кроме того, крупные серверы, например веб-серверы, обычно имеют предел количества одновременных соединений, которые они в состоянии обслужить, и при высоком трафике этот предел может быть превышен. Асинхронность операций, особенно операций чтения, позволяет серверу переключаться между операциями быстрее и обсуживать большее число запросов.

Следующая важная технология, используемая для повышения производительности — *репликация* (копирование) компонентов распределенной системы, перенесение копий данных ближе к центрам их обработки. В сильно рассредоточенных системах наличие близко лежащей копии практически всегда ведет к повышению производительности. Недостаток наличия нескольких копий состоит в том, что изменения в одной из копий должно быть распространено на остальные (проблема непротиворечивости реплик, *data consistency*). Тем не менее реплицирование широко распространено. Частным случаем репликации является *кэширование*, когда реплика создается по инициативе клиента — потребителя реплицируемого ресурса.

Репликация не только повышает доступность, но и помогает выровнять загрузку компонентов, перемещая данные из того места, где они требуются реже, в те места, где они требуются чаще. Репликация также позволяет повысить надежность системы и справляться как со сбоями отдельных узлов, так и с потерей целого кластера. Возможность параллельного чтения реплицированных данных позволяет масштабировать операции чтения, а при определенных условиях — и операции записи.

Одним из технических приемов для повышения производительности распределенной системы является сегментирование данных или *шардинг* — подход, при котором каждый узел системы содержит свою часть данных и выполняет операции над ними. Здесь необходимо правильно разделить компоненты на части и распределить их по системе. Этот метод является распространенным средством обеспечения горизонтальной масштабируемости, однако теперь операции над несколькими объектами могут вовлечь в работу несколько узлов, что требует активной передачи данных по сети. Кроме того, при увеличении числа узлов, хранящих данные, увеличивается вероятность сбоев, что требует наличия избыточности — применения репликации.

Применение шардинга приводит к появлению большой группы специфических задач, связанных с его обеспечением: балансировка нагрузки, распределение данных и ролей, оптимизация сетевого трафика.

Шардинг не гарантирует нахождение искомого блока на выбранном узле, поэтому операции чтения обращаются к некоторому выделенному узлу, а тот уже запрашивает данные или перенаправляет запрос по нужному адресу. Это замедляет чтение. Но зато нет необходимости синхронизовать данные между узлами. Кроме того, части/блоки данных могут параллельно записываться в различные узлы. Это кардинально ускоряет запись.

1.4.2. Особенности проектирования распределенных систем

При проектировании распределенных систем также нужно обратить внимание на следующие функциональные особенности ее построения.

Коммуникационные механизмы. Задача включает в себя проектирование подходящего механизма для коммуникаций между процессами в сети. Это могут быть: удаленный вызов процедуры, обращение к удаленным объектам, синхронный и асинхронный обмен сообщениями, использование распределенной разделяемой памяти и др. Строительными блоками для коммуникаций являются распределенные алгоритмы на графах и алгоритмы маршрутизации. Специфичные алгоритмы нужны для организации групповых коммуникаций и для управления группами процессов и узлов. Когда процессы посылают сообщения группе процессов, важно выдерживать определенный порядок доставки сообщений, чтобы не нарушать семантику распределенной программы.

Синхронизация и координация. Помимо классического приема — взаимоисключения, возможно применение неблокирующих и свободных от ожидания алгоритмов, например для управления параллельным выполнением распределенных транзакций. Для упорядочивания событий и межпроцессных зависимостей важны различные схемы поддержки логического времени. Необходимо организовать наблюдение за глобальным состоянием, в частности для организации контрольных точек, обнаружения завершения и тупиков и фиксации распределенных транзакций. Во многих случаях (инициализация, широковещание, ответственность за вход в критическую секцию, регенерация маркера и др.) один из процессов играет выделенную роль. Такой процесс называется *лидером* или *координатором*, поэтому механизмы выбора лидера могут быть составной частью проектирования распределенных систем.

Репликация и согласованность (непротиворечивость) памяти. Репликация, управление репликами и совместимость реплик крайне важны для быстрого доступа и обработки данных в масштабируемых системах. Репликация создает некоторую избыточность, которая позволяет справляться с частичными отказами и обеспечивает резервную функциональность. Репликация также используется в масштабируемых архитектурах, в которых с целью оптимизации доступа к данным не предусмотрено разделения ресурсов (*share nothing*). Важно помнить,

что для управления обновляемыми репликами и кэшами необходимо постоянно решать проблему их согласованности (консистентности).

Отказоустойчивость. Построение отказоустойчивых систем в условиях системных отказов отдельных узлов и коммуникационных каналов требует поддержки надежного обмена сообщениями, репликации и избыточности служб и данных. Реализация алгоритмов согласия позволяет корректно функционирующим процессам достичь консенсуса, несмотря на существование отказавших или византийских процессов. В случае глобального отказа состояние системы может быть восстановлено путем отката к последней фиксированной контрольной точке.

Безопасность. Защита распределенных систем предполагает активное использование криптографии для аутентификации пользователей и данных, организации защищенных каналов, контроля доступа и управления ключами. Новые архитектуры в распределенных системах, такие как беспроводные соединения, пиринговые сети, GRID и тотальный компьютинг, создают дополнительные сложности для системы безопасности.

Именование и прозрачность. Организация прозрачного доступа к ресурсам, сокрытие факта распределенности помогают придать системе единообразный когерентный вид. Важную роль играет именование ресурсов. Внедрение надлежащей совокупности имен, идентификаторов и адресов позволяет локализовать ресурсы, в том числе и мобильные, прозрачным и масштабируемым образом.

Производительность. Высокая пропускная способность, минимальное время отклика, быстрая и эффективная обработка данных очень желательны для распределенных систем. За счет баланса между вычислениями и коммуникациями, распределения вычислений и репликации данных можно добиться соотношения цена-производительность лучшего, чем в многопроцессорных комплексах. Масштабирование распределенных систем дает возможность гибкого наращивания мощности.

Каждая из перечисленных технологий может служить основой проектных решений при создании распределенных систем, но они также могут находиться в противоречии друг с другом; при достижении одних целей приходится жертвовать другими.

Глава 2
МОДЕЛИ

2.1. Введение в моделирование и понятие модели

Исследование распределенных систем обычно происходит с использованием различных моделей обработки информации. Построение модели позволяет сформулировать набор правил взаимодействия элементов распределенной системы и дает возможность проведения ее последующего формального анализа.

Распределенная система состоит из набора процессов, объединенных коммуникационной подсистемой, обеспечивающей возможность информационного обмена между процессами. Коммуникационные задержки конечны, но их величина непредсказуема. У исполняющихся процессов отсутствует понятие общей памяти, и они общаются посредством сообщений через сеть. Нет глобальных физических часов, к которым все процессы имели бы постоянный доступ. Сообщения, посланные процессами друг другу через коммуникационную систему, могут быть доставлены не в порядке их отправления. Более того, сообщения могут теряться в процессе пересылки, дублироваться, а сами процессы и каналы связи могут исчезать из системы и появляться в ней в произвольные моменты времени. Коммуникации уязвимы для атак безопасности.

Система может быть смоделирована как граф, в котором вершины представляют процессы, ребра представляют коммуникационные каналы. Распределенное приложение выполняется как набор процессов в распределенной системе. Каждое событие в распределенной системе (и это является отличительной чертой таких систем) меняет состояние не всей системы, а всего лишь некоторого подмножества взаимодействующих процессов.

Возникает необходимость в координации процессов (синхронизации и упорядочивании операций). Модель взаимодействия должна отражать тот факт, что коммуникации происходят с задержками, иногда длительными. Точность, с которой независимые процессы могут быть скоординированы, ограничена этими задержками, а также сложностью поддержки единого представления о времени для всех компьютеров в распределенных системах.

Систематический анализ различных моделей параллельной распределенной обработки информации имеется в серии работ ([19], [11]).

Функционирование распределенной системы не детерминировано и может иметь разные истории выполнения для одних и тех же входных данных. Базовой моделью взаимодействия в подобных системах служит асинхронный обмен сообщениями (синхронный обмен трактуется как частный случай асинхронного). Поскольку сообщение нельзя получить прежде, чем оно было отправлено, между событиями посылки и получения возникают отношения предшествования, которые позволяют упорядочить все события в системе. Важную роль играет модель взаимодействия через распределенную разделяемую память. Эта модель также используется для анализа консистентности реплик и кэшей, которые могут быть организованы в распределенной системе с целью ее масштабирования.

Отказы и сбои узлов и коммуникационных каналов могут нарушить корректность работы распределенной системы. Существует классификация возможных отказов, которая описывается собственной моделью. Это создает базу для анализа потенциальных последствий эффектов отказов и для проектирования систем, которые в состоянии корректно функционировать, несмотря на наличие сбоев.

Вследствие модульной природы распределенных систем и их открытости возникает опасность атак системы со стороны внешних и внутренних агентов. Модель безопасности определяет и классифицирует формы таких возможных атак и дает основу для анализа угроз системе и для проектирования систем, способных противостоять атакам.

2.2. Модель распределенного исполнения

2.2.1. Общее описание

В рамках модели состояние системы меняется дискретными шагами, которые называются событиями или переходами. Каждый отдельный вычислительный процесс представляет собой последовательность программных операторов. Фрагменты программы (отдельные операторы, линейные участки кода, подпрограммы и т. д.) могут быть названы событиями (внутренними событиями). Можно считать, что событие мгновенно меняет состояние системы. В промежутке между двумя событиями система остается неизменной.

Глобальное состояние распределенного вычисления состоит из состояний процессов и коммуникационных каналов. Состояние процесса характеризуется состоянием его локальной памяти и зависит от контекста. Состояние канала характеризуется набором сообщений, передающихся по каналу (т. е. еще не достигших адресата).

2.2.2. Модель коммуникационного канала

Говоря о сообщении, передающемся по каналу, нужно сделать уточнение. Как правило, это ситуации временного пребывания сообщения

в буфере передающего или принимающего узла, когда оно еще не принято процессом-получателем. Поэтому формально такое сообщение приписывается связующему каналу. Как мы увидим далее, в распределенных системах принято различать «получение сообщения», т. е. его прибытие на получающий узел, и «доставку сообщения» — передачу сообщения получающему процессу.

2.2.3. Событийное описание

Исполнение процесса состоит из последовательного исполнения его действий. Действия атомарны и моделируются тремя типами событий, а именно:

- внутренние события;
- события посылки сообщения;
- события приема сообщения.

Предполагается, что для каждого сообщения имеются единственные процесс-отправитель и процесс-получатель.

Внутреннее событие меняет состояние процесса, в котором оно случается. Посылка (прием) сообщения меняет состояние процесса, посылающего (принимающего) сообщение, и состояние канала, через который оно проходит. Внутреннее событие влияет только на сам процесс.

Обозначим j-е событие в i-м процессе как e_i^j.

События в процессе линейно упорядочены. Исполнение процесса представляет собой последовательность событий: $e_i^1, e_i^2, \ldots, e_i^n$.

События посылки и приема означают поток информации между процессами и устанавливают потенциальную причинную зависимость между посылающим и принимающим процессами.

Отношение $\rightarrow_{msg}$, которое фиксирует причинную зависимость вследствие обмена сообщениями, определяется следующим образом: для каждого сообщения m, которым обмениваются два процесса, имеем

$$send(m) \rightarrow_{msg} receive(m).$$

Отношение $\rightarrow_{msg}$ определяет отношение между парой соответствующих событий *send* и *receive*.

2.2.4. Упорядочивание событий

Во многих случаях нам интересно знать, произошло ли событие (посылки или получения сообщений) в одном процессе до, после или конкурентно[1] (*concurrent*) с другим событием другого процесса. Исполнение системы может быть описано в терминах событий и их упорядочивания, несмотря на отсутствие глобальных часов, и визуализировано пространственно-временной диаграммой (см. пример на рис. 2.1)[8].

[1] Заметим, что слово «одновременно» здесь не подходит, так как понятие «одновременности» для распределенных систем не определено.

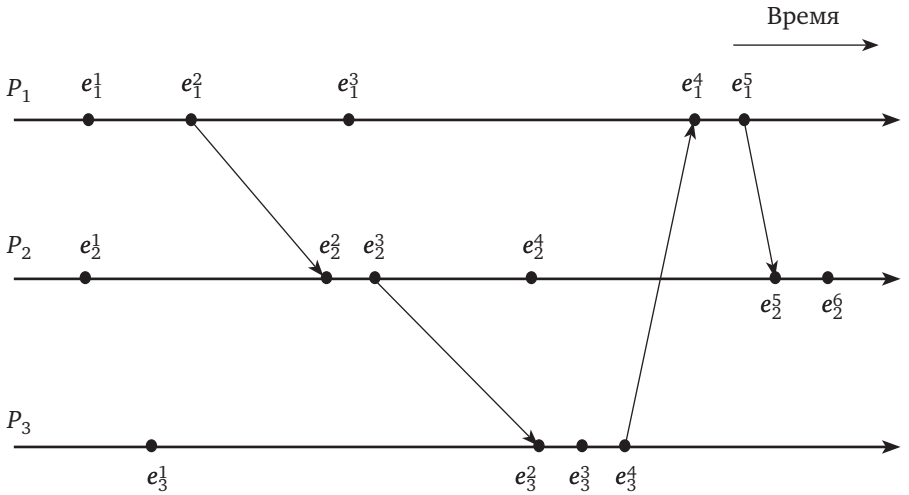

Рис. 2.1. **Пространственно-временная диаграмма распределенного исполнения**

Каждая горизонтальная линия (см. рис. 2.1) соответствует одному процессу, точки показывают события. Наклонные стрелки иллюстрируют передачу сообщения между двумя событиями. Первое из них — событие посылки сообщения, а второе — событие приема того же самого сообщения. Такие события считаются взаимосвязанными. В общем случае исполнение события занимает конечное время; тем не менее предполагается, что события выполняются атомарно и обозначаются точкой на линии процесса. На данном рисунке для процесса P_1 второе событие — событие посылки сообщения, третье — внутреннее, а четвертое — прием сообщения.

Как будет показано ниже, в ряде случаев порядок следования событий можно изменить так, что это никак не отразится на последующих глобальных конфигурациях.

События, которые могут быть следствием предшествующих событий, считаются потенциально зависящими от них.

По определению зависимыми считаются события, происходящие в одном процессе или взаимосвязанные события отправки и приема сообщения. В силу транзитивности зависимыми можно считать события, которые можно соединить ломаной из горизонтальных и наклонных отрезков на пространственно-временной диаграмме. Как уже отмечалось ранее, события в отдельных процессах оказывают влияние только на часть глобальной конфигурации.

Такие события не пересекаются друг с другом, являются независимыми и могут выполняться в другом порядке. С точки зрения постороннего наблюдателя при этом возникают разные глобальные конфигурации, но поведение каждого отдельного процесса при этом не меняется.

Проиллюстрируем этот тезис следующим рассуждением [18].

Рассмотрим два варианта распределенного исполнения представленных на диаграммах (рис. 2.2 и 2.3).

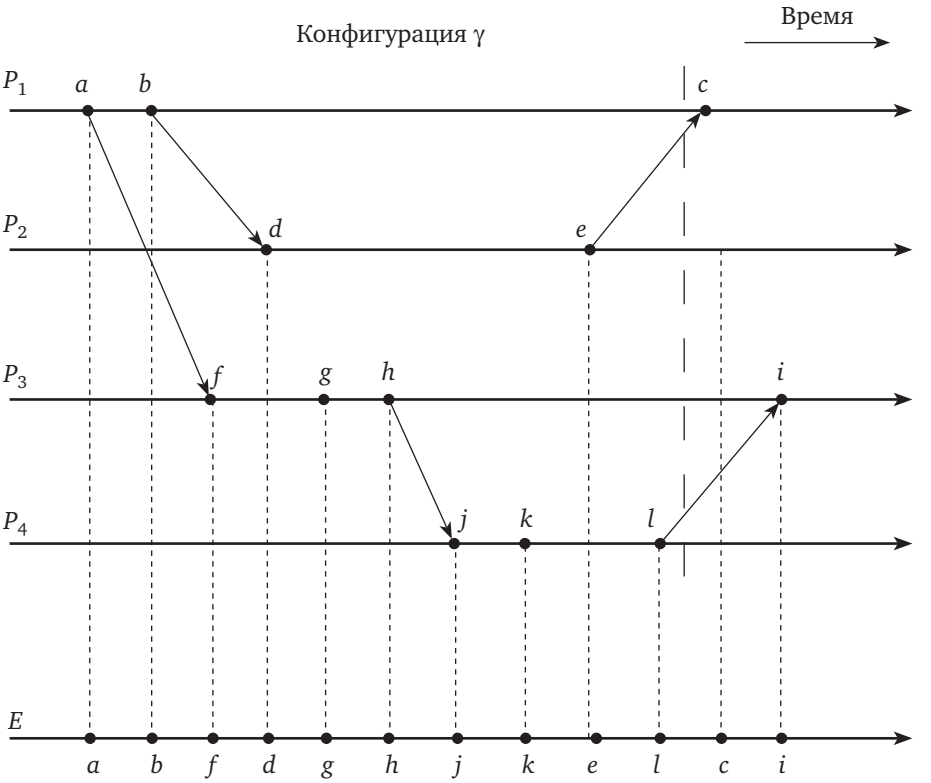

Рис. 2.2. **Пример пространственно-временной диаграммы**

На первый взгляд, кажется, что эти исполнения неэквивалентны и содержат разные совокупности конфигураций. Выполнение, диаграмма которого изображена на рис. 2.2, содержит конфигурацию γ, в которой оба сообщения, отправленные при осуществлении событий *e* и *l*, пребывают на этапе пересылки одновременно. А вот выполнение, диаграмма которого изображена на рис. 2.3, не содержит ни одной такой конфигурации, поскольку сообщение, отправленное при осуществлении события *l*, достигает адресата раньше, чем происходит событие *e*.

Сторонний наблюдатель, который имеет возможность видеть подлинную последовательность событий, способен отличить одно эквивалентное исполнение от другого, т. е. он всякий раз видит только одно из возможных исполнений. Однако процессы не могут отличить эти два исполнения, так как с точки зрения процессов невозможно понять, какое из двух исполнений происходит на самом деле. Таким образом, эти два исполнения можно считать эквивалентными, что строго доказывается в книге [18].

Итак, если исполнение e_1 получено из исполнения *e* путем перестановки событий, сохраняющей причинно-следственный порядок событий, то эти исполнения эквивалентны. Эквивалентные временные диаграммы можно получить из заданной диаграммы путем ее растяжения и сжатия, как резиновой ленты.

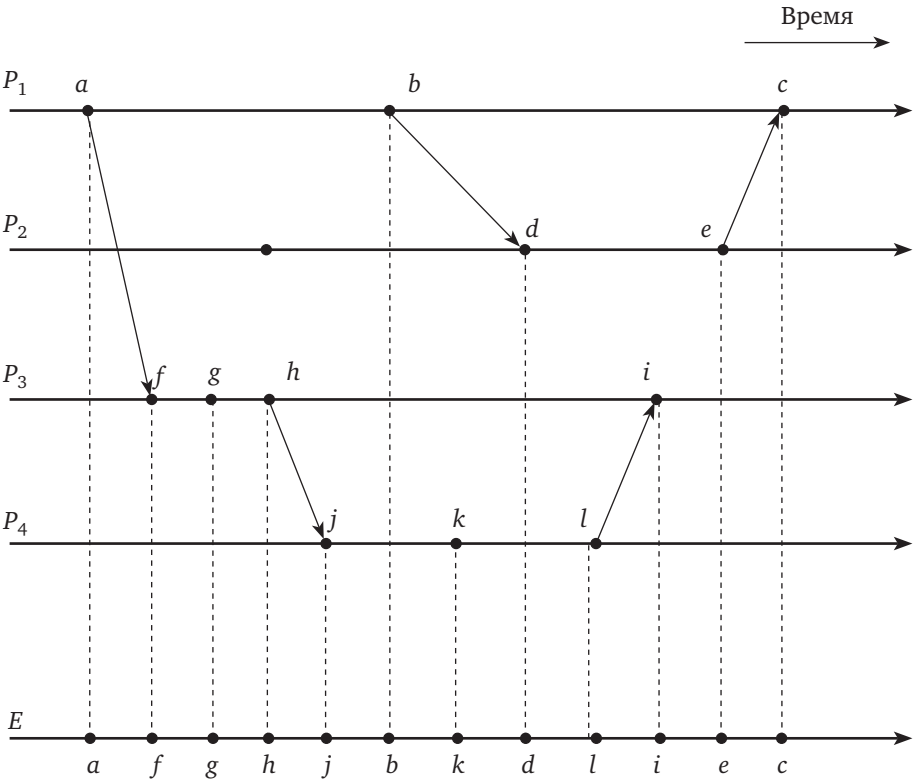

Рис. 2.3. **Пространственно-временна́я диаграмма, эквивалентная диаграмме на рис. 2.2**

2.3. Отношение причинного предшествования

Очевидно, что зависимые события менять местами нельзя. Например, нельзя поменять на временной оси события отправки и приема одного и того же сообщения. Наличие или отсутствие причинной зависимости между отдельными событиями позволяет ввести на множестве событий причинно-следственное бинарное отношение «$\rightarrow$» частичного порядка:

- если e_i^x и e_i^y — события, происходящие в одном и том же процессе, и e_i^x происходит раньше, чем e_i^y, то $e_i^x \rightarrow e_i^y$;
- если e_i^x — это событие отсылки сообщения одним процессом, а e_j^y — взаимосвязанное с ним событие получения того же сообщения другим процессом, то $e_i^x \rightarrow e_j^y$;

Напомним, что бинарное отношение $\preceq$ на множестве называется *отношением частичного порядка*, если оно удовлетворяет свойствам:

1) **рефлексивности** $x \preceq x$ для всех $x \in A$;
2) **антисимметричности** $(x \preceq y) \wedge (y \preceq x) \Rightarrow x = y$ для всех $x, y \in A$;
3) **транзитивности** $(x \preceq y) \wedge (y \preceq z) \Rightarrow x \preceq z$ для всех $x, y, z \in A$.

Отношение «$\rightarrow$» — отношение «*происходит раньше* (*happens before*)». Оно впервые введено в работе Лесли Лампорта[1]. Свойство транзитивности отношения «происходит раньше» показывается фактом, что из $e_i^x \rightarrow e_j^y$ и $e_j^y \rightarrow e_k^z$ следует $e_i^x \rightarrow e_k^z$.

Если e_j^y непосредственно или транзитивно зависит от e_i^x, это означает, что существует путь, состоящий из горизонтальных линий и стрелок на пространственно-временной диаграмме, который начинается событием e_i^x и кончается событием e_j^y (так называемая *причинно-следственная цепочка*). Следует отметить, что причинно-следственная связь $e_i^x \rightarrow e_j^y$ событий e_i^x и e_j^y носит потенциальный характер и означает лишь то, что вся информация, доступная в момент события e_i^x, потенциально доступна в момент события e_j^y.

Поскольку отношение «происходит раньше» накладывает на множество событий частичный порядок, существуют пары событий e_i^x и e_j^y, для которых не выполняются отношения $e_i^x \rightarrow e_j^y$ и $e_j^y \rightarrow e_i^x$. Такие события называются *соисполнимыми* (*concurrent*). В литературе такие события часто называют параллельными, но этот термин в русском языке имеет настолько много различных смыслов, что в применении его к распределенным системам мы его будем избегать. Соответствующее отношение обозначается как $e_i^x \parallel e_j^y$. Например, на рис. 2.1 $e_1^1 \rightarrow e_3^3$ и $e_3^3 \rightarrow e_5^2$, а $e_3^1 \parallel e_3^3$.

Соисполнимость событий, связанных отношением «$\parallel$» трактуется как логическая соисполнимость и не означает, что эти события случаются в один и тот же момент физического времени. Совпадение или несовпадение отметок физического времени для логически соисполнимых событий не должно сказываться на результатах вычислений.

Модель причинной упорядочиваемости существенно упрощает дизайн распределенных алгоритмов, так как обуславливает внутреннюю синхронизацию событий. Отслеживание причинно-зависимых событий позволяет, например, спроектировать корректные алгоритмы обнаружения тупиков и генерации контрольных точек для восстановления, поддержать согласованность реплицированных баз данных и т. д.

2.4. Логическое время. Отметки времени Лампорта

Один из отличительных признаков распределенных систем — отсутствие глобальных физических часов. В централизованной системе события можно упорядочить в зависимости от времени их осуществления. В распределенных системах вследствие того, что таймеры на каждом из входящих в систему компьютеров работают с несколько отличающейся частотой, возникает рассинхронизация физических часов. Поэтому представление о физическом времени как о способе упо-

[1] Lamport L., Time, Clocks, and the Ordering of Events in a Distributed System, Commun. ACM, 21(7):558-565 July 1978.

рядочивания событий не совсем подходит для распределенных систем. Наличие в системе причинно-следственных зависимостей позволяет ввести *логическое время*, аппроксимирующее физические часы, и осуществить таким образом альтернативный способ упорядочивания событий. Для широкого класса алгоритмов непротиворечивость последовательности событий важнее близости времени их осуществления к реальному времени. В таких случаях говорят о логическом времени. Концепция логического времени будет изложена далее, следуя соответствующему разделу монографии [8].

Компьютерные часы и события времени. Каждый компьютер в распределенной системе имеет свои внутренние часы, которые используются локальными процессами для получения текущего времени. Локальные часы разных компьютеров могут поставлять различные временные значения. Это происходит потому, что часы компьютера дрейфуют относительно точного времени, и, что более важно, скорость дрейфа различна. Термин *скорость дрейфа* часов относится к скорости, с которой часы компьютера отклоняются от точного значения часов. Даже если изначально установить часы всех компьютеров в распределенной системе в одно и то же время, их показания могут существенно разойтись за срок до следующей коррекции.

Существование логического времени предполагает, что каждому событию e назначается *временна́я* метка $C(e)$, которая может обладать следующими свойствами.

- **Условие консистентности.** Если событие e_j причинно зависимо от события e_i, временна́я метка события e_i меньше чем метка события e_j:

$$e_i \to e_j \Rightarrow C(e_i) < C(e_j) \text{ для всех } e_i, e_j \in E.$$

- Условие строгой консистентности.

$$e_i \to e_j \Leftrightarrow C(e_i) < C(e_j) \text{ для всех } e_i, e_j \in E.$$

2.4.1. Реализация логических часов

Реализация требует от каждого процесса поддержки двух структур данных:

- локальных логических часов, lc_i;
- глобальных логических часов gc_i, которые отражают локальное представление процесса P_i о логическом глобальном времени.

Протокол поддержки логических часов гарантирует, что логические часы процесса и его представление о глобальном времени управляются согласованно. Протокол состоит из следующих двух правил.

1. ***R1.*** Это правило изменения процессом своих логических часов при наступлении различных событий (внутреннее событие, посылка или получение сообщения).

2. ***R2.*** Это правило изменения процессом своих глобальных логических часов. Оно предполагает, что информация о логическом времени каждого процесса прикладывается к сообщению (техника *piggybacking*) и используется получающим процессом для изменения его представления о глобальном времени.

2.4.2. Скалярное время

Лампорт[1] предложил использовать в качестве отметок времени событий процесса P_i целые неотрицательные числа C_i, которые одновременно являются и представлением процесса о глобальном времени. В этом случае правила ***R1*** и ***R2*** формулируются следующим образом:

- ***R1.*** Перед исполнением события (посылка, прием или внутреннее) процесс P_i делает следующее:

$$C_i \leftarrow Ci + d \quad (d > 0).$$

Величина d обычно равна 1;

- ***R2.*** К каждому сообщению прилагается отметка времени отправителя в момент отправки. Когда процесс P_i получает сообщение с отметкой времени C_{msg}, он выполняет следующие действия:

1) в качестве C_i выбирает $C_i \leftarrow \max(C_i, C_{msg})$;
2) выполняет правило ***R1***;
3) осуществляет доставку сообщения.

В случае $d = 1$ временная метка события e_i представляет собой длину самой протяженной последовательности событий, удовлетворяющей условию: $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_i$. Пример эволюции скалярного времени показан на рис. 2.4.

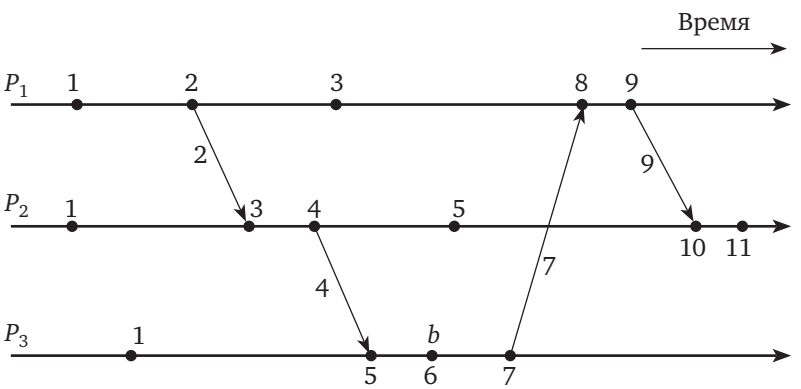

Рис. 2.4. **Эволюция скалярного времени**

Очевидно, что таким образом введенные часы обладают свойством консистентности, т. е. для двух событий e_i и e_j, $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

---

[1] Lamport L., Time, Clocks, and the Ordering of Events in a Distributed System, *Commun. ACM*, 21(7):558-565, July 1978.

Иногда желательно, чтобы никакие два события не имели одинаковых отметок времени. Чтобы выполнить это требование, можно, например, учесть номер процесса, в котором происходят событие. В этом случае при совпадении отметок времени для упорядочивания можно провести дополнительное сравнение номеров процессов.

Организованные таким образом скалярные часы позволяют выполнить тотальное упорядочивание событий в системе и играют роль полезного вспомогательного механизма, встроенного в распределенный алгоритм для контроля очередности осуществления событий.

Тотальный порядок полезен, если необходимо обеспечить свойство «живучести» (см. параграф 2.7), поскольку позволяет организовать обслуживание запросов в соответствии с их временны́ми метками.

2.4.3. Векторное время

К сожалению, показания скалярных не позволяют ничего сказать о взаимосвязи событий путем сравнения их логических временны́х меток. Иными словами скалярные часы не обладают свойством строгой консистентности; т. е. для двух событий e_i и e_j, из того, что $C(e_i) < C(e_j)$ не следует, что $e_i \to e_j$. Например, на рис. 2.4 третье событие процесса P_1 имеет меньшую отметку времени, чем третье событие процесса P_2. Тем не менее они не подчиняются отношению «происходит раньше». Причина в том, что локальные и глобальные логические часы втиснуты в одно понятие, в результате возникает потеря информации о причинной зависимости среди событий разных процессов. Например, на рис. 2.4, когда процесс P_2 получает первое сообщение от процесса P_1, он изменяет свои часы на 3, забывая, что отметка времени последнего события в P_1, от которого он зависит, есть 2.

Причинно-следственная связь, которую не в состоянии уловить скалярные часы, может быть соблюдена при использовании *векторных отметок времени*. В системе векторных часов каждый процесс P_i поддерживает вектор $vt_i[1\ldots n]$, где $vt_i[i]$ есть локальные логические часы процесса P_i. Значение $vt_i[j]$ есть представление процесса P_i о локальном времени процесса P_j. Таким образом, вектор vt_i, который служит для хранения процессом P_i глобального логического времени, является временно́й меткой события и вкладывается в посылаемые сообщения.

Процесс P_i использует следующие два правила ***R1*** и ***R2*** для обновления часов:

- ***R1.*** Перед исполнением события процесс P_i обновляет свое логическое время:

$$vt_i[i] \leftarrow vt_i[i] + d \quad (d > 0);$$

Величина d обычно равна 1;

- ***R2.*** К каждому сообщению прилагается значение вектора vt отправителя во время отправления сообщения. Получив такое сообщение (m, vt), процесс P_i выполняет следующую последовательность действий:

1) изменяет свое глобальное логическое время:

$$1 \leqslant k \leqslant n : vti[k] \leftarrow \max(vti[k], vt[k]);$$

2) исполняет правило ***R*1**;
3) осуществляет доставку сообщения m.

Пример использования системы векторных часов с инкрементом $d = 1$ показан на рис. 2.5. Начальное значение вектора часов есть [0, 0, 0, …, 0].

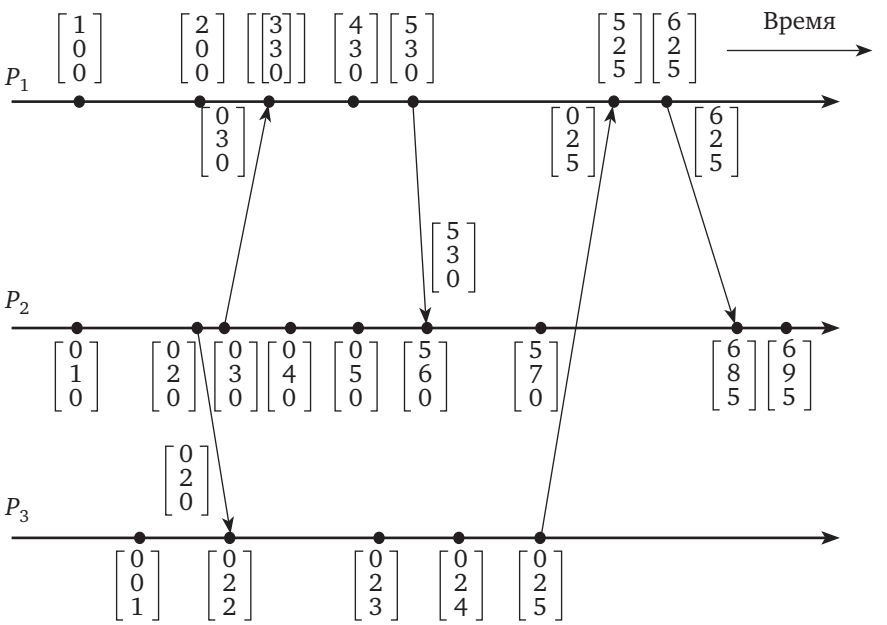

Рис. 2.5. **Эволюция векторного времени**

Система векторных часов строго консистентна, т. е. с помощью проверки векторных отметок времени мы можем определить, связаны ли события причинно. Это полезное и интересное свойство векторных отметок времени, благодаря которому они находят широкое применение. К сожалению, для реализации этого свойства размер вектора не может быть меньше n, где n — число взаимодействующих процессов, и это ограничивает применение алгоритма.

Подсчет событий

Если в правиле R_1 приращение d всегда равно 1, тогда i-й компонент векторных часов в процессе P_i, $vt_i[i]$, означает число событий, которые произошли в P_i к настоящему моменту. Таким образом, если событие e имеет метку vh, то $vh[j]$ означает число событий выполненных процессом P_j, которые причинно предшествуют e. Очевидно, что $\sum vh[j] - 1$ представляет суммарное число событий, которые предшествуют событию e в распределенном вычислении.

2.4.4. Алгоритмы реализации векторных часов

Для реализации векторных часов в распределенной системе, как и в алгоритме Лампорта, требуется добавлять к сообщениям информацию о локальных метках посылающего процесса. Для системы из N процессов требуется иметь массив из N логических отметок времени, одна логическая отметка на процесс. Каждый процесс, наряду со своей локальной меткой времени, поддерживает копии известных ему глобальных времен остальных процессов. Для поддержания консистентности копии применяется следующий набор правил:

- в начальный момент времени все метки нулевые;
- как только у процесса произошло внутреннее событие, он переводит свои часы вперед на единицу;
- передача сообщения — тоже событие, поэтому перед передачей любого сообщения внутренние часы тоже увеличиваются на единицу и копия всех известных процессу часов посылается вместе с сообщением;
- получение сообщения процессом заставляет его перевести все часы в своем векторе на максимум из известных ему и присланных значений.

Возникает вопрос, каким образом, используя векторные метки времени для двух событий, можно определить отношение предшествования для этих событий? Для этого определим операцию предшествования. Ее проще всего описать алгоритмически.

Алгоритм 1. Алгоритм определения предшествования

```
 1. Procedure Precedence(p1, p2: array[1..n] of int)
 2.   if ∀p1[i] ≤ p2[i], i ∈ 1 ... n then
 3.     if ∃I : p1[i] < p2[i] then
 4.       return p1 предшествует p2
 5.     else
 6.       return Предшествование p1 и p2 не определено
 7.     end if
 8.   end if
 9.   if ∀p2[i] ≤ p1[i], i ∈ 1 ... n then
10.    if ∃i : p2[i] < p1[i] then
11.      return p2 предшествует p1
12.    else
13.      return Предшествование p1 и p2 не определено
14.    end if
15.  end if
16. end procedure
```

Неформально этот алгоритм можно описать так: первое событие предшествует второму в том случае, если все значения всех отметок времени для первого события не больше соответствующих отметок времени второго события и при том хотя бы одно из значений отметок времени первого события строго меньше соответствующего значения второго события.

Реализация алгоритма определения предшествования и практические примеры использования векторных часов разбираются в разделе практической части книги 10.3.3.

2.5. Синхронное и асинхронное исполнение

2.5.1. Введение

Вычислительные системы по своей природе асинхронны. Непредсказуемые задержки исполнения возникают из-за переключений контекстов, работы подсистем виртуальной памяти или компьютерных сетей. В распределенных системах трудно задать лимиты на время, необходимое для исполнения процесса: сообщения задерживаются, часы дрейфуют и т. д. Имеет смысл рассмотреть две простых модели (асинхронную и синхронную) — первая не делает никаких предположений о времени, а вторая, наоборот, делает в отношении времени строгие предположения [8].

Определение 1. Асинхронное исполнение — это исполнение, в котором:

— нет синхронности процессоров и нет ограничений на дрейф процессорных часов;

— задержка сообщения (передача + время распространения) конечна, но не ограничена;

— нет верхней границы на время, необходимое процессу для осуществления шага.

Пример асинхронного исполнения четырех процессов приведен на рис. 2.6. Стрелки обозначают сообщения; хвост и голова стрелки означают события *Send* и *Receive* для сообщения, обозначенные кругами и вертикальными линиями соответственно. Не коммуникационные события, также называемые внутренними событиями, показаны закрашенными кругами.

Определение 2. Синхронное исполнение — исполнение, в котором:

— процессы синхронизированы и скорость дрейфа часов между любыми двумя процессорами ограничена;

— время доставки сообщения (передача + доставка) таково, что оно делается за один логический шаг или раунд;

— имеется известная верхняя граница на время выполнения процессом шага исполнения.

Временна́я диаграмма синхронного исполнения показана на рис. 2.7. В представленной системе на каждом шаге каждый процесс посылает несколько сообщений и попутно делает вычисления, обрабатывая полученные значения.

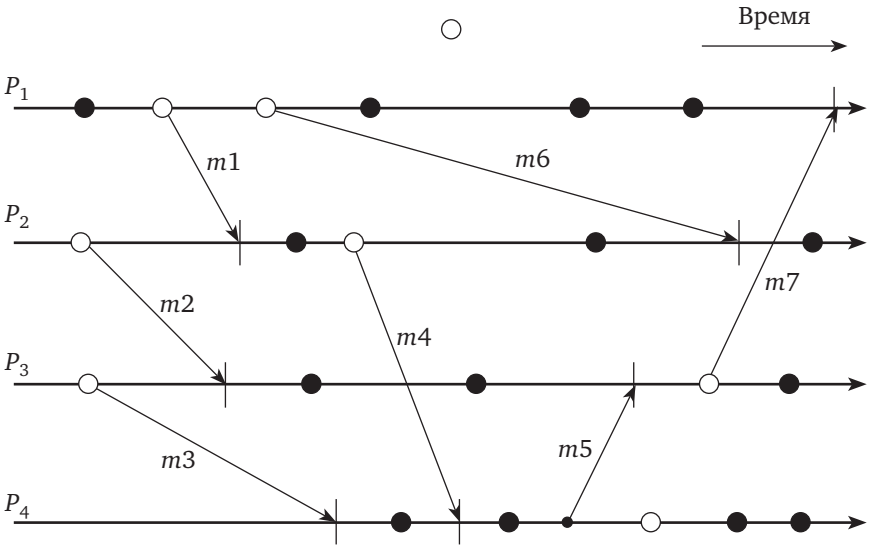

Рис. 2.6. **Пример асинхронного исполнения в системе с обменом сообщениями. Временна́я диаграмма используется для иллюстрации исполнения**

● — Внутреннее событие; ○ — Событие отправки; | — Событие приема

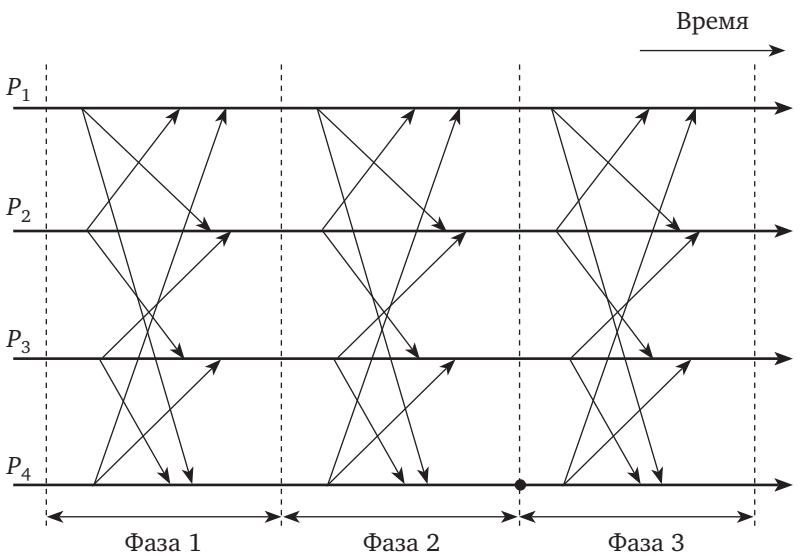

Рис. 2.7. **Пример синхронного исполнения в системе с обменом сообщениями. Все сообщения, посланные в раунде, получены в том же самом раунде**

Как мы убедимся далее, легче проектировать и верифицировать алгоритмы, подразумевая синхронное исполнение, вследствие скоординированной природы исполнений всех процессов. Однако на практике трудно построить синхронную систему и организовать доставку сообщения за ограниченное время. Поэтому синхронность должна быть смоделирована и будет с неизбежностью включать задержки или

блокировки отдельных процессов на некоторое время. Таким образом, синхронное исполнение есть абстракция, реализация которой требует усилий по синхронизации процессов. При этом процессы могут выполнять инструкции асинхронно внутри раунда, с требованием, чтобы после каждого раунда все процессы были синхронизированы и все сообщения доставлены. Это общее понимание синхронного исполнения.

Иногда процессорам разрешается в течение некоторого времени исполняться асинхронно, а затем они синхронизируются. Такое исполнение иногда называется виртуальной синхронностью (см. подпараграф 3.7.4).

2.5.2. Эмуляции синхронных систем асинхронными и наоборот

Синхронность систем — весьма полезное свойство. Обладая знанием глобального времени, можно упростить некоторые алгоритмы. Попробуем задать два вопроса.

1. Все ли алгоритмы, которые можно исполнить на синхронных системах, можно исполнить на асинхронных?
2. Все ли алгоритмы, которые можно исполнить на асинхронных системах, можно исполнить на синхронных?

Оказывается, ответами на оба вопроса будет «да». Множество алгоритмов, которые выполняются на синхронной системе, есть в точности множество алгоритмов, которые выполняются на асинхронной системе. Эти множества эквивалентны, если принять допущение, что системы надежны. Если же системы ненадежны, то существуют алгоритмы, которые корректно исполняются на синхронных системах, но не на асинхронных системах. Мы пока оставим в стороне ненадежные системы до лучших времен и попробуем доказать эквивалентность синхронных и асинхронных систем с точки зрения исполнимости алгоритмов.

Синхронная программа (написанная для синхронной системы) может быть эмулирована в асинхронной системе с использованием инструмента, называемого *синхронизатором*.

Чтобы доказать, что любая синхронная система может быть промоделирована асинхронной, достаточно показать алгоритм, который переводит синхронные системы в асинхронные, алгоритм-*синхронизатор*.

Одним из примеров алгоритмов такого класса являются ABD-синхронизаторы.

ABD-синхронизаторы

Давайте предположим, что время доставки сообщений ограничено сверху, а каждый из процессов снабжен физическими часами, пусть не вполне точными. Это допущение не является очень уж ограничивающим. Если мы рассмотрим гипотетическую распределенную систему из узлов, которые находятся в Канберре, Москве и Нью-Йорке, время доставки сообщений при надежной связи будет заведомо меньше $t_{\max} = L + P_{\max} / T$, где L — латентность, определяющаяся в том чис-

ле расстоянием между узлами и скоростью распространения сигналов (L < 20 000 км / 300 000 км/с ≈ 70 миллисекунд), P_{max} — длина наибольшего из сообщений, T — пропускная способность линии связи. Все значения в числителе конечны, поэтому t_{max} — конечная величина (примем ее за 100 миллисекунд). Обозначим как t_1 и t_2 наибольшее время, требуемое для исполнения фаз отправки и приема соответственно, а как время t_3 — наибольшее время, которое потребуется на фазу вычислений, общее наибольшее время исполнения всех фаз будет $m = t_1 + t_2 + t_3$. Физические часы в системе должны быть синхронизированы. Конечно, каждые часы будут показывать свое локальное время, но мы будем считать, что относительность времени мы уже учли в латентности. Обозначим за d максимальную погрешность показаний часов. Время, которое потребуется каждому из процессов на фазу вычислений, тоже будем полагать конечным ($t_3 < \infty$). Тогда m тоже конечно. Для разработки алгоритмов-синхронизаторов можно положить, что существуют локальные операции, время исполнения которых равно нулю (критически-важные операции), например операции сброса показания часов и операции инициации отправления/получения сообщений. Системы, которые удовлетворяют условию конечности времени всех фаз и наличием неточных согласованных физических часов, называют асинхронными системами с ограниченной задержкой, или *ABD-системами*, и для них можно создать синхронизаторы. Тель [18] разработал алгоритм, превращающий асинхронные ABD-системы в синхронные. Он состоит из двух этапов — этапа сверки часов и этапа моделирования.

На первом этапе каждый из процессов обнуляет свои часы и посылает каждому из соседей сообщение START. Если процесс получает сообщение START, он переходит к первому этапу. После этапа сверки часы соседних процессов будут расходиться не более, чем на m, причем $|CLOCK_{P_1} - CLOCK_{P_2}| < m$.

На i-м раунде в этапе моделирования процессы выполняют действия, который им положено исполнять на i-м этапе синхронного алгоритма. Этот этап начнется во время $(2m + d) \cdot i$. Нетрудно убедиться, что все сообщения обеих фаз будут доставлены до времени $(2m + d) \cdot (i + 1)$.

Проиллюстрируем алгоритм рисунками. На рис. 2.8 изображена простая распределенная система, состоящая из четырех процессов.

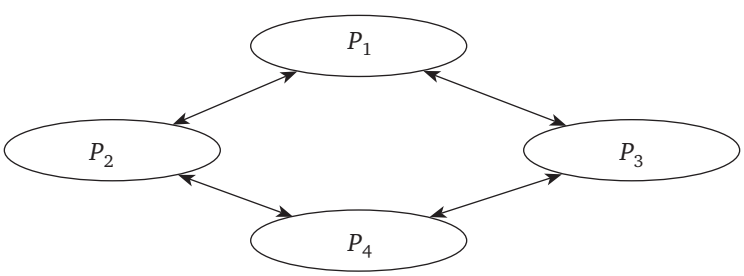

Рис. 2.8. **Схема простой распределенной системы**