МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

М.С. Косяков

ВВЕДЕНИЕ В РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ

Учебное пособие

Санкт-Петербург

2014

Косяков М.С. Введение в распределенные вычисления. – СПб: НИУ ИТМО, 2014. – 155 с.

В пособии излагаются основные понятия и концепции из области распределенных вычислений, для модели асинхронных распределенных систем приводятся методы и алгоритмы решения наиболее важных задач. Особое внимание уделяется механизму логических часов, позволяющему значительно упростить разработку алгоритмов для распределенных систем. Внимательно рассматриваются основные распределенные алгоритмы взаимного исключения. Изучение этих алгоритмов позволяет раскрыть такие важные вопросы, как обеспечение свойств безопасности и живучести распределенных алгоритмов. Материал пособия сопровождается многочисленными примерами, демонстрирующими применение изучаемых методов и алгоритмов для решения реальных задач.

Учебное пособие предназначено, прежде всего, для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника» и 231000 «Программная инженерия», изучающих дисциплину «Распределенные вычисления» и связанные с ней дисциплины. Пособие может быть полезно для специалистов в области информатики, вычислительной техники и программирования в качестве введения в проблематику распределенных вычислений.

Рекомендовано к печати Ученым советом факультета компьютерных технологий и управления 11 марта 2014 г., протокол № 3.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сегодняшние реалии таковы, что разработка практически любого программного обеспечения требует хороших знаний параллельного и распределенного программирования. Обе эти области объединяет то, что и параллельное, и распределенное программное обеспечение состоит из нескольких процессов, которые вместе решают одну общую задачу.

Однако отметим, что понятия параллельной и распределенной обработки данных не являются эквивалентными. Действительно, параллелизм подразумевает одновременное существование и выполнение задач. Распределение же обозначает территориальную удаленность процессов друг от друга. При этом задачи, связанные с распределенной обработкой данных, могут выполняться несколькими последовательными этапами в различные периоды времени. Конечно, в некоторых случаях, параллельная обработка данных может быть эффективно реализована с помощью распределенного программного обеспечения, особенно, при проведении большого объема вычислений, слабо связанных между собой. Поэтому можно сказать, что в общем случае распределенная обработка данных не подразумевает параллелизма, но возможность "параллельного использования" распределенных вычислительных систем существует.

Также обратим внимание, что для совместной работы процессов им необходим некоторый механизм взаимодействия. В мире параллельных систем обычно считается, что взаимодействие процессов обеспечивается с помощью разделяемых переменных. В этом случае один процесс осуществляет запись в одну из таких переменных, а другой – считывает записанное значение. В свою очередь в распределенных системах процессы взаимодействуют путем пересылки сообщений. Поэтому для обмена данными или координации совместных действий процессы должны использовать механизмы отправки и приема сообщений.

В настоящее время имеется немало книг, посвященных вопросам параллельного программирования. Тем не менее, по мнению автора, ощущается определенный недостаток в русскоязычной литературе, знакомящей читателей с основными проблемами в области распределенной обработки данных и существующими подходами к их решению. Среди известных автору источников по данной тематике можно выделить переводы книги Э. Таненбаума [2, 6] и монографии Ж. Теля [3, 7].

В книге Э. Таненбаума основное внимание уделяется архитектуре и технологиям построения распределенных систем, а не распределенным алгоритмам, в сущности, образующим ядро практически любого распределенного программного обеспечения. Поэтому, на взгляд автора, эта книга в большей степени ориентирована на инженеров-практиков, нежели на теоретиков, изучающих фундаментальные принципы и модели,

лежащие в основе всех распределенных систем. В свою очередь монография Ж. Теля посвящена преимущественно распределенным алгоритмам. Материал монографии излагается математическим языком, исследование особенностей распределенной обработки данных проводится на основе понятия системы переходов. Все утверждения и теоремы, приведенные в книге, сопровождаются строгими формальными доказательствами. Как следствие, подобный уровень изложения материала требует от читателя определенной начальной подготовки и, не исключено, что может отпугнуть молодого специалиста, только начинающего изучение распределенных алгоритмов и принципов работы распределенных систем.

В этой связи автор ставил своей целью написание учебного пособия, направленного на изучение фундаментальных аспектов организации распределенной обработки данных и ориентированного на неподготовленного читателя. С этой же целью описание модели распределенного вычисления и доказательство всех утверждений проводится до определенной степени неформально, но на уровне, достаточном для понимания существенных особенностей функционирования распределенных систем, выделяющих их среди систем других классов.

Материал учебного пособия следует рассматривать в качестве введения в проблематику распределенных вычислений. В пособии представлено несколько базовых понятий, концепций и алгоритмов, которые послужат основой для дальнейшего изучения предмета распределенных вычислений заинтересованными студентами. В первом разделе обсуждаются цели и задачи распределенных систем, а также сложности их разработки. Во втором разделе представлена модель распределенного вычисления, используемая при дальнейшем изложении. Кроме того, не опираясь на понятие единого физического времени, вводится отношение "произошло раньше", связывающее события процессов между собой. В третьем разделе рассматриваются различные механизмы логических часов, позволяющие упорядочивать события в одну или несколько последовательностей, которые могли бы происходить в системе, и, как следствие, значительно упрощающие разработку алгоритмов для распределенных систем. Четвертый раздел посвящен изучению основных распределенных алгоритмов взаимного исключения, ключевые идеи которых используются и для решения многих других задач в распределенных системах. Представленный в пособии список литературы содержит перечень источников, которые автор рекомендует в первую очередь для получения более подробной информации по рассматриваемым и дополнительным вопросам из области распределенных вычислений [1-7] и материал из которых в той или иной степени был использован при написании данного пособия [1-30].

Учебное пособие предназначено, прежде всего, для студентов, обучающихся по направлению "Информатика и вычислительная техника" и "Программная инженерия", изучающих дисциплину "Распределенные вычисления" и связанные с ней дисциплины. Пособие может быть полезно для специалистов в области информатики, вычислительной техники и программирования в качестве введения в проблематику распределенных вычислений.

Автор понимает, что, как и "в любой хорошо отлаженной программе всегда есть хотя бы одна ошибка", так и в любой многократно вычитанной книге она также наверняка присутствует. Поэтому автор будет благодарен за все присланные по адресу mkosyakov@gmail.com обнаруженные ошибки и опечатки, а также критические замечания.

РАЗДЕЛ 1. ПРЕДМЕТ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ

1.1. Понятия распределенных вычислений и распределенной системы

Область *распределенных вычислений* (англ. *distributed computing*) представляет собой раздел теории вычислительных систем, изучающий теоретические вопросы организации *распределенных систем* (англ. *distributed systems*).

Также распределенные вычисления иногда определяют в более узком смысле, как применение распределенных систем для решения трудоемких вычислительных задач. В таком контексте распределенные вычисления являются частным случаем параллельных вычислений, т.е. одновременного решения различных частей одной вычислительной задачи несколькими вычислительными устройствами. Отметим, что при изучении параллельных вычислений основной акцент обычно делается на методах разделения решаемой задачи на подзадачи, которые могут рассчитываться одновременно для максимального ускорения вычислений. Основная же особенность в организации параллельных вычислений с использованием распределенных систем будет заключаться в необходимости учитывать различие характеристик доступных вычислительных устройств и наличие существенной временной задержки при обмене данными между ними.

Мы в дальнейшем будем рассматривать распределенные вычисления в широком смысле, как теоретическую основу для построения распределенных систем обработки данных.

Существует множество определений распределенной системы, причем ни одно из них не является строгим или общепринятым.

Весьма оригинальное определение принадлежит американскому ученому в области теории вычислительных систем Лесли Лэмпорту (*Leslie Lamport*). Согласно его утверждению, вы понимаете, что пользуетесь распределенной системой, когда поломка компьютера, о существовании которого вы даже не подозревали, приводит к останову всей системы, а для вас – к невозможности выполнить свою работу. Значительная часть распределенных систем, к сожалению, удовлетворяет такому определению, однако, строго говоря, оно относится только к системам с единой точкой отказа (англ. *single point of failure*).

В свою очередь хорошо известный российской аудитории профессор вычислительной техники Эндрю С. Таненбаум (*Andrew S. Tanenbaum*) определяет распределенную систему как набор *независимых* компьютеров, представляющийся их пользователям *единой* объединенной системой. Здесь необходимо обратить внимание на то, что сами по себе независимые компьютеры не могут представляться пользователю единой системой. Обеспечить это можно только с помощью дополнительного специального

программного обеспечения, называемого *программным обеспечением промежуточного уровня* (англ. *middleware*). Именно с его помощью пользователи полагают, что имеют дело с единой системой, а все различия между компьютерами и способы связи между ними остаются скрытыми для пользователей.

Приведем еще несколько других определений, встречающихся в литературе.

- Распределенная система – это такая система, в которой взаимодействие и синхронизация программных компонентов, выполняемых на независимых сетевых компьютерах, осуществляется посредством передачи сообщений.
- Распределенная система – набор независимых компьютеров, не имеющих общей совместно используемой памяти и общего единого времени (таймера) и взаимодействующих через коммуникационную сеть посредством передачи сообщений, где каждый компьютер использует свою собственную оперативную память и на котором выполняется отдельный экземпляр своей операционной системы. Однако эти операционные системы функционируют совместно, предоставляя свои службы друг другу для решения общей задачи.
- Термин "распределенная система" описывает широкий спектр систем от слабо связанных многомашинных комплексов, представляемых, например, набором персональных компьютеров, объединенных в сеть, до сильно связанных многопроцессорных систем.

Мы будем рассматривать распределенную систему с аппаратной точки зрения в виде совокупности взаимосвязанных *автономных* компьютеров или процессоров, с программной точки зрения – в виде совокупности *независимых* процессов (исполняемых программных компонентов распределенной системы), взаимодействующих посредством передачи сообщений для обмена данными и координации своих действий. Компьютеры, процессоры или процессы будем называть узлами распределенной системы. Чтобы процессоры могли считаться автономными, они должны, по меньшей мере, обладать собственным независимым управлением. По этой причине параллельный компьютер, архитектура которого устроена по схеме "одна команда для многих данных" (англ. *Single Instruction - Multiple Data*, *SIMD*), не может считаться распределенной системой. Под независимостью процессов подразумевается тот факт, что каждый процесс имеет свое собственное состояние, представляемое набором данных, включающим текущие значения счетчика команд, регистров и переменных, к которым процесс может обращаться и которые может изменять. Состояние каждого процесса является полностью закрытым для других процессов: другие процессы не имеют к нему прямого доступа и не могут изменять его.

Скорости выполнения операций разных процессов в распределенной системе различны и заранее неизвестны, а доставка отправленных сообщений может занимать непредсказуемое время.

Поскольку в качестве узлов системы могут выступать процессы, под приведенное нами определение подпадают также и программные системы, представляющие собой совокупность взаимодействующих процессов, выполняемых на одном и том же вычислительном устройстве. В этой ситуации каналы взаимодействия, осуществляющие передачу сообщений между процессами, реализуются с помощью разделяемой памяти вместо сети связи. Однако в большинстве случаев в распределенной системе все же содержится несколько процессоров, взаимосвязанных друг с другом при помощи средств коммуникации. Типичная распределенная система представлена на рис. 1.1; ЦП – центральный процессор, ОП – оперативная (основная) память.

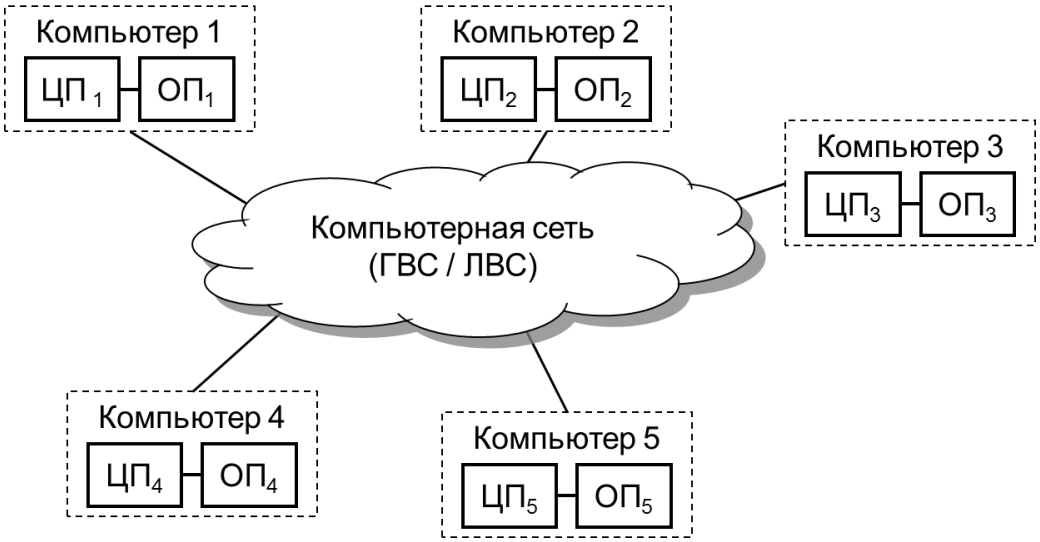


Рис. 1.1. Распределенная система объединяет независимые компьютеры с помощью компьютерной сети.

Возможно, вместо того чтобы рассматривать различные определения, разумнее будет остановиться на основных отличительных признаках, характеризующих распределенные системы. К таким признакам обычно относят:

Отсутствие единого времени для компонентов распределенной системы. Это важное предположение для решения задач проектирования и построения распределенных систем. Оно характеризует территориальное распределение компонентов системы, а именно процессоров, входящих в ее состав, но что более важно, из него следует отсутствие синхронности в их работе.

Отсутствие общей памяти. Это ключевая характеристика, из которой следует необходимость обмена сообщениями между

программными компонентами распределенной системы для их взаимодействия и синхронизации. Кроме того, эта характеристика подразумевает отсутствие единого для всех процессоров физического времени.

Здесь следует отметить, что некоторые распределенные системы могут предоставлять своим пользователям абстракцию единого адресного пространства для всех процессоров с помощью механизмов распределенной разделяемой памяти (англ. *Distributed Shared Memory, DSM*). В этом случае, если не рассматривать сложности конкурентного доступа нескольких процессоров к одному сегменту памяти, для каждого процессора распределенную разделяемую память можно представлять как вполне нормальную организацию виртуальной памяти, где в качестве временного хранилища информации используется не собственный диск, а оперативная память удаленного компьютера. В этой связи в литературе по распределенным системам обычно, среди прочего, рассматриваются различные аспекты организации общей памяти в многопроцессорных системах.

Географическое распределение. Вполне естественно, что чем сильнее удалены процессоры друг от друга территориально, тем понятнее, что система будет рассматриваться как распределенная. Однако совсем не обязательно, чтобы компьютеры были объединены в глобальную вычислительную сеть (*ГВС*). В последнее время кластер из обыкновенных рабочих станций (англ. *Cluster Of Workstation, COW*), соединенных с помощью локальной вычислительной сети (*ЛВС*), также все чаще рассматривается как небольшая распределенная система. При этом все оборудование такой распределенной системы может находиться в одном или нескольких соседних зданиях. Подобные кластеры COW становятся все популярнее из-за относительно низкой стоимости входящих в нее компонентов с одной стороны и неплохой производительности – с другой. Например, ядро поисковой системы компании Google построено по архитектуре COW.

Независимость и гетерогенность. Компьютеры, входящие в состав распределенной системы слабо связаны в том смысле, что они могут иметь различный состав и различную производительность и, следовательно, обеспечивать различное время выполнения идентичных задач. Обычно они не являются частями одной специализированной системы, но функционируют совместно, предоставляя свои службы друг другу для выполнения общей задачи. Более того, в общем случае на компьютерах, составляющих распределенную систему, могут выполняться различные операционные системы.

1.2. Цели построения распределенных систем

За последние несколько лет распределенные системы становились все более популярными и их роль только возрастала. Среди основных причин роста их значимости можно выделить следующие:

Географически распределенная вычислительная среда. Сегодня в большинстве случаев сама вычислительная среда по своей природе представляет собой территориально распределенную систему. В качестве примера можно привести банковскую сеть. Каждый банк обслуживает счета своих клиентов и обрабатывает операции с ними. В случае же перевода денег из одного банка в другой требуется осуществление межбанковской транзакции и взаимодействие систем банков друг с другом. Другим примером географически распределенной вычислительной среды является всем хорошо знакомая сеть Интернет.

Требование увеличения производительности вычислений. Быстродействие традиционных однопроцессорных систем стремительно приближается к своему пределу. Различные архитектуры (такие как суперскалярная архитектура, матричные и векторные процессоры, однокристальные многопроцессорные системы) призваны увеличивать производительность вычислительных систем за счет различных механизмов параллельного исполнения команд. Однако все эти приемы способны повысить производительность максимум в десятки раз по сравнения с классическими последовательными решениями. Кроме того, масштабируемость подобных подходов оставляет желать лучшего. Чтобы повысить производительность в сотни или тысячи раз и при этом обеспечивать хорошую масштабируемость решения необходимо свести воедино многочисленные процессоры и обеспечить их эффективное взаимодействие. Этот принцип реализуется в виде больших многопроцессорных систем и многомашинных комплексов.

Совместное использование ресурсов. Важной целью создания и использования распределенных систем является предоставление пользователям (и приложениям) доступа к удаленным ресурсам и обеспечение их совместного использования. В данной формулировке термин ресурс относится как к компонентам аппаратного обеспечения вычислительной системы, так и к программным абстракциям, с которыми работает распределенная система. Например, пользователь компьютера 1 может использовать дисковое пространство компьютера 2 для хранения своих файлов. Или приложение А может использовать свободную вычислительную мощность нескольких компьютеров для ускорения собственных расчетов. Распределенные базы данных и распределенные системы объектов могут быть отличным примером совместного использования программных компонентов, когда соответствующие программные абстракции распределены по нескольким компьютерам и

согласованно обслуживаются несколькими процессами, образующими распределенную систему.

Отказоустойчивость. В традиционных "нераспределенных" вычислительных системах, построенных на базе единичного компьютера (возможно высокопроизводительного), выход из строя одного из его компонентов обычно приводит к неработоспособности всей системы. Такой сбой в одном или нескольких компонентах системы называют частичным отказом, если он не затрагивает другие компоненты. Характерной чертой распределенных систем, которая отличает их от единичных компьютеров, является устойчивость к частичным отказам, т.е. система продолжает функционировать после частичных отказов, правда, незначительно снижая при этом общую производительность. Подобная возможность достигается за счет избыточности, когда в систему добавляется дополнительное оборудование (аппаратная избыточность) или процессы (программная избыточность), которые делают возможной правильное функционирование системы при неработоспособности или некорректной работе некоторых из ее компонентов. В этом случае распределенная система пытается скрывать факты отказов или ошибок в одних процессах от других процессов. Например, в системах с тройным модульным резервированием (англ. *Triple Modular Redundancy, TMR*) используются три одинаковых вычислительных модуля, осуществляющих идентичные вычисления, а корректный результат определяется простым голосованием.

1.3. Требования к распределенным системам

Эффективная распределенная система должна обладать следующими свойствами: прозрачность (англ. *transparency*), открытость (англ. *openness*), безопасность (англ. *security*), масштабирование (англ. *scalability*). Однако стоит отметить, что, несмотря на кажущуюся простоту и очевидность перечисленных свойств, их реализация на практике часто представляет собой непростую задачу.

1.3.1. Прозрачность (Transparency)

Под прозрачностью распределенной системы понимают ее способность скрывать свою распределенную природу, а именно, распределение процессов и ресурсов по множеству компьютеров, и представляться для пользователей и разработчиков приложений в виде единой централизованной компьютерной системы. Стандарты эталонной модели для распределенной обработки в открытых системах *Reference Model for Open Distributed Processing* (RM-ODP) определяют несколько типов прозрачности. Наиболее важные из них перечислены ниже.

- *Прозрачность доступа* (англ. *access transparency*). Вне зависимости от способов доступа к ресурсам и их внутреннего представления, обращения к локальным и удаленным ресурсам осуществляется одинаковым образом. На базовом уровне скрывается разница архитектур вычислительных платформ, но, что более важно, достигается соглашение о том, как ресурсы разнородных машин, будут представляться пользователям распределенной системы единым образом. В качестве примера можно привести прикладной программный интерфейс (англ. *application programming interface*, *API*) для работы с файлами, хранящимися на множестве компьютеров различных архитектур, который предоставляет одинаковые вызовы операций как с локальными, так и с удаленными файлами.
- *Прозрачность местоположения* (англ. *location transparency*). Позволяет обращаться к ресурсам без знания их физического местоположения. В этом случае имя запрашиваемого ресурса не должно давать никакого представления о том, где ресурс расположен. Поэтому важную роль для обеспечения прозрачности местоположения играет именование ресурсов. Например, чтобы отправить электронное сообщение на адрес user@company.com не требуется знать физического местоположения получателя, его почтового ящика или почтового сервера. В свою очередь обращение к файлу \\server\foo подразумевает знание имени сервера, на котором он расположен, а значит, не является полностью прозрачным с точки зрения местоположения.
- *Прозрачность перемещения* (англ. *migration transparency*). Перемещение ресурса или процесса в другое физическое местоположение остается незаметным для пользователя распределенной системы. Здесь стоит отметить, что выполнение требования прозрачности местоположения не гарантирует прозрачности перемещения. Другими словами, если распределенная система скрывает местоположение ресурса, это не означает, что его можно сменить незаметно для пользователя. Например, распределенные файловые системы позволяют монтировать файловые системы удаленных компьютеров в локальное пространство имен клиента, предоставляя единое дерево каталогов и тем самым обеспечивая прозрачность местоположения. Однако если файлы на удаленных компьютерах будут перемещены в другое место, в большей части распределенных файловых систем они станут недоступны для пользователя.
- *Прозрачность смены местоположения* (англ. *relocation transparency*). Более строгое по отношению к предыдущему требование скрыть факт перемещения ресурса во время его использования. Примером могут служить мобильные пользователи,

использующие сотовые телефоны. В этом случае, если рассматривать вызывающего абонента в качестве пользователя распределенной системы, а вызываемого – в качестве ее ресурса, то система будет прозрачной с точки зрения смены местоположения. Действительно, перемещение "ресурса" из соты в соту в процессе разговора остается незаметным для звонящего.

- *Прозрачность репликации* (англ. *replication transparency*). Если для повышения доступности или увеличения производительности используется несколько копий ресурса (реплик), этот факт остается скрытым от пользователя, и он полагает, что в системе присутствует только один экземпляр ресурса. Для обеспечения прозрачности репликации необходимо, чтобы все реплики имели одно и то же имя, очевидно, не зависящее от местоположения копии ресурса. Таким образом, системы, которые обеспечивают прозрачность репликации, также должны поддерживать и прозрачность местоположения.
- *Прозрачность одновременного доступа* (англ. *concurrency transparency*). Позволяет нескольким пользователям (конкурирующим процессам) одновременно выполнять операции над общим, совместно используемым ресурсом без взаимного влияния друг на друга. Иначе говоря, скрывается факт использования ресурса другими пользователями (процессами). Стоит отметить, что сам ресурс должен оставаться в непротиворечивом состоянии, что может достигаться, например, с помощью механизма блокировок, когда пользователи (процессы) по очереди получают исключительные права на запрашиваемый ресурс.
- *Прозрачность отказов* (англ. *failure transparency*). Подразумевается, что система должна пытаться скрывать частичные отказы, позволяя пользователям и приложениям выполнить свою работу вне зависимости от сбоев в аппаратных или программных компонентах распределенной системы, а также скрывать факт их последующего восстановления. В связи с тем, что любой процесс, компьютер или сетевое соединение могут отказывать независимо от других в произвольные моменты времени, каждый компонент распределенной системы должен быть готов к сбоям в других компонентах и обрабатывать подобные ситуации соответствующим образом.

Степень прозрачности. Важно отметить, что степень, до которой каждое из перечисленных выше свойств должно быть выполнено, может сильно варьироваться в зависимости от задач построения распределенной системы. Действительно, полностью скрыть распределение процессов и ресурсов вряд ли удастся. Из-за ограничения в скорости передачи сигнала, задержка на обращение к ресурсам, территориально удаленным от клиента, всегда будет больше, чем к ресурсам, расположенным поблизости.

Поэтому не каждая система в состоянии или даже должна пытаться скрывать все свои особенности от пользователя. Обычно, это утверждение выражается в поиске компромисса между прозрачностью распределенной системы и ее производительностью.

Например, если для повышения отказоустойчивости в системе присутствуют географически распределенные копии ресурса, то поддержка их идентичного состояния для обеспечения прозрачности репликации потребует гораздо большего времени выполнения каждой операции обновления. Другими словами, каждая операция обновления должна будет распространиться на все реплики до того, как будет разрешена следующая операция с данным ресурсом. Или, например, многие приложения предпринимают несколько последовательных попыток связаться с сервером, пытаясь скрыть его временную недоступность, тем самым замедляя работу системы. Однако, в некоторых случаях, например, если на самом деле сервер вышел из строя, было бы разумнее сразу уведомить пользователя о недоступности ресурса.

1.3.2. Открытость (Openness)

Согласно определению, принятому комитетом IEEE POSIX 1003.0, открытая система – это система, реализующая открытые спецификации (стандарты) на интерфейсы, службы и поддерживаемые форматы данных, достаточные для того, чтобы обеспечить:

- возможность переноса разработанного прикладного программного обеспечения на широкий диапазон систем с минимальными изменениями (*мобильность приложений, переносимость*);
- совместную работу (взаимодействие) с другими прикладными приложениями на локальных и удаленных платформах (*интероперабельность, способность к взаимодействию*);
- взаимодействие с пользователями в стиле, облегчающим последним переход от системы к системе (*мобильность пользователя*).

Ключевой момент в этом определении – использование понятия открытая спецификация, которое, в свою очередь, определяется как общедоступная спецификация, которая поддерживается открытым, гласным согласительным процессом, направленным на постоянную адаптацию к новым технологиям, и соответствует стандартам.

Согласно этому определению открытая спецификация не зависит от конкретной технологии, т.е. не зависит от конкретных технических и программных средств или продуктов отдельных производителей. Открытая спецификация одинаково доступна любой заинтересованной стороне. Более того, открытая спецификация находится под контролем общественного мнения, поэтому заинтересованные стороны могут принимать участие в ее развитии.

В контексте распределенных систем приведенное выше определение означает, что свойство открытости не может быть достигнуто, если спецификация и описание ключевых интерфейсов программных компонентов системы не доступны для разработчиков. Одним словом, ключевые интерфейсы должны быть описаны и опубликованы. Важно отметить, что здесь в первую очередь подразумеваются интерфейсы внутренних компонентов системы, а не только интерфейсы верхнего уровня, с которыми работают пользователи и приложения. При этом синтаксис интерфейсов, т.е. имена доступных функций, типы передаваемых параметров, возвращаемых значений и т.п., обычно описывается посредством языка определения интерфейсов (англ. *Interface Definition Language*, *IDL*). В свою очередь семантика интерфейсов, т.е. то, что на самом деле делают службы, предоставляющие эти интерфейсы, обычно задается неформально, с помощью естественного языка.

Подобное описание позволяет произвольному процессу, нуждающемуся в определенной службе, обратиться к другому процессу, реализующему эту службу, через соответствующий интерфейс. Кроме того, такой подход позволяет создавать несколько различных реализаций одной и той же службы, которые с точки зрения внешних процессов будут работать абсолютно одинаково. Как следствие, несколько реализаций программных компонентов (возможно, от различных производителей) могут взаимодействовать и работать совместно, образуя единую распределенную систему. Таким образом, достигается свойство интероперабельности или, другими словами, способности к взаимодействию. Более того, в этом случае прикладное приложение, разработанное для распределенной системы A, может без изменений выполняться в распределенной системе B, реализующей те же интерфейсы, что и A. То есть достигается свойство переносимости.

Еще одно важное преимущество заключается в том, что открытая распределенная система потенциально может быть образована из разнородного аппаратного и программного обеспечения (опять-таки, возможно, от разных производителей). При этом добавление новых компонентов или замена существующих может осуществляться относительно легко, не затрагивая других компонентов. На аппаратном уровне это выражается в способности простого подключения к системе дополнительных компьютеров или замены существующих на более мощные. На программном – в возможности простого внедрения новых служб или новых реализаций уже существующих. Другими словами, важным свойством открытой распределенной системы является *расширяемость*.

1.3.3. Масштабируемость (Scalability)

В общем случае масштабируемость определяют, как способность вычислительной системы эффективно справляться с увеличением числа пользователей или поддерживаемых ресурсов без потери производительности и без увеличения административной нагрузки на ее управление. При этом систему называют *масштабируемой*, если она способна увеличивать свою производительность при добавлении новых аппаратных средств. Другими словами, под масштабируемостью понимают способность системы расти вместе с ростом нагрузки на нее.

Масштабируемость является важным свойством вычислительных систем, если им может потребоваться работать под большой нагрузкой, поскольку означает, что вам не придется начинать с нуля и создавать абсолютно новую информационную систему. Если у вас есть масштабируемая система, то, скорее всего, вам удастся сохранить то же самое программное обеспечение, попросту нарастив аппаратную часть.

Для распределенных систем обычно выделяют несколько параметров, характеризующих их масштаб: количество пользователей и количество компонентов, составляющих систему, степень территориальной отдаленности сетевых компьютеров системы друг от друга и количество административных организаций, обслуживающих части распределенной системы. Поэтому масштабируемость распределенных систем также определяют по соответствующим направлениям:

- *Нагрузочная масштабируемость.* Способность системы увеличивать свою производительность при увеличении нагрузки путем замены существующих аппаратных компонентов на более мощные или путем добавления новых аппаратных средств. При этом первый случай увеличения производительности каждого компонента системы с целью повышения общей производительности называют *вертикальным масштабированием*, а второй, выражающийся в увеличении количества сетевых компьютеров (серверов) распределенной системы – *горизонтальным масштабированием*.
- *Географическая масштабируемость.* Способность системы сохранять свои основные характеристики, такие как производительность, простота и удобство использования, при территориальном разнесении ее компонентов от более локального взаимного расположения до более распределенного.
- *Административная масштабируемость.* Характеризует простоту управления системой при увеличении количества административно независимых организаций, обслуживающих части одной распределенной системы.

Сложности масштабирования. Построение масштабируемых систем подразумевает решение широкого круга задач и часто сталкивается с ограничениями реализованных в вычислительных системах *централизованных служб, данных* и *алгоритмов*. А именно, многие службы централизованы в том смысле, что они реализованы в виде единственного процесса и могут выполняться только на одном компьютере (сервере). Проблема такого подхода заключается в том, что при увеличении числа пользователей или приложений, использующих эту службу, сервер, на котором она выполняется, станет узким местом и будет ограничивать общую производительность. Если даже предположить возможность неограниченного увеличения мощности такого сервера (вертикальное масштабирование), то тогда ограничивающим фактором станет пропускная способность линий связи, соединяющих его с остальными компонентами распределенной системы. Аналогично, централизация данных требует централизованной обработки, приводя к тем же самым ограничениям. В качестве примера преимуществ децентрализованного подхода можно привести службу доменных имен (англ. *Domain Name Service, DNS*), которая на сегодняшний день является одной из самых больших распределенных систем именования. Служба DNS используется в первую очередь для поиска IP-адресов по доменному имени и обрабатывает миллионы запросов с компьютеров по всему миру. При этом распределенная база данных DNS поддерживается с помощью иерархии DNS-серверов, взаимодействующих по определенному протоколу. Если бы все данные DNS централизовано хранились бы на единственном сервере, и каждый запрос на интерпретацию доменного имени передавался бы на этот сервер, воспользоваться такой системой в масштабах всего мира было бы невозможно.

Отдельно стоит отметить ограничения, создаваемые применением централизованных алгоритмов. Дело в том, что централизованные алгоритмы для своей работы требуют получения всех входных данных и только после этого производят соответствующие операции над ними, а уже затем распространяют результаты всем заинтересованным сторонам. С этой точки зрения проблемы использования централизованных алгоритмов эквивалентны рассмотренным выше проблемам централизации служб и данных. Поэтому для достижения хорошей масштабируемости следует применять *распределенные алгоритмы*, предусматривающие параллельное выполнение частей одного и того же алгоритма независимыми процессами.

В отличие от централизованных алгоритмов, распределенные алгоритмы обладают следующими свойствами, которые на самом деле значительно усложняют их проектирование и реализацию:

- *Отсутствие знания глобального состояния*. Как уже было сказано, централизованные алгоритмы обладают полной информацией о состоянии всей системы и определяют следующие действия, исходя

из ее текущего состояния. В свою очередь, каждый процесс, реализующий часть распределенного алгоритма, имеет непосредственный доступ только к своему состоянию, но не к глобальному состоянию всей системы. Соответственно, процессы принимают решения только на основе своей локальной информации. Следует отметить, что информацию о состоянии других процессов в распределенной системе каждый процесс может получить только из пришедших сообщений, и эта информация может оказаться устаревшей на момент получения. Аналогичная ситуация имеет место в астрономии: знания об изучаемом объекте (звезде / галактике) формируются на основании светового и прочего электромагнитного излучения, и это излучение дает представление о состоянии объекта в прошлом. Например, знания об объекте, находящемся на расстоянии пяти тысяч световых лет, являются устаревшими на пять тысяч лет.

- *Отсутствие общего единого времени*. События, составляющие ход выполнения централизованного алгоритма *полностью упорядочены*: для любой пары событий можно с уверенностью утверждать, что одно из них произошло раньше другого. При выполнении распределенного алгоритма вследствие отсутствия единого для всех процессов времени, события нельзя считать полностью упорядоченными: для некоторых пар событий мы можем утверждать, какое из них произошло раньше другого, для других – нет.
- *Отсутствие детерминизма*. Централизованный алгоритм чаще всего определяется как строго детерминированная последовательность действий, описывающая процесс преобразования объекта из начального состояния в конечное. Таким образом, если мы будем запускать централизованный алгоритм на выполнение с одним и тем же набором входных данных, мы будем получать один и тот же результат и одинаковую последовательность переходов из состояния в состояние. В свою очередь выполнение распределенного алгоритма носит недетерминированный характер из-за независимого исполнения процессов с различной и неизвестной скоростью, а также из-за случайных задержек передачи сообщений между ними. Поэтому, несмотря на то, что для распределенных систем может быть определено понятие глобального состояния, выполнение распределенного алгоритма может лишь ограниченно рассматриваться как переход из одного глобального состояния в другое, т.к. для этого же алгоритма выполнение может быть описано другой последовательностью глобальных состояний. Такие альтернативные последовательности обычно состоят из других глобальных состояний, и поэтому нет особого смысла говорить о том,

что то или иное состояние достигается по ходу выполнения распределенного алгоритма.

- *Устойчивость к отказам*. Сбой в любом из процессов или каналов связи не должен вызывать нарушения работы распределенного алгоритма.

Для обеспечения географической масштабируемости требуются свои подходы. Одна из основных причин плохой географической масштабируемости многих распределенных систем, разработанных для локальных сетей, заключается в том, что в их основе лежит принцип *синхронной связи* (англ. *synchronous communication*). В этом виде связи клиент, вызывающий какую-либо службу сервера, блокируется до получения ответа. Это неплохо работает, когда взаимодействие между процессами происходит быстро и незаметно для пользователя. Однако при увеличении задержки на обращение к удаленной службе в глобальной системе подобный подход становится все менее привлекательным и, очень часто, абсолютно неприемлемым.

Другая сложность обеспечения географической масштабируемости состоит в том, что связь в глобальных сетях по своей природе ненадежна и взаимодействие процессов практически всегда является двухточечным (англ. *point-to-point*). В свою очередь, связь в локальных сетях является высоконадежной и подразумевает использование широковещательных сообщений, что значительно упрощает разработку распределенных приложений. Например, если процессу требуется обнаружить адрес другого процесса, предоставляющего определенную службу, в локальных сетях ему достаточно разослать широковещательное сообщение с просьбой для искомого процесса откликнуться на него. Все процессы получают и обрабатывают это сообщение. Но только процесс, предоставляющий требуемую службу, отвечает на полученную просьбу, указывая свой адрес в ответном сообщении. Очевидно, подобное взаимодействие перегружает сеть, и использовать его в глобальных сетях нереально.

Технологии масштабирования. В большинстве случаев сложности масштабирования проявляются в проблемах с эффективностью функционирования распределенных систем, вызванных ограниченной производительностью ее отдельных компонентов: серверов и сетевых соединений. Существуют несколько основных технологий, позволяющих уменьшить нагрузку на каждый компонент распределенной системы. К таким технологиям обычно относят *распространение* (англ. *distribution*), *репликацию* (англ. *replication*) и *кэширование* (англ. *caching*).

Распространение предполагает разбиение множества поддерживаемых ресурсов на части с последующим разнесением этих частей по компонентам системы. Простым примером распространения может служить распределенная файловая система при условии, что

каждый файловый сервер обслуживает свой набор файлов из общего адресного пространства. Другим примером может являться уже упоминавшаяся служба доменных имен DNS, в которой все пространство DNS-имен разбивается на зоны, и имена каждой зоны обслуживаются отдельным DNS-сервером.

Важную роль для обеспечения масштабируемости играют репликация и кэширование. Репликация не только повышает доступность ресурсов в случае возникновения частичного отказа, но и помогает балансировать нагрузку между компонентами системы, тем самым увеличивая производительность. Кэширование представляет собой особую форму репликации, когда копия ресурса создается в непосредственной близости от пользователя, использующего этот ресурс. Разница заключается лишь в том, что репликация инициируется владельцем ресурса, а кэширование – пользователем при обращении к этому ресурсу. Однако стоит отметить, что наличие нескольких копий ресурса приводит к другим сложностям, а именно к необходимости обеспечивать их *непротиворечивость* (англ. *consistency*), что, в свою очередь, может отрицательно сказаться на масштабируемости системы.

Таким образом, распространение и репликация позволяют распределить поступающие в систему запросы по нескольким ее компонентам, в то время как кэширование уменьшает количество повторных обращений к одному и тому же ресурсу.

Кэширование призвано не только снижать нагрузку на компоненты распределенной системы, но и позволяет скрывать от пользователя задержки коммуникации при обращении к удаленным ресурсам. Подобные технологии, скрывающие задержки коммуникации, важны для достижения географической масштабируемости системы. К ним, в частности, еще можно отнести механизмы асинхронной связи (англ. *asynchronous communication*), в которых клиент не блокируется при обращении к удаленной службе, а получает возможность продолжить свою работу сразу после обращения. Позже, когда будет получен ответ, клиентский процесс сможет прерваться и вызвать специальный обработчик для завершения операции.

Однако асинхронная связь применима не всегда. Например, в интерактивных приложениях пользователь вынужден ожидать реакции системы. В таких случаях можно воспользоваться технологиями *переноса кода*, когда часть кода приложения загружается на сторону клиента и исполняется локально для обеспечения быстрого отклика на действия пользователя. Преимущество подобных решений заключается в том, что они позволяют сократить количество сетевых взаимодействий и снизить зависимость работы приложения от случайных задержек обмена сообщениями через сеть. В настоящее время перенос кода широко используется в Интернете в форме апплетов Java и Javascript.

1.3.4. Сложности разработки распределенных систем

Реализация перечисленных выше свойств распределенных систем на практике представляет непростую задачу. Основные сложности вытекают именно из того факта, что компоненты распределенной системы территориально удалены друг от друга и выполняются на независимых компьютерах в сети. Игнорирование этого обстоятельства на этапе проектирования и разработки соответствующего программного обеспечения часто приводит к существенным проблемам в функционировании распределенной системы.

В этой связи еще в 90-х годах прошлого века сотрудник компании Sun Microsystems Питер Дейч в своей статье "Восемь заблуждений относительно распределенных вычислений" сформулировал основные ошибки, которые допускают при создании распределенных приложений. Он писал: "По существу, каждый, кто впервые создает распределенное приложение, делает следующие предположения. Все они, в конце концов, оказываются ложными, и все вызывают большие неприятности. Вот эти восемь заблуждений:

1. Сеть является надежной.
2. Задержки передачи сообщений равны нулю.
3. Полоса пропускания не ограничена.
4. Сеть является безопасной.
5. Сетевая топология неизменна.
6. Систему обслуживает только один администратор.
7. Издержки на транспортную инфраструктуру равны нулю.
8. Сеть является однородной".

Важно отметить, что перечисленные предположения как раз и отличают разработку традиционных локальных приложений и систем от распределенных в том смысле, что для нераспределенных приложений большинство из этих допущений остается верным, а проблемы, возникающие при нарушении любого из них, скорее всего, не будут проявляться. Бо*л*ьшая же часть принципов, лежащих в основе распределенных систем, имеет непосредственное отношение к перечисленным предположениям. Поэтому в области распределенных вычислений рассматриваются методы и алгоритмы, позволяющие реализовывать распределенные системы, при условии того, что хотя бы одно из представленных допущений является ложным. Например, понятно, что надежных сетей не существует, что, в свою очередь, приводит к невозможности достижения абсолютной прозрачности отказов. Другим примером может являться необходимость принимать во внимание доступную полосу пропускания и ненулевую задержку обмена сообщениями в сети при реализации и использовании механизмов репликации.

1.4. Понятие и назначение программного обеспечения промежуточного уровня

Для поддержки разнородных компьютеров и различных способов связи между ними распределенные системы часто реализуют в виде специального программного уровня, логически располагающего между прикладным программным обеспечением и операционными системами (ОС). Поэтому соответствующее программное обеспечение, собственно реализующее функционал распределенной системы, и называют *программным обеспечением промежуточного уровня* (англ. *middleware*). Этот логический программный уровень призван обеспечивать дополнительное абстрагирование приложений от базовых платформ и скрывать их неоднородность от пользователей и приложений, а также предоставлять подходящую модель программирования для разработчиков. При этом в ПО промежуточного уровня часто предпринимаются серьезные попытки замаскировать от разработчиков прикладного программного обеспечения распределение процессов и ресурсов по множеству компьютеров и скрыть нарушение некоторых допущений из представленных выше восьми заблуждений П. Дейча.

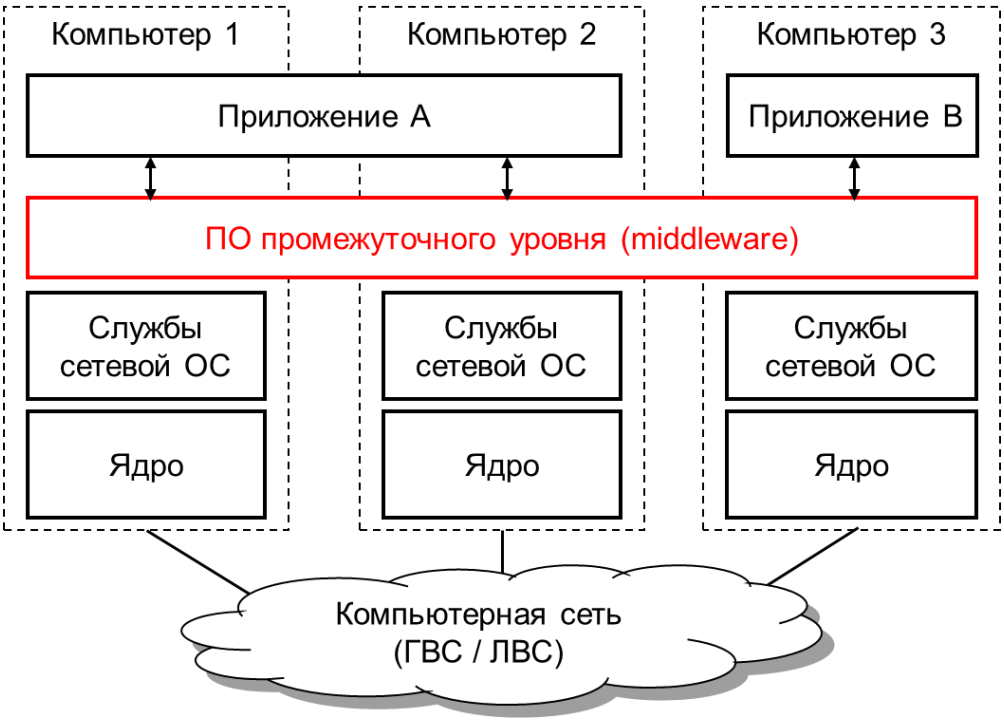


Рис. 1.2. Общая структура программных компонентов в распределенной системе. ПО промежуточного уровня выполняется на нескольких компьютерах и предоставляет приложениям единый унифицированный интерфейс.

На рис. 1.2 показаны три компьютера, объединенные в сеть, и два приложения, причем "Приложение A" является распределенным среди

компьютеров 1 и 2. ПО промежуточного уровня предоставляет всем приложениям единый унифицированный интерфейс, с помощью которого могут взаимодействовать как программные компоненты одного распределенного приложения А, так и различные приложения А и В между собой.

Стоит отметить, что без использования программного обеспечения промежуточного уровня приложения по-прежнему могут взаимодействовать, используя средства *сетевых операционных систем*. В этом случае связь приложений или программных компонентов одного распределенного приложения можно реализовать через операции с сокетами транспортного уровня, которые позволяют процессам на разных компьютерах обмениваться сообщениями с помощью простейших примитивов *send()* и *receive()*. Общая структура взаимодействия с использованием средств сетевых операционных систем приведена на рис. 1.3.

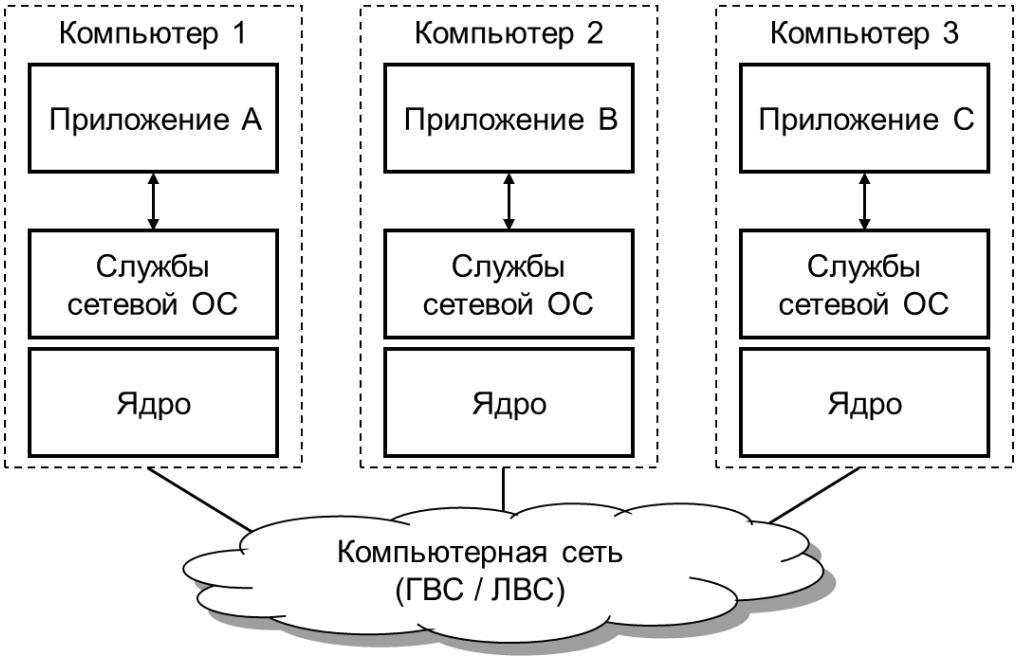


Рис. 1.3. Общая схема взаимодействия с помощью средств сетевой операционной системы.

Однако сложности такого подхода заключаются в том, что наличие распределения становится слишком очевидным. Совершенно ясно, что уровень абстракции сокетов оказывается недостаточным для написания эффективных приложений. В этом случае разработчику необходимо реализовывать взаимодействие в явном виде, а приложению – точно знать конечную точку соединения, т.е. где выполняется другой программный компонент, с которым осуществляется взаимодействие. Кроме того, если требуется, чтобы распределенное приложение работало на различных

операционных системах, разработчику пришлось бы иметь дело с разнообразными платформами. И наконец, с точки зрения эксплуатации подобных систем, отсутствие единого представления часто приводит к тому, что и управлять независимыми компьютерами приходится исключительно независимо. В этом случае страдает простота использования.

В свою очередь, основная задача программного обеспечения промежуточного уровня заключается в обеспечении прозрачности взаимодействия путем предоставления высокоуровневых средств связи, скрывающих низкоуровневую пересылку сообщений по компьютерной сети. В этом случае интерфейс программирования транспортного уровня, предоставляемый сетевой операционной системой, полностью заменяется другими средствами. Например, различные системы ПО промежуточного уровня поддерживают такие абстракции, как удаленный вызов процедур, обращения к удаленным объектам, групповую коммуникацию среди нескольких процессов, механизмы публикации / подписки, уведомления о событиях, прозрачное обращение к реплицированным ресурсам и др.

Поддерживаемые абстракции, по сути, определяют соответствующую модель (и систему) программирования, на которой базируется программное обеспечение промежуточного уровня и которая призвана обеспечивать удобство разработки и отладки распределенных приложений. В качестве примера ранних, но до сих пор популярных моделей ПО промежуточного уровня можно привести модель удаленного вызова процедур, основанную на реализации Sun RPC (англ. *Remote Procedure Call*), и модель групповой коммуникации, представленную в системе ISIS. Другой распространенной моделью является объектно-ориентированное ПО промежуточного уровня, когда приложение представляется в виде распределенных объектов (англ. *distributed objects*), размещенных на различных сетевых компьютерах и взаимодействующих посредством вызова методов, доступных через соответствующие интерфейсы. В этом случае вызывающий процесс может оказаться неосведомленным о том, что обращается к удаленному объекту, и о том, что при этом имеет место факт сетевого обмена. Примерами такого ПО промежуточного уровня являются архитектура CORBA (англ. *Common Object Request Broker Architecture*), созданная Object Management Group, модель удаленного обращения к методам в Java RMI (англ. *Remote Method Invocation*) и модель распределенных объектных компонентов DCOM (англ. *Distributed COM*) компании Microsoft, реализованная поверх различных операционных систем Windows.

Стоить отметить, что использование ПО промежуточного уровня не означает, что приложение не может напрямую обратиться к интерфейсам базовой операционной системы, однако такой подход не приветствуется, и в задачи распределенной системы входит предоставление более или менее

полного набора служб, позволяющего избежать подобного стремления. Например, архитектура CORBA включает в себя службы именования, безопасности, службы событий и уведомлений, службы транзакций службы сохранности и другие службы. Очевидно, в этом случае, реализуемые сервисы тесно связаны с моделью программирования, предоставляемой программным обеспечением промежуточного уровня.

Современные распределенные системы промежуточного уровня обычно создаются для работы на нескольких платформах. При этом прикладные приложения разрабатываются для конкретной системы промежуточного уровня и уже не зависят от платформы (операционной системы). Однако, к сожалению, эта независимость часто подменяется жесткой зависимостью от конкретного ПО промежуточного уровня, т.к. во многих случаях последнее значительно менее открыто, чем утверждается. Тем не менее, инструментарий, предоставляемый распределенными системами промежуточного уровня, оказывается настолько богатым, что целесообразность их применения очень часто не вызывает сомнения.

Таким образом, основная разница между сетевыми операционными системами и распределенными системами промежуточного уровня состоит в том, что в последних делается серьезная попытка максимально скрыть от пользователей и разработчиков прикладного программного обеспечения ее распределенную природу, т.е. добиться представления единой централизованной компьютерной системы.

1.5. Взаимодействие в распределенных системах

1.5.1. Физическое время

В распределенных системах вычисления осуществляются совокупностью независимых процессов, взаимодействующих посредством передачи сообщений для обмена данными и координации своих действий. Время, затрачиваемое каждым процессом на выполнение отдельных действий, различно для разных процессов и заранее неизвестно, а доставка отправленных сообщений занимает непредсказуемое время. Поэтому точность, с которой независимые процессы могут координировать свои действия, ограничена такими временными задержками и сложностью представления единого времени среди всех компьютеров в распределенной системе.

Каждый из компьютеров в распределенной системе имеет свои собственные часы, точнее говоря, системный таймер, который используется его локальными процессами для получения текущего времени. Поэтому два процесса, выполняющиеся на различных компьютерах, могут ассоциировать соответствующую отметку времени с каждым событием. Однако даже если эти процессы будут считывать показания своих часов в одно и то же время, возвращаемые значения,

скорее всего, будут отличаться. Дело в том, что ни одни часы не являются идеальными: кварцевые генераторы, составляющие основу часов для современных компьютеров, не могут иметь абсолютно одинаковую частоту, что ведет к постепенной потере синхронизации и возвращении таймерами различных значений при обращении к ним. Эта разница в показаниях часов называется *рассинхронизацией часов* (англ. *clock skew*), а скорость отклонения часов от точных показаний универсального скоординированного времени UTC (*Coordinated Universal Time*) с течением времени – *скоростью дрейфа* (англ. *clock drift rate*) – см. рис. 1.4.

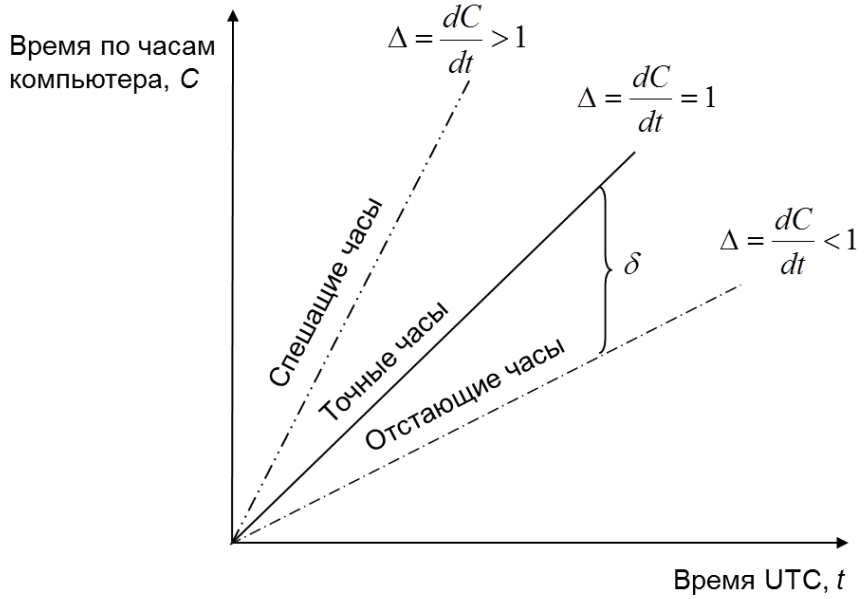


Рис. 1.4. Соотношение времени по часам компьютеров и времени UTC; δ – рассинхронизация часов, Δ – скорость дрейфа.

Таким образом, даже если в какой-то момент на часах всех компьютеров, входящих в распределенную систему, установлено одно и то же значение, с течением времени их показания станут существенно отличаться, если не предпринимать никаких дополнительных мер. В результате приложения, которые ожидают, что временная отметка, ассоциируемая с тем или иным событием, корректна и не зависит от компьютера, на которой она регистрировалась (т.е. часы которого использовались), могут работать неправильно. Чтобы пояснить это приведем следующий пример.

Банковская система. Рассмотрим задачу подсчета полной суммы денег, находящихся на счетах в разных филиалах банка. Предположим, что банковская система не позволяет вносить дополнительные денежные средства на счет филиала и снимать их наличными, а лишь осуществляет перевод различных сумм из одного филиала банка в другой с помощью сообщений.

Для иллюстрации работы такой банковской системы воспользуемся графиками процессов и событий, представленными на рис. 1.5. Горизонтальными линиями на этих графиках изображены временные оси. Точка на оси соответствует событию (например, внутреннему событию процесса, событию отправки или приема сообщения). Квадрат, обведенный вокруг точки, обозначает состояние локального процесса (определяемого состоянием счета данного филиала) в фиксированный момент времени, соответствующий этой точке. Стрелкой обозначается передача сообщения от одного процесса другому.

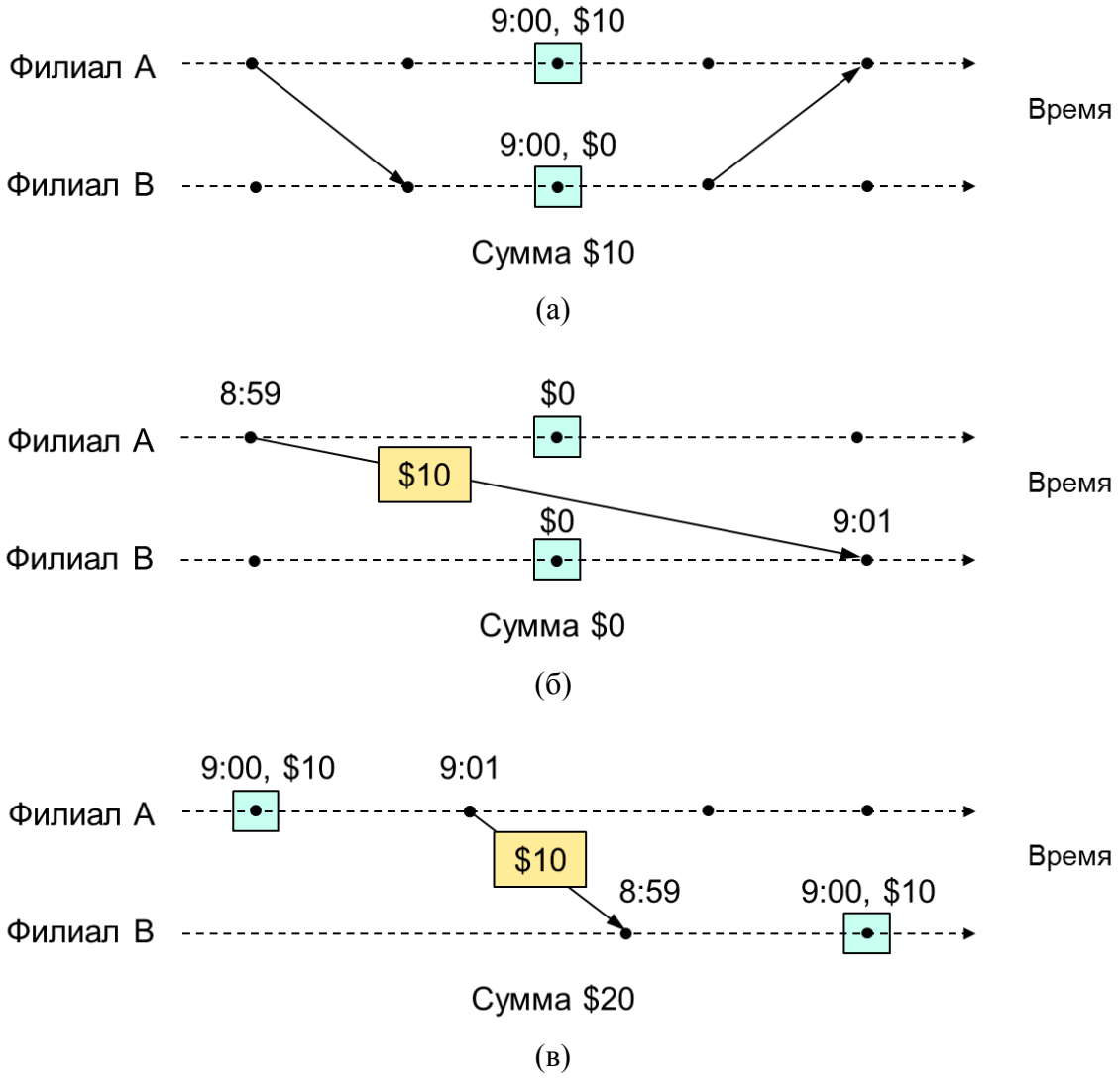


Рис. 1.5. Пример определения суммы денег на счетах банка.

Чтобы определить полную сумму банк должен иметь сведения о количестве денег в каждом из филиалов. Предположим, что требуется подсчитать денежные средства в 9:00 – время, известное всем процессам. В случае, показанном на рис. 1.5а, сумма окажется равной $10. Однако возможна также ситуация, представленная на рис. 1.5б, в которой на

момент определения полной суммы денежные средства филиала A находятся в состоянии передачи в филиал B, т.е. деньги уже отправлены филиалом A, но еще не получены филиалом B. В этом случае будет получен неправильный результат, равный $0. Эту проблему можно решить, изучая все сообщения, находящиеся в состоянии передачи в момент подсчета денег. Например, можно потребовать от филиала A хранить записи обо всех отправленных переводах и их получателях, а от филиала B – хранить записи о всех полученных переводах и их отправителях. Тогда в "состояние" филиала нужно включить не только информацию о количестве денег на его счете, но и указанные записи обо всех отправленных и полученных денежных переводах. Проверяя такие "состояния" процессов, система обнаружит перевод, покинувший филиал A, но еще не полученный филиалом B, и соответствующие средства можно будет добавить в общую сумму. Важно отметить, что денежные средства, уже полученные филиалом B, отдельно учитывать не нужно, т.к. они будут отражены в счете этого филиала.

К сожалению, подход, представленный выше, не гарантирует вычисление верного результата. В примере, приведенном на рис. 1.5в, часы в разных филиалах идут с некоторым относительным сдвигом, т.е. имеет место небольшая рассинхронизация часов. Проверка состояния счета филиала A в момент времени 9:00 дает результат, равный $10. Впоследствии, в 9:01 по часам филиала A, эти деньги переводятся в филиал B, часы которого на момент их получения показывают 8:59. Поэтому при подсчете общей суммы по состоянию на 9:00 эти денежные средства будут учитываться дважды.

1.5.2. Синхронные и асинхронные распределенные системы

В распределенных системах практически невозможно предсказать время выполнения отдельных действий различных процессов, скорость отклонения их локальных часов от точных показаний или задержку доставки сообщения от одного процесса к другому. В этой связи выделяют две диаметрально противоположные модели вычислительных систем: первая накладывает строгие временные ограничения на соответствующие характеристики, во второй не делается никаких предположений относительно времени.

Синхронные распределенные системы. Синхронные распределенные системы – это системы, в которых определены следующие временные ограничения.

- Время выполнения каждого отдельного действия любого процесса ограничено снизу и сверху известными значениями.
- Задержка доставки каждого сообщения от одного процесса к другому не превышает известный предел.

- Каждый процесс имеет свои локальные часы со скоростью отклонения от точных показаний, не превышающей известное значение.

Важно отметить, что для большинства случаев, скорее всего, можно дать некоторые оценки перечисленным временным характеристикам. Однако если на практике все же невозможно гарантировать их выполнение, например, обеспечить доставку всех сообщений в пределах заданного времени, то любые системы или алгоритмы, построенные на основе предполагаемых значений представленных выше ограничений, не будут работать в случае их нарушения.

Благодаря скоординированной работе всех процессов в синхронных системах разрабатывать и отлаживать алгоритмы взаимодействия для этой модели вычислительных систем гораздо проще. Например, в синхронных системах возможно использование таймаутов для определения отказа одного из процессов.

Асинхронные распределенные системы. Многие из распределенных систем, например, построенные на базе сети Интернет, по своей природе не являются синхронными. Поэтому очень часто используется другая модель – асинхронные распределенные системы, в которых не накладывается никаких ограничений на рассматриваемые характеристики, перечисленные ниже.

- Скорость выполнения операций: время выполнения каждого отдельного действия любого процесса конечно, но не имеет известной верхней границы.
- Задержка доставки сообщений: сообщение может быть доставлено через произвольное, но конечное время после его отправки. Например, сообщение от процесса А к процессу В может быть доставлено за несколько миллисекунд, а сообщение от процесса В к процессу С – за несколько минут.
- Скорость отклонения часов также может быть произвольной.

Основная причина асинхронности большинства распределенных систем кроется в совместном использовании несколькими процессами одного процессора и несколькими логическими каналами связи одного физического соединения. Соответственно в случае, когда слишком много процессов неизвестной природы и поведения работают на одном и том же процессоре, результирующая производительность любого из них не может быть гарантирована. Это в точности описывает работу приложений в сети Интернет, где, по сути, нагрузка на сервера и сетевые соединения никак не ограничена. Поэтому, к примеру, нельзя предсказать, сколько времени займет передача файла по протоколу FTP с сервера на компьютер пользователя. Необходимо отметить, что наибольший вклад в задержку доставки сообщения оказывает не собственно время распространения сигнала, а время, затрачиваемое различными уровнями стека протоколов

на обработку операций отправки и получения, а также задержка, вносимая промежуточными узлами на пути передачи сообщения.

Преимущество асинхронной модели заключается в том, что построенные на ее основе распределенные системы и алгоритмы являются более устойчивыми к изменению условий их функционирования и, как следствие, оказываются более универсальными. Однако существует множество архитектурных и алгоритмических проблем, которые не могут быть разрешены в предположении асинхронной модели, в то время как они имеют решение для синхронных систем. Рассмотрим следующий пример задачи двух генералов, призванный проиллюстрировать сложности достижения соглашения в асинхронных системах.

Соглашение в задаче двух генералов. Две армии, каждая руководимая своим генералом, готовятся к штурму города. Лагери обеих армий располагаются на двух холмах, разделенных долиной. Единственным способом связи между генералами является отправка посыльных с сообщениями через долину. Коммуникацию будем считать надежной, т.е. посыльные не могут быть перехвачены и уничтожены противником. Для успешного штурма генералы должны атаковать город вместе. При этом им необходимо согласовать, кто из них будет возглавлять штурм, и в какое время необходимо его начинать.

Легко увидеть, что даже в асинхронной системе, генералы могут прийти к соглашению, кто из них будет возглавлять штурм. Например, каждый из генералов отправляет другому посыльного с информацией о количестве солдат, входящих в его армию. Тот генерал, у которого солдат в армии больше, и будет возглавлять штурм (в случае, если в армиях одинаковое количество солдат, есть договоренность, что возглавлять штурм будет генерал из первой армии). Но открытым остается вопрос, когда начинать штурм? К сожалению, в асинхронных системах посыльные могут добираться до другой армии сколь угодно долго. Предположим, первый генерал отправляет второму сообщение "Атакуем!". Второй может так и не получить это сообщение, скажем, в течение 3 часов, или, все же, посыльный добрался до него за 10 минут? В синхронных системах проблема по-прежнему остается, но генералы уже обладают некоторой дополнительной информацией: посыльный доберется до другой армии за время, не меньшее, чем *min* минут, и не превышающее *max* минут. Поэтому, в этом случае первый генерал может отправить сообщение "Атакуем!", подождать *min* минут и выступать в атаку. Вторая армия после получения сообщения может подождать 1 минуту и тоже выступать в атаку. При этом гарантируется, что вторая армия выступит после первой руководящей армии, но придет вслед за ней не более чем через (*max* – *min* + 1) минут.

1.5.3. Упорядочивание событий

Очень часто имеет значение не точное время наступления того или иного события (например, события отправки или получения сообщения), а порядок, в котором эти события происходили, т.е. требуется определить произошло ли данное событие в данном процессе до или после другого события в другом процессе. В этих случаях выполнение процессов можно описывать в терминах последовательности событий в условиях отсутствия знания точного времени их происхождения.

Рассмотрим следующий пример групповой рассылки сообщений электронной почты между пользователями A, B, C и X. Пользователь A отправляет всей группе письмо с темой "Общее собрание". Пользователи B и C отвечают на него всей группе своими сообщениями с темой "Re: Общее собрание".

В действительности события происходят в следующей последовательности:

1. Первым отправляется сообщение от пользователя A.
2. Пользователь B получает его, читает и отправляет ответ.
3. Пользователь C получает оба сообщения от A и B и затем отправляет свой ответ, опирающийся на оба сообщения от A и B.

Однако в связи с произвольными и независимыми задержками доставки сообщений, некоторые пользователи могут видеть другую последовательность наступления событий. Например, согласно сценарию, приведенному на рис. 1.6, в почтовом ящике пользователя X сообщения будут располагаться в следующем порядке:

1. Сообщение от пользователя C с темой "Re: Общее собрание".
2. Сообщение от пользователя A с темой "Общее собрание".
3. Сообщение от пользователя B с темой "Re: Общее собрание".

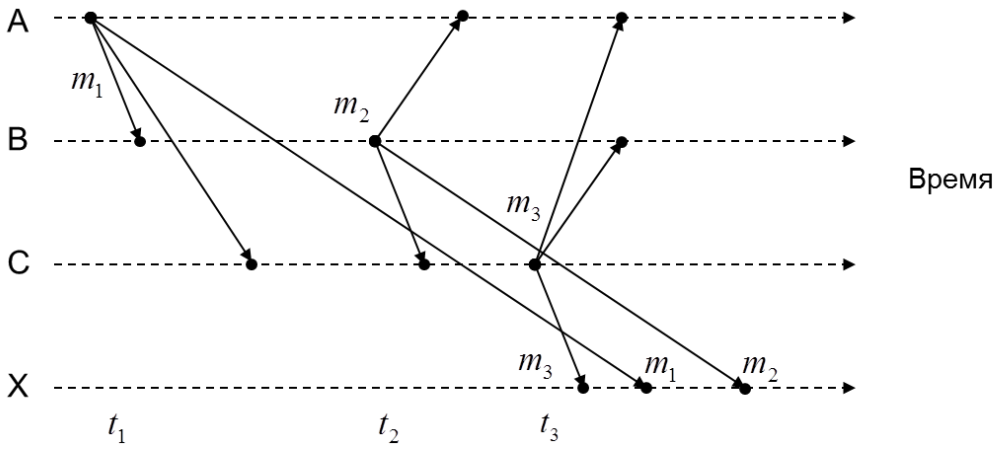


Рис. 1.6. Порядок сообщений, наблюдаемый различными пользователями.

В случае, если часы компьютеров пользователей A, B и C можно было бы точно синхронизировать, каждое отправляемое сообщение могло

бы содержать отметку времени отправки согласно локальным часам компьютера-отправителя. Для нашего примера, сообщения m_1, m_2 и m_3 содержали бы в себе отметки времени t_1, t_2 и t_3, где $t_1<t_2<t_3$. Тогда в почтовом ящике пользователя X полученные сообщения можно было бы отображать согласно порядку, задаваемому их временными отметками.

Однако, как уже обсуждалось, в распределенных системах часы не могут быть синхронизированы с абсолютной точностью. Если рассинхронизация часов невелика, то описанный выше подход с использованием отметок реального времени в большинстве случаев будет давать верный результат. Но не всегда. Для преодоления проблем, связанных с использованием реального времени, Л. Лэмпорт предложил модель логических часов (англ. *logical clock*), отсчитывающих логическое время (англ. *logical time*), которая может быть применена для упорядочивания событий, происходящих в различных процессах распределенной системы. Логическое время позволяет определить порядок событий (или порядок сообщений для рассматриваемого примера рассылки электронной почты) без обращения к часам компьютеров. Вопросы, связанные с моделью логического времени, мы рассмотрим далее в соответствующем разделе.

1.5.4. Примитивы взаимодействия

Базовыми примитивами взаимодействия в распределенных системах являются примитивы *send()* и *receive()*, позволяющие соответственно отправлять и принимать сообщения. В простейшем случае примитив *send()* имеет по меньшей мере три параметра: идентификатор процесса-получателя сообщения, указатель на буфер с передаваемыми данными в адресном пространстве процесса-отправителя и количество передаваемых элементов данных конкретного типа. Например, функция отправки данных может иметь следующий вид:

```
send(void *sendbuf, int count, int dest),
```
где

`sendbuf` – указывает на буфер, содержащий передаваемые данные,
`count` – содержит количество передаваемых элементов данных,
`dest` – идентифицирует процесс-получатель.

В свою очередь вызов *receive()* также должен содержать как минимум три аргумента: идентификатор процесса-отправителя, от которого разрешается принимать сообщения (возможно использование значения, позволяющего принимать сообщения от любого процесса), указатель на буфер в адресном пространстве процесса-получателя, куда следует помещать принимаемые данные, и количество принимаемых элементов данных:

`receive(void *recvbuf, int count, int source)`, где

`recvbuf` – указывает на буфер, сохраняющий принимаемые данные,
`count` – содержит количество принимаемых элементов данных,
`source` – идентифицирует процесс-отправитель.

Для анализа работы команд *send()* и *receive()* рассмотрим следующий пример кода, в котором процесс P_1 отправляет сообщение процессу P_2, а процесс P_2 выводит полученное сообщение в стандартный выходной поток.

P_1	P_2

`a = 100;`
`send(&a, 1, 2);`
`a = 0;`
	`receive(&a, 1, 1);`
	`printf("%d\n", a);`

Стоит обратить внимание, что процесс P_1 изменяет значение переменной "a" сразу после команды *send()*. Очевидно, что семантика операции *send()* должна обеспечивать получение процессом P_2 значения 100, а не нуля. Другими словами, процесс P_2 должен получить значение переменной "a" на момент вызова команды *send()*. В то же время большинство современных компьютерных архитектур доверяют функции приема-передачи данных сетевым интерфейсам и не подразумевают участия центрального процессора (ЦП) в пересылке данных. В результате, если команда *send()* попросту дает задание коммуникационной подсистеме на отправку данных и сразу же возвращает управление вызвавшему ее процессу, не исключены ситуации, когда процесс P_2 получит значение ноль вместо ожидаемого 100.

Простым решением указанной проблемы является возврат управления в отправляющий процесс P_1 только тогда, когда он не окажет влияния на семантику передачи данных. Важно отметить, что под этим не обязательно подразумевается возвращение в вызвавший процесс только после того, как принимающий процесс получит все передаваемые данные. Это всего лишь означает, что отправитель блокируется до момента, когда выполнение семантики операции *send()* сможет быть гарантировано вне зависимости от дальнейшего хода выполнения приложения. Существует два механизма обеспечения этого требования.

Блокирующие операции отправки/ получения без буферизации. В этом случае возврат из команды *send()* не осуществляется до тех пор, пока не будет вызвана соответствующая команда *receive()* в принимающем процессе и пока все данные не будут переданы процессу-получателю. Подобный механизм взаимодействия обычно подразумевает

дополнительный обмен сигналами для установления связи (англ. *handshake*) между отправляющим и принимающим процессами. А именно, когда процесс-отправитель достигает состояния готовности к обмену данными, он отправляет запрос на передачу данных процессу-получателю и переходит в состояние ожидания до тех пор, пока не получит от него соответствующего ответа. Процесс-получатель отвечает на данный запрос только после того, как он сам достигнет состояния готовности к приему данных и вызовет команду *receive()*. Собственно передачу данных процесс-отправитель начинает только после получения ответа от принимающего процесса, что свидетельствует о готовности последнего к приему данных. Описанная схема взаимодействия представлена на рис. 1.7, где А – передающий процесс, а В – принимающий процесс, и носит название *блокирующей отправки/ получения без буферизации*, т.к. в этом случае не используется никаких дополнительных буферов, как на стороне отправителя, так и на стороне получателя.

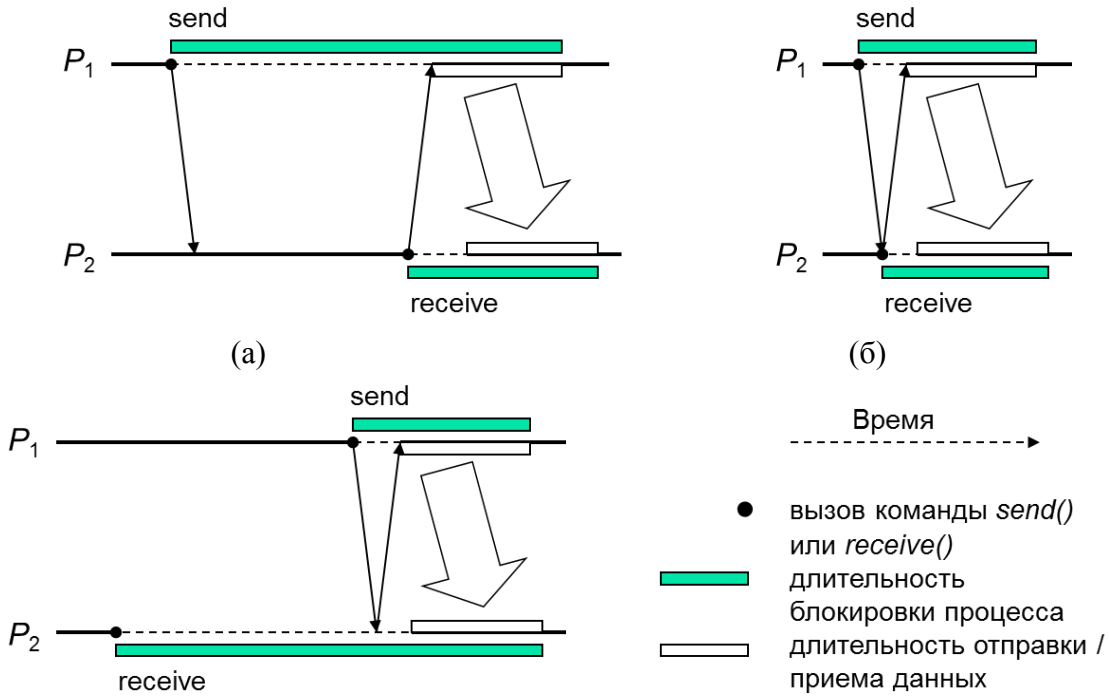


Рис. 1.7. Механизм блокирующей отправки/ получения без буферизации; (а) первым осуществляется вызов *send()*, (б) вызовы *send()* и *receive()* происходят практически одновременно, (в) первым осуществляется вызов *receive()*.

Из рис. 1.7 видно, что блокирующая отправка/ получение без буферизации неплохо работает, когда команды *send()* и *receive()* вызываются приблизительно в одно и то же время. Однако в асинхронных системах достичь подобной синхронности в действиях нескольких процессов практически невозможно. Поэтому накладные расходы,

связанные с ожиданием и соответствующим простоем процессов, становятся основным недостатком данного механизма взаимодействия.

Кроме того, при использовании блокирующей отправки / получения без буферизации возможны ситуации, приводящие к *взаимной блокировке* процессов, или *тупиковой ситуации* (англ. *deadlock*). Рассмотрим следующий простой пример обмена сообщениями, демонстрирующий возникновение тупиковой ситуации:

P_1	P_2
`send(&a, 1, 2);` `receive(&b, 1, 2);`	`send(&a, 1, 1);` `receive(&b, 1, 1);`

Приведенный выше фрагмент кода призван обеспечить передачу значений переменных "a" между процессами P_1 и P_2. Однако в случае, если команды *send()* и *receive()* реализованы согласно механизму блокирующей отправки/ получения без буферизации, то примитив *send()* в процессе P_1 будет ожидать вызова соответствующей команды *receive()* в процессе P_2, а примитив *send()* в процессе P_2 будет ожидать вызова *receive()* в P_1. В результате оба процесса будут находиться в состоянии ожидания и оказываются заблокированными навсегда.

Блокирующие операции буферизованной отправки/ получения. Простым решением проблем, связанных с возникновением взаимных блокировок, указанных выше, и простоем процессов, ожидающих готовности к приему-передаче данных от других процессов, является использование дополнительных буферов на отправляющей и принимающей стороне. В этом случае передаваемые данные просто копируются из буфера процесса-отправителя в дополнительный буфер отправки (обычно размещаемый в адресном пространстве операционной системы). При этом команда *send()* возвращает управление вызвавшему ее процессу сразу после того, как завершится операция копирования, и процесс-отправитель сможет продолжать свою работу, т.к. любые изменения данных с его стороны уже не будут влиять на семантику передачи сообщения.

Стоить отметить, что принимаемые данные не могут быть сохранены непосредственно в адресном пространстве процесса-получателя пока он не будет готов к приему и не вызовет команду *receive()*. Иначе его нормальное исполнение может быть нарушено. Поэтому наличие дополнительного буфера приема на стороне получателя, куда следует помещать принимаемые данные, также является обязательным. Когда же процесс-получатель достигнет состояния готовности к приему данных, он вызовет команду *receive()*, которая проверит буфер приема, и если в нем

находится новое сообщение, скопирует его содержимое в целевой буфер в адресном пространстве принимающего процесса.

Описанная выше схема взаимодействия опирается на наличие двух дополнительных буферов отправки и приема, соответственно, на стороне отправителя и на стороне получателя. Кроме того подразумевается, что сам обмен данными осуществляется специальной коммуникационной подсистемой, работающей независимо от исполняемых процессов. Если подобные средства недоступны, то проблемы блокирующей отправки/ получения без буферизации могут быть решены путем введения только одного буфера, располагающегося на одной стороне, например, на стороне получателя. В этом случае операция *send()* должна будет прервать работу принимающего процесса, оба процесса должны будут самостоятельно осуществить передачу и сохранение данных в дополнительный буфер приема на стороне получателя, после чего процесс-получатель продолжит работу. В дальнейшем, когда принимающий процесс вызовет команду *receive()*, данные будут скопированы из буфера приема в целевой буфер в адресном пространстве получателя.

Механизм блокирующего буферизованного взаимодействия для случая двух дополнительных буферов отправки и приема иллюстрируется рис. 1.8.

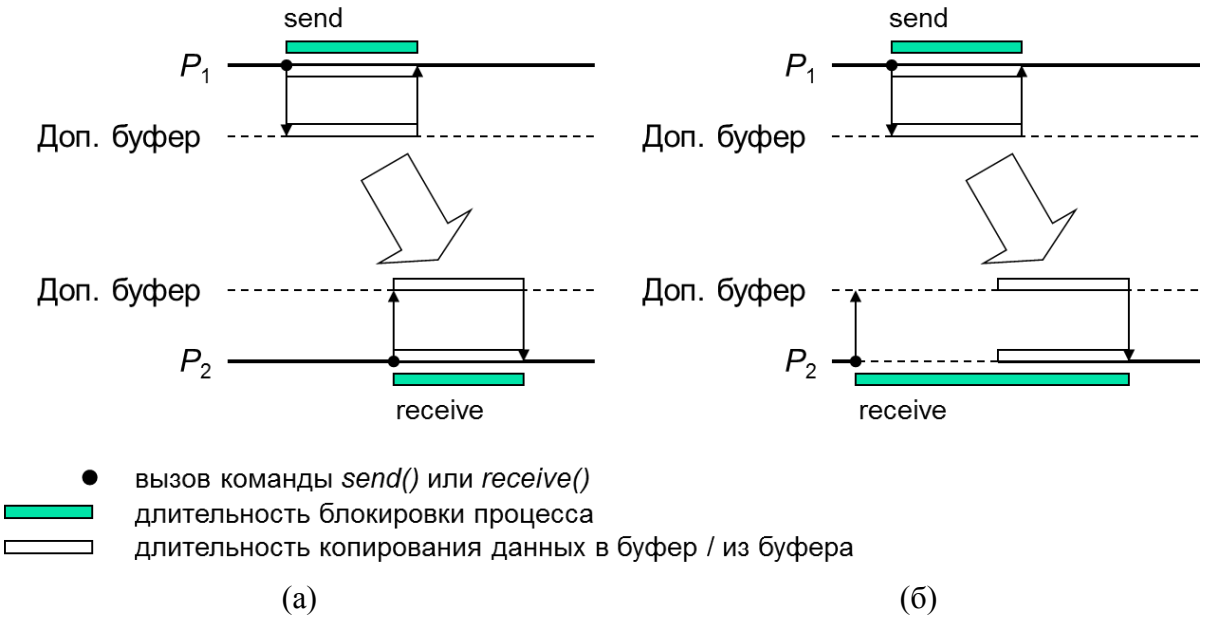


Рис. 1.8. Механизм блокирующей буферизованной отправки/ получения;
(а) первым осуществляется вызов *send()*, (б) первым осуществляется вызов *receive()*.

Легко видеть, что при использовании механизма буферизованной отправки и получения вместо издержек, вызванных простоем процессов, ожидающих приема или передачи данных, появляются накладные расходы, связанные с выделением дополнительной памяти, копированием данных в буфер / из буфера, управлением дополнительными буферами. В случае,

когда распределенная система является синхронной (т.е. когда команды *send()* и *receive()* вызываются приблизительно в одно и то же время), отправка и получение данных без буферизации может быть даже более эффективной и производительной, чем буферизованная передача. Тем не менее, для более общего случая асинхронных систем буферизованное взаимодействие будет являться предпочтительным до тех пор, пока емкость буфера не станет ограничивающим фактором. Для понимания влияния буферов конечного размера на передачу сообщений рассмотрим следующий пример:

P_1	P_2

`for (i = 0; i < 1000; i++)`
`{`
`produce_data(&a);`
`send(&a, 1, 2);`
`}`
	`for (i = 0; i < 1000; i++)`
	`{`
	`receive(&a, 1, 1);`
	`consume_data(&a);`
	`}`

В этом фрагменте кода процесс P_1 производит 1000 элементов данных, а процесс P_2 потребляет их. Предположим, что процесс P_2 исполняется существенно медленнее, чем процесс P_1. Если в дополнительных буферах отправки и приема достаточно места для размещения всех производимых элементов данных, то оба процесса могут выполняться без проблем. Однако, если свободного пространства в буфере недостаточно, возникает ситуация переполнения буфера. В зависимости от реализации команды *send()*, она может либо вернуть ошибку вызвавшему процессу, либо заблокировать отправителя до тех пор, пока не будет вызвана соответствующая команда *receive()*, освобождающая место в буфере. В первом случае ответственность за обработку таких ошибочных ситуаций ложится на разработчика ПО, и в приложении должны быть реализованы дополнительные механизмы синхронизации между отправляющим и принимающим процессами. Во втором случае возможная блокировка отправляющего процесса может приводить к непредсказуемому снижению производительности системы. Поэтому хорошим правилом является написание программ с ограниченными требованиями к размеру буферов.

В то время как буферизация помогает избежать многих тупиковых ситуаций при исполнении процессов, написание кода, приводящего к взаимным блокировкам, все равно остается возможным. Дело в том, что как и в случае взаимодействия без буферизации, вызовы команды *receive()* по-прежнему являются блокирующими для сохранения семантики передачи сообщений. Поэтому фрагмент кода, представленный ниже,

приведет к взаимной блокировке, т.к. оба процесса будут ожидать поступления данных, но ни один из них не будет их передавать:

P_1	P_2
`receive(&a, 1, 2);` `send(&b, 1, 2);`	`receive(&a, 1, 1);` `send(&b, 1, 1);`

Стоит еще раз отметить, что в данном примере взаимная блокировка вызвана ожиданием завершения команд *receive()*.

Тупиковые ситуации также могут возникать при использовании команды *send()*, блокирующей отправителя до вызова соответствующей команды *receive()* в случае если свободного пространства в буфере оказывается недостаточно. Еще раз рассмотрим пример обмена сообщениями между двумя процессами:

P_1	P_2
`send(&a, 1, 2);` `receive(&b, 1, 2);`	`send(&a, 1, 1);` `receive(&b, 1, 1);`

Команда *send()* вернет управление вызывающему процессу, когда передаваемые данные будут скопированы из памяти процесса-отправителя в дополнительный буфер отправки. Если хотя бы одно сообщение сможет быть скопировано в дополнительный буфер отправки, то представленный выше код выполнится успешно. Если же свободного пространства в обоих буферах окажется недостаточно, возникнет взаимная блокировка процессов. Как и в предыдущем случае блокирующей отправки/ получения без буферизации подобные ситуации циклического ожидания должны быть распознаны и разрешены.

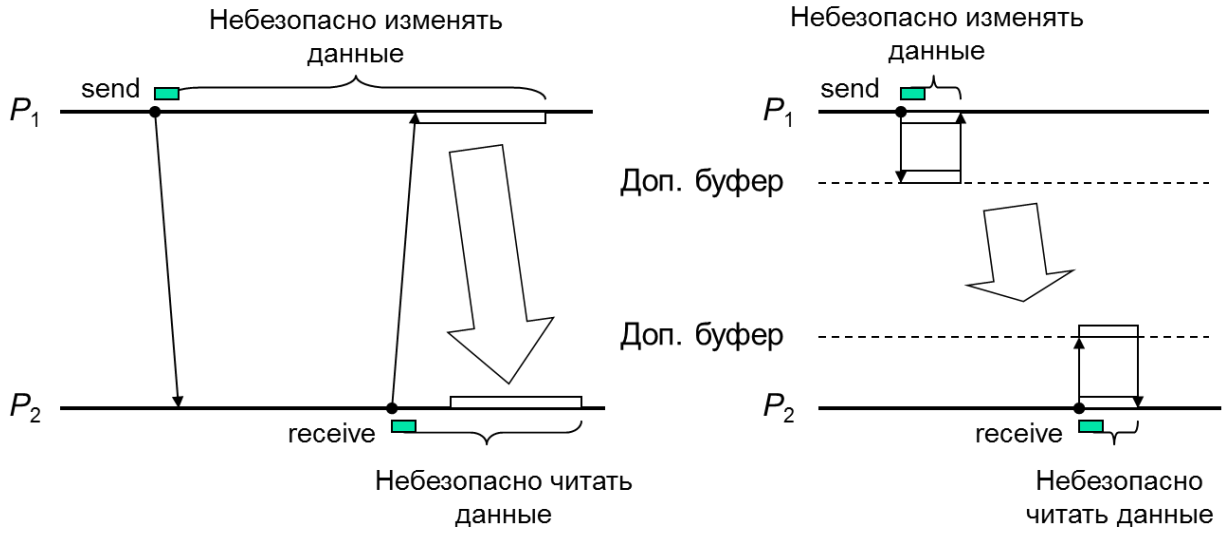
Стоит отметить, что возникновение взаимных блокировок при использовании блокирующих примитивов взаимодействия – явление нередкое, и предотвращение подобного циклического ожидания требует аккуратности и внимательности от разработчиков ПО.

Неблокирующие операции отправки/ получения. При использовании блокирующих примитивов взаимодействия семантика операции передачи данных обеспечивается либо за счет простоя процессов при обмене данными, либо за счет использования дополнительных буферов. Инициируемый такими операциями переход процесса в следующее состояние после отправки или получения сообщения завершается до того, как процессу возвращается управление. Поэтому блокирующие операции *send()* и *receive()* можно считать атомарными.

В некоторых случаях ответственность за обеспечение семантики передачи данных можно переложить на разработчика ПО и, тем самым, обеспечить быстрое выполнение операций *send()* и *receive()* без дополнительных издержек. Для таких неблокирующих примитивов отправки/ получения возврат управления в исполняемый процесс осуществляется сразу после вызова соответствующей команды и, следовательно, до того, как выполнение семантики передачи данных может быть гарантировано, и до того, как процесс перейдет в состояние после отправки или получения сообщения. Другими словами, вызов команды *send()* и *receive()* лишь *начинает* соответствующую операцию. Поэтому разработчик должен внимательно следить за тем, чтобы его код не изменял отправляемые данные и не использовал принимаемые данные, до завершения команд *send()* и *receive()*.

Неблокирующие операции отправки/ получения дополняются командами, позволяющими проверить признак их завершения. Сразу после вызова неблокирующей операции *send()* или *receive()* исполнение процесса будет безопасным, если оно не зависит от передаваемых или получаемых данных. Позже, процесс сможет проверить завершено ли выполнение неблокирующей операции отправки или приема, и при необходимости дождаться ее завершения. Таким образом, при правильном использовании неблокирующие примитивы позволяют скрыть издержки обмена данными за счет других вычислений.

Как показано на рис. 1.9 существуют механизмы неблокирующей отправки / получения как без буферизации, так и с использованием дополнительных буферов на стороне отправителя и / или на стороне получателя.



(а) (б)

Рис. 1.9. Механизмы неблокирующей отправки/ получения;
(а) без буферизации, (б) с использованием буферов.

В случае взаимодействия без буферизации, процесс-отправитель направляет получателю запрос на передачу данных и продолжает свою работу. Этот запрос будет ожидать от процесса-получателя готовности к приему данных и соответствующего вызова команды *receive()*, после чего начнется собственно передача данных. Взаимодействующие процессы смогут узнать об окончании сетевого обмена и, следовательно, о возможности работать с передаваемыми и принимаемыми данными без риска их повреждения с помощью команд, проверяющих признак завершения операций *send()* или *receive()*. Эта ситуация иллюстрируется рис. 1.9а.

В случае буферизованного взаимодействия, отправитель инициирует копирование передаваемых данных из своего буфера в дополнительный буфер отправки и сразу продолжает работать. При этом дальнейшая обработка передаваемых данных процессом-отправителем становится безопасной в момент завершения операции копирования. На стороне получателя команда *receive()* инициирует перемещение данных из дополнительного буфера приема в адресное пространство получателя и возвращает управление в вызвавший процесс. Стоит отметить, что возращение управления произойдет даже в том случае, если данные еще не поступили от отправителя в дополнительный буфер приема. При этом запрос от команды *receive()* будет ожидать поступления данных. Как видно из рис. 1.9б, использование механизма неблокирующей буферизованной отправки/ получения позволяет уменьшить время, в течение которого работа с передаваемыми и принимаемыми данными является небезопасной.

Таким образом, блокирующие примитивы взаимодействия обеспечивают простоту программирования и гарантируют выполнение семантики передачи данных, в то время как неблокирующие операции применимы для увеличения производительности компьютерных систем за счет маскирования издержек обмена данными. Однако в последнем случае разработчикам ПО приходится следить за тем, чтобы приложение не обращалось к передаваемым и принимаемым данным до завершения операций отправки или приема. Поэтому большинство современных библиотек, реализующих операции обмена сообщениями, обычно предоставляют как блокирующие, так и неблокирующие примитивы.

1.5.5. Синхронный и асинхронный обмен сообщениями

Основной смысл определения блокирующих и неблокирующих операций отправки и приема данных заключается в описании *локального* поведения вызывающего процесса без учета хода выполнения других процессов. А именно, блокирующие примитивы возвращают управление вызывающему процессу при завершении копирования данных из адресного пространства процесса-отправителя или в адресное

пространство процесса-получателя, неблокирующие – лишь начинают соответствующую операцию. Имеет смысл рассматривать взаимодействие также с *глобальной* точки зрения. В этом контексте различают синхронный и асинхронный механизм обмена сообщениями.

Синхронный обмен сообщениями. При синхронном обмене сообщениями отправитель и получатель дожидаются друг друга для передачи каждого сообщения, и операция отправки считается завершенной только после того, как получатель закончит прием сообщения.

Завершение синхронной операции *send()* свидетельствует не только о том, что семантика передачи данных оказывается обеспеченной, но и о том, что получатель достиг определенной точки в ходе своего выполнения, а именно вызова соответствующей операции *receive()*. Поэтому процессы не только обмениваются данными, но и синхронизируют свое выполнение во времени.

Другое преимущество синхронного обмена сообщениями заключается в том, что для передачи сообщения не требуется использования дополнительных буферов, т.к. синхронная операция *send()* все равно не сможет завершиться, пока не будет вызвана соответствующая операция *receive()*. Хотя, конечно, возможна реализации этих примитивов и с использованием буферизации. Важно отметить, что синхронная операция *receive()* после получения сообщения должна отослать соответствующее подтверждение (англ. *acknowledgement*) отправителю для завершения операции *send()*.

В случае использования блокирующих примитивов взаимодействия передача сообщения с помощью пары вызовов *send()* и *receive()* может рассматриваться как одна атомарная операция.

Асинхронный обмен сообщениями. При асинхронном обмене сообщениями не происходит никакой координации между отправителем и получателем сообщения. Для завершения операции *send()* отправителю не требуется дожидаться приема сообщения процессом-получателем. При отправке нового сообщения, отправителю неизвестно, получено ли его предыдущее сообщение, направленное этому же или, возможно, другому получателю. Поэтому, если канал связи между отправителем и получателем не сохраняет порядок передаваемых по нему сообщений, т.е. не обеспечивает свойство FIFO (англ. *First In First Out*), получатель может принимать сообщения в другом порядке, нежели они были переданы отправителем. Асинхронной операции *receive()* нет необходимости отсылать отправителю подтверждение (англ. *acknowledgement*) о приеме сообщения.

Преимуществом асинхронного обмена сообщениями является возможность перекрывать вычисления отправителя и получателя во времени, т.к. процессы не будут ожидать друг друга для передачи каждого сообщения.

В связи с тем, что отправка и прием сообщения не синхронизированы во времени, передача сообщения с помощью блокирующих примитивов требует наличия дополнительных буферов для размещения отправленных, но еще не полученных сообщений. Как уже обсуждалось, в этом случае необходимо определить поведение операции *send()* в ситуациях переполнения буфера. Если блокировать отправителя до освобождения требуемого пространства, то непредсказуемо может возникнуть ситуация взаимной блокировки процессов. Если сообщение отбрасывать, то коммуникацию нельзя считать надежной.

РАЗДЕЛ 2. МОДЕЛЬ РАСПРЕДЕЛЕННОГО ВЫЧИСЛЕНИЯ

Для изучения различных аспектов функционирования распределенных систем часто используют несколько моделей распределенной обработки данных. При этом та или иная модель выбирается в зависимости от исследуемой задачи из области распределенных вычислений.

Распределенное вычисление (не следует путать с областью *распределенных вычислений* теории вычислительных систем) удобно рассматривать в виде совокупности дискретных событий, каждое из которых вызывает небольшое изменение состояния всей системы. Система становится "распределенной" благодаря тому обстоятельству, что каждое событие приводит к изменению только *части* глобального состояния всей системы. А именно, при наступлении события изменяется лишь локальное состояние одного процесса и, возможно, состояние одного или нескольких каналов связи (или локальные состояния некоторого подмножества процессов, взаимодействующих при помощи синхронного механизма обмена сообщениями).

В данном разделе мы представим модель распределенной обработки данных и несколько основных понятий и концепций, которые будут использованы в дальнейшем. Мы будем рассматривать асинхронные распределенные системы, в которых процессы взаимодействуют с помощью асинхронного механизма обмена сообщениями. Описание модели и доказательство утверждений проводится до определенной степени неформально, но на уровне, достаточном для понимания существенных особенностей функционирования распределенных систем, выделяющих их среди систем других классов.

2.1. Модель распределенной системы

Распределенная система состоит из конечного множества N независимых процессов $\{P_1, P_2, \ldots P_N\}$. Если процесс P_i может напрямую отправлять сообщения процессу P_j, то мы будем говорить, что между процессом P_i и процессом P_j имеется *канал* C_{ij}. Для удобства все каналы будем считать однонаправленными. Чтобы два процесса имели возможность обмениваться сообщениями, между ними необходимо установить два разнонаправленных канала, каждый из которых служит для передачи сообщений в одном направлении. Структуру распределенной системы можно представить в виде ориентированного графа, в котором вершины графа соответствуют процессам, а ребра – каналам связи между процессами. Указанный граф также называется *топологией сети* распределенной системы. Далее, если не оговорено противное, мы будем

подразумевать, что топология является полносвязной, т.е. любые две вершины соединены ребром, и каждый процесс может непосредственно взаимодействовать со всеми другими процессами.

Состояние канала C определяется совокупностью сообщений отправленных по этому каналу, но еще не полученных принимающим процессом. (*Внутреннее*) *состояние процесса P* по-своему формализуется в контексте каждой решаемой задачи. Неформально можно сказать, что в своем состоянии процесс фиксирует свое "концентрированное прошлое", т.е. все существенные для рассматриваемой задачи аспекты своего предыдущего выполнения. Кроме того, состояние процесса – это характеристика, однозначно определяющая его дальнейшее поведение, все последующие реакции на внешние события. В общем случае состояние процесса определяется текущими значениями его счетчика команд, регистров и переменных.

Теоретическую модель процесса определим в виде множества его возможных *состояний* (англ. *states*), некоторого подмножества этого множества, элементы которого называются *начальными состояниями* (англ. *initial states*) и множества дискретных *событий* (англ. *events*). Событие *e*, происходящее в процессе *P*, представляет собой атомарное действие, которое может изменить состояние самого процесса *P* и состояние канала *C*, инцидентного этому процессу: состояние *C* будет изменено при отправке сообщения по этому каналу (если ребро, соответствующее *C*, направлено от вершины с процессом *P*) или при получении сообщения из этого канала (если ребро, соответствующее *C*, направлено к вершине с процессом *P*). Поэтому все события могут быть классифицированы как внутренние события (англ. *internal events*), события отправки (англ. *send events*) и события получения сообщения (англ. *receive events*). Следует отметить, что в качестве одного атомарного события также часто рассматривают отправку сразу нескольких сообщений по нескольким каналам связи, инцидентным *P*, например, при широковещательной или групповой рассылке. При этом для удобства мы будем полагать, что получение сообщения не может совпадать с отправкой или получением других сообщений в виде одного события.

Событие *e* определяется (1) процессом *P*, в котором оно произошло; (2) состоянием *s* процесса *P* непосредственно перед событием *e*; (3) состоянием *s'* процесса *P* непосредственно после события *e*; (4) каналом *C*, состояние которого изменяется при наступлении события *e*; (5) сообщением *m*, отправленным по каналу *C* (если ребро, соответствующее *C*, направлено от вершины с процессом *P*), или полученным из канала *C* (если ребро, соответствующее *C*, направлено к вершине с процессом *P*). Поэтому событие *e* мы будем определять как пятерку (*P*, *s*, *s'*, *m*, *C*), где *m* и *C* принимают специальное значение *null*, в случае, когда событие *e* не изменяет состояние ни одного из каналов связи.

Множество событий процесса P линейно упорядочено согласно порядку их наступления. Через e_i^x обозначим x-ое по порядку событие в процессе P_i, $1 \le i \le N$. Мы будем опускать верхний или нижний индекс, когда его значение очевидно из контекста, или когда его употребление не имеет смысла. Запись $e_i^x(s)$ будет обозначать состояние s', в которое переходит процесс P_i из состояния s при наступлении события e_i^x.

Выполнением (англ. *run*, *execution*) процесса P_i называется последовательность событий e_i^0, e_i^1, … e_i^x, e_i^{x+1}, … в которой s_i^0 – начальное состояние процесса P_i и для каждого $x \ge 0$ выполняется соотношение $s_i^{x+1} = e_i^x(s_i^x)$. Выполнение процесса P_i будем обозначать через R_i:

$$R_i = (E_i, \rightarrow_i),$$

где E_i – множество событий процесса P_i, а бинарное отношение $\rightarrow_i$ задает линейный порядок на этом множестве согласно порядку наступления событий в P_i. Запись $e_i \rightarrow_i e_i'$ обозначает, что событие e_i предшествует e_i' в выполнении процесса P_i.

Нас, однако, будет интересовать не выполнение отдельного процесса, а выполнение всей распределенной системы в целом.

Глобальное состояние S (англ. *global state*) распределенной системы определяется состояниями всех процессов и всех каналов, входящих в ее состав. *Начальными глобальными состояниями* (англ. *initial global state*) считаются все те состояния, в которых все процессы пребывают в своих начальных состояниях, а каналы связи – пусты. Функционирование распределенной системы порождает множество распределенных событий, происходящих в процессах, и оказывающих влияние только на *часть* глобального состояния системы.

Обозначим через $E = \cup_i E_i$ множество всех событий, происходящих при выполнении распределенной системы. Событие $e \in E$, такое что $e = (P, s, s', m, C)$, считается *допустимым* в глобальном состоянии S тогда и только тогда, когда (1) в глобальном состоянии S процесс P находится в состоянии s и (2) если ребро, соответствующее каналу C, направлено к вершине с процессом P, и сообщение m находится в канале C и может быть принято из него. Обозначим через $e(S)$ глобальное состояние распределенной системы непосредственно после наступления события e в глобальном состоянии S. Состояние $e(S)$ определено только в случае, если событие *e допустимо в глобальном состоянии S*. Тогда $e(S)$ будет отличаться от состояния S только состоянием процесса P и, возможно, состоянием канала C.

1. В глобальном состоянии $e(S)$ процесс P будет находиться в состоянии s'.
2. Если ребро, соответствующее каналу C, направлено к вершине с процессом P, то в глобальном состоянии $e(S)$ состояние канала C

будет совпадать с его состоянием в S за исключением сообщения m, удаленного из этого канала.

3. Если ребро, соответствующее каналу C, направлено от вершины с процессом P, то в глобальном состоянии $e(S)$ состояние канала C будет совпадать с его состоянием в S за исключением сообщения m, добавленного в этот канал.

Последовательность событий e^0, e^1, … e^x, e^{x+1}, … будем называть *выполнением распределенной системы* тогда и только тогда, когда событие e^x допустимо в глобальном состоянии S^x, S^0 – начальное глобальное состояние распределенной системы, и для каждого $x \geq 0$ выполняется соотношение $S^{x+1} = e^x(S^x)$. Выполнение всей распределенной системы будем обозначать через R. Глобальное состояние S называется достижимым в выполнении R, если для этой последовательности событий существует такое $k \geq 0$, что $e^k(S^k) = S$.

Для иллюстрации выполнения распределенной системы рассмотрим следующий пример передачи маркера (англ. *token*). Пусть распределенная система состоит из двух процессов P_1 и P_2 и двух каналов C_{12} и C_{21}, см. рис. 2.1.

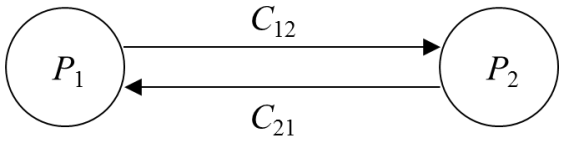


Рис. 2.1. Пример распределенной системы из двух процессов.

Маркер – это уникальный объект, который процессы передают друг другу с помощью сообщения. Каждый процесс может находиться в одном из двух состояний s или s', где s – состояние, в котором процесс не владеет маркером, а s' – состояние, в котором процесс обладает маркером. Пусть начальное состояние процесса P_1 – s', а начальное состояние процесса P_2 – s, т.е. маркер находится в процессе P_1. В каждом процессе могут происходить только два события: (1) событие отправки маркера по каналу с переходом процесса из состояния s' в состояние s и (2) событие получения маркера из канала с переходом процесса из состояния s в s'. Диаграмма переходов процесса из одного состояния в другое показана на рис. 2.2.

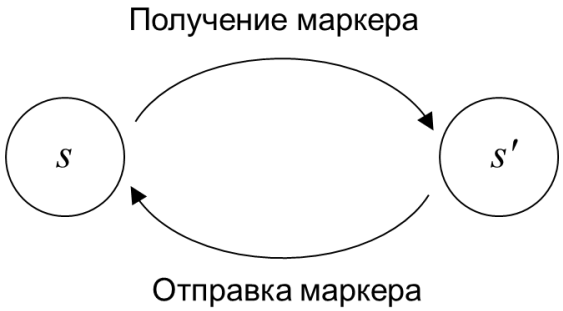


Рис. 2.2. Диаграмма состояний процесса.

Глобальные состояния распределенной системы и переходы между ними представлены на рис. 2.3. На этой диаграмме выполнению распределенной системы соответствует путь из начального глобального состояния S^0 в одно из других глобальных состояний S. Примерами выполнения распределенной системы могут служить следующие последовательности событий: (1) пустая последовательность или (2) последовательность "P_1 отправляет маркер, P_2 получает маркер, P_2 отправляет маркер". Последовательность "P_1 отправляет маркер, P_2 отправляет маркер" не может являться выполнением распределенной системы, т.к. событие "P_2 отправляет маркер" недопустимо в глобальном состоянии, в котором P_2 находится в состоянии s.

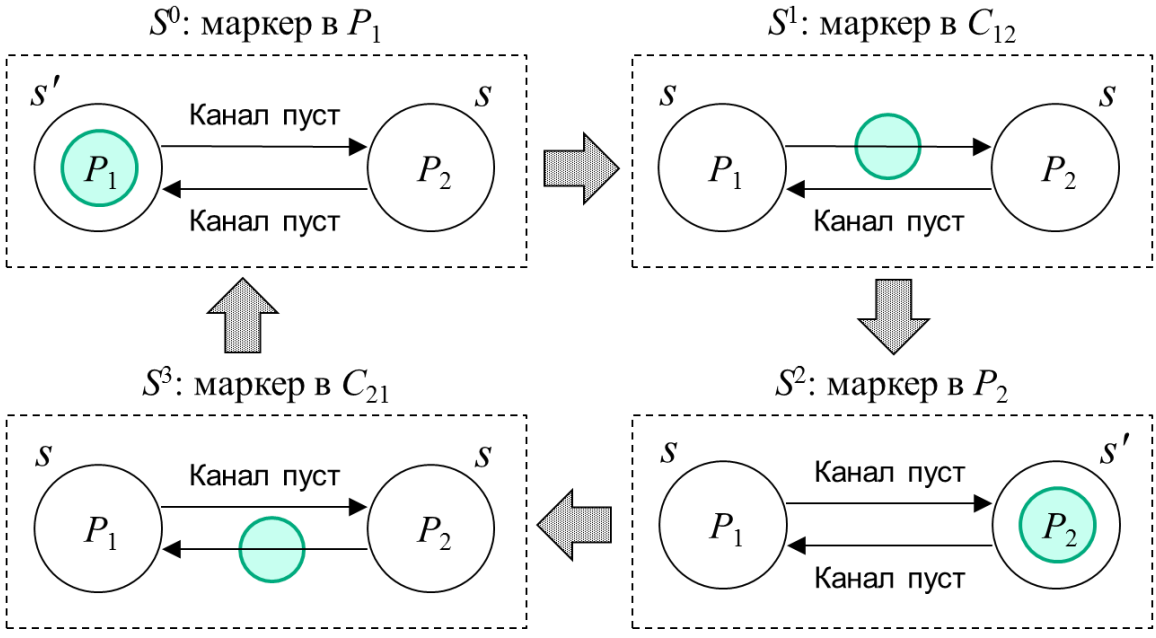


Рис. 2.3. Глобальные состояния и переходы между ними.

В представленном выше примере передачи маркера в каждом глобальном состоянии распределенной системы допустимо ровно одно событие. Для иллюстрации недетерминированного выполнения распределенных систем рассмотрим систему с такой же топологией (см.

рис. 2.1), но в которой состояния процессов P_1 и P_2 и переходы между ними определяются согласно рис. 2.4. Начальное состояние процесса P_1 – s_1, начальное состояние процесса P_2 – s_2.

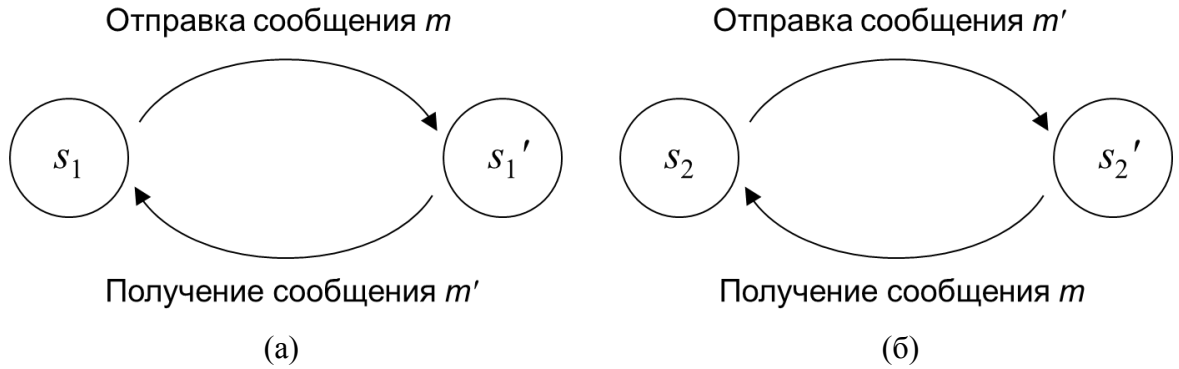


Рис. 2.4. Диаграмма состояний (а) процесса P_1 и (б) процесса P_2.

Пример выполнения такой системы показан на рис. 2.5. Легко видеть, что в некоторых глобальных состояниях допустимо более одного события и, соответственно, более одного перехода из этого глобального состояния.

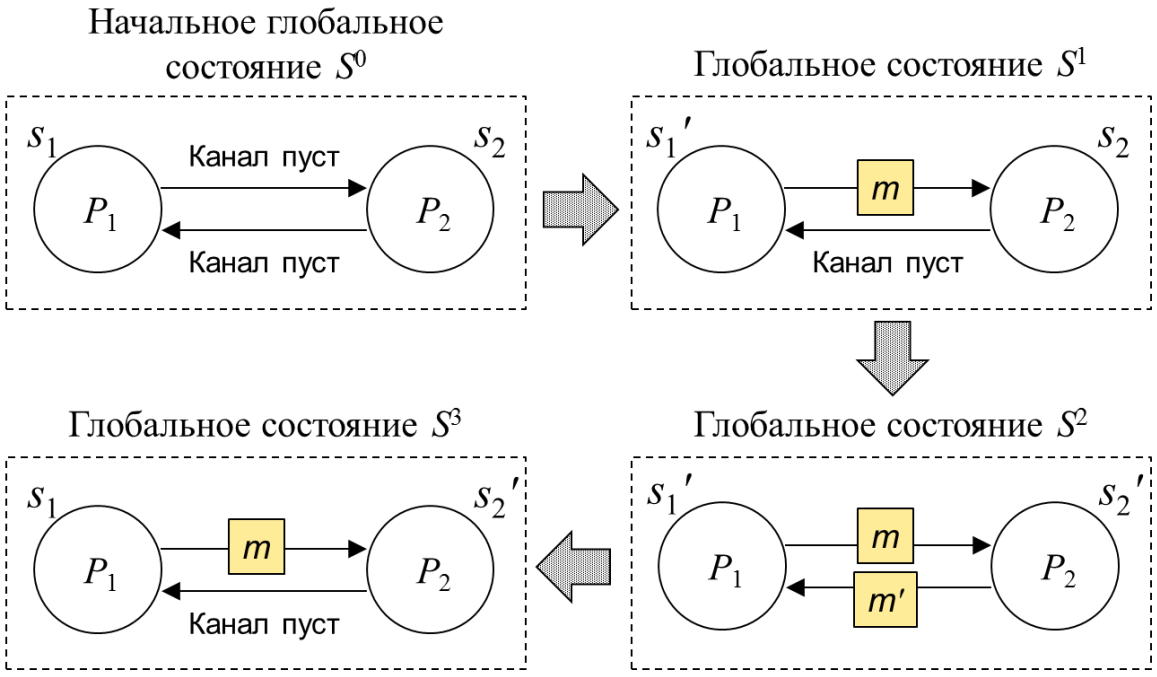


Рис. 2.5. Глобальные состояния и переходы между ними.

Например, в начальном глобальном состоянии S^0 могут произойти оба события "P_1 отправляет m" и "P_2 отправляет m' ". Важно отметить, что глобальные состояния, в которые система может перейти из состояния S^0 при наступлении одного из этих событий, различны. А именно, при наступлении события "P_1 отправляет m" система перейдет из состояния S^0 в состояние S^1, в котором P_1 находится в состоянии s_1', P_2 – в состоянии s_2,

канал C_{12} содержит сообщение m, а канал C_{21} пуст. В свою очередь при наступлении события "P_2 отправляет m' " система перейдет из состояния S^0 в другое глобальное состояние $S^{1'}$, в котором P_1 находится в состоянии s_1, P_2 – в состоянии s_2', канал C_{12} пуст, а канал C_{21} содержит сообщение m'. Поэтому выполнение такой распределенной системы может представлять собой другую по сравнению с рис. 2.5 последовательность событий и, как следствие, последовательность *других глобальных состояний*.

Справедливость (англ. *fairness*). При исследовании поведения распределенных систем часто возникает необходимость ограничиться рассмотрением только так называемых *справедливых выполнений* (англ. *fair executions*). Условия *слабой* справедливости позволяют исключить из рассмотрения такие выполнения, в которых некоторые события оказываются *постоянно допустимыми* для бесконечного числа идущих подряд глобальных состояний, но при этом ни разу не происходят в виде переходов из этих состояний из-за того, что выполнение продолжается всякий раз за счет других допустимых событий. Условия *сильной* справедливости позволяют исключить из рассмотрения такие выполнения, в которых некоторые события оказываются *бесконечно часто допустимыми* для глобальных состояний в выполнении, но при этом ни разу не происходят в виде переходов из этих состояний из-за того, что выполнение продолжается всякий раз за счет других допустимых событий. Другими словами, выполнение считается *слабо справедливым*, если ни одно событие не может оставаться постоянно допустимым без того, чтобы, в конце концов, не произойти в виде перехода. Выполнение считается *сильно справедливым*, если ни одно событие не может оказываться допустимым бесконечно часто без того, чтобы, в конце концов, не произойти в виде перехода.

Безопасность (англ. *safety*) и *живучесть* (англ. *liveness*). При разработке распределенного алгоритма, решающего ту или иную задачу, необходимо показать, что полученный алгоритм действительно представляет собой корректное решение поставленной задачи. Существует два фундаментальных типа свойств, проверяемых для подтверждения корректности распределенного алгоритма: безопасность и живучесть.

Неформально можно сказать, что алгоритм удовлетворяет свойству безопасности, если "что-то плохое *никогда* не случается". Другими словами, условие безопасности требует, чтобы определенное утверждение оставалось истинным *в каждом* достижимом глобальном состоянии *в любом* выполнении. Для иллюстрации этого требования можно привести пример системы управления сигналами светофоров; тогда требование безопасности будет выражаться в том, что при любом распределении

сигналов светофоров одновременное движение по пересекающимся направлениям на перекрестке должно быть запрещено.

Однако из того, что в системе не может произойти ничего плохого, совсем не следует, что в ней может случиться что-то хорошее. Так, программные системы, которые ничего не делают, всегда удовлетворяют требованиям безопасности: в них ничего не происходит, и, как следствие, ничего плохого тоже не случается. Или, например, для системы управления сигналами светофоров требование безопасности можно выполнить путем разрешения проезда только в одном направлении и полного запрещения движения в другом.

Таким образом требования безопасности должны сопровождаться требованиями прогресса вычислительной системы в нужном направлении. Такие требования называются живучестью. Неформально можно сказать, что алгоритм удовлетворяет свойству живучести, если "что-то хорошее *рано или поздно* случится". Другими словами, условие живучести требует, чтобы определенное утверждение становилось истинным *хотя бы в одном* достижимом глобальном состоянии *в любом* выполнении. Для нашего примера системы управления сигналами светофоров это свойство должно выражаться в обеспечении возможности, рано или поздно, проехать по любому направлению на перекрестке.

2.2. Причинно-следственный порядок событий

Представляя выполнение в виде последовательности событий, мы тем самым естественно привносим в модель распределенной обработки данных понятие времени. Будем говорить, что событие e наступает раньше события e', если в последовательности событий e предшествует e'.

Пусть задано выполнение $R = (e^0, e^1, \ldots e^x, e^{x+1}, \ldots)$. С этой последовательностью событий связана последовательность глобальных состояний $(S^0, S^1, \ldots S^x, S^{x+1}, \ldots)$ такая, что $S^{x+1} = e^x(S^x)$ для всех $x \geq 0$. Важно отметить, что последовательность R не всегда однозначно определяет последовательность глобальных состояний: если в R не содержится ни одного события процесса P_i для некоторого i, $1 \leq i \leq N$, то начальное состояние этого процесса остается неопределенным и может быть выбрано неоднозначным образом. Однако если в R содержится хотя бы одно событие процесса P_i, то первое событие $e_i = (P_i, s, s', m, C)$ указывает на то, что начальным состоянием процесса P_i является s. Таким образом, если в R содержится хотя бы одно событие каждого процесса, то начальное состояние S^0 определяется однозначно и за счет этого однозначно определяется и вся последовательность $(S^0, S^1, \ldots S^x, S^{x+1}, \ldots)$.

Для визуализации выполнения распределенной системы можно использовать *пространственно-временные диаграммы*. Пример одной из таких диаграмм приведен на рис. 2.6. Ход выполнения каждого процесса P_i

представляется в виде горизонтальной прямой линии. Каждое событие e_i процесса P_i отмечается точкой на прямой, соответствующей этому процессу. Вообще говоря, в реальности выполнение того или иного события занимает некоторое ненулевое конечное время, однако, из-за предположения атомарности событий их можно изображать точками. На прямой процесса P_i событие e_i изображается левее события e_i' тогда и только тогда, когда $e_i \rightarrow_i e_i'$, т.е. если событие e_i наступает раньше события e_i' в процессе P_i. Поэтому на диаграмме ось времени направлена слева направо. Например, на рис. 2.6 первым по порядку событием, произошедшим в процессе P_2, является внутреннее событие, вторым событием является событие получения сообщения, а третьим – событие отправки сообщения. Стрелкой обозначается передача сообщения от одного процесса другому. Если отправка сообщения m происходит при наступлении события e_i, а получение этого же сообщения m – при наступлении события e_j', то события отправки e_i и получения e_j' называют *взаимосвязанными*. Вертикальная проекция событий на ось R описывает выполнение распределенной системы. Так как событие получения сообщения не может наступить раньше связанного с ним события отправки этого сообщения, стрелки, обозначающие передачу сообщения, всегда будут направлены слева направо.

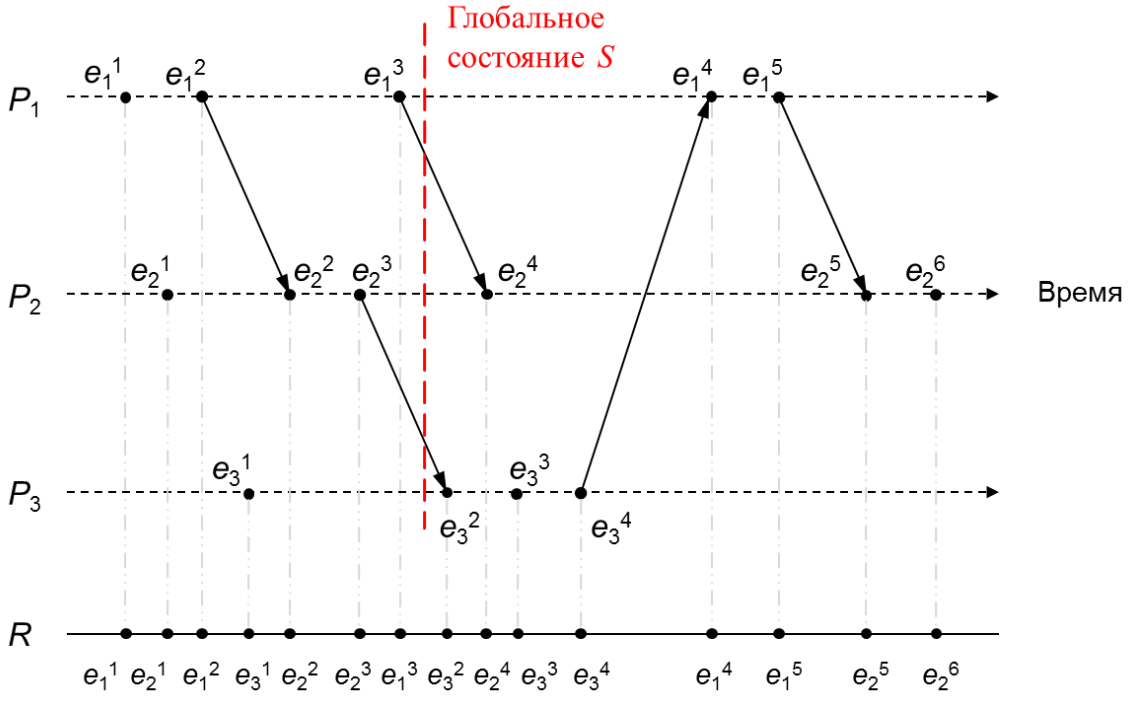


Рис. 2.6. Пример пространственно-временной диаграммы распределенного выполнения.

Последовательность глобальных состояний, связанная с данным выполнением R, может быть представлена на пространственно-временной диаграмме в виде последовательности вертикальных "срезов" состояний

процессов и каналов, проходящих между событиями, отмеченными точками (см. рис. 2.6). Ниже мы покажем, что порядок следования некоторых событий в выполнении R может быть изменен так, что это никак не отразится на последующем глобальном состоянии, достижимом в этом выполнении. Поэтому представление о времени как о линейном порядке на множестве E всех событий, происходящих при выполнении распределенной системы, не совсем подходит для распределенной обработки данных.

Происходящие события оказывают влияние только на часть глобального состояния системы. Поэтому два события, идущие друг за другом в выполнении и влияющие на раздельные части глобального состояния, оказываются *независимыми* и могут происходить в другом порядке. Действительно, пусть S – глобальное состояние распределенной системы, и пусть e_i и e_j – события, которые происходят в разных процессах P_i и P_j, т.е. $i \neq j$, и при этом оба события допустимы в глобальном состоянии S. Тогда событие e_i допустимо в глобальном состоянии $e_j(S)$, а событие e_j – в глобальном состоянии $e_i(S)$ и при этом $e_i\,(e_j(S)) = e_j\,(e_i(S))$.

Для доказательства этого утверждения рассмотрим два события $e_i = (P_i,\ s_i,\ s_i',\ m_i,\ C_i)$ и $e_j = (P_j,\ s_j,\ s_j',\ m_j,\ C_j)$, где C_i и C_j – каналы, инцидентные процессам P_i и P_j. Важным обстоятельством оказывается тот факт, что e_i и e_j не являются взаимосвязанными событиями отправки и получения одного и того же сообщения, т.е. m_i и m_j – *различные сообщения*. В противном случае оба взаимосвязанных события отправки и получения не могли бы одновременно являться допустимыми в глобальном состоянии S, что противоречило бы условию. Обозначим через S_i глобальное состояние, в которое перейдет система из состояния S после наступления события e_i, т.е. $S_i = e_i(S)$. Как указывалось в п. 2.1, глобальное состояние S_i будет отличаться от S только состоянием процесса P_i и, возможно, состоянием канала C_i. А именно, процесс P_i будет находиться в состоянии s_i', а сообщение m_i будет либо добавлено в канал C_i, если e_i – событие отправки, либо удалено из канала C_i, если e_i – событие получения сообщения. Так как m_i и m_j – различные сообщения, а состояние s_j процесса P_j одинаково в глобальных состояниях S и S_i, то событие e_j является допустимым также и в состоянии S_i. С помощью аналогичных рассуждений мы можем показать, что и событие e_i является допустимым в состоянии $S_j = e_j(S)$.

Рассмотрим теперь глобальное состояние $S_{ij} = e_j(S_i)$. В нем процессы P_i и P_j находятся в состояниях s_i' и s_j', а сообщение m_i либо добавлено в канал C_i, либо удалено из него, в зависимости от типа события e_i, равно как и сообщение m_j либо добавлено в канал C_j, либо удалено из него, в зависимости от типа события e_j. Эта же ситуация соответствует глобальному состоянию $S_{ji} = e_i(S_j)$, т.е. $S_{ij} = S_{ji}$, что и требовалось доказать.

Таким образом, события не могут произойти в другом порядке, только в двух случаях:

1. $i = j$, т.е. события e_i и e_j происходят в одном и том же процессе, или
2. e_i и e_j – взаимосвязанные события отправки и получения одного и того же сообщения.

В остальных случаях порядок, в котором они происходят, может быть изменен, но в результате будет достигнуто то же самое глобальное состояние. Важно отметить, что с глобальной точки зрения при этом система будет проходить через *разные* глобальные состояния, т.е. переходы $S - S_i - S_{ij}$ отличаются от переходов $S - S_j - S_{ji}$ т.к. $S_i \neq S_j$, хотя $S_{ij} = S_{ji}$.

Тот факт, что какие-то два события нельзя поменять местами в выполнении, означает, что эти события связаны *отношением причинно-следственной зависимости*. Это отношение можно распространить на все множество событий E в выполнении, введя тем самым *причинно-следственное отношение частичного порядка*.

Отношением *причинно-следственного порядка* $\rightarrow$ называется наименьшее отношение, удовлетворяющее следующим условиям:

1. если e_i и e_i' – два разных события одного и того же процесса P_i и при этом $e_i \rightarrow_i e_i'$, т.е. e_i наступает раньше события e_i' в P_i, то $e_i \rightarrow e_i'$;
2. если e_i и e_j' – взаимосвязанные события отправки и получения одного и того же сообщения, то $e_i \rightarrow e_j'$;
3. отношение $\rightarrow$ транзитивно, т.е. если $e_i \rightarrow e_j'$ и $e_j' \rightarrow e_k''$, то $e_i \rightarrow e_k''$.

Для любых двух событий e_i и e_j' таких, что $e_i \rightarrow e_j'$, событие e_j' напрямую или транзитивно зависит от e_i. Графически это означает, что на пространственно-временной диаграмме существует направленный в сторону увеличения времени путь из e_i в e_j', состоящий из стрелок, обозначающих передачу сообщений, и сегментов горизонтальных прямых, отражающих ход выполнения каждого отдельного процесса. Например, на рис. 2.6 $e_1^1 \rightarrow e_3^3$ и $e_3^3 \rightarrow e_2^6$.

Отношение причинно-следственного порядка событий было впервые введено Л. Лэмпортом и получило название "произошло раньше" (англ. *happened before*). Запись $e_i \rightarrow e_j'$ читается как "e_i произошло раньше e_j'". Чтобы подчеркнуть причинно-следственную зависимость между событиями иногда говорят, что "событие e_i потенциально влияет на событие e_j'", а "событие e_j' потенциально зависит от e_i". Кроме того, отношение $\rightarrow$ отражает обмен информацией при выполнении распределенной системы, и выражение $e_i \rightarrow e_j'$ свидетельствует о том, что любая информация, доступная при наступлении события e_i потенциально доступна и для события e_j'. Поэтому также говорят, что "событие e_j' знает о событии e_i". Например, на рис. 2.6 событие e_2^6 "знает" о наступлении всех других событий, представленных на рисунке.

Отношение причинно-следственного порядка иррефлексивно и задает отношение *частичного* порядка на множестве E событий выполнения распределенной системы, поскольку могут найтись события e_i и e_j', для которых не выполняется ни $e_i \rightarrow e_j'$, ни $e_j' \rightarrow e_i$.

Для любых событий e_i и e_j' запись $e_i \not\rightarrow e_j'$ обозначает тот факт, что e_j' не зависит от e_i ни напрямую, ни транзитивно, и наступление события e_i не влияет на событие e_j'. Важно отметить, что, если $e_i \not\rightarrow e_j'$, то событие e_j' "не знает" не только о событии e_i, но и обо всех других событиях, произошедших после e_i в том же процессе P_i. Например, на рис. 2.6 $e_1^3 \not\rightarrow e_3^3$ и $e_2^4 \not\rightarrow e_3^1$.

Стоит обратить внимание на следующие два правила:

- для любых двух событий e_i и e_j': $e_i \not\rightarrow e_j' \not\Rightarrow e_j' \not\rightarrow e_i$;
- для любых двух событий e_i и e_j': $e_i \rightarrow e_j' \Rightarrow e_j' \not\rightarrow e_i$.

События e_i и e_j', для которых $e_i \not\rightarrow e_j'$ и $e_j' \not\rightarrow e_i$, будем называть *параллельными* (англ. *concurrent*) или независимыми (англ. *independent*) и для обозначения этого факта будем использовать запись $e_i \parallel e_j$. Для выполнения распределенной системы, представленного на рис. 2.6, $e_1^3 \parallel e_3^3$ и $e_2^4 \parallel e_3^1$.

Следует отметить, что отношение $\parallel$ не является транзитивным, т.е. $(e_i \parallel e_j') \wedge (e_j' \parallel e_k'') \not\Rightarrow e_i \parallel e_k''$. Например, на рис. 2.6 $e_3^3 \parallel e_2^4$ и $e_2^4 \parallel e_1^5$, однако $e_3^3 \nparallel e_1^5$.

Таким образом для любых двух событий e_i и e_j', происходящих при выполнении распределенной системы, либо $e_i \rightarrow e_j'$, либо $e_j' \rightarrow e_i$, либо $e_i \parallel e_j'$.

2.3. Эквивалентные выполнения

В этом подразделе мы покажем, что любое изменение порядка следования событий в выполнении, сохраняющее причинно-следственный порядок, не оказывает влияния на результат выполнения. Такая перестановка событий приводит к другой последовательности глобальных состояний, но полученное при этом выполнение будет считаться эквивалентным исходному выполнению.

Рассмотрим выполнение $R = (e^0, e^1, \ldots e^x, e^{x+1}, \ldots)$ и связанную с ним последовательность глобальных состояний $(S^0, S^1, \ldots S^x, S^{x+1}, \ldots)$, где $S^{x+1} = e^x(S^x)$ для всех $x \geq 0$. Перестановкой $\hat{R}$ элементов R является последовательность $\hat{R} = (\hat{e}^0, \hat{e}^1, \ldots \hat{e}^x, \hat{e}^{x+1}, \ldots)$ такая, что $\hat{e}^x = e^{\sigma(x)}$, где σ – перестановка на множестве натуральных чисел или на конечном множестве $\{0, \ldots, k-1\}$, если R – конечное выполнение, в котором происходит k событий. Перестановка $\hat{R}$ событий выполнения R *сохраняет причинно-следственный порядок*, если $\hat{e}^x \rightarrow \hat{e}^y \Rightarrow x < y$, т.е. ни одно событие

$\hat{e}^y$ не появляется в этой последовательности перед событием $\hat{e}^x$, от которого $\hat{e}^y$ зависит.

Пусть $\hat{R} = (\hat{e}^0, \hat{e}^1, \ldots \hat{e}^x, \hat{e}^{x+1}, \ldots)$ – перестановка событий выполнения R, сохраняющая причинно-следственный порядок событий в выполнении. Тогда $\hat{R}$ является выполнением распределенной системы с начальным состоянием S^0. При этом если R – конечное выполнение, то последнее глобальное состояние $\hat{S}^k$ в выполнении $\hat{R}$ для последовательности ($\hat{S}^0 = S^0$, $\hat{S}^1$, … $\hat{S}^k$), где $\hat{S}^{x+1} = \hat{e}^x(\hat{S}^x)$, $0 \le x \le k-1$, будет совпадать с последним глобальным состоянием S^k в выполнении R.

Для доказательства этого утверждения будем формировать глобальные состояния $\hat{S}^x$ в последовательности ($\hat{S}^0 = S^0$, $\hat{S}^1$, … $\hat{S}^x$, $\hat{S}^{x+1}$, …) одно за другим, и для построения $\hat{S}^{x+1}$ достаточно показать, что событие $\hat{e}^x$ допустимо в состоянии $\hat{S}^x$.

Предположим, что для всех $0 \le y \le x-1$ событие $\hat{e}^y$ допустимо в состоянии $\hat{S}^y$ и $\hat{S}^{y+1} = \hat{e}^y(\hat{S}^y)$. Пусть событие $\hat{e}^x$ является событием, происходящим в процессе P_i, и $\hat{e}^x = (P_i, s, s', m, C)$. Тогда событие $\hat{e}^x$ будет допустимым в глобальном состоянии $\hat{S}^x$, если (1) в состоянии $\hat{S}^x$ процесс P_i находится в состоянии s и (2) если $\hat{e}^x$ – событие получения сообщения, то сообщение m находится в канале C и может быть принято из него.

Чтобы показать, что в состоянии $\hat{S}^x$ процесс P_i находится в состоянии s необходимо рассмотреть два случая. В обоих случаях важно обратить внимание на то, что причинно-следственный порядок событий *линейно* упорядочивает события процесса P_i для любого i. Это означает, что порядок следования событий процесса P_i в перестановке $\hat{R}$ будет таким же, что и в выполнении R.

Случай 1: $\hat{e}^x$ – первое событие процесса P_i в перестановке $\hat{R}$. Тогда в глобальном состоянии $\hat{S}^x$ процесс P_i находится в своем начальном состоянии s_i^0. Событие $\hat{e}^x$ также является первым событием процесса P_i в выполнении R, т.е. переводит P_i из его начального состояния s_i^0 в другое. Поэтому $s = s_i^0$.

Случай 2: $\hat{e}^x$ – не является первым событием процесса P_i в перестановке $\hat{R}$. Пусть последним событием процесса P_i, предшествующим $\hat{e}^x$ в перестановке $\hat{R}$, является событие $e_i = (P_i, s_i, s_i', m_i, C_i)$. Тогда в глобальном состоянии $\hat{S}^x$ процесс P_i находится в состоянии s_i'. Событие e_i также является последним событием процесса P_i, предшествующим $\hat{e}^x$ в выполнении R, и поэтому $s = s_i'$.

Чтобы показать, что сообщение m находится в канале C и может быть принято из него, если $\hat{e}^x$ – событие получения сообщения, важно обратить внимание на то, что взаимосвязанные события отправки и получения в перестановке $\hat{R}$ расположены в том же порядке, что и в выполнении R. Поэтому событию $\hat{e}^x$ в перестановке $\hat{R}$ предшествуют те же самые события отправки и получения сообщений по каналу C, что и в выполнении R. Следовательно, состояние канала C перед наступлением $\hat{e}^x$

в перестановке $\hat{R}$ идентично состоянию канала C перед наступлением этого же события $e^{\sigma(x)} = \hat{e}^x$ в выполнении R.

Таким образом мы показали, что для каждого x событие $\hat{e}^x$ допустимо в глобальном состоянии $\hat{S}^x$, и тогда в качестве $\hat{S}^{x+1}$ можно взять $\hat{e}^x(\hat{S}^x)$.

Нам осталось показать, что, если выполнение R – конечно, то последнее глобальное состояние $\hat{S}^k$ в выполнении $\hat{R}$ будет совпадать с последним глобальным состоянием S^k в выполнении R. Если в выполнении R не содержится ни одного события процесса P_i, то в S^k процесс P_i будет находиться в своем начальном состоянии s_i^0. Так как в этом случае $\hat{R}$ также не будет содержать ни одного события процесса P_i, состояние P_i в $\hat{S}^k$ также будет s_i^0. В противном случае состояние процесса P_i в S^k – это то состояние, в которое P_i переходит при наступлении последнего события этого процесса в выполнении R. Это событие также является последним событием P_i и в выполнении $\hat{R}$. Значит, состояние P_i в $\hat{S}^k$ будет точно таким же, что и в S^k.

Состояние каналов распределенной системы в глобальном состоянии S^k определяется совокупностью сообщений, отправленных по этим каналам, но еще не полученных принимающими процессами в выполнении R. Но коль скоро в R и $\hat{R}$ содержится одна и та же совокупность событий, те же сообщения окажутся в состоянии пересылки и для последнего глобального состояния $\hat{S}^k$ в выполнении $\hat{R}$.

Таким образом мы показали, что $\hat{S}^k = S^k$, что и требовалось доказать.

В обоих выполнениях R и $\hat{R}$ происходит одна и та же совокупность событий, и причинно-следственный порядок этих событий одинаков для R и $\hat{R}$. В этом случае, мы будем говорить, что выполнения R и $\hat{R}$ *эквиваленты* и обозначать это отношение $R \sim \hat{R}$.

На рис. 2.7. изображена пространственно-временная диаграмма выполнения, эквивалентного выполнению, представленному на рис 2.6. На оси $\hat{R}$ цветом выделены события с измененным порядком следования по сравнению с выполнением R. Диаграммы эквивалентных выполнений можно получить из заданной диаграммы путем "растягивания" и "сжимания" осей времени отдельных процессов, при условии, что стрелки, обозначающие передачу сообщений, остаются направленными слева направо.

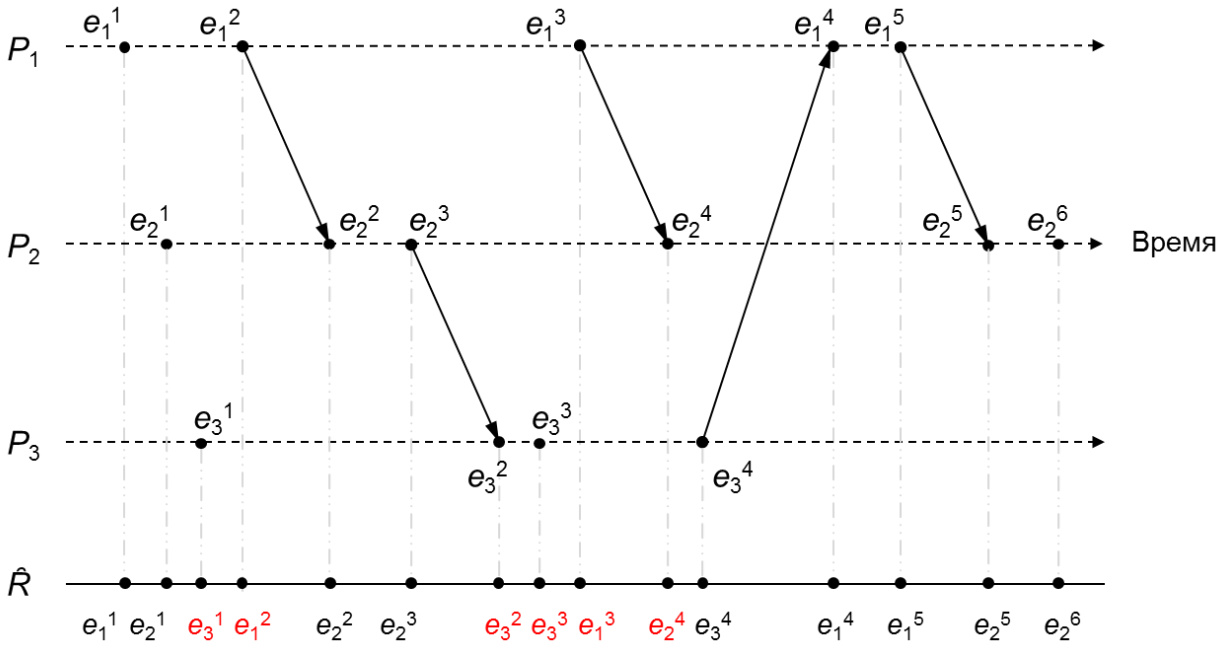


Рис. 2.7. Пространственно-временная диаграмма распределенного выполнения, эквивалентного выполнению на рис 2.6.

Несмотря на то, что изображенные на рис. 2.6 и рис. 2.7 выполнения эквивалентны и в них происходит одна и та же совокупность событий, эти выполнения порождают *разные множества глобальных состояний*. Например, выполнение на рис. 2.6 содержит глобальное состояние S, в котором сообщения, отправленные при наступлении событий e_1^3 и e_2^3, одновременно пребывают в состоянии пересылки. В свою очередь выполнение на рис. 2.7 не содержит ни одного такого глобального состояния, т.к. сообщение, отправленное при событии e_2^3, оказывается полученным раньше, чем наступает событие e_1^3.

Сторонний наблюдатель, которому одномоментно доступна вся картина происходящего и который имеет возможность видеть фактическую последовательность событий, способен отличить одно эквивалентное выполнение от другого, т.е. он всякий раз видит только одно из возможных выполнений. Однако процессы не могут отличить два эквивалентных выполнения, т.к. с точки зрения процессов невозможно понять, какое из двух эквивалентных выполнений происходит на самом деле. Поясним это на следующем примере. Предположим, что необходимо определить, действительно ли сообщения, отправленные при наступлении событий e_1^3 и e_2^3, находились в состоянии пересылки одновременно. Пусть в одном из процессов поддерживается булева переменная *isSim*, которая принимает значение *true*, если сообщения находились в состоянии пересылки одновременно, и *false* – в противном случае. Тогда в последнем глобальном состоянии выполнения на рис 2.6, значение *isSim* должно быть равно *true*, а в последнем глобальном состоянии выполнения на рис. 2.7 –

false. Однако, как было показано выше, последние глобальные состояния для эквивалентных конечных выполнений должны совпадать, поэтому ожидаемого присваивания значений переменной *isSim* достичь невозможно.

Класс эквивалентности выполнений по отношению ~ однозначно определяется множеством событий и действующим на нем причинно-следственным порядком. Такое частично упорядоченное множество событий называют распределенным *вычислением* (англ. *computation*) и обозначают через $\mathbb{C}$:

$$\mathbb{C} = (E, \rightarrow).$$

Выполнение распределенной системы R можно рассматривать как линеаризацию (англ. *linear extension*) частичного порядка, т.е. введение на множестве событий отношения линейного порядка $<$, сохраняющего частичный причинно-следственный порядок событий $\rightarrow$, т.е. для любых двух событий e_i и e_j': $e_i \rightarrow e_j' \Rightarrow e_i < e_j'$.

Из теории частично упорядоченных множеств следует, что для любых двух событий e_i и e_j' таких, что $e_i \not\rightarrow e_j'$, существует такая линеаризация $<$ частичного порядка $\rightarrow$, для которой выполняется соотношение $e_j' < e_i$. Это означает, что если e_i и e_j' – параллельные события вычисления $\mathbb{C}$, то существуют два таких выполнения R и $\hat{R}$ этого вычисления, в одном из которых событие e_i наступает раньше события e_j', а в другом – наоборот, событие e_j' наступает раньше события e_i. Процессы же по ходу выполнения не могут определить, какое из этих двух событий происходит первым, и, стало быть, через какое глобальное состояние проходит система.

Таким образом имеет смысл говорить о совокупности *событий вычисления*, т.к. во всех выполнениях одного и того же вычисления происходят одни и те же события. И не имеет смысла говорить о совокупности *глобальных состояний вычисления*, потому что различные выполнения одного и того же вычисления могут порождать разные множества глобальных состояний.

Существенные события (англ. *relevant events*). Не все события, происходящие в распределенной системе, могут оказаться важными или *существенными* с точки зрения конкретного прикладного приложения. Например, в задачах восстановления распределенной системы после сбоя существенными могут являться только внутренние события установки контрольных точек (англ. *checkpoints*). Обозначим через $E_R \subseteq E$ непустое подмножество всех существенных событий, и через $\rightarrow_R$ – *ограничение отношения* $\rightarrow$ на подмножество E_R, т.е. $\forall e_i, e_j' \in E_R$: $e_i \rightarrow_R e_j' \Leftrightarrow e_i \rightarrow e_j'$. Тогда распределенное вычисление, определяемое существенными событиями и порядком $\rightarrow_R$, обозначается через $\mathbb{C}_R = (E_R, \rightarrow_R)$.

2.4. Конус прошлого и конус будущего для события

В распределенном вычислении $\mathbb{C}$ на событие e_j' потенциально влияют только такие события e_i, для которых $e_i \rightarrow e_j'$, и событие e_j' "знает" о наступлении только этих событий e_i. Поэтому множество таких событий e_i формирует "прошлое" события e_j', которое обозначается через $Past(e_j')$:

$$Past(e_j') = \{e_i: \forall e_i \in \mathbb{C}, e_i \rightarrow e_j'\}.$$

Множество $Past(e_j')$ называют (*световым*) *конусом прошлого* для события e_j', как проиллюстрировано на рис. 2.8. Подмножество множества $Past(e_j')$ событий, происходящих в процессе P_i, обозначим через $Past_i(e_j')$. Множество $Past_i(e_j')$ линейно упорядочено отношением $\rightarrow_i$. Пусть $\max(Past_i(e_j'))$ – наибольший элемент этого множества. Очевидно, что событие $\max(Past_i(e_j'))$ является последним событием процесса P_i, потенциально влияющим на событие e_j' (см. рис. 2.8). Важно отметить, что событием $\max(Past_i(e_j'))$ при $i \neq j$ всегда будет являться событие *отправки* сообщения.

Совокупность $P(e_j') = \cup_i \max(Past_i(e_j'))$ всех последних событий процессов, которые могут оказывать влияние на событие e_j', называют *поверхностью конуса прошлого* для события e_j'. Здесь обратим внимание, что если событие e_i произошло раньше, чем e_j', то все события, в свою очередь наступившие в P_i раньше e_i, также произошли раньше e_j'. Поэтому поверхность конуса прошлого определяет *все* множество $Past(e_j')$ событий, произошедших раньше e_j'.

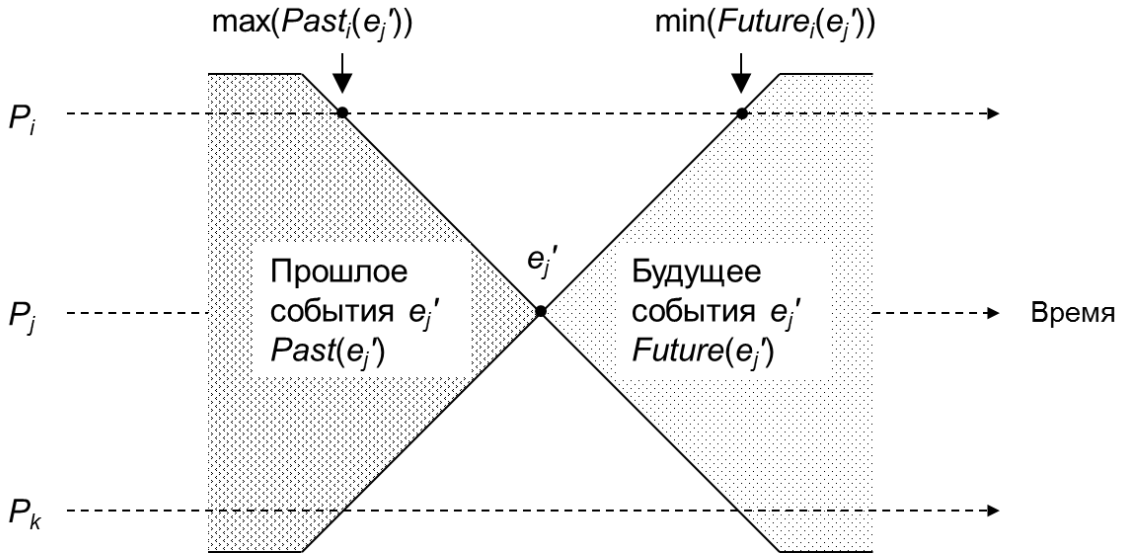


Рис. 2.8. Конус прошлого и конус будущего для события.

Аналогично конусу прошлого для события определяется его конус будущего. А именно, (*световому*) *конусу будущего* $Future(e_j')$ события e_j'

принадлежат все события e_i, на которые e_j' может оказывать влияние (см. рис. 2.8):

$$Future(e_j') = \{e_i: \forall e_i \in \mathbb{C}, e_j' \rightarrow e_i\}.$$

Если через $Future_i(e_j')$ обозначить все события множества $Future(e_j')$, происходящие в процессе P_i, то наименьшим элементом $\min(Future_i(e_j'))$ этого множества будет являться первое событие процесса P_i, на которое влияет e_j'. Следует обратить внимание, что событием $\min(Future_i(e_j'))$ при $i \neq j$ всегда будет являться событие *получения* сообщения.

Совокупность $F(e_j') = \cup_i \min(Future_i(e_j'))$ всех первых событий процессов, на которые e_j' может оказывать влияние, называют *поверхностью конуса будущего* для события e_j'. Аналогично $P(e_j')$ поверхность конуса будущего $F(e_j')$ определяет *все* множество событий $Future(e_j')$, которые потенциально зависят от e_j'.

Очевидно, что все события процесса P_i, которые наступают после события $\max(Past_i(e_j'))$, но перед событием $\min(Future_i(e_j'))$, являются параллельными с событием e_j'. Поэтому параллельными с e_j' событиями являются те и только те события, которые принадлежат множеству $E \setminus Past(e_j') \setminus Future(e_j')$.

Логический и физический параллелизм. Для распределенного вычисления два события считаются логически параллельными, когда они не связаны отношением причинно-следственной зависимости друг с другом. В свою очередь, физический параллелизм событий подразумевает наступление событий практически в одно и то же время. Важно отметить, что события могут быть логически параллельными даже в случае, когда они происходят в разные моменты времени. Например, на рис. 2.7 события e_2^4 и e_3^1 происходят в различные моменты времени, однако, являются логически параллельными. Тем не менее, если бы скорости выполнения процессов и задержки доставки сообщений были бы другими, указанные события вполне могли бы произойти практически в одно и то же время. При этом результат вычисления не будет зависеть от того, совпадает ли выполнение логически параллельных событий во времени или нет. Поэтому, несмотря на то, что два логически параллельных события могут происходить в различные моменты времени, для всех практических и теоритических задач можно считать, что такие события происходят одновременно. Однако здесь важно еще раз оговориться, что в отличие от физического параллелизма, отношение логического параллелизма не является транзитивным, т.е. $(e_i \parallel e_j') \wedge (e_j' \parallel e_k'') \not\Rightarrow e_i \parallel e_k''$. Поэтому, например, для стороннего наблюдателя события e_2^4 и e_3^1 на рис. 2.7 могли бы происходить практически одновременно. Также событие e_2^4 могло бы произойти практически в одно и то же время с событием e_1^5. Однако $e_3^1 \nparallel e_1^5$, и потому эти события всякий раз будут происходить друг за

другом, и между ними в выполнении всегда будут находиться другие события.

2.5. Свойства каналов

Свойства каналов связи определяются допущениями относительно взаимосвязи событий отправки и получения сообщений.

Свойство очередности. Канал, сохраняющий порядок передаваемых по нему сообщений, называется *очередью*, или каналом FIFO (англ. *First In First Out*). Это означает, что если отправка сообщения m_1 происходит раньше чем отправка сообщения m_2 по одному и тому же каналу, то получение сообщения m_1 также происходит раньше, чем получение сообщения m_2. При отправке сообщения по такому каналу оно помещается в конец очереди, а при получении сообщения оно извлекается из начала очереди.

Канал, не являющийся очередью (non-FIFO канал), можно представить в виде мультимножества, в которое отправляющий процесс добавляет элементы (сообщения), а принимающий процесс изымает элементы (сообщения) в случайном порядке.

Емкость каналов. Емкостью канала называется количество сообщений, которые могут находиться в канале в состоянии пересылки. Канал считается *переполненным*, если число содержащихся в нем сообщений в точности совпадает с его емкостью.

Для асинхронного механизма обмена сообщениями событие отправки сообщения будет допустимо только в тех состояниях, в которых канал не является переполненным. Стоить отметить, что при описании модели распределенной системы в п. 2.1, мы предполагали, что допустимость события отправки сообщения не зависит от состояния канала, тем самым неявно подразумевая, что каналы имеют неограниченную емкость и, как следствие, никогда не переполняются. В дальнейшем мы будем рассматривать именно такие каналы.

Для случая синхронного механизма обмена сообщениями можно считать, что процессы взаимодействуют через каналы с нулевой емкостью.

Надежность. Канал считается надежным, если каждое сообщение, отправленное по этому каналу, обязательно будет доставлено получателю в единственном экземпляре и в том виде, в котором оно было отправлено (естественно, при условии, что принимающий процесс оказывается способным получить это сообщение).

В ненадежном канале возможно возникновение ошибок разных типов: потеря сообщения, искажение сообщения, дублирование или спонтанное порождение сообщения. Потеря сообщения приводит к тому, что отправленное сообщение никогда не сможет быть получено. Искажение сообщения происходит, когда полученное сообщение

оказывается отличным от отправленного. Дублирование сообщения возникает в том случае, когда сообщения принимаются чаще, чем отправляются. Спонтанное порождение сообщения происходит, если получатель принимает сообщение, которое вообще никогда не было отправлено.

Мы все каналы будем считать надежными.

РАЗДЕЛ 3. ЛОГИЧЕСКИЕ ЧАСЫ

Понятие времени является фундаментальным для нашего мышления; его течение отделяет все более раннее от всего более позднего. Большинство, наверное, скажет, что событие *a* произошло раньше события *b*, если *a* произошло в более ранее время, чем *b*.

В распределенных системах отсутствуют глобальные часы, отсчитывающие общее для всех процессов время, и к показаниям которых процессы могли бы получать мгновенный доступ. Поэтому нам пришлось определять отношение "произошло раньше", связывающее события процессов между собой, не опираясь на понятие единого физического времени. Мы показали, что традиционное представление о времени как о линейном порядке на множестве событий не совсем подходит для распределенного вычисления, т.к. отношение "произошло раньше" оказывается отношением *частичного порядка*, и с точки зрения процессов невозможно понять, какая последовательность событий происходит на самом деле.

В этом разделе мы рассмотрим механизм *логических часов*, который позволяет отслеживать причинно-следственный порядок событий распределенного вычисления и, как следствие, упорядочивать события в одну или несколько последовательностей, которые *могли бы* происходить в системе. Использование *логического времени*, отсчитываемого такими часами, значительно упрощает разработку алгоритмов для распределенных систем.

Следует подчеркнуть, что основная задача логических часов заключается только в отслеживании порядка событий, а не в определении каких-либо других свойств, обычно ассоциируемых с понятием времени. Например, логические часы не дают никакого количественного представления о физическом времени, прошедшем между двумя событиями. В отличие от физического времени, чей ход нельзя остановить или изменить, логическое время не течет само по себе. Оно поддается учету только при наступлении событий в распределенной системе, и потому – дискретно. Невозможно ничего не делать и ожидать наступления того или иного момента логического времени в будущем: если не происходит никаких событий, логическое время "останавливается", и ожидаемый момент может никогда не наступить.

3.1. Общие принципы построения логических часов

В обычной жизни мы говорим, что некоторое событие произошло в 9:30, если оно наступило *после* того, как наши часы показали 9:30, и *до* того, как показания часов изменились на 9:31. При этом с каждым

интересующим нас событием мы ассоциируем *отметку времени* (англ. *timestamp*), соответствующую этому событию.

Описание принципов построения логических часов мы также начнем с понимания того, что такие часы должны обеспечивать механизм, позволяющий каждому событию в распределенной системе приписывать некоторое числовое значение, которое можно интерпретировать как время наступления этого события. Поэтому *логическими часами* (англ. *logical clock*) назовем всякую функцию Θ, отображающую множество событий распределенного вычисления $\mathbb{C}$ в некоторое упорядоченное множество $(T, <)$, где T представляет собой совокупность допустимых значений *логического времени* (англ. *time domain*). В распределенных системах каждый процесс P_i может управлять работой только своих часов Θ_i независимо от часов других процессов. При этом отметка логического времени события e_i, происходящего в процессе P_i, будет определяться показаниями часов этого процесса на момент наступления e_i, т.е. $\Theta(e_i) = \Theta_i(e_i)$ для всех событий e_i процесса P_i. Как было указано выше, показания часов Θ_i должны изменяться *между* наступлением событий процесса P_i, т.е. сам факт изменения показаний Θ_i событием не является.

Несмотря на то, что такие часы работают в независимых процессах, отметки логического времени, назначаемые событиям, должны удовлетворять фундаментальному условию монотонного возрастания времени. А именно, все события, принадлежащие конусу прошлого любого события e_i, должны быть отмечены временем меньшим, чем время $\Theta(e_i)$ события e_i. И наоборот, все события из конуса будущего события e_i должны иметь время наступления большее, чем $\Theta(e_i)$:

$$\forall e_i, e_j' \in \mathbb{C}: e_i \rightarrow e_j' \Rightarrow \Theta(e_i) < \Theta(e_j').$$

Данное требование монотонного возрастания времени называют условием *непротиворечивости логических часов* (англ. *clock consistency*) или условием *непротиворечивости отметок времени* событий. Оно отражает последовательную природу выполнения каждого отдельного процесса и соглашение о том, что на передачу любого сообщения затрачивается ненулевое конечное время. Действительно, из определения отношения причинно-следственного порядка следует, что требование непротиворечивости логических часов будет выполнено при выполнении следующих двух условий:

Условие 1: если e_i и e_i' – два разных события одного и того же процесса P_i, и событие e_i наступает в P_i раньше события e_i', то $\Theta(e_i) < \Theta(e_i')$.

Условие 2: если e_i и e_j' – взаимосвязанные события отправки и получения одного и того же сообщения, передаваемого из процесса P_i в процесс P_j, то $\Theta(e_i) < \Theta(e_j')$.

В случае, когда Θ и T удовлетворяют условию

$$\forall e_i, e_j' \in \mathbb{C}: e_i \rightarrow e_j' \Leftrightarrow \Theta(e_i) < \Theta(e_j'),$$

логические часы называют *строго непротиворечивыми* (англ. *strongly consistent*).

Реализация механизма логических часов подразумевает (1) определение структуры данных, поддерживаемой локально каждым процессом для представления логического времени из множества T и хранения текущих показаний своих часов, и (2) описание метода продвижения логического времени, гарантирующего выполнение условия непротиворечивости.

В логических часах каждого процесса P_i часто выделяют две составляющие.

- *Логические локальные часы* для измерения собственного хода выполнения процесса. То есть логические локальные часы используются процессом P_i для записи информации о ходе своего собственного выполнения.
- *Логические глобальные часы* для описания локального представления процесса P_i о глобальном времени. То есть логические глобальные часы используются процессом P_i для записи информации о ходе выполнения других процессов. Логические глобальные часы позволяют процессу назначать непротиворечивые отметки времени для собственных событий.

Обычно показания логических локальных и логических глобальных часов хранятся в одной структуре данных, описывающей логическое время.

Основной целью метода продвижения логического времени является обеспечение условия непротиворечивости логических часов. Описание такого метода складывается из определения следующих двух правил.

Правило 1: определяет, как процесс изменяет показания своих логических локальных часов при наступлении в нем любого события.

Правило 2: определяет, как процесс изменяет показания своих логических глобальных часов для отражения своего представления о ходе выполнения других процессов. Точнее говоря, это правило определяет, какая информация о текущем логическом времени процесса включается в каждое отправляемое им сообщение, и как процесс, получающий такое сообщение, использует эту информацию для обновления своего локального представления о глобальном времени.

Нетрудно видеть, что для гарантии непротиворечивости логических часов указанные Правила 1 и 2 должны соответственно удовлетворять Условиям 1 и 2, перечисленным выше.

Различные механизмы логических часов отличаются друг от друга как структурой данных, описывающей логическое время, так и методом его продвижения. Однако все они в той или иной форме реализуют

указанные Правила 1 и 2 для выполнения условия непротиворечивости и, иногда, обеспечивают некоторые дополнительные возможности.

3.2. Скалярное время Лэмпорта

Для линейного упорядочивания событий распределенного вычисления Л. Лэмпорт предложил использовать часовую функцию, отображающую множество событий распределенного вычисления в множество неотрицательных целых чисел. В этом случае логическое локальное время процесса P_i и его представление о логическом глобальном времени выражаются одной скалярной величиной, обозначаемой как L_i. Отметку времени произвольного события e_i будем обозначать через $L(e_i)$.

Правила 1 и 2 продвижения логического времени определяются следующим образом.

Правило 1: перед выполнением любого события процесс P_i увеличивает показания своих часов L_i:

$$L_i = L_i + d, \text{ где } d > 0.$$

В общем случае d может принимать любое значение, каждый раз разное для каждого последующего события. Однако чаще всего значение d полагают всегда равным единице. Логические часы L_i инициализируется нулем.

Очевидно, что такое правило продвижения логического времени удовлетворяет Условию 1 непротиворечивости логических часов. Однако при получении сообщения кроме Условия 1 необходимо еще удовлетворить Условию 2. Поэтому перед выполнением события получения сообщения процесс вынужден произвести немного другие действия согласно следующему правилу.

Правило 2: в каждое передаваемое сообщение добавляется значение логического времени L_i процесса-отправителя P_i на момент отправки этого сообщения. Когда процесс P_j получает такое сообщение, содержащее отметку времени L_{msg}, он выполняет следующие шаги:

1. $L_j = \max(L_j, L_{msg})$;
2. исполняет **Правило 1**;
3. доставляет сообщение и приступает к его обработке.

Нетрудно видеть, что это правило обеспечивает выполнение обоих Условий 1 и 2 непротиворечивости логических часов для события получения сообщения.

Поэтому такие логические часы являются непротиворечивыми, т.е. для любых двух событий e_i и e_j':

$$e_i \rightarrow e_j' \Rightarrow L(e_i) < L(e_j').$$

Пример работы алгоритма скалярных часов Лэмпорта для $d = 1$ приведен на рис. 3.1. Рядом с каждым событием представлена его отметка времени. На стрелках указаны отметки времени, передаваемые с сообщениями.

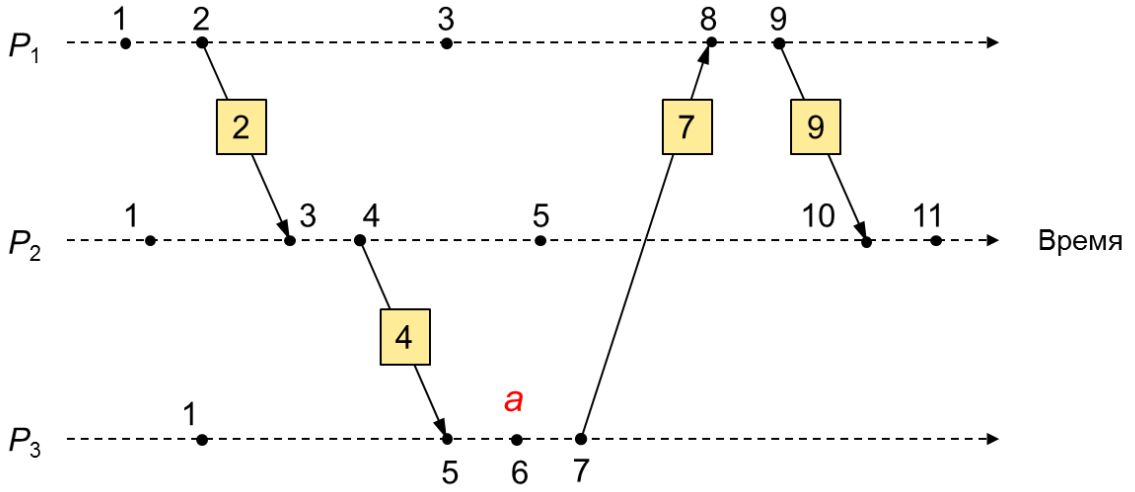


Рис. 3.1. Пример работы алгоритма скалярных часов Лэмпорта.

3.2.1 Основные свойства

Линейное упорядочивание событий. Скалярные часы Лэмпорта могут быть использованы для введения на множестве событий распределенного вычисления $\mathbb{C}$ отношения линейного порядка $<$, сохраняющего частичный причинно-следственный порядок событий $\rightarrow$. Для этого мы просто будем упорядочивать события согласно их отметкам времени. Единственная сложность, с которой мы можем столкнуться, заключается в том, что некоторые события, происходящие в различных процессах, могут иметь одинаковую отметку времени. Например, на рис. 3.1 третье событие в процессе P_1 и второе событие в процессе P_2 имеют идентичную скалярную отметку времени, равную трем.

Для упорядочивания таких событий воспользуемся произвольным линейным порядком, заданным для процессов распределенной системы. К примеру, можно использовать линейно упорядоченные идентификаторы процессов. Тогда отношение линейного порядка $<$, связывающее любые два события e_i и e_j', происходящие в процессах P_i и P_j, будет определяться выражением:

$$e_i < e_j' \Leftrightarrow (L(e_i) < L(e_j')) \vee ((L(e_i) = L(e_j')) \wedge (i < j)).$$

Благодаря тому, что скалярные часы удовлетворяют требованию непротиворечивости, событиями с одинаковыми отметками времени могут являться только параллельные события, т.е. $L(e_i) = L(e_j') \Rightarrow e_i \parallel e_j'$. Такие события могут быть упорядочены любым способом без нарушения причинно-следственного порядка $\rightarrow$, в том числе так, как представлено

выше. Поэтому, введенное нами отношение линейного порядка $<$ сохраняет причинно-следственный порядок событий $\rightarrow$: $e_i \rightarrow e_j' \Rightarrow e_i < e_j'$.

Таким образом, мы показали, что скалярные часы могут быть использованы для определения *одного из эквивалентных* выполнений распределенной системы, т.е. для упорядочивания событий в последовательность, которая *могла бы* происходить в системе. В общем случае рассмотренный линейный порядок используется для обеспечения свойства живучести при построении различных распределенных алгоритмов: а именно, запросы, требующие обслуживания, снабжаются отметками скалярного времени и обслуживаются по порядку, задаваемому этими отметками, что позволяет каждому запросу, в конце концов, получить свое право на обслуживание.

Стоит отметить, что линейный порядок $<$, определенный выше, устанавливает статические приоритеты для процессов распределенной системы согласно значениям их идентификаторов. Если требуется "более справедливый метод", то отношение линейного порядка между процессами можно было бы задать как функцию логического времени. Например, в случае, когда $L(e_i) = L(e_j')$ и $i < j$, мы могли бы определить, что $e_i < e_j'$, если значение $L(e_i) \bmod N$ попадает в полуинтервал $(i, j]$, и $e_j' < e_i$ в противном случае.

Подсчет событий. Если значение прироста логических часов всегда равно единице ($d = 1$), то скалярные часы обладают следующим свойством: если событие e имеет отметку времени $L(e)$, то величина $L(e) - 1$ будет определять количество событий, которые должны произойти последовательно, друг за другом, до наступления события e независимо от процессов, в которых эти события происходят. Например, на рис. 3.1 событию a предшествуют пять последовательно выполняемых событий вдоль наибольшего направленного пути, ведущего к a, в то время как, например, первые события представленных процессов происходят одновременно согласно показаниям логического часов, и, следовательно, могли бы происходить практически в одно и то же физическое время. Величину $L(e) - 1$ часто называют *высотой* (англ. *height*) события e.

Отсутствие строгой непротиворечивости. Скалярные часы не являются строго непротиворечивыми, т.е. для событий e_i и e_j' таких, что

$$L(e_i) < L(e_j') \not\Rightarrow e_i \rightarrow e_j'.$$

Например, на рис. 3.1 третье по счету событие e_1^3 процесса P_1 имеет скалярную отметку времени $L(e_1^3) = 3$, т.е. меньшую, чем отметка времени $L(e_2^3) = 4$ третьего события e_2^3 процесса P_2. Однако событие процесса P_1 не "происходит раньше" события процесса P_2. Действительно, если бы выполнялось свойство строгой непротиворечивости, то из этого бы следовало, что любые два параллельных события должны иметь одинаковые отметки скалярного времени. Однако это невозможно, т.к.

отношение $\|$ не является транзитивным, т.е. может иметь место ситуация, когда $(e_i \| e_j') \wedge (e_j' \| e_k'')$, но $e_i \rightarrow e_k''$.

Таким образом, если для событий e_i и e_j' выполняется неравенство $L(e_i) < L(e_j')$, мы можем утверждать, что $e_j' \not\rightarrow e_i$, но мы не можем определить, выполняется ли $e_i \rightarrow e_j'$ или $e_i \| e_j'$, т.е. являются ли эти события зависимыми или параллельными. Математически это можно объяснить тем, что функция $L(e)$ отображает частично упорядоченное множество событий распределенного вычисления $\mathbb{C} = (E, \rightarrow)$ в линейно упорядоченное множество неотрицательных целых чисел $Z_{\geq 0}$. Поэтому скалярные часы не позволяют зафиксировать параллелизм событий.

Отсутствие строгой непротиворечивости является основным недостатком скалярных часов, т.е., исходя из знания (неравных) отметок времени различных событий, мы не можем заключить, связаны эти события отношением причинно-следственного порядка или нет.

3.2.2 Примеры использования

Для иллюстрации применения скалярных часов при решении различных задач рассмотрим следующие примеры.

Банковская система. Вернемся к представленной в п. 1.5.1 задаче подсчета полной суммы денег, находящихся на счетах в разных филиалах банка. Как и прежде будем предполагать, что банковская система не позволяет вносить дополнительные денежные средства на счет филиала и снимать их наличными, а лишь осуществляет перевод различных сумм из одного филиала банка в другой с помощью сообщений. Будем считать, что в каждом филиале работает ровно один процесс P_i, который отсылает бесконечно много сообщений каждому другому процессу распределенной системы. Удовлетворить этому требованию несложно: процесс всегда может отправить "пустое" сообщение с переводом \$0 в другой филиал. Для простоты будем считать, что все каналы обладают свойством FIFO.

Однако теперь, чтобы избежать трудностей, указанных в п. 1.5.1, мы будем подсчитывать денежные средства, опираясь не на физическое время, отсчитываемое компьютерами филиалов, а на введенное выше логическое время. А именно, общую сумму денег будем определять в момент логического времени $t \in Z_{\geq 0}$, т.е. *после* наступления всех событий с отметками времени, меньшими или равными t, и *до* наступления событий с отметками времени, строго большими, чем t, где t – время, известное всем процессам. Для этого потребуем, чтобы каждый процесс P_i записывал состояние своего счета непосредственно перед выполнением первого события e_i, для которого $L(e_i) > t$. Такое событие обязательно должно наступить, т.к. по нашему предположению в каждом процессе происходит бесконечно много событий. Чтобы учесть денежные средства, находящиеся в состоянии передачи между филиалами, необходимо

определить все сообщения, у которых события отправки имеют время меньшее или равное t, а события получения – время, превышающее t. Для этого, начиная с первого события e_i, имеющего время $L(e_i) > t$, т.е. начиная именно с того события, перед которым процесс зафиксировал состояние своего счета, каждый процесс P_i должен записывать все поступающие ему сообщения с отметками логического времени $L_{msg} \le t$. Когда из канала C_{ji} придет первое сообщение с отметкой времени $L_{msg} > t$, процесс P_i зафиксирует состояние канала C_{ji} на момент времени t, подсчитывая денежную сумму, переданную по этому каналу в записанных сообщениях. Сообщение с отметкой времени $L_{msg} > t$ обязательно должно поступить, т.к. по нашему предположению процесс P_j отправляет бесконечно много сообщений процессу P_i. Общая сумма денег будет определяться подсчетом всех денежных средств, записанных в зафиксированных состояниях счетов и каналов.

Чтобы показать, что представленный алгоритм дает правильный результат, рассмотрим произвольное выполнение R банковской системы. Перестановка $\hat{R}$ событий выполнения R в порядке, задаваемом отметками скалярного времени, представляет собой выполнение, эквивалентное R, т.е. с точки зрения процессов невозможно понять, какое из этих выполнений происходит на самом деле. Работа рассматриваемого алгоритма заключается в том, что он делает *снимок глобального состояния* S в выполнении $\hat{R}$ сразу после всех событий, имеющих время t, т.е. записывает состояние всех процессов и каналов на этот момент. Поэтому, очевидно, что в результате его работы будет получена корректная сумма денег, циркулирующих в банковской системе.

На рис. 3.2 представлен пример выполнения банковской системы, использующей скалярные часы. Слева показана начальная сумма денег на счете каждого филиала. Размер денежного перевода указан на стрелке, обозначающей передачу сообщения.

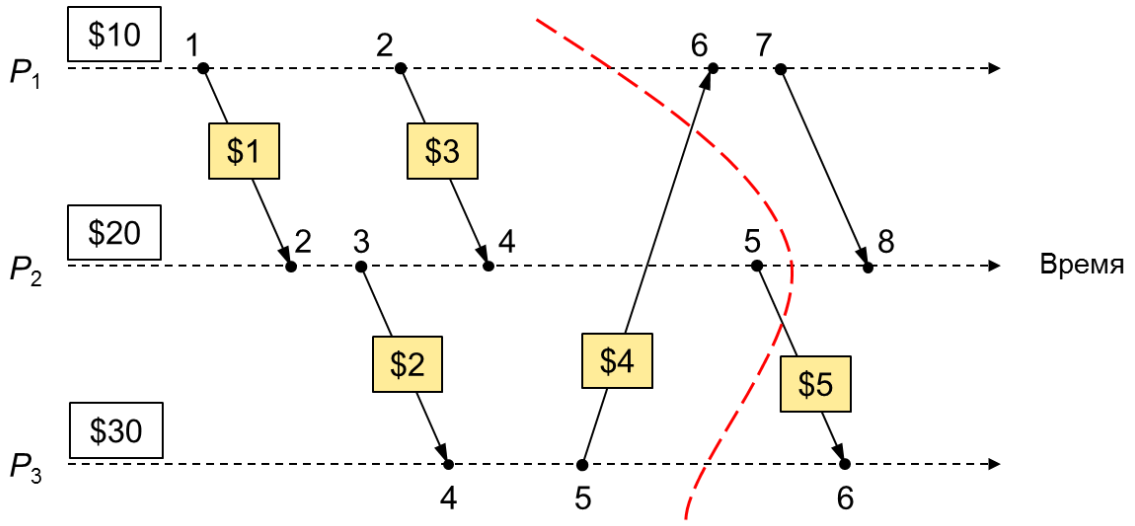


Рис. 3.2. Пример выполнения банковской системы, использующей скалярные часы.

Предположим, что подсчитать общую сумму денег требуется в момент времени $t = 5$, что на рис. 3.2 обозначено штриховой линией, проходящей через временные оси каждого из процессов. При подсчете денег на своем счете процесс P_1 получит результат, равный \$10 – \$1 – \$3 = \$6. На счете процесса P_2 окажется \$20 + \$1 – \$2 + \$3 – \$5 = \$17. Счет процесса P_3 будет составлять \$30 + \$2 – \$4 = \$28. Кроме того, процесс P_1 обнаружит в канале C_{31} сообщение с переводом \$4, а процесс P_3 – в канале C_{23} сообщение с переводом \$5. Общая сумма будет составлять \$6 + \$17 + \$28 + \$4 + \$5 = \$60, что является верным результатом.

На рис. 3.3 представлено эквивалентное выполнение банковской системы, в котором все события упорядочены согласно отметкам скалярного времени. Штриховая линия на рис. 3.3 соответствует штриховой линии на рис. 3.2 и демонстрирует результаты работы алгоритма, а именно, зафиксированные состояния процессов и каналов. Нетрудно видеть, что в выполнении на рис 3.3 совокупность этих состояния есть не что иное, как глобальное состояние системы, в которой она находится после наступления всех событий со временем 5 и до наступления любого события со временем 6.

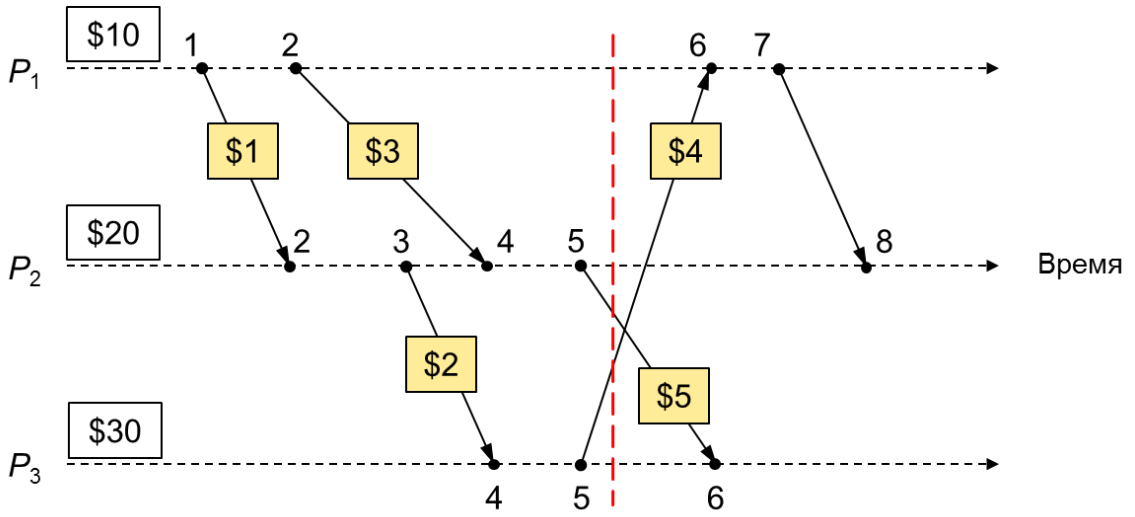


Рис. 3.3. Выполнение банковской системы, эквивалентное выполнению на рис. 3.2.

Полностью упорядоченная групповая рассылка. Обратимся теперь к рассмотренной в п. 1.5.3 задаче полного упорядочивания сообщений групповой рассылки. Пусть существует группа процессов, рассылающих друг другу широковещательные сообщения, которые требуется доставить всем процессам данной группы, включая отправителей этих сообщений, причем в порядке, одинаковом для всех процессов. Каждую такую рассылку будем рассматривать как одно атомарное событие соответствующего процесса, и будем считать, что все каналы обладают свойством FIFO.

Чтобы обеспечить единый для всех процессов порядок доставки широковещательных сообщений можно воспользоваться механизмом скалярных часов Лэмпорта, позволяющим задать линейный порядок на множестве событий рассылки этих сообщений. В этом случае останется лишь убедиться, что каждый процесс имеет необходимую информацию о событиях рассылки, происходящих в других процессах. Для этого каждый процесс локально поддерживает очередь, в которую помещает все принимаемые широковещательные сообщения и упорядочивает их по возрастанию значений отметок времени, содержащихся в этих сообщениях. Затем сообщения доставляются одно за другим по порядку из очереди каждого процесса. Здесь под отметкой времени сообщения m будем подразумевать упорядоченную пару $(L(m), i)$, где $L(m)$ – логическое время события рассылки этого сообщения, i – идентификатор процесса-отправителя. Линейный порядок таких отметок времени будет определяться правилом, введенным в п. 3.2.1.

Следует обратить внимание, что из-за задержек передачи сообщений, возможна ситуация, когда в начале очередей двух различных процессов будут находиться разные сообщения от разных отправителей. Эта ситуация проиллюстрирована на рис. 3.4. Локальная очередь каждого процесса изображена прямоугольником рядом с его временной осью. Видно, что имеется некоторый временной интервал, в течение которого сообщение m_1 уже получено процессами P_1 и P_2, а сообщение m_2 – процессами P_3 и P_4, но оба сообщения еще не дошли до остальных процессов данной группы. В течение этого временного интервала процессы P_1 и P_2 считают, что первым в очереди является сообщение m_1, а процессы P_3 и P_4 считают, что первым является сообщение m_2. При этом процессам, уже получившим широковещательное сообщение, не ведомо, направлено ли им еще какое-либо сообщение, и стоит ли ждать его прихода. Поэтому прежде чем процесс примет решение о доставке сообщения m, стоящего в начале его собственной очереди, он должен получить гарантии, что в каналах не осталось ни одного сообщения с отметкой времени меньшей, чем отметка времени сообщения m. Благодаря свойству FIFO каналов связи это условие будет выполнено, когда процесс получит сообщения от всех остальных процессов в группе с отметками времени б*о*льшими, чем отметка времени сообщения m. Для этого при получении широковещательного сообщения каждый процесс должен разослать всем процессам соответствующее подтверждение. Очевидно, отметка времени подтверждения будет больше, чем отметка времени самого сообщения, на которое это подтверждение высылается. Поэтому рано или поздно каждый процесс сможет получить от всех остальных процессов сообщения с отметками времени б*о*льшими, чем отметка времени сообщения m, и, следовательно, доставить m и приступить к его обработке.

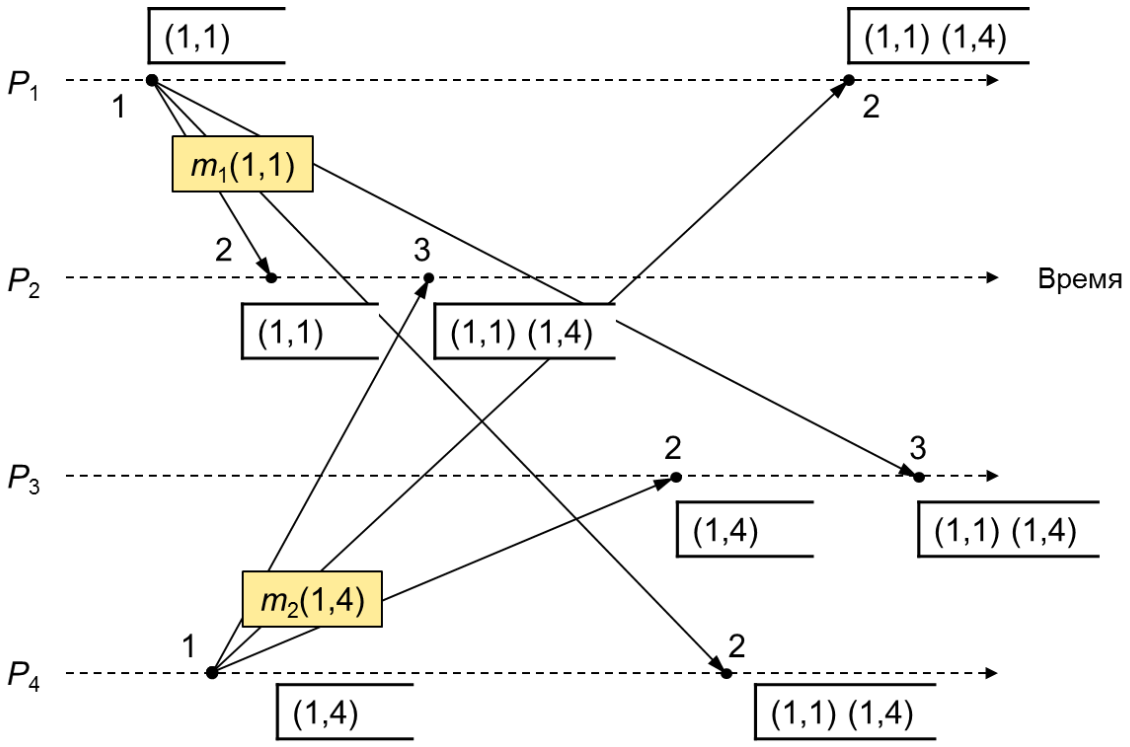


Рис. 3.4. Пример формирования очередей в процессах.

Следует еще раз обратить внимание, что подтверждения используются исключительно для информирования процессов о том, что в каналах не осталось ни одного сообщения, которое может потеснить сообщение *m* в их очередях и занять первое место. Поэтому при приеме широковещательного сообщения процесс может не отправлять соответствующее подтверждение, если к этому моменту он уже разослал свое собственное широковещательное сообщение с отметкой времени большей, чем отметка времени принимаемого сообщения.

Таким образом, механизм работы скалярных часов Лэмпорта гарантирует, что никакие два сообщения не будут иметь одинаковые отметки времени. При этом линейный порядок отметок времени будет определять общий порядок доставки сообщений групповой рассылки.

3.3. Векторное время

Скалярные часы не являются строго непротиворечивыми и не позволяют зафиксировать параллелизм событий. События, которые могли бы происходить одновременно, могут получить различные отметки времени как будто они происходили в каком-то определенном порядке. Для некоторых задач, например, рассмотренных в п. 3.2.2, такое ограничение не является существенным. Однако, к примеру, при отладке распределенных программ (англ. *distributed debugging*) информация о том,

оказывают ли события потенциальное влияние друг на друга или нет, становится важной.

Параллелизм событий можно зафиксировать, если часовые отметки времени параллельных событий оказываются несравнимыми. Другими словами, выявление параллельных событий возможно только тогда, когда множество T допустимых значений логического времени оказывается упорядоченным частично, но не линейно. Чтобы понять, из каких элементов может состоять такое множество, предположим, что в каждом процессе работают логические локальные часы, увеличивающие свои показания на единицу перед наступлением каждого события. Тогда для стороннего наблюдателя, которому одномоментно доступна вся картина происходящего, ход выполнения распределенной системы будет определяться совокупностью показаний локальных часов всех процессов. Подходящей структурой данных для представления такого глобального времени будет являться N-мерный вектор, в котором i-й компонент содержит текущее значение локального времени процесса P_i. Пример изменения глобального векторного времени V с точки зрения стороннего наблюдателя приведен на рис. 3.5.

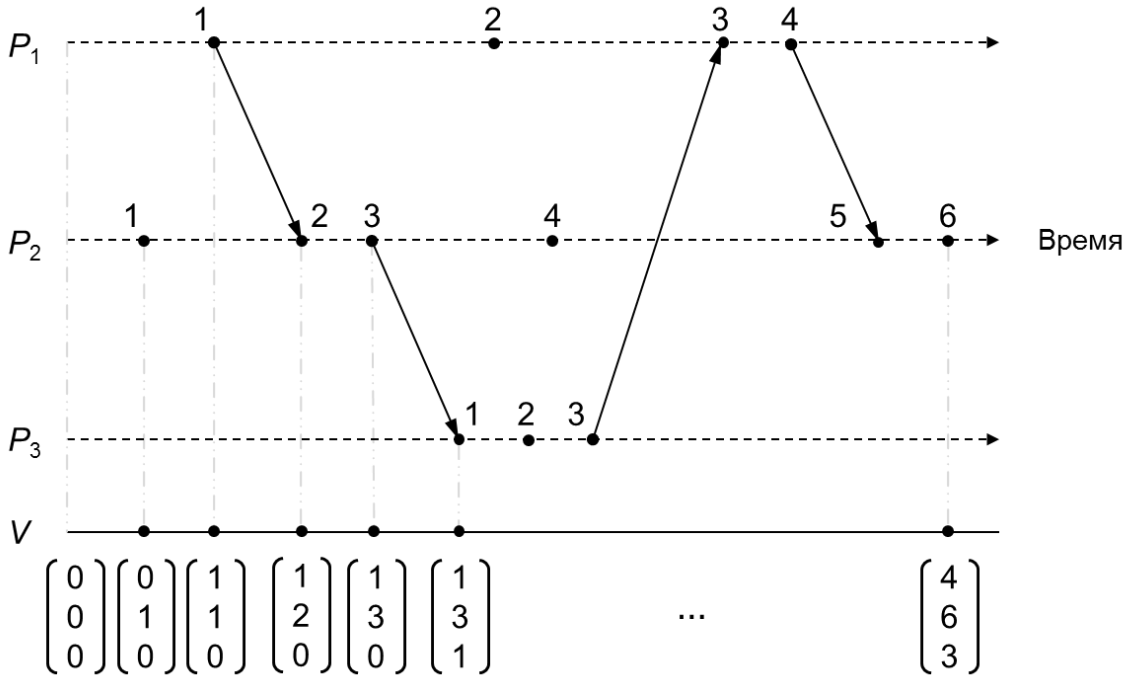


Рис. 3.5. Пример изменения глобального векторного времени.

Процессам же текущее значение глобального времени недоступно. Каждый процесс может лишь сформировать *свое собственное представление* о ходе выполнения процессов распределенной системы через поступающие ему сообщения. Для этого каждый процесс P_i должен локально поддерживать работу с вектором $V_i[1..N]$. В компоненте $V_i[i]$ этого вектора хранятся показания логических локальных часов процесса P_i,

измеряющих ход его собственного выполнения. Логические глобальные часы, используемые для записи информации о ходе выполнения других процессов, представлены остальными компонентами вектора V_i, а именно, компонент $V_i[j]$ содержит последние сведения, полученные процессом P_i о локальном времени процесса P_j. Другими словами, если $V_i[j] = x$, то процесс P_i "знает" о том, что локальное время процесса P_j достигло значения x. Стоит отметить, что к настоящему моменту локальные часы процесса P_j могли уйти вперед, т.е. текущее локальное время процесса P_j может превышать значение x, однако, процесс P_i не получал об этом никакой информации через направляемые ему сообщения. Весь вектор V_i можно рассматривать как локальное представление процесса P_i о глобальном векторном времени, отражающем ход выполнения *всех* процессов в распределенной системе.

Таким образом, областью значений часовой функции Θ является множество N-мерных векторов неотрицательных целых чисел, где N – количество процессов в распределенной системе. Поэтому такие часы называют *векторными часами*. Отметка времени события e_i, происходящего в процессе P_i, определяется всеми компонентами вектора V_i на момент наступления e_i. Такую отметку часто называют *векторной отметкой времени* события e_i и обозначают через $V(e_i)$.

Для сравнения двух векторных отметок времени $V(e_i)$ и $V(e_j')$ событий e_i и e_j' введем следующие отношения:

$$V(e_i) = V(e_j') \Leftrightarrow \forall k\colon V(e_i)[k] = V(e_j')[k];$$

$$V(e_i) \le V(e_j') \Leftrightarrow \forall k\colon V(e_i)[k] \le V(e_j')[k];$$

$$V(e_i) < V(e_j') \Leftrightarrow V(e_i) \le V(e_j') \text{ и } \exists k\colon V(e_i)[k] < V(e_j')[k].$$

Очевидно, что введенное выше отношение < определяет частичный порядок на множестве векторных отметок времени для $N \ge 2$. Например, вектора $V(e_i) = [2, 3, 0]$ и $V(e_j') = [0, 4, 1]$ являются несравнимыми. Для обозначения этого факта будем использовать запись $V(e_i) \parallel V(e_j')$:

$$V(e_i) \parallel V(e_j') \Leftrightarrow \neg(V(e_i) < V(e_j')) \wedge \neg(V(e_j') < V(e_i)).$$

Для выполнения условия непротиворечивости логических часов каждый процесс P_i использует следующие правила продвижения своего векторного времени.

Правило 1: перед выполнением любого события процесс P_i увеличивает показания своих локальных часов $V_i[i]$:

$$V_i[i] = V_i[i] + d, \text{ где } d > 0.$$

Как и для случая скалярного времени, каждый раз d может принимать любые значения. Однако обычно d всегда полагают равным единице.

Перед выполнением события получения сообщения процесс вынужден произвести немного другие действия согласно следующему правилу.

Правило 2: в каждое передаваемое сообщение добавляется значение векторного времени V_i процесса-отправителя P_i на момент отправки этого сообщения. Когда процесс P_j получает сообщение с отметкой времени V_{msg}, он выполняет следующие шаги:

1. обновляет свое логическое глобальное время по правилу:

$$V_j[k] = \max(V_j[k], V_{msg}[k]), 1 \le k \le N;$$

2. исполняет **Правило 1**;

3. доставляет сообщение и приступает к его обработке.

Компоненты векторного времени $V_i[k]$ инициализируется нулем, $1 \le k \le N$.

Нетрудно видеть, что представленные Правила 1 и 2 вместе с введенным отношением строгого порядка < на множестве векторных отметок времени удовлетворяют Условиям 1 и 2 непротиворечивости логических часов, т.е. такие векторные часы являются непротиворечивыми:

$$\forall e_i, e_j' \in \mathbb{C}: e_i \rightarrow e_j' \Rightarrow V(e_i) < V(e_j').$$

Пример работы алгоритма векторных часов для $d = 1$ приведен на рис. 3.6. Рядом с каждым событием представлена его отметка времени. На стрелках указаны отметки времени, передаваемые с сообщениями.

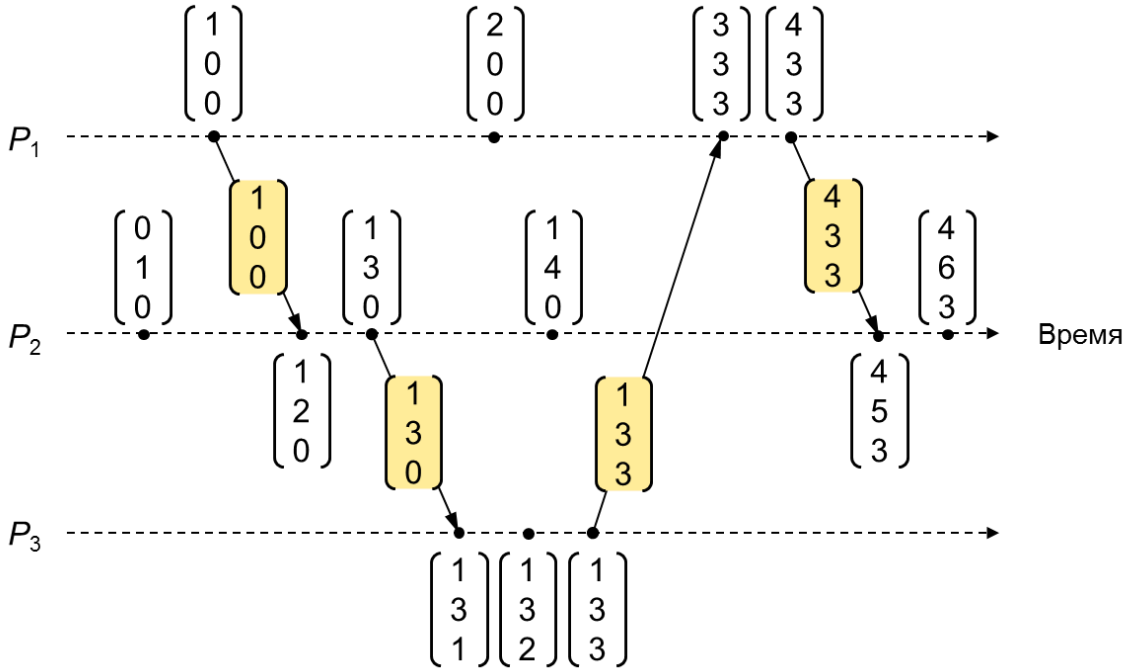


Рис. 3.6. Пример работы алгоритма векторных часов.

3.3.1 Основные свойства

Изоморфизм. Если события распределенного вычисления отмечаются с помощью механизма векторных часов, то для любых двух событий e_i и e_j' с векторными отметками времени $V(e_i)$ и $V(e_j')$, соответственно, имеем:

$$e_i \rightarrow e_j' \Leftrightarrow V(e_i) < V(e_j'),$$

$$e_i \parallel e_j' \Leftrightarrow V(e_i) \parallel V(e_j').$$

Необходимое условие для $e_i \rightarrow e_j'$ обеспечивается правилами продвижения векторного времени процесса.

Для доказательства того, что $V(e_i) < V(e_j')$ является достаточным условием для $e_i \rightarrow e_j'$, вначале покажем, что если события e_i и e_j' различны, то

$$e_i \not\rightarrow e_j' \Rightarrow V(e_j')[i] < V(e_i)[i].$$

Неформально, можно сказать, что в этом случае событие e_j' "не знает" о событии e_i, а "знает" лишь о каком-то событии, предшествующим событию e_i в процессе P_i (если $V(e_j')[i] \neq 0$).

Действительно, если $i = j$, т.е. e_i и e_j' – события одного процесса P_i, то из $e_i \not\rightarrow e_j'$ следует, что e_j' произошло раньше e_i. Поэтому на основании Правила 1 продвижения локального времени процесса P_i имеем $V(e_j')[i] < V(e_i)[i]$. Теперь предположим, что $i \neq j$, т.е. события e_i и e_j' происходят в разных процессах. В этом случае $V(e_i)[i]$ представляет собой значение локального времени процесса P_i в момент наступления события e_i. Очевидно, процесс P_j не мог получать это значение через поступающие к нему сообщения, т.к. $e_i \not\rightarrow e_j'$, т.е. $V(e_j')[i] < V(e_i)[i]$.

Отсюда имеем, что $e_i \not\rightarrow e_j' \Rightarrow \neg(V(e_i) < V(e_j'))$, что равносильно $V(e_i) < V(e_j') \Rightarrow e_i \rightarrow e_j'$, т.е. $V(e_i) < V(e_j')$ является достаточным условием для $e_i \rightarrow e_j'$.

Истинность утверждения $e_i \parallel e_j' \Leftrightarrow V(e_i) \parallel V(e_j')$ следует из определений параллельных событий и несравнимых векторных отметок времени.

Таким образом векторные часы являются *изоморфизмом*, т.е. взаимно однозначным соответствием, сохраняющим порядок, между множеством событий распределенного вычисления $(E, \rightarrow)$ и множеством их векторных отметок времени $(V, <)$.

Стоит отметить, что если кроме векторных отметок времени событий e_i и e_j' еще известны процессы P_i и P_j, в которых эти события происходили (т.е. мы знаем значения i и j), то проверка на наличие причинно-следственной зависимости между событиями может быть упрощена согласно следующим выражениям:

$$e_i \rightarrow e_j' \Leftrightarrow V(e_i)[i] \leq V(e_j')[i],$$

$$e_i \parallel e_j' \Leftrightarrow (V(e_i)[i] > V(e_j')[i]) \wedge (V(e_i)[j] < V(e_j')[j]).$$

Первое выражение свидетельствует о том, что событие e_j' "знает" о наступлении события e_i в процессе P_i. Второе выражение говорит о том, что на момент наступления события e_j' процесс P_j "не знает" о событии e_i в процессе P_i (первая часть этого выражения), равно как и процесс P_i "не знает" о событии e_j' в процессе P_j на момент наступления e_i (вторая часть этого выражения). Поэтому при известных значениях i и j двух событий e_i и e_j' достаточно сравнить всего два компонента их векторных отметок времени.

Строгая непротиворечивость. Векторные часы являются строго непротиворечивыми, что позволяет по отметкам времени двух событий судить о том, оказывают ли эти события потенциальное влияние друг на друга или нет. Поэтому, в отличие от скалярных часов, позволяющих построить *одно из эквивалентных* выполнений распределенной системы, векторные часы могут быть использованы для определения *всех возможных* выполнений.

Свойство строгой непротиворечивости логических часов является очень важным: благодаря ему векторные часы широко применяются при построении распределенных алгоритмов. Например, механизм векторных часов используется в коммуникационных сетях, сохраняющих причинно-следственный порядок доставки сообщений, в хранилищах данных, поддерживающих причинную непротиворечивость, а также применяется для отладки распределенных программ (англ. *distributed debugging*) и при установке непротиворечивых точек восстановления (англ. *checkpoints*) распределенной системы в случае сбоя.

Основным недостатком механизма векторных часов является необходимость хранить для каждого события и пересылать в каждом сообщении отметки времени, состоящие из N целых чисел, что может приводить к значительным накладным расходам, особенно, в случае большого числа N процессов, работающих в распределенной системе. Однако было показано, что векторами меньшей длины для определения причинно-следственной зависимости между событиями по их отметкам времени обойтись невозможно. А именно, если события распределенного вычисления системы, состоящей из N процессов, отобразить в пространство векторов длины n так, чтобы $V(e_i) < V(e_j')$ было необходимым и достаточным условием для $e_i \rightarrow e_j'$, то неизбежно будем иметь $n \geq N$.

Подсчет событий. Если значение прироста логических локальных часов всегда равно единице ($d = 1$), то i-й компонент векторных часов процесса P_i, $V_i[i]$, будет определять количество событий, произошедших в P_i до настоящего момента. Если событие e_i имеет векторную отметку времени $V(e_i)$, то j-й компонент вектора $V(e_i)[j]$ содержит в себе число

событий, выполненных в процессе P_j и предшествующих e_i согласно отношению причинно-следственного порядка. Очевидно, величина $\sum_j V(e_i)[j] - 1$ представляет собой общее число событий, произошедших раньше e_i в распределенном вычислении, т.е. принадлежащих конусу прошлого данного события. Здесь стоит обратить внимание, что поверхность конуса прошлого и, как следствие, *все множество* событий, произошедших раньше e_i, определяется вектором, совпадающим с $V(e_i)$ за исключением i-го компонента: для поверхности конуса прошлого i-й компонент будет на единицу меньше $V(e_i)[i]$.

Если событие e_i представляет собой внутреннее событие останова процесса при отладке распределенной системы (англ. *breakpoint event*), то глобальное состояние, следующее сразу за поверхностью конуса прошлого события e_i, будет определять контрольную точку, фиксирующую состояние системы *непосредственно после всех событий*, потенциально влияющих на e_i, (англ. *casual distributed breakpoint*). Другими словами, такое глобальное состояние не включает в себя результат наступления событий, не влияющих на e_i, и, как следствие, позволяет дать неискаженную информацию о причинах возникновения e_i. Такая распределенная контрольная точка является естественным расширением обычного понятия контрольной точки, устанавливаемой для последовательно выполняемого процесса с целью определения его состояния сразу после всех событий, предшествующих этой контрольной точке.

3.3.2 Пример использования

Банковская система. Для иллюстрации применения векторных часов при решении различных задач вернемся к рассмотренной в п. 3.2.2 задаче подсчета полной суммы денег, находящихся на счетах в разных филиалах банка. При решении этой задачи с помощью скалярных часов предполагалось, что всем процессам известен некоторый момент логического времени t, в который процессы будут подсчитывать денежные средства на своих счетах и в каналах между филиалами. Однако сама процедура достижения соглашения между процессами о таком будущем моменте t представляет определенные сложности. Действительно, при использовании механизма скалярных часов процесс не может произвольным образом выбрать значение t для согласования с другими процессами, т.к. не исключено, что на момент выбора t логические часы других процессов ушли далеко вперед, и их текущее логическое время потенциально может превышать любое предлагаемое значение t. Кроме того, если t окажется слишком большим, то придется долго ожидать наступления этого момента.

Использование векторных часов может помочь в решении этой задачи. А именно, для любых двух процессов P_i и P_j в любой момент верно

неравенство $V_i[i] \geq V_j[i]$, т.е. представление других процессов о ходе выполнения процесса P_i не может опережать ход его собственного выполнения. Поэтому в момент продвижения процессом P_i своего логического локального времени $V_i[i]$ не существует такого процесса P_j, у которого $V_j > V_i$. Это означает, что текущие показания векторных часов других процессов не могут превосходить показания векторных часов процесса P_i при выполнении в нем внутреннего события. То есть в качестве будущего момента логического времени t процесс P_i может выбрать время своего *следующего* события.

С использованием механизма векторных часов процедура согласования времени t для подсчета денежных средств могла бы определяться таким образом.

1. Процесс P_i вычисляет векторное время своего следующего, еще не наступившего события $t = V_i + (0,\ldots 0, 1, 0,\ldots 0)$, и рассылает это значение всем остальным процессам распределенной системы.
2. Затем процесс P_i приостанавливает свою работу до тех пор, пока не получит подтверждения от всех других процессов о том, что они приняли значение t. Тем самым гарантируется, что в системе не окажется ни одного процесса, показания часов которого превзойдут t до момента окончания процедуры согласования.
3. После того как значение t становится известным всем процессам распределенной системы, процесс P_i увеличивает показания своих логических локальных часов, т.е. V_i становится равным t, и рассылает пустое сообщение всем другим процессам с тем, чтобы показания их векторных часов превзошли t.

В остальном алгоритм подсчета денежных средств остается таким же, как и в п. 3.2.2.

Основной недостаток предложенной выше схемы заключается в том, что процесс P_i вынужден приостанавливать свою работу на время информирования других процессов о выбранном значении t. Избежать этого можно с помощью дополнительного $(N + 1)$-го виртуального процесса, чьи логические часы управляются процессом P_i, предлагающим t. В этом случае P_i может назначить t с помощью увеличения логического локального времени этого $(N + 1)$-го виртуального процесса, тем самым получая возможность продолжать свою работу наравне со всеми другими процессами распределенной системы.

3.4. Методы эффективной реализации векторных часов

Размер векторных отметок времени определяется числом N процессов, участвующих в распределенном вычислении, и растет линейно с увеличением N. В случае, когда число процессов велико, использование

механизма векторных часов будет приводить к необходимости пересылки значительного объема данных с каждым сообщением. Поэтому если в распределенной системе работают сотни процессов даже при небольшом числе событий, происходящих всего в нескольких процессах, доля накладных расходов в каждом пересылаемом сообщении может достигать недопустимых величин. Несмотря на то, что для выполнения условия строгой непротиворечивости длина векторных отметок времени не может быть меньше числа процессов в распределенной системе, существует ряд методов, позволяющих реализовать механизм векторных часов более эффективно.

3.4.1 Дифференциальная пересылка векторного времени

Метод дифференциальной пересылки векторного времени основан на том наблюдении, что между последовательными отправками сообщений одному и тому же получателю обычно изменяется лишь небольшое число компонентов векторного времени процесса-отправителя. Это становиться особенно заметным, когда распределенная система состоит из большого числа процессов, но информацией друг с другом активно обмениваются лишь немногие из них. Поэтому, когда процесс P_i отправляет сообщение процессу P_j, предлагается вместе с этим сообщением передавать только те компоненты векторного времени процесса P_i, которые изменились с момента последней отправки сообщения процессу P_j.

Таким образом, если с момента последней отправки сообщения процессу P_j в процессе P_i изменились только компоненты i_1, i_2, … i_n его векторного времени V_i на значения соответственно v_1, v_2, … v_n, то в новом сообщении процесс P_i пересылает лишь множество пар: $\{(i_1, v_1), (i_2, v_2), \ldots (i_n, v_n)\}$. Когда процесс P_j получает такое сообщение, он обновляет свое векторное время согласно следующему правилу:

$$V_j[i_k] = \max(V_j[i_k], v_k),\ 1 \le k \le n.$$

Данный метод позволяет уменьшить накладные расходы на передачу сообщений и, как следствие, снизить требования к емкости каналов связи и размеру буфера приема-передачи на отправляющей и принимающей стороне. Конечно, в наихудшем случае процессу придется передавать все компоненты своего векторного времени, однако, в среднем можно ожидать, что размер такой дифференциальной отметки времени, передаваемой с каждым сообщением, будет меньше, чем размер всего вектора V_i. Стоит обратить внимание, что для правильной работы этого метода каналы должны обладать свойством FIFO.

Метод дифференциальной пересылки векторного времени требует, чтобы каждый процесс хранил значения всех своих отметок времени для последних событий отправки сообщений каждому другому процессу в распределенной системе. Прямая реализация данного требования приводит

к накладным расходам на хранение отметок времени в каждом процессе с порядком роста $O(N^2)$. Однако существует метод, позволяющий уменьшить накладные расходы, связанные с хранением дополнительной информации в каждом процессе, до $O(N)$. Суть его заключается в следующем.

Каждый процесс P_i дополнительно поддерживает работу с двумя массивами:

- $LS_i[1..N]$ (от англ. *Last Sent*). Элемент $LS_i[j]$ содержит значение логического локального времени процесса P_i, $V_i[i]$, на момент последней отправки процессом P_i сообщения процессу P_j.
- $LU_i[1..N]$ (от англ. *Last Update*). Элемент $LU_i[j]$ содержит значение логического локального времени процесса P_i, $V_i[i]$, на момент последнего изменения процессом P_i j-го компонента своего векторного времени $V_i[j]$.

Очевидно, $LU_i[i] = V_i[i]$ в каждый момент выполнения процесса P_i, а значение $LU_i[j]$ обновляется при получении процессом P_i сообщения, вызывающего изменение $V_i[j]$. В свою очередь значение элемента $LS_i[j]$ обновляется каждый раз при отправке процессом P_i сообщения процессу P_j.

Используя эти два массива, можно заключить, что с момента последней отправки процессом P_i сообщения процессу P_j в процессе P_i изменились только те k компонентов его векторного времени $V_i[k]$, для которых $LS_i[j] < LU_i[k]$. Поэтому, согласно методу дифференциальной пересылки векторного времени, только эти компоненты должны быть переданы в новом сообщении из P_i в P_j. Таким образом, когда процесс P_i отправляет сообщение процессу P_j, вместе с этим сообщением в качестве отметки времени он пересылает лишь множество пар

$$\{(k, V_i[k]): LS_i[j] < LU_i[k]\},$$

вместо пересылки всего вектора V_i, состоящего из N компонентов.

Пример работы метода дифференциальной пересылки векторных отметок времени приведен на рис. 3.7. На этом рисунке второе сообщение, переданное процессом P_3 в процесс P_2 с отметкой времени $\{(3,2)\}$, информирует процесс P_2 о том, что третий компонент векторного времени был изменен, и его новое значение равно двум. Передаваемая отметка времени выглядит именно так, потому что с момента последней отправки сообщения из P_3 в P_2 процесс P_3 изменил показания своих локальных часов с единицы до двух.

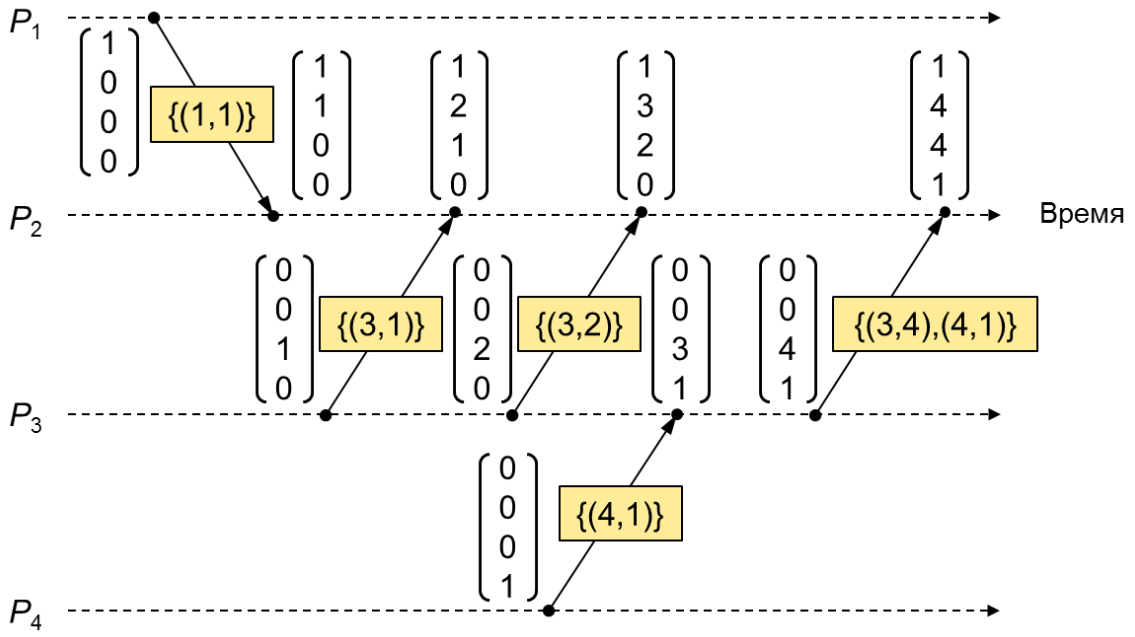


Рис. 3.7. Пример работы метода дифференциальной пересылки векторного времени.

Таким образом можно сделать вывод, что применение представленного метода дифференциальной пересылки векторного времени может привести к уменьшению накладных расходов, связанных с передачей сообщений, особенно, в случае, когда взаимодействие процессов в распределенной системе носит локальный характер.

3.4.2 Часы, фиксирующие прямую зависимость

Механизм логических часов, фиксирующих только прямую зависимость между процессами, позволяет снизить накладные расходы на передачу сообщений путем пересылки всего одной скалярной величины с каждым сообщением. В этой связи "полноценное" векторное время, позволяющее отслеживать причинно-следственный порядок событий распределенного вычисления, становится недоступным для процессов непосредственно во время их выполнения, т.е. в режиме *on-line*. Вместо этого, каждый процесс поддерживает лишь информацию, позволяющую выявлять только *прямую зависимость* его событий от событий в других процессах. При этом говорят, что событие e_j' процесса P_j находится в прямой зависимости от события e_i другого процесса P_i, если между наступлением событий e_i и e_j' состоялась *прямая* передача сообщения из P_i в P_j. Более формально: событие e_j' оказывается в прямой зависимости от события e_i тогда и только тогда, когда $i = j$ и $e_i \rightarrow_i e_j'$, или когда $i \neq j$ и существуют взаимосвязанные события s и r отправки и получения одного и того же сообщения, такие что $(e_i \rightarrow_i s) \vee (e_i = s)$ и $(e_j' = r) \vee (r \rightarrow_j e_j')$. При учете только прямых зависимостей векторная отметка времени любого события, позволяющая определить его транзитивные зависимости, может

быть рекурсивно вычислена в режиме *off-line* при помощи алгоритма, который исследует ранее зафиксированную процессами информацию о прямой зависимости их событий.

В механизме логических часов, фиксирующих прямую зависимость, каждый процесс P_i вместо вектора V_i поддерживает работу с так называемым *вектором прямых зависимостей* (англ. *direct dependency vector*) $D_i[1..N]$. В компоненте $D_i[k]$ этого вектора хранится локальное время процесса P_k на момент отправки последнего сообщения из P_k в P_i. Отметка времени события e_i определяется всеми компонентами D_i на момент наступления e_i и обозначается через $D(e_i)$. Компоненты $D_i[k]$ инициализируется нулем, $1 \le k \le N$, и каждый процесс P_i использует следующие правила для работы с D_i.

- Перед выполнением любого события процесс P_i вычисляет новое значение своего логического локального времени и записывает его в компонент $D_i[i]$, т.е. $D_i[i] = D_i[i] + 1$.
- Когда процесс P_i отправляет сообщение процессу P_j, вместе с сообщением передается только значение его локального времени $D_i[i]$.
- Когда процесс P_j получает от процесса P_i сообщение, содержащее значение времени d, он обновляет i-й компонент своего вектора зависимостей согласно следующему правилу:

$$D_j[i] = \max(D_j[i], d).$$

Отметим, что новое значение $D_j[i]$ вычисляется именно как максимум из старого значения $D_j[i]$ и времени d на случай канала, не обладающего свойством FIFO, т.е. когда сообщение от процесса P_i с б*о*льшим значением d может опередить предшествующее ему сообщение с меньшим значением d.

Из представленных выше правил работы с вектором D_i видно, что D_i характеризует только прямые зависимости текущего состояния процесса P_i от событий в других процессах. Другими словами, в любой момент значение $D_i[j]$ соответствует порядковому номеру последнего события в процессе P_j, от которого текущее состояние процесса P_i зависит напрямую. Важно отметить, что такое событие процесса P_j могло наступить раньше другого события в P_j, от которого текущее состояние P_i зависит согласно отношению причинно-следственного порядка.

Пример работы часов, фиксирующих прямую зависимость, приведен на рис. 3.8. На этом рисунке, когда процесс P_4 отправляет сообщение процессу P_3, вместе с этим сообщением он передает только значение своего логического времени, равное двум, что позволяет отследить прямую зависимость второго события e_3^2 процесса P_3 от второго события e_4^2 процесса P_4. Затем процесс P_3 отправляет сообщение в процесс P_2, тем

самым устанавливая прямую зависимость P_2 от P_3. Таким образом, в действительности, четвертое событие e_2^4 процесса P_2 становится зависимым от второго события e_4^2 процесса P_4 согласно отношению причинно-следственного порядка. Однако, процесс P_2 не получал об этом никакой информации и непосредственно по ходу своего выполнения установить эту зависимость не может. Более того, в четвертом элементе вектора зависимостей процесса P_2, $D_2[4]$, содержится информация о более раннем событии e_4^1 процесса P_4, от которого e_2^4 зависит напрямую.

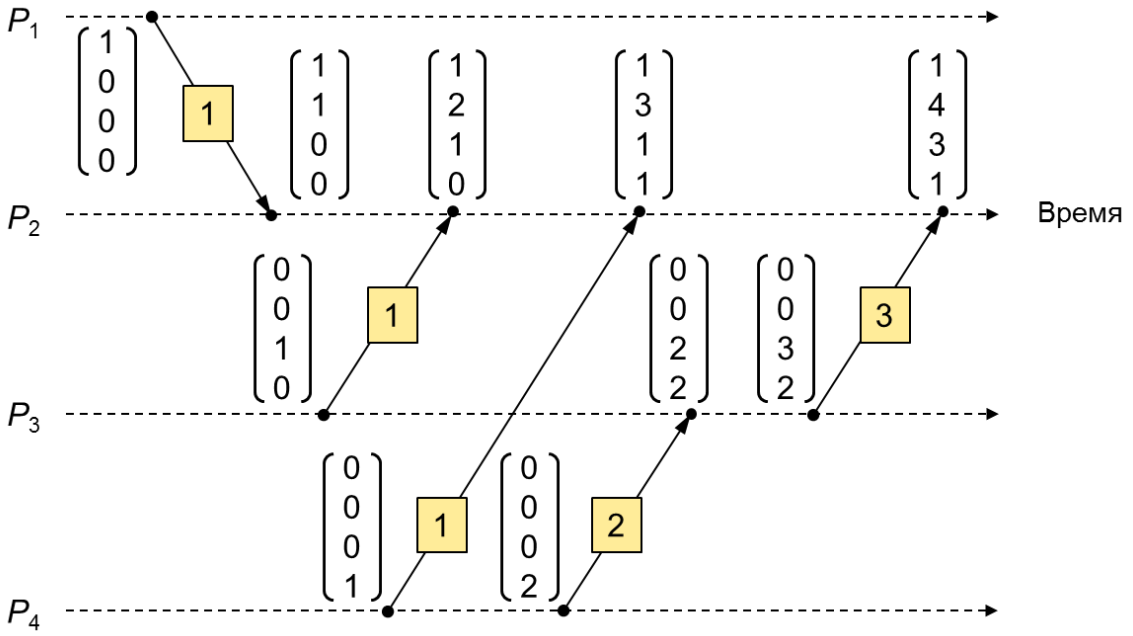


Рис. 3.8. Пример работы часов, фиксирующих прямую зависимость.

Этот пример наглядно демонстрирует тот факт, что рассматриваемый механизм логических часов сам по себе не сохраняет транзитивные зависимости причинно-следственной связи между событиями. Такие зависимости могут быть определены только в режиме *off-line* с помощью рекурсивного поиска на множестве векторов прямых зависимостей, зафиксированных процессами для своих событий. Очевидно, такой поиск приводит к дополнительным вычислительным затратам и задержкам определения векторного времени того или иного события. Поэтому данный механизм логических часов применяется только для решения таких задач, которые не требуют выявления транзитивной причинно-следственной зависимости между событиями непосредственно "на лету" по ходу выполнения процессов. К таким задачам, например, можно отнести отладку распределенных программ (англ. *distributed debugging*), когда анализ причины, по которой программа ведет себя не так, как этого ожидают, проводится в режиме *off-line*, исходя из результатов, полученных при ошибочном выполнении.

Все множество событий, от которых заданное событие e_i зависит транзитивно, может быть определено исходя из определения свойства транзитивности, т.е. путем последовательного объединения информации о событиях e_j', от которых зависит e_i, с информацией о событиях e_k'', от которых в свою очередь зависят события e_j'. Для иллюстрации данного принципа обратимся к рис. 3.8. Здесь вывод о транзитивной зависимости четвертого события e_2^4 процесса P_2 от второго события e_4^2 процесса P_4 можно сделать, объединяя зафиксированную в $D(e_2^4)$ информацию о зависимости e_2^4 от третьего события e_3^3 процесса P_3 и зафиксированную в $D(e_3^3)$ информацию о зависимости e_3^3 от e_4^2.

Общая схема рекурсивного алгоритма, определяющего множество событий, произошедших раньше x-го по порядку события e_i^x процесса P_i, приведена в Листинге 3.1. Данный алгоритм находит события других процессов из поверхности конуса прошлого события e_i^x, то есть, по сути, формирует векторную отметку времени для e_i^x и сохраняет ее в массиве `DTV`. Функция `DependencyTrack(i,x)` инициализирует i-й элемент массива `DTV` порядковым номером x заданного события e_i^x. Это же значение является логическим локальным временем события e_i^x в процессе P_i и хранится в i-ом компоненте отметки времени $D(e_i^x)$, зафиксированной для e_i^x. Остальные элементы `DTV` инициализируются нулем. Затем алгоритм анализирует отметку времени каждого события, указанного в каждом элементе `DTV`, при необходимости рекурсивно вызывая функцию `VisitEvent(j,y)`, где `j` – идентификатор процесса, а `y` – порядковый номер события в этом процессе. Каждый раз, когда `VisitEvent(j,y)` находит в векторе $D(e_j^y)$ значение, превосходящее порядковый номер, хранящийся в соответствующем элементе `DTV`, в этот элемент `DTV` записывается найденное новое значение. После этого с помощью `VisitEvent` осуществляется переход к новому найденному (более позднему) событию и сравнение его отметки времени с `DTV`. Таким образом, функция `VisitEvent` рекурсивно проходит события из конуса прошлого события e_i^x. При окончании поиска массив `DTV` будет содержать порядковые номера всех последних событий, произошедших раньше заданного события e_i^x.

Листинг 3.1. Общая схема алгоритма рекурсивного поиска транзитивных зависимостей.

```
void DependencyTrack(ProcessID i, EventIndex x) {
     /* Построение векторной отметки времени события e_i^x */
     /* Результат помещается в массив DTV */
     EventIndex DTV[N];

     void VisitEvent(ProcessID j, EventIndex y) {
     /* Сохраняет зависимости события e_j^y в DTV */
```

```
    /* Массив D представляет собой отметку времени
    события e_j^y */

    for (int k=0; k<N; k++)
         if (k != j)
              if (D[k] > DTV[k])
              { DTV[k] = D[k]; VisitEvent(k, D[k]) }
    }

    /* Инициализация DTV */
    for (int k=0; k<N; k++)
         if (k != i) DTV[k] = 0; else DTV[k] = x;

    VisitEvent(i, x);
}
```

Проиллюстрируем работу представленного алгоритма на примере поиска транзитивных зависимостей для четвертого события процесса P_2 (см. рис. 3.8). Вызов функции `DependencyTrack` происходит с параметрами `i` = 2, `x` = 4 и массив `DTV` приобретает вид (0 4 0 0). Затем вызывается функция `VisitEvent(2,4)`. Для этого события вектор прямых зависимостей D_2 имеет вид (1 4 3 1), поэтому `DTV` изменяется на (1 4 0 0) и осуществляется переход к первому событию процесса P_1 с помощью вызова `VisitEvent(1,1)`. У этого события вектор D_1 = (1 0 0 0). Так как ни один элемент вектора D_1 не превышает соответствующий элемент `DTV`, то происходит возврат из функции `VisitEvent(1,1)` и продолжается проверка вектора D_2. На этот раз обнаруживается, что третий элемент вектора D_2 превышает соответствующий элемент `DTV` (а именно, 3 > 0). Поэтому `DTV` принимает вид (1 4 3 0), и с помощью `VisitEvent(3,3)` осуществляется переход к третьему событию процесса P_3. Вектор зависимостей этого события D_3 = (0 0 3 2). Поскольку четвертый элемент D_3 больше, чем соответствующий элемент `DTV` (а именно, 2 > 0), `DTV` изменяется на (1 4 3 2), и вызывается `VisitEvent(4,2)`. Поскольку ни один элемент D_4 = (0 0 0 2) не превосходит `DTV`, происходит возврат из `VisitEvent(4,2)`, равно как и из `VisitEvent(3,3)`, т.к. все элементы D_3 оказываются просмотренными. Значение четвертого элемента D_2 оказывается меньше полученного значения четвертого элемента `DTV` (а именно, 1 < 2), поэтому функция `DependencyTrack` завершается. Полученное значение векторной отметки времени (1 4 3 2) полностью отражает все транзитивные причинно-следственные зависимости для заданного четвертого события процесса P_2.

Таким образом можно заключить, что механизм часов, фиксирующих прямую зависимость, обладает значительно меньшими накладными расходами на передачу сообщений в сравнении с

традиционными векторными часами, т.к. позволяет передавать с каждым сообщением всего одну скалярную величину. При этом данный механизм предоставляет возможность восстановить все транзитивные причинно-следственные зависимости в режиме *off-line* с помощью рекурсивного анализа векторов прямой зависимости, зафиксированных процессами.

3.4.3 Адаптивный метод Жарда-Жордана

В рассмотренном выше механизме логических часов зависимость между процессами устанавливается с помощью передачи одной единственной скалярной величины, что позволяет зафиксировать только прямую зависимость событий одного процесса от событий в других процессах. Поэтому для определения транзитивной причинно-следственной зависимости между событиями возникает необходимость для каждого процесса *регистрировать каждое* событие получения сообщения, т.е. обновлять и сохранять вектор прямых зависимостей в момент наступления такого события (или, по крайней мере, до наступления следующего события отправки сообщения в этом процессе). В противном случае транзитивную зависимость между событиями будет невозможно восстановить, т.к. цепочка прямых зависимостей окажется разорванной. Если в ходе распределенного вычисления процессы активно обмениваются сообщениями, требование регистрации каждого события получения сообщения может привести к необходимости сохранения большого объема данных и дополнительной нагрузке на процессы.

Метод Жарда-Жордана позволяет адаптивно регистрировать только часть событий, т.е. сохранять информацию об их зависимостях, обеспечивая при этом возможность определять транзитивную причинно-следственную зависимость между *зарегистрированными* событиями. Другими словами, данный метод позволяет выделять непустое подмножество существенных (регистрируемых) событий $E_R \subseteq E$ и фиксировать причинно-следственный порядок $\rightarrow_R$, индуцированный исходным порядком $\rightarrow$ на множестве E. Для этого процессы по ходу своего выполнения должны отслеживать не только прямую зависимость, но и более длинные цепочки зависимостей между любыми двумя регистрируемыми событиями. Это приводит к необходимости пересылать в сообщениях больше информации, нежели несет в себе одна скалярная величина. Поэтому метод Жарда-Жордана можно рассматривать как промежуточное решение между отслеживанием только прямой зависимости (когда с каждым сообщением передается всего один скаляр, но требуется регистрировать каждое событие получения) и отслеживанием полной транзитивной зависимости в векторных часах (когда с каждым сообщением передается вся векторная отметка времени, полностью описывающая все зависимости передающего процесса).

В основе метода Жарда-Жордана лежит следующее наблюдение: если вместе с регистрацией события e_i сохраняется вся информация обо всех его зависимостях, то последующие регистрируемые события смогут определить свои транзитивные зависимости, используя информацию, сохраненную при наступлении e_i. Поэтому, когда событие e_i регистрируется, вся информация о зависимостях, накопленная процессом P_i к этому моменту, сохраняется вместе с этим событием, а соответствующие структуры данных процесса P_i, поддерживающие эту информацию, сбрасываются в состояние зависимости только от события e_i. В дальнейшем, когда процесс P_i распространяет информацию о зависимостях, он распространяет только информацию о зависимостях между событиями, произошедшими после e_i. При этом следующее регистрируемое событие (в этом или в любом другом процессе) сможет "узнать" о других зарегистрированных событиях, произошедших раньше e_i, с помощью информации, сохраненной вместе с e_i. Данный метод по-прежнему не позволяет определять транзитивную причинно-следственную зависимость непосредственно по ходу выполнения процессов в режиме *on-line*, однако, в отличие от механизма логических часов, фиксирующих только прямую зависимость, позволяет избежать необходимости сохранения длинной истории событий.

Для реализации представленного подхода по аналогии с отношением прямой зависимости, рассмотренным в предыдущем пункте, вводится отношение *псевдо-прямой зависимости* $<<$ между событиями распределенного вычисления. А именно, любые два зарегистрированных события e_i и e_j', происходящие в различных процессах P_i и P_j, связаны отношением псевдо-прямой зависимости, $e_i << e_j'$, тогда и только тогда, когда между e_i и e_j' имеется *цепочка из взаимосвязанных событий отправки и получения* сообщений, не содержащая других зарегистрированных событий. Более формально: $e_i << e_j'$, тогда и только тогда, когда $i = j$ и $e_i \rightarrow_i e_j'$, или когда $i \neq j$ и существуют цепочка s_1, r_1, s_2, r_2, … s_n, r_n взаимосвязанных событий s_k и r_k отправки и получения сообщений, такая что $(e_i \rightarrow_i s_1) \vee (e_i = s_1)$, события r_k и s_{k+1} происходят в одном и том же процессе, и r_k наступает раньше s_{k+1}, $(e_j' = r_n) \vee (r_n \rightarrow_j e_j')$, и на этой цепочке не встречается ни одного другого зарегистрированного события.

Для отслеживания этой зависимости между событиями каждый процесс P_i поддерживает работу с так называемыми *частично-векторными часами* PV_i (англ. *partial vector clock*), представляющими собой множество упорядоченных пар вида (j, x), каждая из которых указывает на то, что текущее состояние P_i находится в псевдо-прямой зависимости от зарегистрированного события e_j^x с порядковым номером x в процессе P_j. Пара (i, v) определяет текущее логическое локальное время v процесса P_i. Зависимости регистрируемого события e_i сохраняются в его отметке

времени, которая определяется показаниями частично-векторных часов на момент наступления e_i и обозначается через $PV(e_i)$. Исходные показания частично-векторных часов для P_i: $PV_i = \{(i, 0)\}$.

Пусть $PV_i = \{(i_1, v_1), \ldots (i, v), \ldots (i_n, v_n)\}$ – текущие показания частично-векторных часов процесса P_i. По методу Жарда-Жордана каждый процесс P_i использует следующие правила для работы со своими логическими часами:

- При регистрации очередного $(v + 1)$-го события e_i^{v+1} процесс P_i вычисляет новое значение своего логического локального времени и записывает его в PV_i:

  $$PV_i = \{(i_1, v_1), \ldots (i, v + 1), \ldots (i_n, v_n)\}.$$

  После этого показания частично-векторных часов процесса P_i сохраняются в отметке времени $PV(e_i^{v+1})$ и затем сбрасываются в состояние зависимости только от события e_i^{v+1}:

  $$PV_i = \{(i, v + 1)\}.$$

- При отправке сообщения из P_i в P_j вместе с сообщением передаются текущие показания PV_i.

- Когда процесс P_i принимает сообщение с отметкой времени $PV_{msg} = \{(i_{m1}, v_{m1}), (i_{m2}, v_{m2}), \ldots (i_{mk}, v_{mk})\}$, он устанавливает показания своих частично-векторных часов PV_i в виде множества следующих пар:
  1. все пары (i_{mx}, v_{mx}), такие что $(i_{mx}, .)$ не содержатся в PV_i;
  2. все пары (i_x, v_x), такие что $(i_x, .)$ не содержатся в PV_{msg};
  3. пары $(i_x, \max(v_x, v_{mx}))$ для всех $(i_x, .)$, которые содержатся и в PV_i и в PV_{msg}.

Пример работы адаптивного метода Жарда-Жордана приведен на рис. 3.9; регистрируемые события отмечены жирным кружочком. Рассмотрим работу процесса P_2. Исходные показания логических часов PV_2 этого процесса равны (2, 0). При регистрации события e_2^1 процесс P_2 вычисляет новое значение своего локального времени и копирует его в отметку времени $PV(e_2^1)$. При получении сообщения от процесса P_1 в показания логических часов PV_2 добавляется пара (1, 0). Аналогичная ситуация возникает при получении сообщения от процесса P_3 с отметкой времени {(3, 1), (4, 0)}. В итоге $PV_2 = \{(1 ,0), (2, 1), (3, 1), (4, 0)\}$. Когда же регистрируется событие e_2^2, процесс P_2 увеличивает свое локальное время, и полученные показания логических часов сохраняются в отметке времени $PV(e_2^2)$ данного события. После этого показания PV_2 сбрасываются в значение (2, 2). Таким образом легко видеть, что событие e_2^2 напрямую зависит от событий e_1^0, e_2^1 и e_3^1. Кроме того, данное событие находится в отношении псевдо-прямой зависимости от события e_4^0.

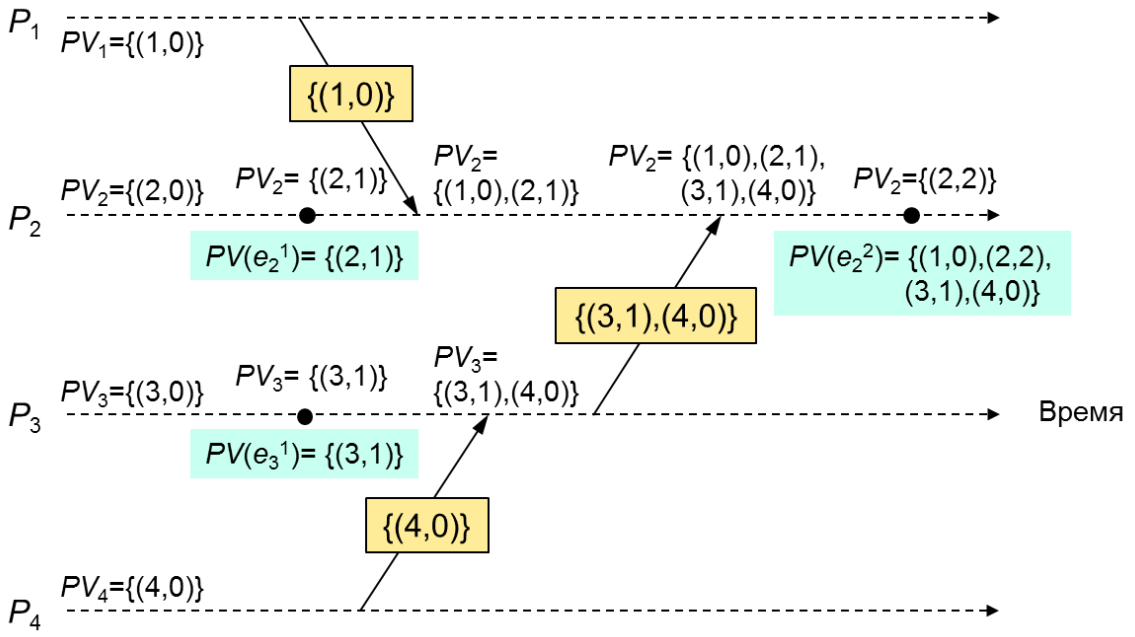


Рис. 3.9. Пример работы адаптивного метода Жарда-Жордана.

Транзитивные зависимости заданного события e_i могут быть восстановлены путем анализа отметок времени событий, с которыми e_i связано отношением псевдо-прямой зависимости. Если такой анализ провести рекурсивно, то могут быть найдены *все* события, произошедшие раньше e_i.

Основное преимущество метода Жарда-Жордана заключается в том, что этот метод позволяет настраивать оптимальный размер отметки времени, передаваемой с каждым сообщением. А именно, количество пар, содержащихся в логических частично-векторных часах, может быть ограничено сверху путем регистрации дополнительных событий. Например, если требуется, чтобы размер логического времени не превышал k пар, достаточно, чтобы процесс зарегистрировал дополнительное событие, когда число накопленных им зависимостей достигнет k. Такое свойство адаптивной регистрации событий позволяет гибко контролировать накладные расходы на передачу сообщений при распределенном вычислении.

3.5. Матричное время

В механизме матричных часов логическое время представлено в виде матрицы неотрицательных целых чисел размером N x N. Каждый процесс P_i локально поддерживает работу с матрицей $M_i[1..N, 1..N]$ со следующими элементами.

- В элементе $M_i[i, i]$ хранятся показания локальных часов процесса P_i для измерения своего собственного хода выполнения.

- В элементе $M_i[i, j]$ содержатся последние сведения, полученные процессом P_i о локальном времени процесса P_j. Другими словами, строка $M_i[i, .]$ является векторным временем процесса P_i.
- В элементе $M_i[j, k]$ содержатся последние сведения, полученные процессом P_i о том, какой информацией обладает процесс P_j о локальном времени процесса P_k.

Таким образом всю матрицу M_i можно рассматривать как локальное представление процесса P_i не только о текущем глобальном времени, но и о том, что процессы "знают" о локальном времени друг друга. Отметка времени события e_i, происходящего в процессе P_i, определяется всеми элементами матрицы M_i на момент наступления этого события и обозначается через $M(e_i)$.

Каждый процесс P_i использует следующие правила для работы со своими матричными часами.

Правило 1: перед выполнением любого события процесс P_i увеличивает показания своих локальных часов $M_i[i, i]$:

$$M_i[i, i] = M_i[i, i] + d, \text{ где } d > 0.$$

Как и для случая скалярного времени, каждый раз d может принимать любые значения. Однако обычно d всегда полагают равным единице.

Перед выполнением события получения сообщения процесс вынужден произвести немного другие действия согласно следующему правилу.

Правило 2: в каждое передаваемое сообщение добавляется значение матричного времени M_i процесса-отправителя P_i на момент отправки этого сообщения. Когда процесс P_j получает сообщение от процесса P_i с отметкой времени M_{msg}, он выполняет следующие шаги:

1. обновляет показания своих векторных часов в строке $M_j[j, .]$ согласно векторному времени процесса P_i, полученному в строке $M_{msg}[i, .]$:

   $$1 \le k \le N: M_j[j, k] = \max(M_j[j, k], M_{msg}[i, k]);$$

   остальные элементы матрицы M_j обновляются по правилу:

   $$1 \le k, l \le N: M_j[k, l] = \max(M_j[k, l], M_{msg}[k, l]).$$

2. исполняет **Правило 1**;
3. доставляет сообщение и приступает к его обработке.

Элементы матрицы $M_i[k, l]$ инициализируется нулем, $1 \le k, l \le N$.

Пример работы алгоритма матричных часов для $d = 1$ приведен на рис. 3.10. Рассмотрим матричную отметку времени для события e_2^6 процесса P_2. Событие e_1^5 является последним событием в процессе P_1, которое предшествует e_2^6 согласно отношению причинно-следственного порядка. Поэтому $M(e_2^6)[2,1] = M(e_2^6)[1,1] = 5$. Аналогично, $M(e_2^6)[2,3] =$

$M(e_2^6)[3,3] = 4$. По сведениям процесса P_2 последним событием процесса P_1, о котором "знает" процесс P_3, является событие e_1^2. Поэтому $M(e_2^6)[3,1] = 2$. Аналогично, e_3^4 – последнее событие в P_3, о котором "знает" P_1 по сведениям P_2, т.е. $M(e_2^6)[1,3] = 4$.

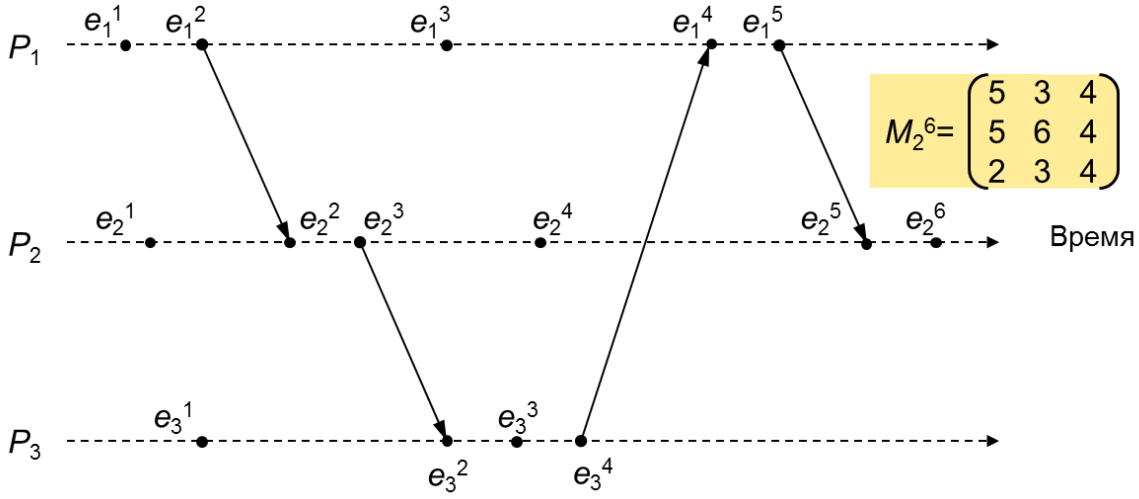


Рис. 3.10. Пример работы алгоритма матричных часов.

3.5.1 Основные свойства

По определению строка $M_i[i, .]$ обладает всеми свойствами векторных часов. Кроме этого, если для матричного времени процесса P_i верно неравенство $\min_k(M_i[k, l]) \geq t$, то по сведениям процесса P_i каждый из процессов P_k "знает" о том, что локальное время процесса P_l достигло значения t или превышает его. Например, исходя из матричной отметки времени события e_2^6 на рис. 3.10 процессу P_2 известно, что все процессы в системе обладают информацией о том, что ход выполнения процесса P_1 достиг события e_1^2. Из данного неравенства процесс P_i может сделать вывод, что всем другим процессам в системе известно о том, что процесс P_l уже не будет передавать информацию с отметкой своего логического локального времени меньшей, либо равной t. Во многих приложениях это означает, что процессам больше не нужно хранить какие-либо данные, полученные ранее от процесса P_l, и они могут без проблем удалять устаревшую информацию (англ. *obsolete information*).

Если значение прироста локальных логических часов всегда равно единице ($d = 1$), то элемент матрицы $M_i[k, l]$ будет содержать в себе число событий, выполненных в процессе P_l и известных процессу P_k настолько, насколько об этом "знает" процесс P_i.

РАЗДЕЛ 4. ВЗАИМНОЕ ИСКЛЮЧЕНИЕ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

Процессам распределенной системы часто приходиться координировать свои действия. Например, они могут соревноваться за возможность работы с разделяемыми ресурсами, требующими эксклюзивного доступа. Для организации такой работы, во-первых, нужно в коде каждого процесса выделять некоторую его часть, называемую критической секцией (англ. *critical section*), в которой есть обращения к таким ресурсам. Во-вторых, необходимо реализовать тот или иной механизм взаимного исключения (англ. *mutual exclusion*), выражающийся в удовлетворении следующего основного требования: во время выполнения критической секции одного процесса с обращениями к разделяемым ресурсам ни один другой процесс не должен выполняться в своей критической секции, работающей с любым из этих ресурсов.

В некоторых случаях совместно используемые ресурсы управляются специализированными процессами (серверами), которые в том числе обеспечивают взаимное исключение при доступе к управляемым ресурсам со стороны других процессов (клиентов). Однако бывают ситуации, когда равноправные процессы должны координировать обращения к общему ресурсу самостоятельно между собой. В таких случаях требуются отдельные механизмы взаимного исключения, независимые от конкретной схемы управления ресурсами.

Отсутствие общей памяти в распределенных системах не позволяет использовать разделяемые переменные (такие как семафоры) для решения рассматриваемой задачи: распределенные алгоритмы взаимного исключения должны опираться исключительно на обмен сообщениями между процессами. Разработка таких алгоритмов осложняется тем, что приходится иметь дело с произвольными задержками передачи сообщений и отсутствием полной информации о состоянии всей системы. Также не делается никаких предположений об относительной скорости выполнения процессов и об их количестве.

В данном разделе мы рассмотрим основные распределенные алгоритмы взаимного исключения, ключевые идеи которых используются и для решения многих других задач в распределенных системах. Кроме того, изучение этих алгоритмов позволяет раскрыть такие важные вопросы, как обеспечение свойств безопасности и живучести распределенных алгоритмов.

4.1. Общие концепции

При рассмотрении распределенных алгоритмов взаимного исключения мы будем оставаться в рамках следующих допущений:

процессы выполняются и взаимодействуют асинхронно, ни один из процессов не выходит из строя, все каналы связи являются надежными и коммуникационная сеть не разбивается на несвязанные друг с другом части. Мы не будем делать никаких предположений о том, являются ли каналы между процессами очередями (FIFO) или нет (non-FIFO). Требование к свойству очередности каналов будет определяться отдельно для каждого обсуждаемого алгоритма. Также мы будем полагать, что обращения к разделяемым ресурсам происходят из одной единственной критической секции. Распространение представленных ниже алгоритмов на случаи работы с более чем одной критической секцией не вызывает затруднений.

С позиции организации работы с разделяемыми ресурсами в коде каждого процесса можно выделить следующие части: (1) критическая секция, в которой есть обращения к разделяемым ресурсам, и (2) предшествующая и последующая часть кода, где такие обращения отсутствуют. Общая структура кода процессов распределенной системы, рассматриваемая далее для изучения механизмов взаимного исключения, приведена в Листинге 4.1. В коде каждого процесса используются две функции: *request_CS()* для получения разрешения на вход в критическую секцию и блокировки процесса в случае необходимости и *release_CS()* для выхода из критической секции и предоставления возможности остальным процессам войти в нее. Каждая из этих функций принимает в качестве аргумента общий для всех процессов идентификатор критической секции *R*. Суть разработки распределенных алгоритмов взаимного исключения как раз и состоит в построении механизмов, обеспечивающих работу этих двух функций.

Листинг 4.1. Структура кода с критической секцией.

P_1		P_N
`void P1(int R) {`	…	`void PN(int R) {`
`while(true) {`		`while(true) {`
`/* предшествующий код */;`		`/* предшествующий код */;`
`request_cs``(R);`		**`request_cs`**`(R);`
`/* критическая секция */;`		`/* критическая секция */;`
`release_cs``(R);`		**`release_cs`**`(R);`
`/* последующий код */;`		`/* последующий код */;`
`}`		`}`
`}`		`}`

Исходя из организации работы с критической секцией (КС) множество возможных состояний процесса будет определяться тремя состояниями: (1) запрос на вход в КС, (2) выполнение внутри КС и (3) выполнение вне КС. При этом возможны только переходы из состояния выполнения вне КС в состояние запроса на вход в КС (вызов функции *request_CS*), из состояния запроса на вход в КС в состояние выполнения внутри КС (возврат из функции *request_CS*) и из состояния выполнения внутри КС вновь в состояние выполнения вне КС (вызов функции *release_CS* и возврат из нее). Важно отметить, что в состоянии выполнения внутри КС процесс может находиться лишь ограниченное время, в то время как в состоянии выполнения вне КС процесс может находиться сколь угодно долго. В состоянии запроса на вход в КС процесс блокируется до получения разрешения на вход в КС и не должен формировать новых запросов для доступа к КС.

Основные требования к алгоритмам взаимного исключения формулируются следующим образом.

- Безопасность. В каждый момент времени в КС может выполняться не более одного процесса.
- Живучесть. Каждый запрос на вход в КС рано или поздно должен быть удовлетворен.

Условие живучести алгоритма взаимного исключения говорит о том, что нельзя допускать ситуации *взаимной блокировки*, т.е. бесконечного ожидания прихода сообщения, которое никогда не придет, и ситуации *голодания*, т.е. бесконечного ожидания разрешения на вход в КС одним процессом, в то время как другие процессы неоднократно получают доступ к КС.

Отсутствие голодания также является условием справедливости. Другой контекст определения справедливости обслуживания запросов на вход в КС связан с установлением порядка вхождения процессов в КС и ограничением на число позволяемых входов-выходов в КС для одного процесса, пока другой процесс ожидает входа в КС. А именно, логично потребовать, что в случае, если один запрос на вход в КС произошел раньше другого, то и доступ к КС должен быть предоставлен в таком же порядке. Если механизм взаимного исключения удовлетворяет этому требованию, и все запросы на вход в КС связаны отношением причинно-следственного порядка, то ни один процесс не сможет войти в КС более одного раза, пока другой процесс ожидает получения разрешения на вход в КС.

Все многообразие алгоритмов взаимного исключения в распределенных системах на самом деле базируется на реализации одного из двух основных принципов.

1. Алгоритмы на основе получения разрешений (англ. *permission-based algorithms*). В этом случае для входа в КС процессу требуется собрать "достаточное" число разрешений от других процессов распределенной системы. Свойство безопасности будет выполнено, если такое число разрешений сможет получить лишь один процесс из всех процессов, желающих войти в КС. Свойство живучести обычно обеспечивается за счет упорядочивания запросов по их отметкам логического времени или с помощью ациклического графа предшествования, в котором вершины соответствуют процессам распределенной системы, а направленные ребра описывают приоритет процессов друг над другом для определения порядка предоставления доступа к КС.
2. Алгоритмы на основе передачи маркера (англ. *token-based algorithms*). Для таких алгоритмов право войти в КС материализуется в виде уникального объекта – маркера, который в каждый момент времени может содержаться только у одного процесса или же находиться в канале в состоянии пересылки от одного процесса к другому. Свойство безопасности в этом случае будет гарантировано ввиду уникальности маркера. Поэтому основные усилия при разработке алгоритмов данного класса направлены на обеспечение свойства живучести, а именно, на управление перемещением маркера таким образом, чтобы он рано или поздно оказался в каждом процессе, желающем войти в КС. Существуют два подхода к решению этой задачи: непрерывное перемещение маркера среди всех процессов распределенной системы (лат. *perpetuum mobile*) или перемещение маркера лишь в ответ на получение запроса от заинтересованного в нем процесса (англ. *token asking methods*). Отметим, что при формировании запросов на владение маркером необходимо создать такие условия их распространения, при которых каждый запрос рано или поздно достигнет процесса, в котором находится маркер, вне зависимости от перемещений самого маркера.

Обратим внимание, что два перечисленных принципа построения алгоритмов взаимного исключения объединяются в централизованном алгоритме, когда доступ к КС управляется специально выделенным процессом, называемым координатором. В этом случае для входа в КС процессу достаточно получить одно единственное разрешение от координатора, который становится ответственным за поддержку информации о состояниях всех процессов и за выдачу разрешений на вход в КС таким образом, чтобы гарантировать выполнение свойств безопасности и живучести. С другой стороны, такое уникальное разрешение, выдаваемое координатором, можно рассматривать как маркер, передаваемый координатором процессу, желающему войти в КС, и

возвращаемый обратно координатору после его использования. Более подробно работу централизованного алгоритма взаимного исключения мы рассмотрим в следующем подразделе.

Централизованные и распределенные алгоритмы. Мы уже упоминали некоторые характерные свойства централизованных и распределенных алгоритмов в п. 1.3.3 при обсуждении сложностей масштабирования. Здесь мы перечислим их основные отличительные особенности.

Централизованный алгоритм предполагает, что основная часть работы выполняется одним процессом или небольшой группой процессов (в сравнении с их общим числом), в то время как остальные процессы играют незначительную роль в достижении общего результата. Обычно эта роль сводится к получению информации от главного процесса или предоставлению ему нужных данных. Поэтому централизованный алгоритм всегда является *асимметричным*: различные процессы выполняют логически разные функции, хотя, возможно, в чем-то и совпадающие. Наиболее подходящей для централизованных алгоритмов архитектурой вычислительных систем является архитектура клиент-сервер, в которой сервер владеет и распоряжается информационными ресурсами, а клиент имеет возможность воспользоваться ими. Большинство коммерческого программного обеспечения разрабатывается именно по такой архитектуре. Отметим, что с теоретической точки зрения централизованный сервер потенциально является узким местом всей системы как из-за ограничений в собственной производительности, так и из-за ограничений в пропускной способности его линий связи. Кроме того, такой сервер становится единой точкой отказа распределенной системы. На практике перечисленные проблемы чаще всего решаются путем использования реплицированных серверов, разнесенных друг от друга территориально. Однако такую конфигурацию уже нельзя считать полностью централизованной.

В распределенных алгоритмах каждый процесс играет одинаковую роль в достижении общего результата и в разделении общей нагрузки. Поэтому распределенные алгоритмы являются *симметричными*: все процессы исполняют один и тот же код и выполняют одни и те же логические функции. На самом деле абсолютно симметричные алгоритмы представляются практически идеальной конструкцией и для реализации в реальных системах более предпочтительными являются частично распределенные алгоритмы, обладающие естественной асимметрией в функциях своих узлов. К примеру, если процессы распределенной системы логически организованы в дерево, то его корень и листья обычно выполняют немного разные функции, при этом отличающиеся от функций вершин ветвления. Однако отметим, что возрастающая популярность

мобильных систем, одноранговых (англ. *peer-to-peer*) и самоорганизующихся (англ. *ad-hoc*) сетей будет требовать полностью распределенных решений.

4.2. Централизованный алгоритм

Наиболее простой способ организации взаимного исключения в распределенных системах состоит в том, что один процесс выбирается координатором, который единолично предоставляет разрешение на вход в КС. Таким процессом, например, может быть процесс с наибольшим идентификатором. Каждый раз, когда процесс распределенной системы собирается войти в КС, он посылает координатору сообщение REQUEST с соответствующим запросом и ожидает получения разрешения. Поступающие координатору запросы помещаются в очередь согласно порядку их поступления. Если на момент получения запроса ни один из процессов не находится в КС, т.е. поступивший запрос занимает первое место в очереди, координатор немедленно посылает ответ REPLY с разрешением на вход в КС. После получения ответа процесс, запросивший доступ, входит в КС. Эта ситуация проиллюстрирована на рис. 4.1а: процесс P_1 запрашивает вход в КС у координатора P_3 и получает ответ; очередь координатора P_3 изображена прямоугольником.

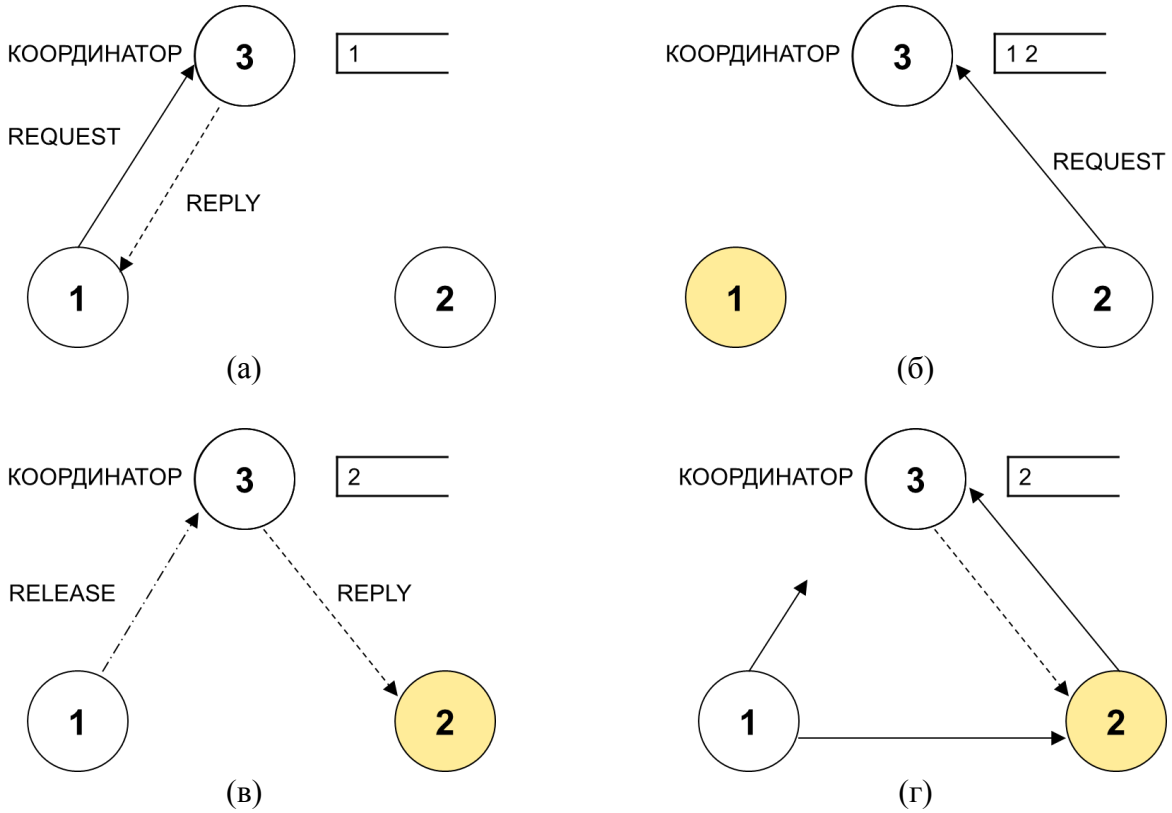


Рис. 4.1. Пример работы централизованного алгоритма взаимного исключения.

Предположим теперь, что пока процесс P_1 выполняется в КС, другой процесс P_2 запрашивает у координатора разрешение на вход в КС (см. рис. 4.1б; выполняющийся в КС процесс P_1 отмечен серым цветом). Т.к. очередь не пуста, координатор знает, что в КС уже находится процесс P_1, и не дает процессу P_2 разрешения на вход. Конкретный способ запрещения доступа зависит от особенностей системы. На рис. 4.1б координатор просто не отвечает, тем самым блокируя процесс P_2 в состоянии запроса на вход в КС. Также, координатор может послать ответ, гласящий "доступ запрещен". В любом случае запрос от P_2 помещается в конец очереди координатора. Когда процесс P_1 выходит из КС, он отправляет координатору сообщение RELEASE, тем самым предоставляя остальным процессам возможность войти в КС. При получении сообщения RELEASE координатор удаляет запрос процесса P_1 из своей очереди и отправляет разрешающее сообщение процессу, запрос которого окажется первым в его очереди; в нашем примере – процессу P_2. Увидев разрешение, P_2 войдет в КС, как показано на рис. 4.1в.

Легко заметить, что представленный алгоритм гарантирует свойство безопасности: координатор позволяет войти в КС только одному процессу за раз. Кроме того, никакой процесс никогда не ждет разрешения на вход в КС вечно, т.к. запросы обслуживаются координатором согласно порядку их поступления. Поэтому алгоритм также удовлетворяет условию живучести. Однако, строго говоря, такая схема не обеспечивает выполнение одного из аспектов свойства справедливости, согласно которому запросы на вход в КС должны удовлетворяться *в порядке их возникновения в системе,* а не в порядке их получения координатором. Как следствие, становится возможной ситуация, в которой один процесс сможет несколько раз войти и выйти из КС, в то время как другой процесс ожидает входа в КС, даже при условии того, что запрос от второго процесса произошел раньше всех запросов от первого процесса. Рассмотрим, например, случай, когда процесс P_1 отправляет запрос на вход в КС координатору P_3 и после этого отправляет сообщение процессу P_2. После получения сообщения от P_1 процесс P_2 отправляет координатору P_3 свой запрос на вход в КС. Очевидно, что если запрос от P_2 достигнет P_3 раньше запроса от P_1, то и доступ к КС будет предоставлен P_2 вперед P_1 (см. рис. 4.1г). Этот пример иллюстрирует тот факт, что порядок поступления запросов к координатору может отличаться от причинно-следственного порядка их возникновения в системе.

Тем не менее, описанный алгоритм прост в реализации и использует для работы с КС всего три сообщения: для входа в КС требуется два сообщения – REQUEST и REPLY, для выхода из КС – одно сообщение RELEASE.

4.3. Алгоритмы на основе получения разрешений

Далее мы рассмотрим некоторые *распределенные* алгоритмы взаимного исключения и начнем с обсуждения алгоритмов на основе получения разрешений.

4.3.1 Алгоритм Лэмпорта

Алгоритм, предложенный Л. Лэмпортом, является одним из самых ранних распределенных алгоритмов взаимного исключения и призван проиллюстрировать применение скалярных часов для линейного упорядочивания операций, производимых процессами в распределенной системе, таких, например, как вход в КС и выход из КС. В его основе лежат предположения о том, что все каналы связи обладают свойством FIFO, и сеть является полносвязной. Данный алгоритм обобщает рассмотренный выше централизованный алгоритм, в котором запросы на вход в КС помещаются в очередь и обслуживаются из нее по порядку, в том смысле, что позволяет рассматривать такую очередь в виде отдельного объекта, разделяемого между всеми процессами, а не находящегося под управлением одного единственного процесса (координатора). Важным преимуществом алгоритма Лэмпорта является также то, что запросы на вход в КС обслуживаются не в произвольном порядке, как в централизованном алгоритме, а *в порядке их возникновения в системе*, т.е. в порядке, определяемом их отметками логического времени. Поэтому, если один запрос на вход в КС произошел раньше другого, то и доступ к КС по первому запросу будет предоставлен раньше, чем по второму. Здесь под отметками времени запросов на вход в КС мы будем подразумевать упорядоченные пары (L_i, i), где L_i – скалярное время процесса P_i на момент формирования запроса, i – идентификатор процесса P_i. Линейный порядок таких отметок времени будет определяться правилом, введенным в п. 3.2.1.

Алгоритм Лэмпорта опирается на тот же принцип, что и алгоритм полностью упорядоченной групповой рассылки, рассмотренный в п. 3.2.2. А именно, чтобы реализовать представление общей совместно используемой очереди, каждый процесс работает с ее локальной "копией" и для определения единого для всех процессов порядка обслуживания запросов использует их отметки логического времени, упорядоченные линейно. Чтобы каждый запрос оказался в каждой из таких локальных "копий", всякий раз, когда процесс собирается войти в КС, он помещает свой запрос вместе с отметкой времени в свою локальную очередь и рассылает сообщение с этим же запросом всем остальным процессам. При получении запроса остальные процессы помещают его уже в свои локальные очереди. По порядку обслуживания запросов процесс получит право войти в КС, когда значение отметки времени его запроса окажется наименьшим среди всех других запросов. Остается лишь ответить на

вопрос: как процессу убедиться в том, что его запрос имеет наименьшую отметку времени? Действительно, из-за задержек передачи сообщений, возможна ситуация, когда в локальных очередях двух различных процессов в течение некоторого временного интервала наименьшей отметкой времени будут обладать разные запросы, что может привести к нарушению требования взаимного исключения. Эта ситуация была подробно рассмотрена при описании алгоритма полностью упорядоченной групповой рассылки в п. 3.2.2 и проиллюстрирована на рис. 3.4. Поэтому прежде чем процесс примет решение о входе в КС исходя из состояния своей собственной очереди, он должен получить от всех других процессов сообщения, гарантирующие, что в каналах не осталось ни одного сообщения с запросом, имеющим время меньшее, чем его собственный запрос. Благодаря свойству FIFO каналов связи и правилам работы логических часов такими сообщениями могут быть ответные сообщения, высылаемые при получении сообщения с запросом, или же любые другие сообщения, направляемые процессу, запрашивающему вход в КС, с отметкой времени большей, чем отметка времени его запроса.

Обозначим через Q_i локальную очередь процесса P_i, в которую помещаются все запросы на доступ к КС вместе с их отметками времени. Тогда алгоритм Лэмпорта будет определяться следующими правилами.

1. **Запрос на вход в КС.** Когда процессу P_i нужен доступ к КС, он рассылает сообщение REQUEST(L_i, i) со значением своего логического времени (L_i, i) всем остальным процессам и, кроме того, помещает этот запрос в свою очередь Q_i. Каждая такая рассылка представляет собой одно атомарное событие процесса P_i. Когда процесс P_j получает запрос REQUEST(L_i, i) от процесса P_i, он помещает его в свою очередь Q_j и отправляет процессу P_i ответ REPLY(L_j, j) со значением своего логического времени (L_j, j).

2. **Вход в КС.** Процесс P_i может войти в КС, если одновременно выполняются два условия.

   - Запрос REQUEST(L_i, i) процесса P_i обладает наименьшим значением отметки времени среди всех запросов, находящихся в его собственной очереди Q_i.
   - Процесс P_i получил сообщения от всех остальных процессов в системе с отметкой времени большей, чем (L_i, i). При условии того, что каналы связи обладают свойством FIFO, соблюдение этого правила гарантирует, что процессу P_i известно обо всех запросах, предшествующих его текущему запросу.

3. **Выход из КС.** При выходе из КС процесс P_i удаляет свой запрос из собственной очереди Q_i и рассылает всем другим процессам сообщение RELEASE(L_i, i) с отметкой своего логического времени.

По-прежнему каждая такая рассылка представляет собой одно атомарное событие процесса P_i. Получив сообщение RELEASE(L_i, i), процесс P_j удаляет запрос процесса P_i из своей очереди Q_j. После удаления процессом P_j запроса процесса P_i из Q_j отметка времени его собственного запроса может оказаться наименьшей в Q_j, и P_j сможет рассчитывать на вход в КС.

Обратим внимание, что представленный алгоритм является распределенным: все процессы следуют одним и тем же правилам, и каждый процесс принимает решение о входе в КС только на основе своей локальной информации. Также отметим, что когда получение всех сформированных запросов на доступ к КС подтверждено всеми процессами, все локальные очереди будут пребывать в одинаковом состоянии, что как раз и позволяет рассматривать их как "копии" некоторой отдельной очереди, разделяемой между процессами. Поэтому можно сказать, что решение на вход в КС принимается на основе локальной информации, которая, однако, глобально согласована.

Легко увидеть, что алгоритм Лэмпорта обладает свойствами безопасности и живучести.

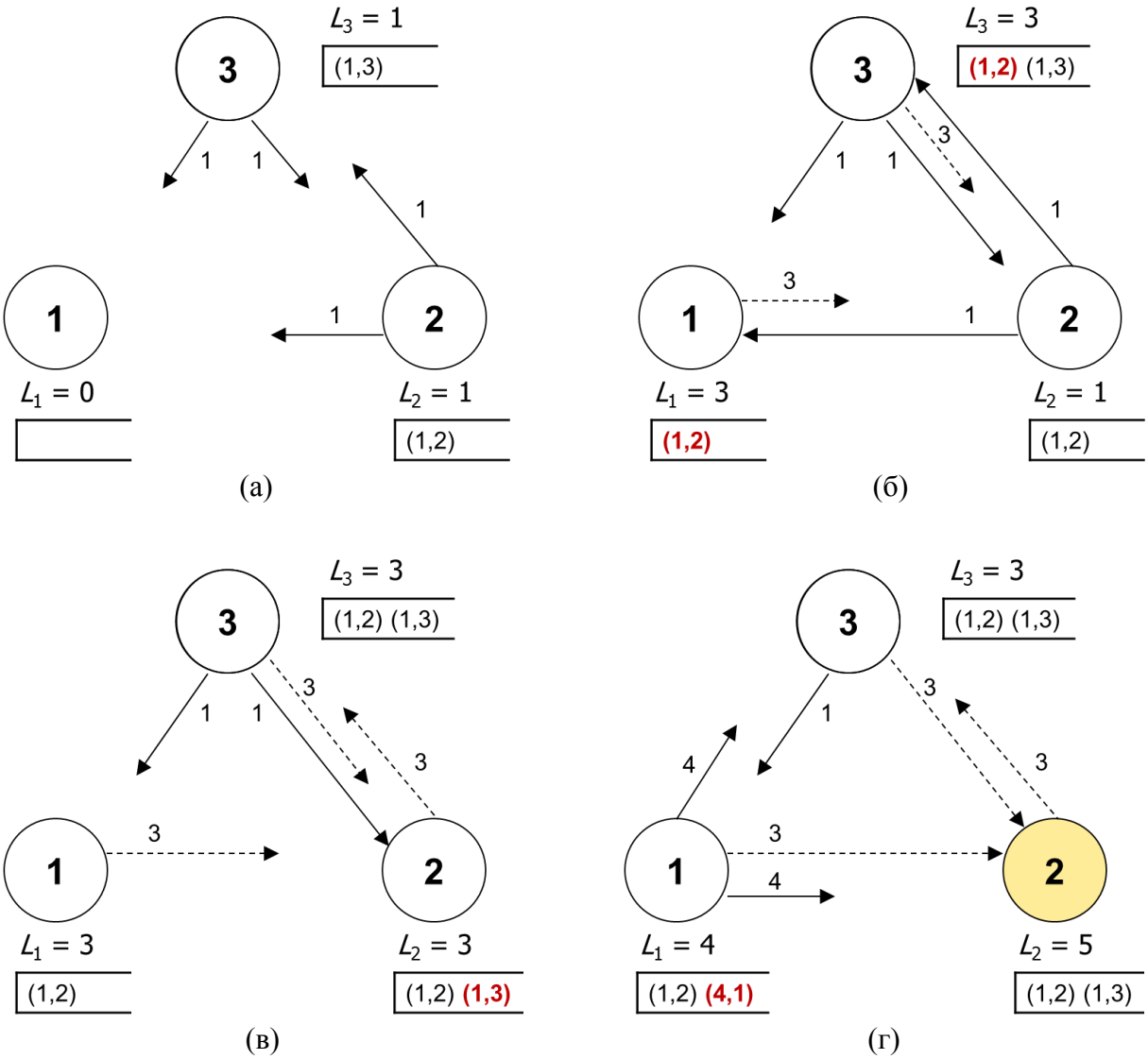
Докажем выполнение свойства безопасности от противного. Пусть два процесса P_i и P_j выполняются в КС одновременно. Это означает, что отметка времени запроса P_i оказалась наименьшей в его очереди Q_i, а отметка времени запроса P_j – наименьшей в очереди Q_j, и при этом оба процесса получили друг от друга ответные сообщения со временем б*о*льшим, чем время их запросов. Без потери общности будем считать, что значение времени запроса P_i меньше времени запроса P_j. Тогда, опираясь на свойство FIFO каналов связи и правила работы логических часов, можно утверждать, что в момент получения процессом P_j ответного сообщения от P_i в очереди Q_j уже должен был находиться запрос процесса P_i, и запрос P_j не мог иметь наименьшую отметку времени в Q_j.

Покажем теперь, что алгоритм Лэмпорта обладает свойством живучести. Действительно, механизм ответных сообщений гарантирует, что любой процесс P_i, запрашивающий доступ к КС, рано или поздно получит от всех других процессов сообщения с отметкой времени б*о*льшей, чем время его запроса. Поэтому второе условие входа в КС рано или поздно будет выполнено. Кроме того перед запросом P_i в очереди может оказаться не более $(N-1)$ запросов от других процессов с меньшими значениями отметки времени. При выходе из КС процесс P_j с наименьшим временем запроса разошлет сообщение RELEASE, которое приведет к удалению запроса P_j из очередей всех процессов, в том числе из очереди Q_i. Последующие запросы на вход в КС от P_j будут иметь время б*о*льшее, чем время запроса P_i, и будут обслужены после P_i. Поэтому за ограниченное число шагов отметка времени запроса P_i окажется наименьшей в Q_i и

первое условие входа в КС также окажется выполненным. Следовательно, любой процесс, запрашивающий доступ к КС, в конце концов, сможет войти в нее.

Также отметим, что доступ к КС предоставляется строго в соответствии с отметками времени запросов по порядку, сохраняющему частичный причинно-следственный порядок →. Поэтому можно утверждать, что запросы на вход в КС обслуживаются в порядке их возникновения в системе.

Пример работы распределенного алгоритма Лэмпорта приведен на рис. 4.2. На рис. 4.2а приблизительно в одно и то же время процессы P_2 и P_3 запрашивают вход в КС, изменяя показания своих скалярных часов L_2 и L_3; локальная очередь каждого процесса изображена прямоугольником (будем считать, что в очередях запросы сортируются в порядке возрастания их отметок времени).



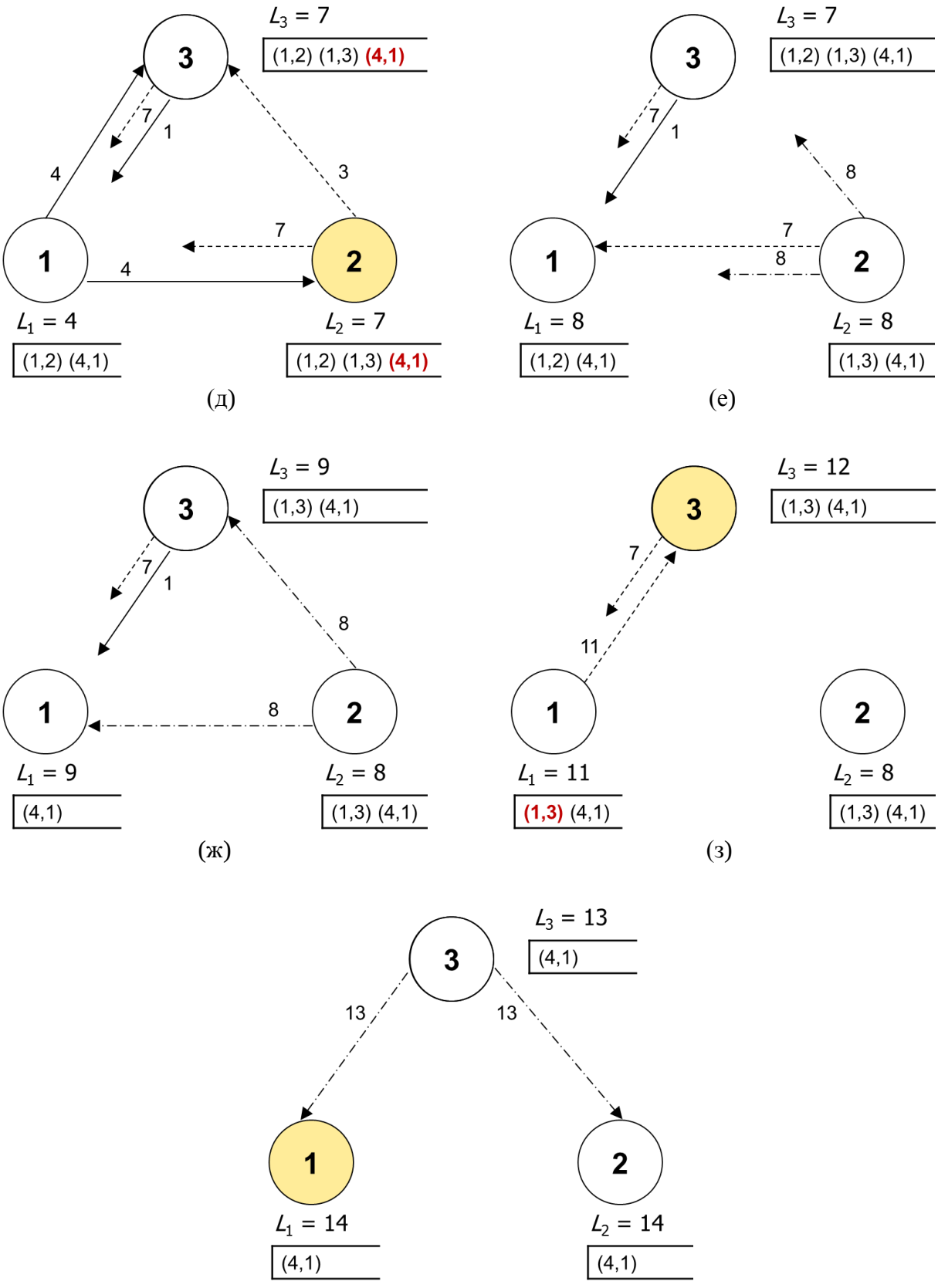


Рис. 4.2. Пример работы алгоритма Лэмпорта.

На рис. 4.2б процессы P_1 и P_3 получают запрос от P_2 и отправляют ему ответные сообщения. Т.к. отметка времени запроса P_2 меньше отметки времени запроса P_3, запрос от P_2 помещается перед запросом от P_3 в его собственной очереди Q_3. Это дает возможность P_2 войти в КС вперед P_3. При этом процесс P_2 сможет рассчитывать на вход к КС только тогда, когда получит ответные сообщения от P_1 и P_3. Отметим, что благодаря свойству FIFO каналов связи, P_2 не сможет получить ответное сообщение от P_3 вперед запроса, отправленного ранее процессом P_3. Поэтому вначале P_2 получит запрос от P_3, поместит его в свою локальную очередь Q_2 и отправит P_3 ответное сообщение. Поскольку отметка времени поступившего от P_3 запроса больше отметки времени запроса P_2, запрос от P_3 помещается в конец очереди Q_2, как показано на рис. 4.2в. Получив ответные сообщения от P_1 и P_3, процесс P_2 войдет в КС, т.к. его запрос находится во главе его собственной очереди Q_2 – см. рис. 4.2г. Далее в нашем сценарии процесс P_1 переходит в состояние запроса на вход в КС и рассылает соответствующее сообщение остальным процессам. При получении запроса от P_1 процессы P_2 и P_3 отправляют ответные сообщения – см. рис. 4.2д. Обратим внимание, что к этому моменту запрос на вход в КС от процесса P_3 был получен и подтвержден процессом P_2. Однако этот же запрос, направляемый процессу P_1, так и не был получен. На рис. 4.2е процесс P_2 выходит из КС, удаляет свой запрос из очереди Q_2 и рассылает всем процессам сообщение RELEASE. Желающий войти в КС процесс P_1 получает от P_2 ответное сообщение и, затем, сообщение RELASE, что приводит к удалению запроса от P_2 из очереди Q_1 – см. рис. 4.2ж. Отметим, что на рис. 4.2ж в состоянии запроса на вход в КС находится и процесс P_1, и процесс P_3. В локальной очереди каждого из этих процессов на первом месте расположен собственный запрос этого процесса. Оба этих процесса получили ответные сообщения от P_2. Тем не менее, получить доступ к КС процесс P_3 сможет раньше, чем P_1, т.к. его запрос произошел раньше запроса P_1. Действительно, P_1 не сможет войти в КС пока не получит ответное сообщение от P_3. Благодаря свойству FIFO канала связи между P_3 и P_1 это сообщение придет в P_1 только после поступления запроса от P_3, что проиллюстрировано на рис. 4.2ж. Когда же P_1 получит запрос от P_3, он поместит его вперед своего собственного запроса в своей очереди Q_1, т.к. запрос от P_3 имеет меньшую отметку времени, чем запрос от P_1 – см. рис. 4.2з. Теперь P_1 не сможет получить доступ к КС, пока P_3 не войдет и не выйдет из КС. В свою очередь, P_3 войдет в КС при получении ответного сообщения от P_1 – см. рис. 4.2з. При выходе из КС P_3 разошлет всем процессам сообщение RELEASE, которое приведет к удалению его запроса из всех очередей, в том числе из очереди процесса P_1. После этого запрос процесса P_1 окажется первым в его собственной очереди Q_1, и он сможет войти в КС, как показано на рис. 4.2и.

Чтобы оценить эффективность работы алгоритма Лэмпорта, отметим, что для обеспечения взаимного исключения необходимо переслать $3(N-1)$ сообщений: для входа в КС требуется $(N-1)$ сообщений REQUEST и $(N-1)$ ответных сообщений REPLY, для выхода из КС требуется $(N-1)$ сообщений RELEASE.

Оптимизация алгоритма Лэмпорта. В алгоритме Лэмпорта в некоторых ситуациях нет необходимости посылать ответные сообщения REPLY. А именно, если процесс P_j получит запрос REQUEST(L_i, i) от процесса P_i после того, как он сам отправил процессу P_i некоторое сообщение с отметкой времени $(L_j, j) > (L_i, i)$ (таким сообщением, например, может быть сообщение REQUEST(L_j, j), если P_j соберется входить в КС практически одновременно с P_i), то P_j нет необходимости отправлять P_i ответное сообщение. Действительно, благодаря свойству FIFO канала связи между P_j и P_i поступление такого сообщения к процессу P_i будет гарантировать, что ему от P_j больше не поступит запросов с отметкой времени меньшей, чем (L_i, i). К примеру, на рис. 4.2ж процесс P_1 получает от процесса P_3 запрос с отметкой времени (1, 3) уже после того, как сам отправил запрос с отметкой времени (4, 1) – см. рис. 4.2д. Поэтому показанное на рис. 4.2з ответное сообщение из процесса P_1 в процесс P_3 с отметкой времени (11, 1) можно было не пересылать.

С учетом представленной оптимизации число сообщений, необходимых для обеспечения взаимного исключения в алгоритме Лэмпорта, будет варьироваться от $2(N-1)$ до $3(N-1)$.

4.3.2 Алгоритм Рикарта-Агравала

Рассмотренный выше алгоритм Лэмпорта явным образом выстраивает запросы на доступ к КС в очередь в порядке возрастания их отметок времени. Еще раз обратим внимание на роль сообщений REPLY и RELEASE в этом алгоритме. Ответные сообщения REPLY используются исключительно для информирования процесса, запрашивающего вход в КС, о том, что в каналах не осталось ни одного другого запроса, который мог бы встать перед его запросом в его локальной очереди. Роль сообщения RELEASE сводится к удалению обслуженного запроса из всех локальных очередей для предоставления другим процессам возможности войти в КС. Отсюда видно, что отметки логического времени, переносимые этими сообщениями, не играют существенной роли в функционировании алгоритма и могут быть опущены. В этом случае при проверке второго условия входа в КС процесс будет опираться не на время получаемых сообщений, а на число поступивших сообщений REPLY: второе условие входа в КС в алгоритме Лэмпорта будет выполнено, когда процесс, запрашивающий доступ к КС, получит все сообщения REPLY от всех остальных процессов.

В работе Г. Рикарта и А. Агравала предпринята попытка оптимизировать алгоритм Лэмпорта, исключив из него сообщения RELEASE за счет объединения функций сообщений RELEASE и REPLY в одном сообщений. Более того, алгоритм Рикарта-Агравала не требует, чтобы каналы связи обладали свойством FIFO. Основная идея этого алгоритма заключается в том, что для входа в КС процессу требуется собрать разрешения *от всех* других процессов распределенной системы. Для этого процесс, желающий получить доступ к КС, рассылает свой запрос REQUEST всем остальным процессам и ожидает от них поступления разрешений в виде сообщений REPLY. Условия, при которых процессы дают свое разрешение на вход в КС, сформулированы таким образом, чтобы удовлетворить требованиям безопасности и живучести. А именно, если все остальные процессы находятся в состоянии выполнения вне КС, они немедленно отправляют процессу, запрашивающему доступ к КС, сообщения REPLY. Если же один из процессов выполняется в КС, то он откладывает отправку своего разрешения до тех пор, пока сам не покинет КС, тем самым не позволяя двум процессам выполняться в КС одновременно. Если же несколько процессов запрашивают разрешение на вход в КС в одно и то же время, то, как и в алгоритме Лэмпорта, для разрешения конфликта доступа к КС используются линейно упорядоченные отметки времени запросов (L_i, i), где L_i – скалярное время процесса P_i на момент формирования запроса, i – идентификатор процесса P_i. Процесс, запрос которого имеет наименьшую отметку времени, получит право войти в КС первым. Это достигается за счет того, что каждый находящийся в состоянии запроса на вход в КС процесс P_i при получении запроса от другого процесса P_j сравнивает отметку времени (L_i, i) своего запроса с отметкой времени (L_j, j) поступившего запроса, и откладывает отправку REPLY, если $(L_i, i) < (L_j, j)$. Если же $(L_i, i) > (L_j, j)$, то процесс P_i отправляет сообщение REPLY процессу P_j немедленно. Поэтому процесс с наименьшей отметкой времени запроса получит ответные сообщения от всех остальных процессов и сможет войти в КС.

Для реализации алгоритма Рикарта-Агравала каждый процесс P_i поддерживает работу с массивом $DR_i[1..N]$, содержащим признаки *отложенного ответа* (от англ. *deferred reply*). Если процесс P_i откладывает отправку ответного сообщения процессу P_j, он устанавливает $DR_i[j] = 1$; после отправки сообщения REPLY элемент $DR_i[j]$ сбрасывается в ноль. Все элементы $DR_i[k]$ инициализируется нулем, $1 \le k \le N$. С учетом этого массива алгоритм Рикарта-Агравала будет определяться следующими правилами.

1. **Запрос на вход в КС.** Когда процессу P_i нужен доступ к КС, он переходит в состояние запроса на вход в КС, фиксируя показания своих скалярных часов L_i, и рассылает сообщение REQUEST(L_i, i) всем остальным процессам. Когда процесс P_j получает запрос

REQUEST(L_i, i) от процесса P_i, он отправляет процессу P_i ответ REPLY в случае, если P_j находится в состоянии выполнения вне КС, или в случае, если P_j находится в состоянии запроса на вход в КС и отметка времени (L_j, j) его запроса больше отметки времени (L_i, i) запроса процесса P_i. В противном случае P_j откладывает отправку ответного сообщения REPLY и устанавливает $DR_j[i] = 1$.

2. **Вход в КС.** Процесс P_i может войти в КС, если он получил ответные сообщения REPLY от всех остальных процессов.

3. **Выход из КС.** При выходе из КС процесс P_i рассылает отложенные сообщения REPLY всем ожидающим процессам, т.е. всем процессам P_j, для которых $DR_i[j] = 1$, и затем устанавливает $DR_i[j] = 0$.

Представленный алгоритм удовлетворяет требованиям безопасности и живучести для решения задачи взаимного исключения.

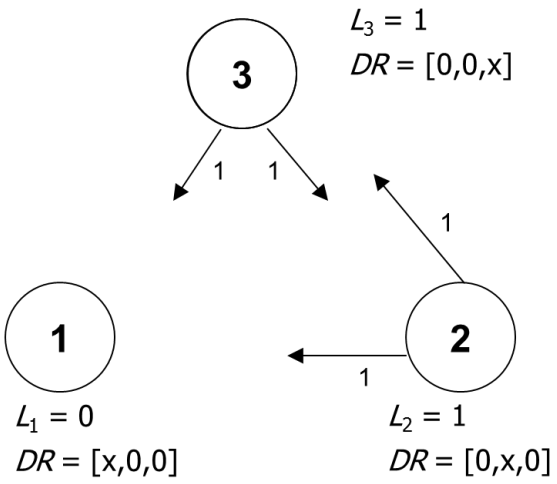
Докажем выполнение свойства безопасности от противного. Пусть два процесса P_i и P_j выполняются в КС одновременно, и отметка времени запроса P_i меньше отметки времени запроса P_j. Процесс P_j может войти в КС, только когда он получит ответные сообщения от всех остальных процессов. При этом сообщение с запросом от P_j достигло P_i *после того*, как P_i отправил свой запрос; в противном случае отметка времени запроса P_i должна была бы быть больше отметки времени запроса P_j по правилам работы логических часов. По условиям алгоритма процесс P_i не может отослать ответное сообщение REPLY процессу P_j, находясь в состоянии запроса на вход в КС и имея меньшее значение отметки времени своего запроса, до тех пор, пока P_i не выйдет из КС. Таким образом мы получаем противоречие с предположением, что P_j получил ответные сообщения REPLY от всех процессов в системе, в том числе, от процесса P_i.

Покажем теперь, что алгоритм Рикарта-Агравала обладает свойством живучести. Процесс P_i не сможет получить доступ к КС, если он отправил свой запрос, но не получил необходимого ответа хотя бы от одного процесса P_j. В этом случае мы будем говорить, что P_i ждет P_j. Эта ситуация возможна, если процесс P_j находится либо в состоянии выполнения внутри КС, либо в состоянии запроса на вход в КС. При этом отметка времени запроса P_j должна быть меньше отметки времени запроса P_i. Если P_j выполняется внутри КС, то через ограниченное время он выйдет из КС и отправит ответное сообщение процессу P_i. Если же P_j находится в состоянии запроса на вход в КС, то либо он войдет в КС, либо должен существовать другой процесс P_k такой, что P_j ждет P_k. Продолжая эти рассуждения, мы построим цепь из процессов, ожидающих друг друга: P_i ждет P_j, P_j ждет P_k, P_k ждет P_m, и т.д. Важно отметить, что эта цепь имеет ограниченную длину и является ациклической: процессы располагаются в ней в порядке убывания отметок времени запросов. Последний процесс P_l

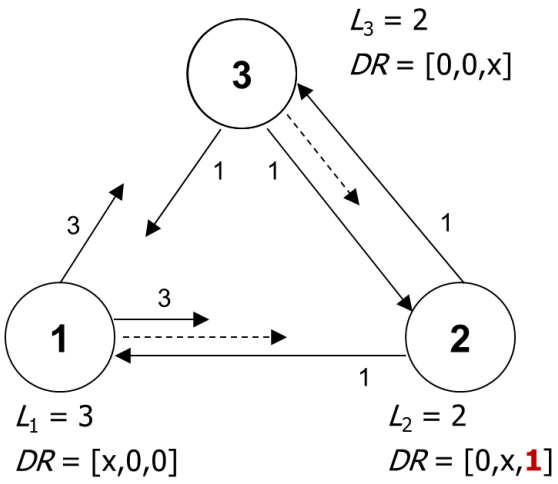
в этой цепи с наименьшим временем запроса либо выполняется в КС, либо получит разрешения от всех остальных процессов и сможет войти в КС. При выходе из КС P_l разошлет ответные сообщения всем ожидающим процессам, в том числе процессу, стоящему непосредственно слева от него, тем самым позволяя этому процессу войти в КС. Поэтому за ограниченное число шагов любой процесс окажется последним в цепи ожидающих процессов и сможет войти в КС.

Из этих рассуждений видно, что в алгоритме Рикарта-Агравала, также как и в алгоритме Лэмпорта, доступ к КС предоставляется строго в соответствии с отметками времени запросов по порядку, сохраняющему частичный причинно-следственный порядок →. Отсюда следует, что *после* получения сообщения REQUEST от процесса P_i всеми процессами ни один из них не сможет войти в КС более одного раза вперед P_i. Обратим также внимание, что в отличие от алгоритма Лэмпорта, в котором запросы на доступ КС явным образом выстраивались в очередь, в алгоритме Рикарта-Агравала процессы, желающие войти в КС, неявно упорядочиваются в ациклическую цепь, в которой каждый процесс ожидает поступления разрешений от процессов, стоящих справа от него.

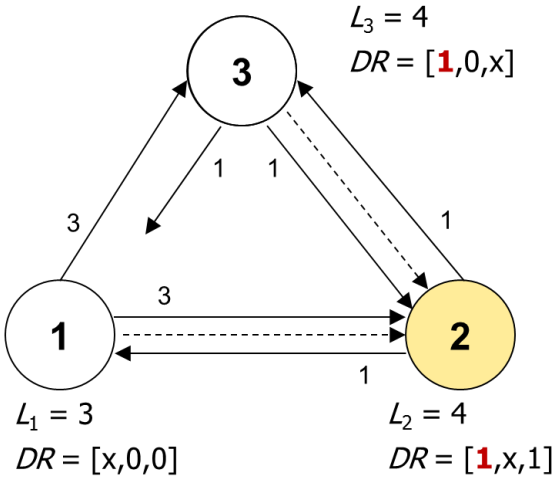
Пример работы алгоритма Рикарта-Агравала приведен на рис. 4.3. Также как и в сценарии, представленном выше для алгоритма Лэмпорта, на рис. 4.3а процессы P_2 и P_3 запрашивают вход в КС приблизительно в одно и то же время. На рис. 4.3б процессы P_1 и P_3 получают запрос от P_2 и отправляют ему ответные сообщения. В свою очередь процесс P_2, получив запрос от P_3, откладывает отправку ответного сообщения, т.к. P_2 находится в состоянии запроса на вход в КС и отметка времени его запроса меньше отметки времени запроса P_3. Соответствующий элемент массива DR_2 устанавливается в единицу. Поэтому процесс P_3 не сможет получить доступ к КС до тех пор, пока P_2 не войдет и не выйдет из нее. Далее в нашем сценарии процесс P_1 переходит в состояние запроса на вход в КС и рассылает соответствующее сообщение остальным процессам, как показано на рис. 4.3б. Между тем, получив необходимые ответы от P_1 и P_3, процесс P_2 входит в КС – см. рис. 4.3в. Затем запрос от процесса P_1 достигает процессов P_2 и P_3. Обратим внимание, что при получении запроса от P_1 процессы P_2 и P_3 не отправляют ответные сообщения, т.к. P_2 находится в состоянии выполнения в КС, а P_3 находится в состоянии запроса на вход в КС и отметка времени его запроса меньше отметки времени запроса P_1. В итоге цепь ожидающих процессов будет выглядеть следующим образом: P_1 ждет P_3, P_3 ждет P_2. Эта ситуация проиллюстрирована на рис. 4.3в. Действительно, процесс P_3 знает о том, что его ждет процесс P_1, т.к. $DR_3[1] = 1$, а процесс P_2 знает о том, что его ждут процессы P_1 и P_3, т.к. $DR_2[1] = 1$ и $DR_2[3] = 1$.


$L_3 = 1$
$DR = [0,0,x]$
3
1
1
1
1
2
1
$L_1 = 0$
$DR = [x,0,0]$
$L_2 = 1$
$DR = [0,x,0]$

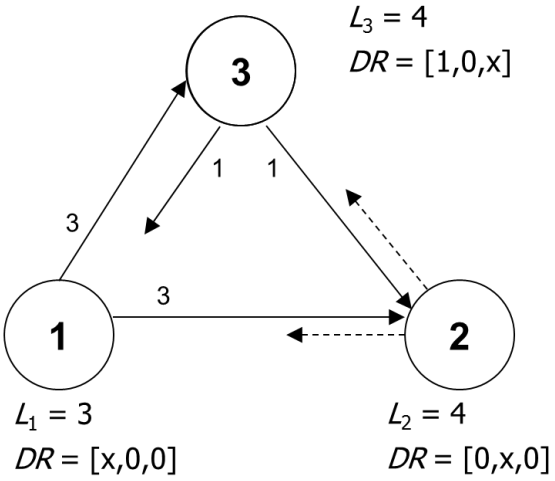
(а)


$L_3 = 2$
$DR = [0,0,x]$
3
1
1
3
1
3
1
2
1
$L_1 = 3$
$DR = [x,0,0]$
$L_2 = 2$
$DR = [0,x,1]$

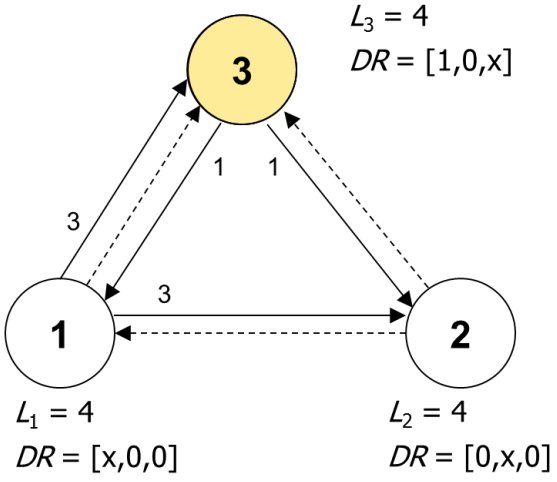
(б)


$L_3 = 4$
$DR = [1,0,x]$
3
1
1
3
1
3
1
2
1
$L_1 = 3$
$DR = [x,0,0]$
$L_2 = 4$
$DR = [1,x,1]$

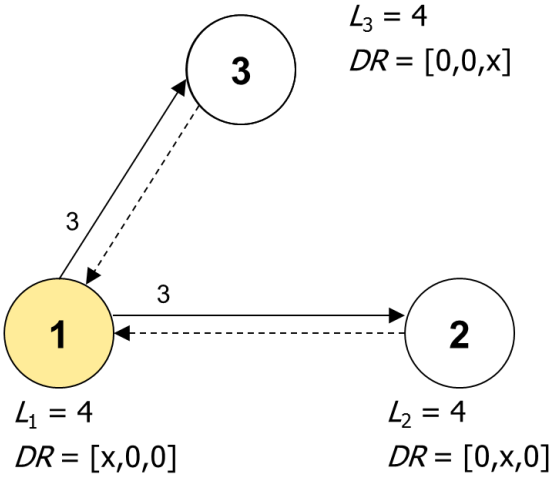
(в)


$L_3 = 4$
$DR = [1,0,x]$
3
1
1
3
3
1
2
$L_1 = 3$
$DR = [x,0,0]$
$L_2 = 4$
$DR = [0,x,0]$

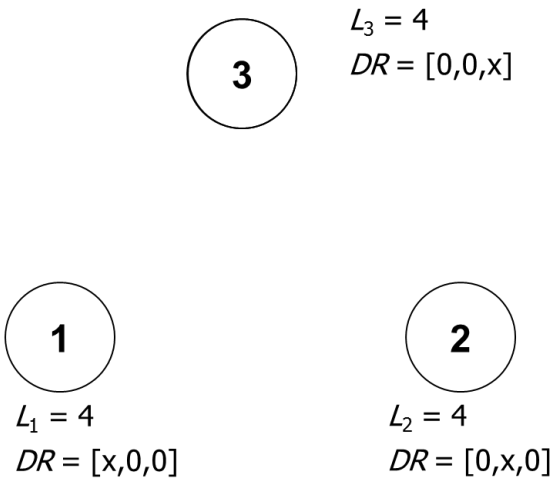
(г)


$L_3 = 4$
$DR = [1,0,x]$
3
1
1
3
3
1
2
$L_1 = 4$
$DR = [x,0,0]$
$L_2 = 4$
$DR = [0,x,0]$

(д)


$L_3 = 4$
$DR = [0,0,x]$
3
3
3
1
2
$L_1 = 4$
$DR = [x,0,0]$
$L_2 = 4$
$DR = [0,x,0]$

(е)



(ж)

Рис. 4.3. Пример работы алгоритма Рикарта-Агравала.

На рис. 4.3г процесс P_2 выходит из КС и рассылает отложенные ответы ожидающим процессам P_1 и P_3. Теперь последним в цепи ожидающих процессов оказывается процесс P_3, и он войдет в КС следующим. На рис. 4.3д запрос процесса P_3 достигает процесса P_1, и, несмотря на то, что сам P_1 находится в состоянии запроса на вход в КС, он высылает P_3 ответ, т.к. отметка времени запроса P_1 больше отметки времени запроса P_3. По получению необходимых ответов P_3 входит в КС. Когда P_3 выйдет из КС, он отправит P_1 отложенный ответ, и P_1 сможет войти в КС, как показано на рис. 4.3е. На рис. 4.3ж процесс P_1 выходит из КС. При этом ему не требуется рассылать никаких сообщений.

Отметим, что в отличие от алгоритма Лэмпорта, алгоритм Рикарта-Агравала использует для работы с КС $2(N-1)$ сообщений: $(N-1)$ сообщений требуется для входа в КС и $(N-1)$ сообщений требуется для выхода из КС.

4.3.3 Алгоритм обедающих философов

В алгоритме Рикарта-Агравала для работы с КС требовался обмен $2(N-1)$ сообщениями. Ниже мы опишем алгоритм, требующий $2(N-1)$ сообщений только в наихудшем случае. Для этого предположим, что распределенная система состоит из большого числа процессов, однако число процессов, активно работающих с КС, невелико. Представленный в данном пункте алгоритм позволяет процессам, незаинтересованным в работе с КС, в конце концов, не участвовать в обмене сообщениями.

В отличие от алгоритма Рикарта-Агравала, в котором процесс запрашивает разрешение у всех других процессов *каждый раз* перед входом в КС, в рассматриваемом алгоритме такие разрешения представлены в виде специальных сущностей (вспомогательных ресурсов), которые процесс собирает для получения доступа к КС. При этом если

другие процессы не забирают эти разрешения обратно между последовательными обращениями процесса к КС, процесс сохраняет полученные разрешения у себя и может не запрашивать их при повторном входе в КС.

Важно также отметить, что в отличие от алгоритмов взаимного исключения, представленных выше, рассматриваемый алгоритм решает более общую задачу – задачу обедающих философов. Постановка этой задачи допускает одновременное выполнение нескольких процессов в своих КС при условии, что между этими процессами не может возникнуть конфликта доступа из-за обращений к любому из разделяемых ресурсов. В задаче обедающих философов такая ситуация моделируется разрешением нескольким философам питаться одновременно, если они не являются соседями.

Общие концепции разрешения конфликтов. Как уже отмечалось, проблема взаимного исключения возникает в тех случаях, когда несколько процессов претендуют на некоторый общий ресурс, который в каждый момент времени может быть использован только одним процессом. Если несколько процессов одновременно обращаются к такому разделяемому ресурсу, то имеет место конфликт доступа, который должен быть разрешен в пользу одного из этих процессов и, соответственно, в ущерб остальным процессам. Когда все конфликтующие процессы неразличимы между собой, т.е. в случае, когда какое-либо свойство одного из этих процессов присутствует также у всех остальных процессов, то разрешить конфликт становится невозможным без обращения к случайному выбору. Дело в том, что любой детерминированный метод, отдающий предпочтение одному процессу из группы для разрешения конфликта в его пользу, должен осуществлять такой выбор на основании какого-либо свойства, присущего именного этому процессу и отсутствующего у других. Таким образом, если не пользоваться методами случайного выбора, для разрешения конфликтов необходимо обеспечить выполнение следующих требований.

1. **Различимость**. В *каждом* состоянии системы для *любого множества* конфликтующий процессов как минимум один процесс должен быть отличим от других процессов в таком множестве. Примером такого свойства может являться уникальный идентификатор процесса.
2. **Справедливость**. Естественным требованием является не только возможность разрешать конфликт в пользу того или иного процесса, но и обеспечение справедливости такого выбора, т.е. гарантии того, что конфликты доступа постоянно не разрешаются в ущерб определенным процессам.

Если конфликты доступа возникают в одном и том же состоянии системы, то детерминированная схема разрешения конфликтов будет давать одни и те же результаты, потому как результат работы детерминированного метода полностью определяется состоянием системы. В этом случае схема разрешения конфликтов не будет справедливой. Таким образом для удовлетворения требования справедливости разрешения конфликтов необходимо, чтобы при возникновении конфликтов состояние системы не было одинаковым. Примером информации, учет которой позволяет гарантировать то, что конфликты возникают при различных состояниях системы, является *время*. При этом под временем можно понимать как реальное физическое время, так и логическое время, рассмотренное ранее. Так, в представленных выше алгоритмах взаимного исключения Лэмпорта и Рикарта-Агравала конфликты доступа к КС разрешались с помощью механизма скалярных часов. Процессы использовали присваиваемые запросам отметки времени для разрешения конфликта доступа в пользу процесса с наименьшим временем возникновения запроса. Если же показания скалярных часов нескольких процессов совпадали, то для определения процесса, победившего в конфликте, использовалось другое отличительное свойство, а именно, уникальный идентификатор процесса. Ниже мы рассмотрим другой подход к разрешению конфликтов доступа, основанный на расположении *вспомогательных ресурсов*. Важно отметить, что вспомогательные ресурсы используются исключительно для целей разрешения конфликтов и не имеют прямого отношения к реальным разделяемым ресурсам, из-за которых может возникнуть конфликт доступа.

Возможность возникновения конфликтов между процессами в распределенной системе можно задать в виде неориентированного графа G, где каждому процессу P_i поставлена в соответствие вершина графа. Ребро, соединяющее две вершины, соответствующие процессам P_i и P_j, существует в G тогда и только тогда, когда между процессами P_i и P_j может возникнуть конфликт доступа: другими словами, когда существует один или несколько разделяемых ресурсов, требующих эксклюзивного доступа, к которым эти процессы могут обратиться одновременно. Такой граф G называют *графом конфликтов* (англ. *conflict graph*).

Для иллюстрации построения и использования графа конфликтов рассмотрим следующий пример.

Представим какой-нибудь законодательный орган, в который входят пять комитетов. Каждый комитет сформирован из пяти человек, таким образом, как показано в Таблице 4.1. Общее собрание этого законодательного органа можно провести в любое время, когда не проходит заседание любого из его комитетов. Пусть каждый из представленных пяти комитетов собирается запланировать еженедельное четырехчасовое заседание для своих членов. При этом было бы

желательно, чтобы такие заседания проводились по возможности одновременно. В противном случае, если все комитеты будут собираться в непересекающиеся периоды времени, общее время занятости всех комитетов составит 20 часов, что ограничивает возможность проведения общего собрания законодательного органа. Из Таблицы 4.1 следует, что не все комитеты могут заседать одновременно, т.к. некоторые сотрудники входят сразу в несколько комитетов и, очевидно, не могут одновременно присутствовать на нескольких заседаниях: в Таблице 4.1. такие сотрудники отмечены серым цветом. Например, сотрудник E одновременно входит в состав комитета 1 и комитета 2.

Таблица 4.1. Пример построения графа конфликтов: пять комитетов, в каждый из которых входит по пять человек.

Комитет 1	**Комитет 2**	**Комитет 3**	**Комитет 4**	**Комитет 5**
A	F	J	A	Q
B	G	K	K	R
C	H	L	N	N
D	I	I	O	D
E	E	M	P	S

Для решения задачи планирования расписания каждого из представленных комитетов можно построить соответствующий граф конфликтов, в котором вершины будут соответствовать комитетам, а ребра – возможным конфликтам между ними. В этом случае возможность возникновения конфликта между двумя комитетами определяется наличием одного или более сотрудников, которые одновременно входят в состав обоих комитетов. Например, между вершинами, соответствующими комитету 1 и комитету 2, будет существовать ребро, т.к. они имеют в своем составе общего для них сотрудника E. В свою очередь комитет 1 и комитет 3 имеют разные и непересекающиеся составы, поэтому в графе конфликтов ребро между соответствующими вершинами будет отсутствовать.

Граф конфликтов для представленной задачи планирования расписания заседаний пяти комитетов приведен на рис. 4.4. По постановке задачи комитеты, соответствующие соседним вершинам в этом графе, не могут проводить заседания одновременно. Поэтому из графа конфликтов сразу видно, что для проведения заседаний потребуется минимум три четырехчасовых интервала, т.е. минимум 12 часов: например,

одновременно могут заседать комитеты 1 и 3, комитеты 2 и 5, и еще один интервал необходимо выделить для заседания комитета 4. Если же комитеты не планируют свою занятость наперед, а действуют независимо и выражают свое желание провести заседание в произвольный момент времени, при этом возможно вызывая "конфликты доступа" к своим сотрудникам, то проведение заседания становится эквивалентным входу процесса в КС, для обеспечения которого необходимо получить доступ к ресурсам – членам комитета.

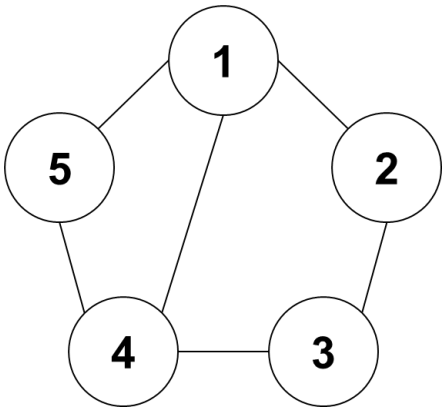


Рис. 4.4. Граф конфликтов для задачи заседаний пяти комитетов.

Для разрешения конфликтов между процессами требуется некоторый механизм, позволяющий при любом состоянии системы задавать приоритет процессов друг над другом таким образом, что при возникновении конфликта он бы разрешался в пользу процесса с б*о*льшим приоритетом. В данном случае приоритет процесса, по сути, определяет порядок предоставления разделяемых ресурсов в пользование тому или иному процессу, т.е. задает отношение предшествования между обращениями процессов к разделяемым ресурсам в случае возникновения конфликтов. Будем представлять такое отношение предшествования для любой пары потенциально конфликтующих процессов с помощью ориентированного графа предшествования H (англ. *precedence graph*), который может быть получен из графа G путем задания каждому ребру направления от процесса с б*о*льшим приоритетом к процессу с меньшим приоритетом. На рис. 4.5 показан граф H для состояния системы, в котором процесс P_1 имеет приоритет над процессами P_2, P_4 и P_5, а процесс P_3 имеет приоритет над P_2 и P_4.

Очевидно, что для обеспечения различимости процессов между собой, граф H должен быть ациклическим. Действительно, если все процессы, составляющие цикл, оказываются в конфликтной ситуации, то они будут неразличимы между собой, т.к. для любых двух процессов P_i и P_j, входящих в цикл, P_i будет иметь приоритет над P_j и, одновременно, P_j будет иметь приоритет над P_i.

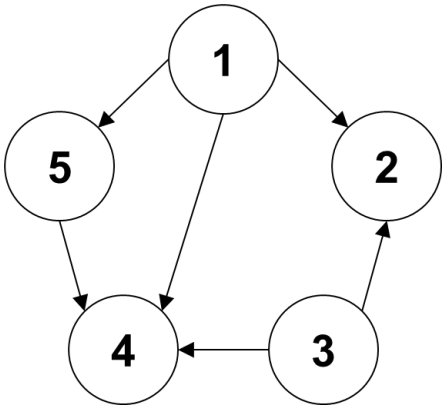


Рис. 4.5. Граф предшествования для задачи заседаний пяти комитетов.

В качестве примера рассмотрим ситуацию, когда граф предшествования H образует цикл между вершинами, соответствующими процессам P_1, P_5 и P_4, как показано на рис. 4.6. На этом рисунке процесс P_1 имеет приоритет над процессом P_5, P_5 имеет приоритет над P_4, и P_4 имеет приоритет над P_1. Если конфликт доступа к разделяемому ресурсу возникает только между двумя процессами из перечисленных выше, он может быть легко разрешен в пользу процесса с б*о*льшим приоритетом. Проблема же возникнет тогда, когда все три процесса одновременно захотят использовать разделяемые ресурсы и войдут в конфликт. Тогда выбрать один процесс из трех детерминированным способом не представится возможным.

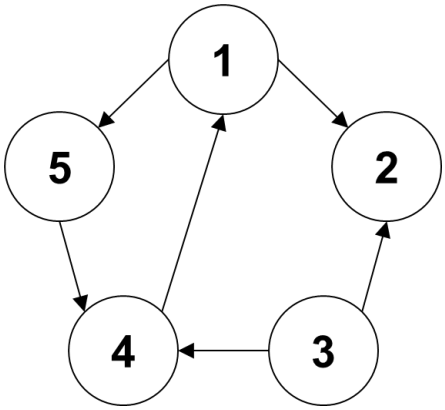


Рис. 4.6. Пример цикла в графе предшествования.

Если граф H не содержит циклов, то вершину, в которую не входит ни одно ребро, будем называть корнем. Тогда в качестве отличительного свойства процессов можно использовать значение глубины соответствующей ему вершины (англ. *depth*), определяемой как наибольшее число ребер по любому направленному пути от корня графа к данной вершине. Глубина корня графа равна нулю. По определению две соседние вершины не могут иметь одну и ту же глубину. Например, на

рис. 4.5 глубина процессов P_1, P_5 и P_4 соответственно равна 0, 1 и 2. Обратим внимание, что, опираясь на понятие глубины вершины, граф H задает отношение частичного порядка на множестве процессов.

Таким образом для обеспечения свойства различимости процессов, алгоритм разрешения конфликтов должен гарантировать ацикличность графа H при *любом* состояния системы. Однако отсутствие циклов в графе H в любом состоянии системы само по себе не обеспечивает свойство справедливости в разрешении конфликтов. Необходимо, чтобы даже при постоянном возникновении конфликтных ситуаций каждый процесс за конечное время получал бы возможность разрешать все свои конфликты в свою пользу. Это требование может быть выполнено при условии, что каждый процесс, находящийся в конфликтной ситуации, будет "подниматься вверх" по графу H до тех пор, пока не станет первым в порядке предшествования среди всех процессов, с которыми он конфликтует. Фраза "процесс будет подниматься вверх по графу H" подразумевает под собой тот факт, что состояние системы будет меняться, и вместе с ним будет меняться граф H так, что в некотором будущем состоянии рассматриваемый процесс сможет разрешить все свои конфликты в свою пользу. В крайнем случае, процессу может понадобиться "подняться в самый верх" графа H до его корня. Если процесс станет корнем графа H, т.е. будет иметь нулевую глубину, то любой его конфликт с любым другим процессом разрешится в его пользу, т.к. корневой процесс имеет приоритет над всеми своими соседями.

Каким же образом мы можем изменять граф H? Единственный способ заключается в изменении ориентации его ребер. Кроме того, для построения именно распределенного алгоритма работы с графом H необходимо реализовать соответствующее представление этого графа в распределенной системе и каждое его изменение производить локально одним из процессов этой системы.

Таким образом, обобщая все вышесказанное, можно заключить, что распределенный алгоритм работы с графом предшествования H должен удовлетворять следующим требованиям.

1. При любом изменении граф H должен оставаться ациклическим.
2. Граф H должен изменяться таким образом, чтобы каждый процесс, участвующий в разрешении конфликтов, в конце концов, оказался бы вверху графа, т.е. стал первым среди конфликтующих процессов в порядке предшествования, и смог бы разрешить все свои конфликты в свою пользу.
3. Каждое изменение графа H должно производиться локально одним из процессов распределенной системы.

Для реализации распределенного представления графа H и алгоритма работы с ним рассмотрим обобщенную задачу обедающих философов, которая моделирует проблему разрешения конфликтов в распределенных системах. Решение этой задачи определяет представление графа предшествования H в распределенной системе с помощью *вспомогательных ресурсов* и демонстрирует алгоритм работы с этим графом.

Решение обобщенной задачи обедающих философов. В общей формулировке задача обедающих философов предполагает наличие конечного неориентированного графа G, вершины которого представляют собой процессы распределенной системы. В терминах данной задачи процессы называют "философами" и, обычно, обозначают символами u, v, w. Два философа являются соседями тогда и только тогда, когда между ними существует ребро в графе G. Каждый из философов может находиться в одном из трех состояний: состоянии размышления, голодном состоянии и состоянии поглощения пищи. При этом возможны только переходы из состояния размышления в голодное состояние, из голодного состояния в состояние приема пищи и из состояния приема пищи вновь в состояние размышления. Важно отметить, что философ принимает пищу в течение ограниченного времени, в то время как в состоянии размышления он может находиться сколь угодно долго. Суть задачи состоит в разработке детерминированной схемы координации действий философов таким образом, чтобы ни один из философов не принимал бы пищу одновременно с любым из своих соседей. Когда два соседних философа одновременно оказываются в голодном состоянии, между ними возникает конфликт, который должен быть разрешен в пользу одного из них, т.к. только один философ сможет приступить к приему пищи. Поэтому граф G представляет собой граф конфликтов для обедающих философов.

Кроме выполнения основного требования, запрещающего соседним философам питаться одновременно, распределенный алгоритм решения задачи обедающих философов должен обладать следующими свойствами.

1. **Отсутствие голодания.** Ни один из философов не должен вечно голодать, т.е. философ должен чередовать прием пищи и размышления.
2. **Симметричность.** Все философы подчиняются одинаковым правилам и выполняют одинаковые роли при координации своих действий. Не существует каких-либо заданных извне неизменных приоритетов, установленных для философов.
3. **Экономия.** Между переходами из состояния в состояние философ должен отправлять и получать ограниченное число сообщений. В частности, постоянно размышляющий философ не должен неограниченно отсылать или получать сообщения.

4. **Параллелизм.** Алгоритм координации действий философов не должен исключать возможность одновременного приема пищи для нескольких философов, при условии, что они не являются соседями.

5. **Ограниченность.** Количество сообщений, находящихся в каждый момент времени в процессе пересылки между любыми двумя философами, должно быть ограничено.

Такая постановка задачи в качестве частного случая включает в себя классическую формулировку проблемы обедающих философов, когда пять философов сидят вокруг круглого стола, и, соответственно, каждый из них имеет по два соседа. В этом случае граф G будет представлять собой кольцо с пятью вершинами. Кроме того, рассмотренная в предыдущих пунктах традиционная задача взаимного исключения для N процессов также может быть описана в терминах обобщенной задачи обедающих философов. А именно, в этом случае в каждый момент времени принимать пищу будет разрешено максимум одному философу. Поэтому, для задачи взаимного исключения граф G будет представлять собой полный граф из N вершин, а состояние приема пищи будет соответствовать выполнению процесса внутри КС.

Для разрешения конфликтов между собой философам необходим механизм определения приоритетов друг относительно друга, очевидно, удовлетворяющий рассмотренным ранее требованиям различимости и справедливости, чтобы не допустить вечного голодания философов. А именно, за конечное время голодный философ либо должен получить возможность поесть, обнаружив всех своих соседей в состоянии размышления, либо в случае возникновения конфликтов с соседями, рано или поздно, должен получить возможность разрешать все конфликты в свою пользу. Этот механизм может быть реализован в виде графа предшествования H, изменение которого осуществляется по следующим правилам.

- Направление ребра в графе H изменяется только тогда, когда философ, находящийся в вершине, инцидентной данному ребру, переходит из голодного состояния в состояние приема пищи. То есть философ понижает приоритет по отношению к своему соседу только с началом употребления пищи.
- При переходе философа в состояние приема пищи *все* ребра, инцидентные его вершине, *одновременно* изменяют свое направление в сторону этой вершины. Другими словами, кушающий философ понижает свой приоритет по отношению ко всем своим соседям.

Перечисленные правила основываются на следующем свойстве ориентированных графов. В результате переориентации всех ребер по

направлению к какой-либо вершине u не может быть создано никаких новых путей из вершины w в вершину v, для любых w и $v \neq u$, т.к. через u не может пройти ни один путь (u не будет являться промежуточной вершиной на любом пути). Поэтому такие правила преобразования графа H не позволяют голодному философу, находящемуся в вершине v, опускаться вниз в порядке предшествования. Кроме того, очевидно, что эти правила сохраняют свойство ацикличности графа H, т.к. при переориентации всех ребер по направлению к вершине u не может быть получено ни одного цикла, содержащего u.

Покажем теперь, что при таких правилах изменения H гарантируется, что голодный философ, в конце концов, поест, т.к. поднимется вверх в порядке предшествования и сможет разрешить все конфликты в свою пользу. Пусть философ в вершине v с глубиной k, $k > 0$, голоден и находится выше всех своих голодных соседей в порядке предшествования. Тогда либо v начнет прием пищи, либо какой-нибудь из его размышляющих соседей, стоящих выше v в порядке предшествования, перейдет в голодное состояние. Голодный философ, являющийся корнем графа H, имеет приоритет над всеми своими соседями, поэтому сможет принимать пищу вне зависимости от того, голодны ли его соседи или нет. Кроме того, согласно правилам изменения H, любой голодный философ, начав прием пищи, опускается в самый низ в порядке предшествования, тем самым продвигая других голодных философов вверх. Таким образом, по индукции, голодный философ с глубиной k, в конце концов, приступит к приему пищи, потому что он стоит выше всех своих голодных соседей с глубиной, большей чем k, а все философы с глубиной, меньшей чем k, если они станут голодными, рано или поздно опустятся в порядке предшествования ниже рассматриваемого философа.

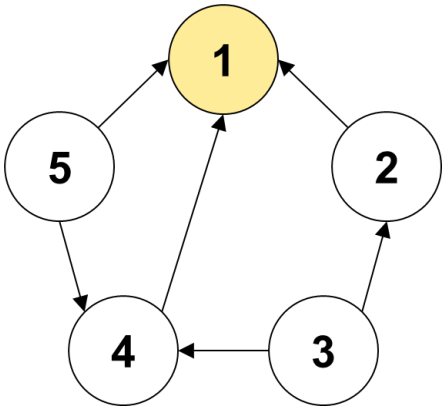


Рис. 4.7. Пример изменения графа предшествования.

На рис. 4.7 показан пример изменения графа предшествования H, представленного ранее на рис. 4.5. На рис. 4.7 бывший корень P_1 начинает принимать пищу, что в терминах задачи заседаний пяти комитетов

соответствует заседанию комитета 1, и понижает свой приоритет по отношению к своим соседям P_2, P_4 и P_5. Вершина P_5 становится корнем.

Приведенные выше правила работы с графом H обеспечивают выполнение требований различимости философов и справедливости разрешения конфликтов. Однако сами по себе они не дают представления о способе реализации графа H в распределенной системе и не определяют, можно ли голодному философу приступить к приему пищи, не нарушая при этом требования задачи, согласно которому соседи не должны питаться одновременно. Действительно, голодный философ может перейти в состояние приема пищи, когда он находится выше всех своих голодных соседей в порядке предшествования, и при этом *ни один из его соседей не ест*. Если же хотя бы один из его соседей принимает пищу, голодному философу необходимо ожидать до тех пор, пока кушающий сосед не закончит питаться. Как философам определить, питается ли его сосед или нет?

Для ответа на этот вопрос в решение задачи вводятся *вспомогательные ресурсы*, *вилки* (англ. *fork*), таким образом, что с каждым ребром графа G ассоциируется ровно одна вилка. Вилка, ассоциируемая с ребром, соединяющим философов u и v, находится в их совместном доступе, но в каждый момент времени может принадлежать только одному из них. Для того чтобы поесть, потребуем от философа собрать все вилки по всем ребрам графа, инцидентным вершине, в которой он находится. Из этого требования будет следовать, что любые два соседних философа не смогут одновременно находиться в состоянии приема пищи, т.к. они не смогут одновременно владеть вилкой, находящейся в их совместном доступе.

Очевидно, что с введением вспомогательных ресурсов, правило перехода из голодного состояния в состояние приема пищи и правило передачи вилок от одного философа к другому можно сформулировать следующим образом.

1. Голодный философ u может начать принимать пищу только тогда, когда u владеет всеми вилками от всех своих соседей, и для любого его соседа v верно, что u имеет приоритет над v или v находится в состоянии размышления.
2. Не принимающий пищу философ u должен отдавать вилку своему голодному соседу v только в случаях, когда v имеет приоритет над u, или когда u находится в состоянии размышления, т.е. не заинтересован в обладании этой вилкой.

Для реализации этих правил философам необходимо определять взаимный приоритет друг относительно друга с помощью графа предшествования H. Отметим, что особенность представления графа H в распределенной системе как раз и состоит в том, что философ должен

определять наличие или отсутствие приоритета над своим соседом исходя только из своего локального состояния, и приоритет между парой соседних философов должен согласованно изменяться обоими философами.

Задать граф H в распределенной системе можно путем введения для каждой вилки атрибута "грязная" / "чистая" и определения относительного приоритета таким образом, что философ u имеет приоритет над философом v, т.е. ребро в графе H ориентировано из вершины u в вершину v, тогда и только тогда, когда 1) философ u владеет общей с v вилкой, и она "чистая", или 2) философ v владеет общей с u вилкой, и она "грязная", или 3) вилка находится в процессе передачи от философа v к философу u.

Соответственно представленные выше правила изменения графа H будут иметь следующее выражение.

- Философ, принимающий пищу, владеет всеми общими с его соседями вилками, и все эти вилки "грязные". Это утверждение соответствует правилу, согласно которому кушающий философ получает более низкий приоритет по отношению ко всем своим соседям.
- Философ, владеющий "чистой" вилкой, продолжает ее удерживать до тех пор, пока он не перейдет в состояние приема пищи. Все это время вилка остается "чистой". Это утверждение соответствует правилу, согласно которому приоритет философа по отношению к своему соседу понижается только с началом употребления пищи.
- Если вилка использовалась для приема пищи, она становится "грязной" и остается "грязной" до тех пор, пока она не будет передана соседнему философу. Вилка становится "чистой" только при передаче соседнему философу. Важно отметить, что передача "очищенных" вилок не изменяет ориентацию ребер графа H, поэтому это утверждение также соответствует правилу, согласно которому приоритет философа по отношению к своему соседу понижается только с началом употребления пищи.
- "Чистыми" вилками владеют только голодные философы.

Последнее утверждение следует из следующих соображений. Предположим, что "чистой" вилкой может владеть размышляющий философ. Философ имеет право передать вилку своему соседу только, если вилка "грязная". Поэтому размышляющий философ будет удерживать вилку у себя в течение всего периода размышления. Однако по условиям задачи философ может размышлять бесконечно долго и, как следствие, может удерживать вилку в течение произвольного времени, тем самым приводя своих соседей к бесконечному голоданию.

Таким образом, в случае представления графа H с помощью атрибутов вспомогательных ресурсов, а именно, вилок, правило перехода

из голодного состояния в состояние приема пищи и правило передачи вилок от одного философа к другому принимают вид:

1. Голодный философ *u* может начать принимать пищу только тогда, когда *u* владеет всеми вилками от всех своих соседей, и для любого его соседа *v* верно, что общая с *v* вилка – "чистая", или *v* находится в состоянии размышления.

2. Не принимающий пищу философ *u* должен отдавать вилку своему голодному соседу *v* только в случае, когда общая с *v* вилка – "грязная".

Согласно второму правилу философ *u* передает "грязную" вилку своему соседу *v* только, если *v* голоден. Отсюда, кстати, следует, что закончивший прием пищи философ, не будет возвращать вилки своим соседям, если они находятся в состоянии размышления и, следовательно, не заинтересованы в них. Соответственно, если философ, обладающий всеми вилками, повторно проголодается раньше любого из своих соседей, он сможет поесть без необходимости вновь собирать все вилки. Но как философу определить, голоден его сосед или нет?

Для ответа на этот вопрос потребуем от голодного философа, не обладающего какой-либо вилкой, явным образом запрашивать ее у своего соседа. Для каждой пары соседних философов будем использовать маркер запроса (англ. *request-token*), соответствующий общей для этих философов вилке. В каждый момент времени маркер запроса может находиться только у одного философа из пары или находиться в состоянии пересылки от одного философа к другому. Обладатель маркера имеет право запрашивать вилку у своего соседа, пересылая ему маркер. Тогда, если у философа *u* находится одновременно и общая с философом *v* вилка, и маркер запроса, соответствующий этой вилке, то *u* может заключить, что *v* голоден.

Таким образом, решение задачи обедающих философов определяется следующими правилами.

1. **Передача маркера запроса.** Философ *u* отправляет соседнему философу *v* соответствующий маркер запроса, если 1) *u* владеет маркером, 2) у философа *u* отсутствует общая с *v* вилка и 3) *u* голоден.

2. **Передача вилки.** Философ *u* пересылает философу *v* общую с ним вилку, если 1) у *u* вместе с этой вилкой находится соответствующий ей маркер запроса, 2) вилка – "грязная" и 3) *u* не принимает пищу. При передаче вилки она "очищается".

3. **Переход из голодного состояния в состояние приема пищи.** Голодный философ *u* может начать принимать пищу только тогда, когда *u* владеет всеми вилками от всех своих соседей, и для любого его соседа *v* верно, что общая с ним вилка – "чистая", либо у *u* не

находится маркера запроса от *v*. Когда философ *u* переходит в состояние приема пищи, все находящиеся у него вилки становятся "грязными".

Начальное состояние распределенной системы. Представленное выше решение задачи обедающих философов опирается на следующие инварианты:

- граф *H* остается ациклическим;
- принимающий пищу философ владеет всеми вилками со всех инцидентных ему ребер;
- "чистыми" вилками владеют только голодные философы;
- философ владеет вилкой и соответствующим ей маркером запроса только тогда, когда его сосед голоден.

Поэтому начальное состояние распределенной системы также должно обеспечивать выполнение перечисленных утверждений. А именно, изначально вилки должны быть распределены среди философов таким образом, чтобы в *H* не было циклов. Кроме того, если предположить, что в начальном состоянии все философы находятся в размышлениях, то все вилки должны быть "грязными", и ни у одного из философов одновременно с вилкой не должен находиться соответствующий ей маркер запроса.

Теперь продемонстрируем, что представленное решение задачи обедающих философов обладает требуемыми свойствами.

1. **Отсутствие голодания.**

Для начала покажем, что, если философ *u* имеет приоритет над всеми своими голодными соседями, то для каждого соседа *v* с б*о*льшим, чем у *u*, приоритетом верно, что философ *u* владеет общей с ним вилкой, но не владеет соответствующим ей маркером запроса. Действительно, предположим, что вилка находится у философа *v*. Тогда она должная быть "чистой", т.к. *v* имеет приоритет над *u*. Но чистые вилки могут находиться только у голодных философов, а по условию, *v* не голоден. Поэтому, если *u* стоит выше всех своих голодных соседей в порядке предшествования, то у него содержатся все вилки всех его соседей с б*о*льшим приоритетом. Кроме того, если вместе с вилкой *u* также содержал бы маркер запроса от своего соседа *v*, это означало бы, что *v* голоден. Поэтому, у *u* отсутствуют маркеры запросов, соответствующие вилкам от соседей с б*о*льшим приоритетом.

Теперь покажем, что голодный философ *u*, в конце концов, получит "очищенные" вилки от своих соседей, обладающих меньшими приоритетами. Если *u* еще не владеет такой вилкой, он должен переслать своему соседу *v*, имеющему меньший приоритет, соответствующий маркер

запроса. Если v кушает, то по постановке задачи, рано или поздно, он должен прекратить принимать пищу. Если v голоден, то он не имеет права перейти в состояние приема пищи, т.к. содержит "грязную" вилку и соответствующий ей маркер запроса (см. правило перехода из голодного состояния в состояние приема пищи). Поэтому, в конце концов, v должен переслать "очищенную" вилку философу u (см. правило передачи вилки).

Из аргументов, приведенных в этих двух абзацах, следует, что если философ u голоден и одновременно продолжает оставаться выше всех своих голодных соседей в порядке предшествования, то, рано или поздно, условие перехода из голодного состояния в состояние приема пищи будет выполнено, и он сможет поесть. Поэтому по индукции, опираясь на рассмотренные выше правила изменения графа H, можно заключить, что любой голодный философ, в конце концов, приступит к приему пищи.

2. **Симметричность.**

Из описания алгоритма следует, что все философы подчиняются одинаковым правилам.

3. **Экономия.**

Между переходами из состояния в состояние каждый философ отправляет и получает ограниченное число сообщений. А именно, если число соседей у философа равно d, то тогда этот философ отправит не более d и получит не более d сообщений, содержащих маркеры запроса или вилки. Действительно, предположим, что когда философ переходит в голодное состояние, у него содержится e "грязных" вилок. Тогда для получения недостающих вилок он должен отправить $(d - e)$ запросов своим соседям. Кроме того, в худшем случае, до перехода в состояние приема пищи он может потерять все свои "грязные" вилки, и ему потребуется запросить их вновь. Т.к. философ находится в голодном состоянии при пересылке своей "грязной" вилки в этом же сообщении он может переслать и соответствующий ей маркер запроса. Поэтому для перехода из голодного состояния в состояние приема пищи потребуется не более чем $2d$ сообщений. В состоянии приема пищи и в состоянии размышления каждый философ может лишь получать сообщения с маркером запроса и отправлять вилки своим соседям, поэтому число получаемых и отправляемых сообщений также не превосходит $2d$. В лучшем случае, когда все соседи философа постоянно находятся в состоянии размышления, он не будет получать от них запросов на вилки и может чередовать прием пищи и размышления вовсе без взаимодействия с ними.

4. **Параллелизм.**

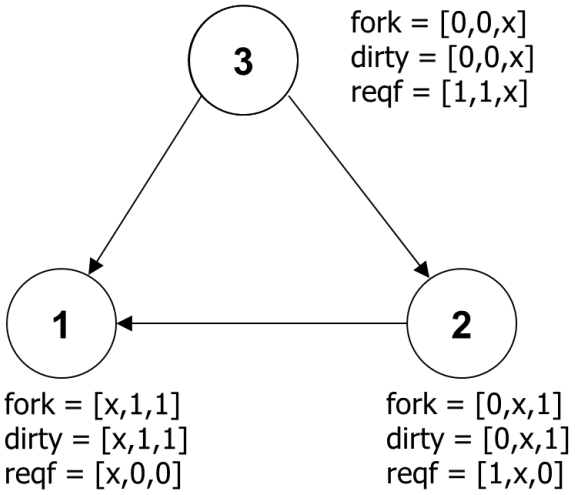
Из правила перехода в состояние приема пищи следует, что представленный алгоритм координации действий философов не исключает возможности одновременного приема пищи для нескольких философов при условии, что они не являются соседями.

5. **Ограниченность.**

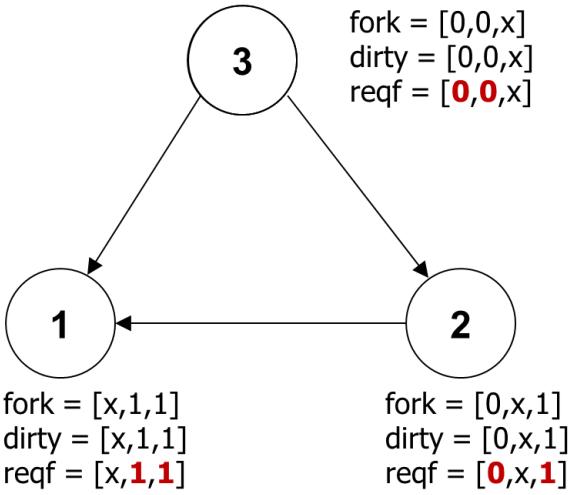
Число сообщений, находящихся в каждый момент времени в состоянии пересылки между любыми двумя философами, не превосходит двух: сообщение с маркером запроса и сообщение, содержащее передаваемую вилку.

В заключение еще раз отметим, что для традиционной задачи взаимного исключения граф конфликтов представляет собой полный граф, т.к. по постановке этой задачи при попытке доступа к КС процесс потенциально входит в конфликт с любым другим процессом, который может претендовать на доступ к этой же КС. В наихудшем случае для входа в КС процессу придется собрать (N – 1) вилок: по одной от каждого другого процесса в распределенной системе. Поэтому при использовании алгоритма обедающих философов для входа в КС потребуется обмен максимум 2(N – 1) сообщениями. Если же, начиная с некоторого момента, процесс перестанет запрашивать вход в КС, он рано или поздно отдаст все свои вилки другим процессам и не будет участвовать в обмене сообщениями. Таким образом, число сообщений, требуемых для работы с КС, в среднем будет пропорционально только числу процессов, активно работающих с этой КС.

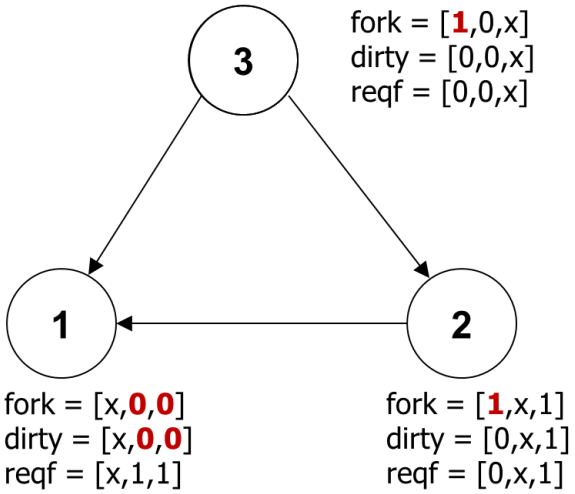
Пример работы алгоритма обедающих философов для традиционной задачи взаимного исключения приведен на рис. 4.8. Каждый процесс (философ) P_i поддерживает работу с массивами $fork_i[1..N]$, $dirty_i[1..N]$ и $reqf_i[1..N]$ такими, что $fork_i[j] = 1$, когда P_i владеет общей с P_j вилкой, $dirty_i[j] = 1$, когда эта вилка "грязная", и $reqf_i[j] = 1$, когда у P_i находится маркер запроса, соответствующий общей с P_j вилке. В противном случае элементы перечисленных массивов содержат ноль. Направление ребер в графе предшествования H обозначено стрелками. На рис. 4.8а изображено начальное состояние системы: вилки распределены так, что граф H не содержит циклов, и все процессы находятся в состоянии выполнения вне КС (в размышлениях), т.е. все вилки "грязные", и ни у одного из процессов одновременно с вилкой не находится соответствующий ей маркер запроса. На рис. 4.8б процессы P_2 и P_3 переходят в состояние запроса на вход в КС (голодное состояние) и запрашивают недостающие вилки у своих соседей. Когда процесс P_i запрашивает вилку у процесса P_j, элемент массива $reqf_i[j]$ сбрасывается в ноль. При получении процессом P_j запроса от P_i элемент массива $reqf_j[i]$ устанавливается в единицу. По правилу передачи вилки процесс P_1 вынужден расстаться со своими вилками и передать их процессам P_2 и P_3, т.к. у P_1 находятся "грязные" вилки вместе с поступившими на них запросами, и, кроме того, P_1 не выполняется в КС (не принимает пищу). При передаче вилок они "очищаются" – см. рис. 4.8в.


3
fork = [0,0,x]
dirty = [0,0,x]
reqf = [1,1,x]
1
2
fork = [x,1,1]
dirty = [x,1,1]
reqf = [x,0,0]
fork = [0,x,1]
dirty = [0,x,1]
reqf = [1,x,0]

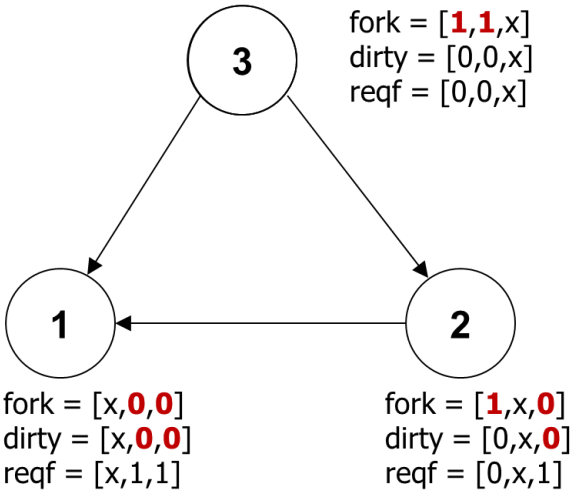
(а)


3
fork = [0,0,x]
dirty = [0,0,x]
reqf = [0,0,x]
1
2
fork = [x,1,1]
dirty = [x,1,1]
reqf = [x,1,1]
fork = [0,x,1]
dirty = [0,x,1]
reqf = [0,x,1]

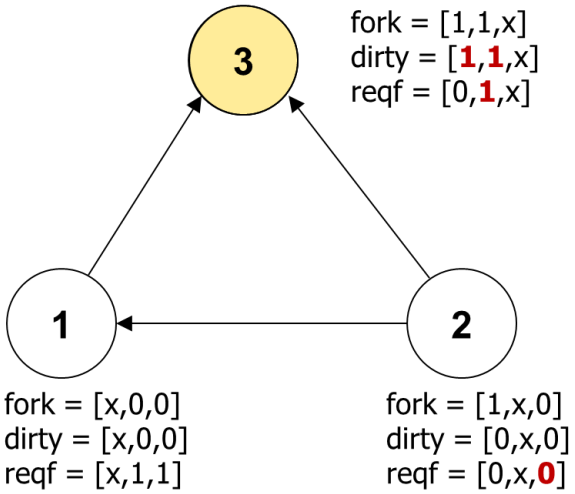
(б)


3
fork = [1,0,x]
dirty = [0,0,x]
reqf = [0,0,x]
1
2
fork = [x,0,0]
dirty = [x,0,0]
reqf = [x,1,1]
fork = [1,x,1]
dirty = [0,x,1]
reqf = [0,x,1]

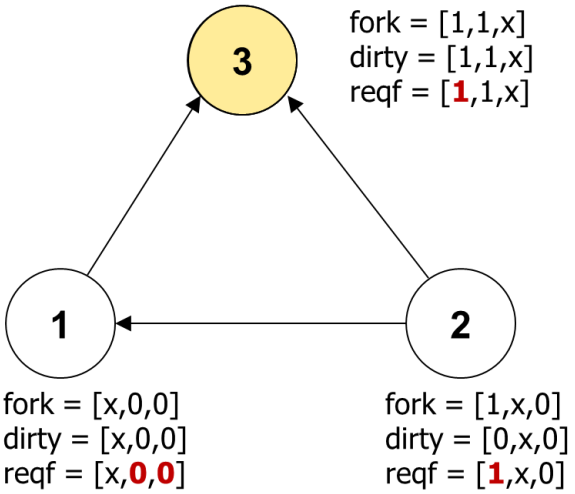
(в)


3
fork = [1,1,x]
dirty = [0,0,x]
reqf = [0,0,x]
1
2
fork = [x,0,0]
dirty = [x,0,0]
reqf = [x,1,1]
fork = [1,x,0]
dirty = [0,x,0]
reqf = [0,x,1]

(г)


3
fork = [1,1,x]
dirty = [1,1,x]
reqf = [0,1,x]
1
2
fork = [x,0,0]
dirty = [x,0,0]
reqf = [x,1,1]
fork = [1,x,0]
dirty = [0,x,0]
reqf = [0,x,0]

(д)


3
fork = [1,1,x]
dirty = [1,1,x]
reqf = [1,1,x]
1
2
fork = [x,0,0]
dirty = [x,0,0]
reqf = [x,0,0]
fork = [1,x,0]
dirty = [0,x,0]
reqf = [1,x,0]

(е)

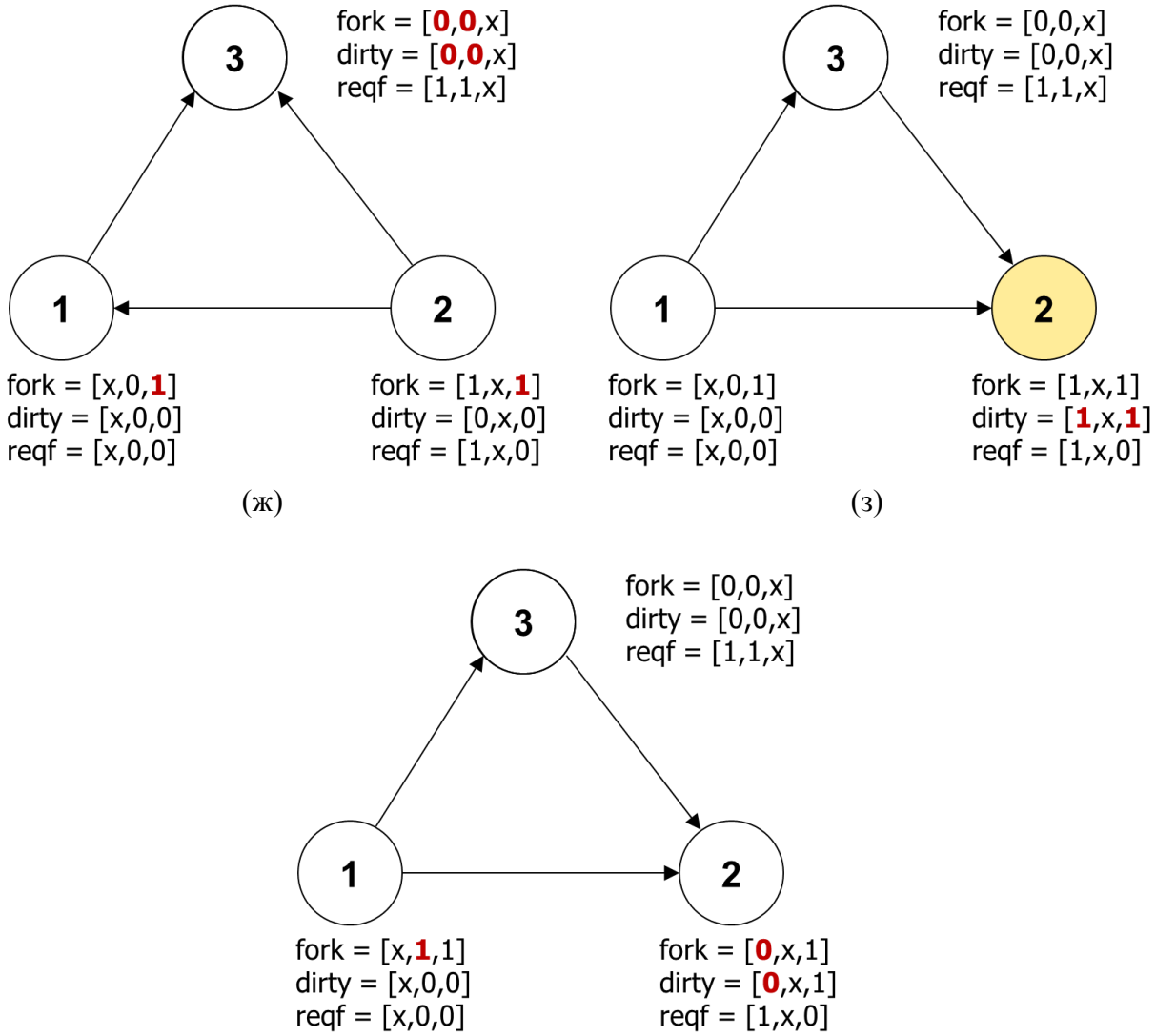


Рис. 4.8. Пример работы алгоритма обедающих философов.

Обратим внимание, что на рис. 4.8в процесс P_2 содержит все необходимые ему вилки. При этом общая с P_1 вилка – "чистая", а общая с P_3 вилка – "грязная". Однако процесс P_2 не может войти в КС (начать прием пищи), т.к. вместе с "грязной" вилкой у него содержится запрос на нее от процесса P_3. Поэтому P_2 отправляет эту вилку процессу P_3, не забывая перед этим ее "почистить", как показано на рис. 4.8г. В итоге процесс P_3 получит "чистые" вилки от P_1 и P_2 и сможет войти в КС. С началом работы в КС все вилки процесса P_3 становятся "грязными": его приоритет по отношению к P_1 и P_2 понижается. Т.к. процесс P_2 находится в состоянии запроса на вход в КС, он отправляет запрос на только что отданную вилку процессу P_3. В свою очередь процесс P_3 не возвращает "грязную" вилку процессу P_2, т.к. в момент получения запроса процесс P_3 выполняется внутри КС. Эта ситуация проиллюстрирована на рис. 4.8д. На рис. 4.8е процесс P_1 переходит в состояние запроса на вход в КС и запрашивает вилки у своих соседей. Отметим, что процесс P_3 не отдает процессу P_1 "грязную" вилку,

т.к. выполняется в КС, а процесс P_2 не отдает процессу P_1 вилку, т.к. она – "чистая". Поэтому процессу P_1 придется ожидать до тех пор, пока P_3 не завершит свое выполнение в КС, и пока P_2 не войдет и не выйдет из КС. После завершения выполнения в КС процесс P_3 "очищает" все свои вилки и отдает их запрашивающим соседям – см. рис. 4.8ж. Обладая чистыми вилками, процесс P_2 сможет войти в КС, как показано на рис. 4.8з. С началом работы в КС P_2 понизит свой приоритет по отношению к P_1 и P_3. После завершения выполнения внутри КС, имея маркер запроса от P_1, процесс P_2 отдаст P_1 общую с ним вилку, после чего P_1 также получит возможность войти в КС, как показано на рис. 4.8и.

4.4. Алгоритмы на основе передачи маркера

Для алгоритмов данного класса право войти в КС материализуется в виде уникального объекта – маркера, который в каждый момент времени может содержаться только у одного процесса или же находиться в канале в состоянии пересылки от одного процесса к другому. Свойство безопасности алгоритмов взаимного исключения в этом случае будет гарантировано ввиду уникальности маркера. При этом процесс, владеющий маркером, может неоднократно входить в КС до тех пор, пока маркер не будет передан другому процессу. Очевидно, что в течение всего времени выполнения внутри КС процесс должен удерживать маркер у себя.

Главные различия алгоритмов, основывающихся на передаче маркера, заключаются в методах поиска и получения маркера, причем эти методы должны гарантировать, что маркер рано или поздно окажется в каждом процессе, желающем войти в КС. Самым простым решением, обеспечивающим такие гарантии, является организация непрерывного перемещения маркера среди *всех* процессов распределенной системы (лат. *perpetuum mobile*). Чтобы не пропустить ни одного процесса, желающего войти в КС, все процессы распределенной системы обычно организуют в направленное логическое кольцо, по которому и циркулирует маркер. Алгоритмы с таким механизмом перемещения маркера называют алгоритмами маркерного кольца (англ. *token ring*). Любой процесс, желающий войти в КС, ожидает прихода маркера, и после выхода из КС передает его дальше по кольцу. Если процесс, получивший маркер, не заинтересован в работе с КС, он просто передает маркер своему соседу. Поэтому даже если ни один из процессов не работает с КС, маркер продолжает непрерывно циркулировать между процессами. Поскольку маркер перемещается от процесса к процессу в общеизвестном порядке, ситуации голодания возникнуть не может. Когда процесс решает войти в КС, в худшем случае ему придется ожидать, пока все остальные процессы последовательно войдут в КС и выйдут из нее.

Далее мы подробно рассмотрим два других алгоритма, в которых перемещение маркера осуществляется лишь в ответ на получение запроса на владение маркером от заинтересованного в нем процесса (англ. *token asking methods*). Отметим, что в этом случае необходимо создать такие условия распространения запросов, при которых каждый запрос рано или поздно достигнет процесса, в котором находится маркер, вне зависимости от перемещений самого маркера.

4.4.1 Широковещательный алгоритм Сузуки-Касами

Одним из решений, позволяющих каждому запросу на владение маркером гарантированно достичь процесса, в котором находится маркер, является широковещательная рассылка таких запросов всем процессам распределенной системы. Данный алгоритм был предложен И. Сузуки и Т. Касами, и суть его заключается в следующем. Если процесс, не обладающий маркером, собирается войти в КС, он рассылает всем другим процессам сообщение REQUEST с запросом на владение маркера. При получении сообщения REQUEST процесс, владеющий маркером, пересылает его запрашивающему процессу. Если в момент получения сообщения REQUEST процесс, владеющий маркером, выполняется внутри КС, он откладывает передачу маркера до тех пор, пока не выйдет из нее. Отметим, что для работы алгоритма Сузуки-Касами не требуется, чтобы каналы связи обладали свойством FIFO.

Несмотря на кажущуюся простоту описанной схемы взаимодействия, представленный алгоритм передачи маркера должен эффективно справляться с решением двух связанных между собой задач.

1. Необходимо различать устаревшие запросы на получение маркера от текущих, еще необслуженных запросов. Действительно, запрос на владение маркером получают все процессы, однако, удовлетворить этот запрос может только один процесс, владеющий маркером. В результате, после того как запрос будет обслужен, у остальных процессов будут находиться устаревшие запросы, в ответ на которые не нужно передавать маркер. Более того, из-за различных и меняющихся задержек передачи сообщений, процесс может получить запрос уже после того, как этот запрос был удовлетворен, т.е. после того, как процесс, рассылавший этот запрос, уже получал маркер в свое владение. В случае, если процесс не имеет возможности определить, был ли находящийся у него запрос уже обслужен или нет, он может передать маркер процессу, который на самом деле в нем не нуждается. Это не приведет к нарушению корректности работы алгоритма взаимного исключения, но может серьезно сказаться на производительности системы.

2. Необходимо вести перечень процессов, ожидающих получения маркера. После завершения процессом своего выполнения внутри КС, он должен определить список процессов, находящихся в состоянии запроса на вход в КС, для того, чтобы передать маркер одному из них.

Для решения этих задач в алгоритме Сузуки-Касами каждый процесс использует порядковые номера (англ. *sequence numbers*) запросов на вход в КС. Порядковый номер n (n = 1, 2, …) увеличивается процессом P_i *независимо* от других процессов каждый раз, когда P_i формирует новый запрос на вход в КС. Порядковый номер запроса P_i передается в рассылаемом сообщении REQUEST в виде REQUEST(i, n). Кроме того, каждый процесс P_i поддерживает работу с массивом $RN_i[1..N]$ (от англ. *request number*), где в элементе $RN_i[j]$ содержится наибольшее значение порядкового номера запроса, полученного от процесса P_j. Другими словами, при получении сообщения REQUEST(j, n) процесс P_i изменяет значение j-го элемента своего массива RN_i согласно выражению $RN_i[j] = \max(RN_i[j], n)$.

Сам маркер переносит внутри себя очередь Q идентификаторов процессов, находящихся в состоянии запроса на вход в КС, и массив $LN[1..N]$ (от англ. *last number*), в элементе $LN[i]$ которого содержится порядковый номер КС, в которой в последний раз выполнялся процесс P_i. Чтобы поддерживать элементы массива $LN[1..N]$ в актуальном состоянии, при выходе из КС процесс P_i обновляет элемент $LN[i] = RN_i[i]$, тем самым указывая, что его запрос с порядковым номером $RN_i[i]$ был удовлетворен. Если все процессы будут следовать этому правилу, то находящийся у P_i запрос процесса P_j с порядковым номером $RN_i[j]$ будет устаревшим, если $RN_i[j] \le LN[j]$. Если же $RN_i[j] = LN[j] + 1$, то это значит, что процесс P_j ожидает получения маркера, и запрос P_j с порядковым номером $RN_i[j]$ еще не был обслужен. В этом случае процесс P_i, владея маркером, помещает идентификатор процесса P_j в конец очереди Q при условии, что идентификатор P_j еще не присутствует в Q. После этого P_i передает маркер процессу, идентификатор которого является первым в очереди Q.

Таким образом, алгоритм Сузуки-Касами будет определяться следующими правилами.

1. **Запрос на вход в КС.**

   - Если желающий войти в КС процесс P_i не владеет маркером, он увеличивает на единицу порядковый номер своих запросов $RN_i[i]$ и рассылает всем другим процессам сообщение REQUEST(i, n), где n – обновленное значение $RN_i[i]$.
   - Когда процесс P_j получает запрос REQUEST(i, n) от процесса P_i, он изменяет значение элемента массива $RN_j[i] = \max(RN_j[i], n)$. Если при этом P_j владеет маркером и находится

в состоянии выполнения вне КС, то P_j передает маркер процессу P_i при условии, что $RN_j[i] = LN[i] + 1$.

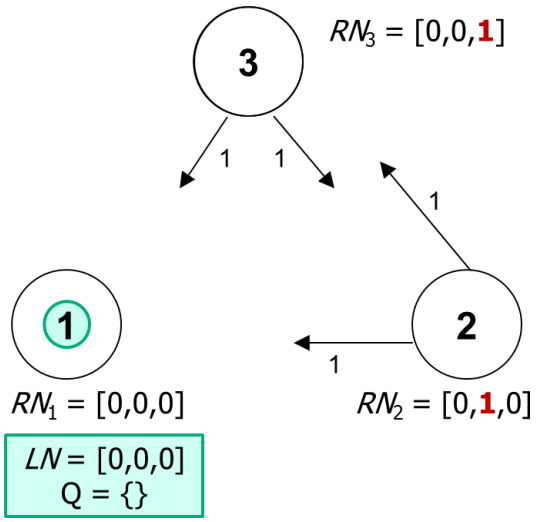
2. **Вход в КС.** Процесс P_i может войти в КС, если он обладает маркером.

3. **Выход из КС.**

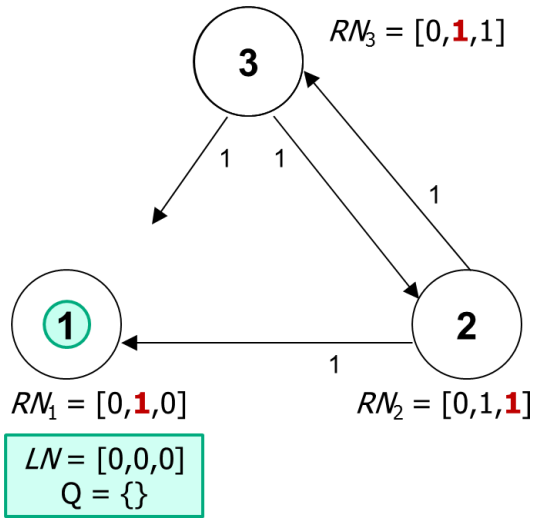
   - При выходе из КС процесс P_i обновляет в маркере значение элемента массива $LN[i]$: $LN[i] = RN_i[i]$.
   - Для каждого процесса P_j, идентификатор которого не присутствует в очереди Q, если $RN_i[j] = LN[j] + 1$, то процесс P_i добавляет идентификатор P_j в конец очереди Q.
   - Если после выполнения представленных выше операций с очередью Q она оказывается не пустой, процесс P_i выбирает первый идентификатор из Q (при этом удаляя его из очереди) и передает маркер процессу с этим идентификатором.

Покажем, что алгоритм Сузуки-Касами обладает свойством живучести. Действительно, запрос процесса P_i на вход в КС за конечное время достигнет других процессов в системе. Один из этих процессов обладает маркером (или, в конце концов, получит его, если маркер находится в состоянии пересылки между процессами). Поэтому запрос процесса P_i рано или поздно будет помещен в очередь Q. Перед ним в очереди может оказаться не более $(N - 1)$ запросов от других процессов, и, следовательно, в конце концов, запрос от P_i будет обслужен, и P_i получит маркер.

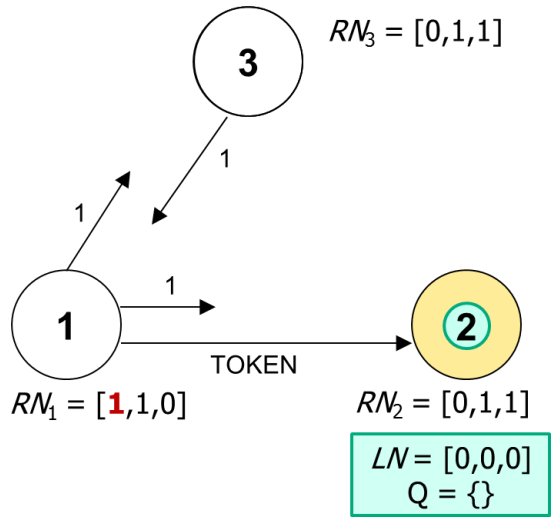
Пример работы алгоритма Сузуки-Касами приведен на рис. 4.9. В начальном состоянии системы маркером владеет процесс P_1, и все процессы находятся в состоянии выполнения вне КС. Также как и в сценарии, представленном в примерах выше, процессы P_2 и P_3 запрашивают вход в КС приблизительно в одно и то же время, как показано на рис. 4.9а. На рис. 4.9б в процесс P_1 первым поступает запрос от процесса P_2, поэтому маркер будет передан именно P_2. Обратим внимание, что если бы в P_1 первым поступил запрос от процесса P_3, то маркер был бы отправлен P_3. Такая зависимость порядка входа в КС от меняющихся задержек передачи сообщений отличает алгоритм Сузуки-Касами от алгоритмов Лэмпорта и Рикарта-Агравала, предоставляющих доступ к КС строго в соответствии с отметками времени запросов и независимо от задержек передачи сообщений. На рис. 4.9в процесс P_2 получает маркер от P_1 в виде сообщения TOKEN и входит в КС. Далее в нашем сценарии процесс P_1 переходит в состояние запроса на вход в КС и рассылает соответствующее сообщение остальным процессам.


3
RN3 = [0,0,1]
1
1
1
1
2
1
RN1 = [0,0,0]
RN2 = [0,1,0]
LN = [0,0,0]
Q = {}

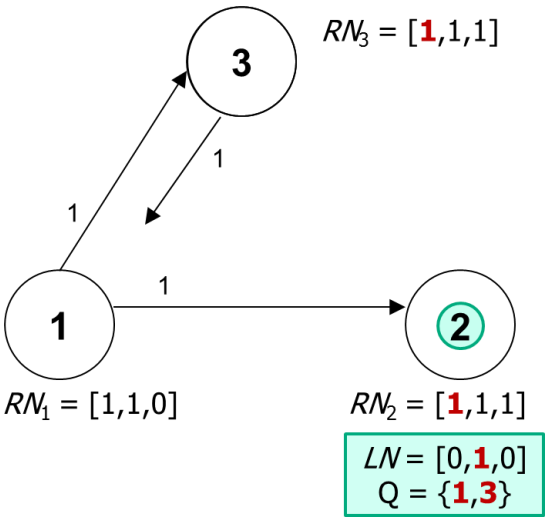
(а)


3
RN3 = [0,1,1]
1
1
1
1
2
1
RN1 = [0,1,0]
RN2 = [0,1,1]
LN = [0,0,0]
Q = {}

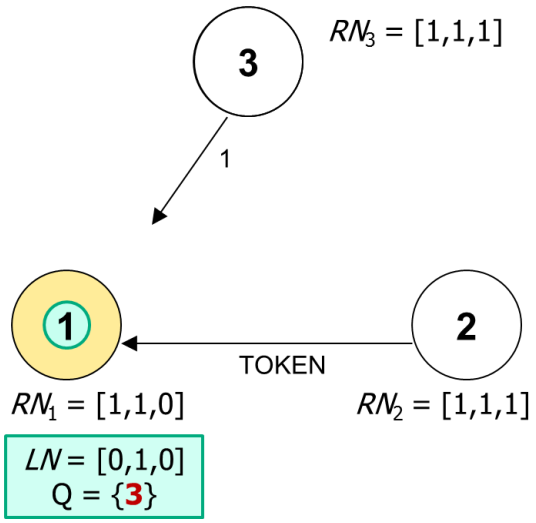
(б)


3
RN3 = [0,1,1]
1
1
1
1
TOKEN
2
RN1 = [1,1,0]
RN2 = [0,1,1]
LN = [0,0,0]
Q = {}

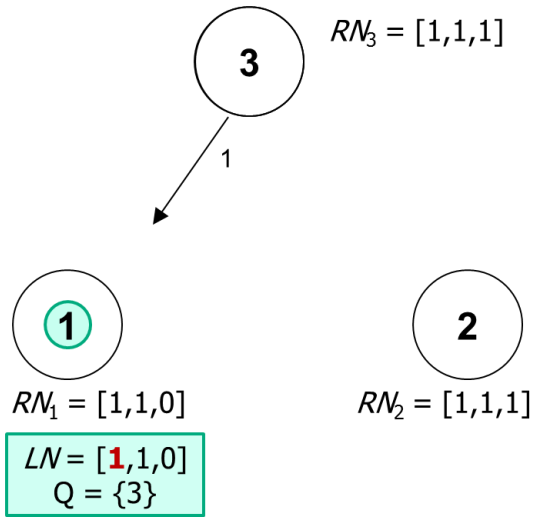
(в)


3
RN3 = [1,1,1]
1
1
1
1
2
RN1 = [1,1,0]
RN2 = [1,1,1]
LN = [0,1,0]
Q = {1,3}

(г)


3
RN3 = [1,1,1]
1
1
TOKEN
2
RN1 = [1,1,0]
RN2 = [1,1,1]
LN = [0,1,0]
Q = {3}

(д)


3
RN3 = [1,1,1]
1
1
2
RN1 = [1,1,0]
RN2 = [1,1,1]
LN = [1,1,0]
Q = {3}

(е)

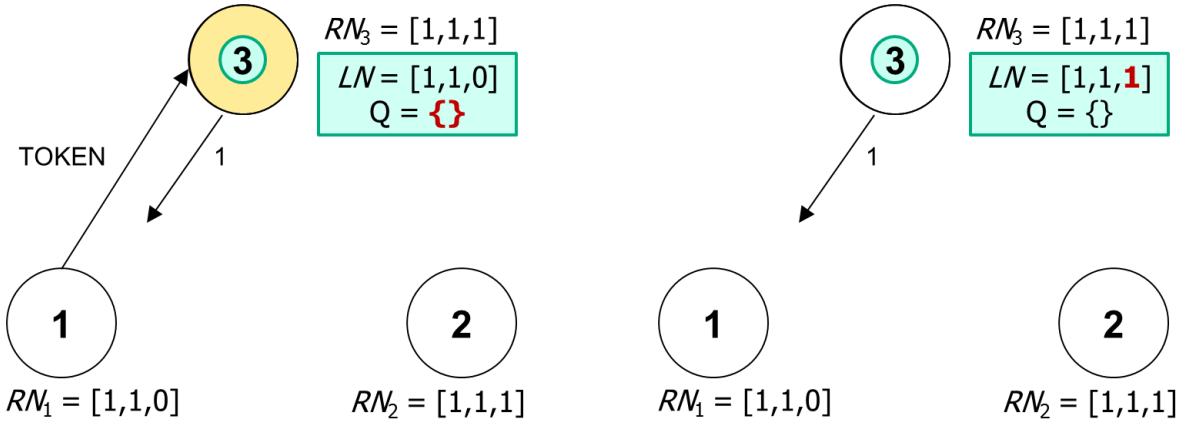


Рис. 4.9. Пример работы алгоритма Сузуки-Касами.

Запрос процесса P_1 достигает процессов P_2 и P_3, как показано на рис. 4.9г. После этого процесс P_2 выходит из КС и обновляет значение $LN[2]$ и содержимое очереди Q. А именно, значение элемента $LN[2]$ меняется на единицу, тем самым, указывая, что процесс P_2 уже выполнялся в своей КС с порядковым номером один, а в очередь Q добавляются идентификаторы процессов P_1 и P_3, ожидающих получения маркера. Важно отметить, что порядок добавления идентификаторов процессов в очередь Q и, как следствие, порядок обслуживания запросов на вход в КС, определяется порядком просмотра массива запросов RN_i. В нашем примере процесс P_2 при выходе из КС просматривает RN_2 с начала, поэтому идентификатор процесса P_1 будет помещен в очередь Q вперед идентификатора P_3 даже несмотря на то, что P_1 запросил доступ к КС после P_3. Если бы при выходе из КС процесс P_i просматривал массив RN_i начиная с позиции $i + 1$ и до N, а затем с позиции 1 до $i - 1$, то для нашего примера процесс P_2 поместил бы идентификатор P_3 перед идентификатором P_1. На рис. 4.9д процесс P_2 передает маркер процессу, находящемуся во главе очереди Q, т.е. процессу P_1, и процесс P_1 входит в КС. Интересно отметить, что на рис. 4.9д процесс P_1 знает, что процесс P_3 ожидает маркер, хотя P_1 так и не получал запроса от P_3: сообщение с запросом на владение маркером от P_3 еще не поступило в P_1. Эта информация доступна P_1 из состояния очереди Q, которую он получил вместе с маркером, и в которую эти сведения были добавлены ранее процессом P_2. Поэтому при выходе из КС P_1 передаст маркер процессу P_3, т.к. в очереди Q содержится только идентификатор P_3. На рис. 4.9е и 4.9ж процесс P_3 наконец получит маркер и войдет в КС. При выходе из КС P_3 обновит значение $LN[3]$ и сохранит маркер у себя, т.к. в системе нет необслуженных запросов, как показано на рис. 4.9з. В свою очередь запрос процесса P_3 с порядковым номером $n = 1$, который рано или поздно будет получен процессом P_1, будет рассматриваться как устаревший, т.к. $RN_1[3] = 1$ и $LN[3] = 1$.

В заключение отметим, что если процесс не владеет маркером, то для входа в КС алгоритм Сузуки-Касами требует обмен N сообщениями: (N – 1) сообщений REQUEST плюс одно сообщение для передачи маркера. Если же процесс владеет маркером, то для входа в КС ему не потребуется ни одного сообщения.

4.4.2 Алгоритм Реймонда на основе покрывающего дерева

Для уменьшения числа сообщений, обмен которыми требуется для работы с КС, в алгоритме Реймонда предполагается, что все процессы распределенной системы логически организованы в дерево таким образом, что сообщения пересылаются только вдоль его ненаправленных ребер, каждое из которых соответствует паре разнонаправленных каналов связи между процессами. Такое дерево может представлять собой минимальное покрывающее дерево графа, описывающего физическую топологию сетевых соединений, или может быть построено для сети с полносвязной логической топологией, используемой в алгоритме Сузуки-Касами. Отметим, что между любыми двумя вершинами в дереве, например, между вершинами, соответствующими процессу, запрашивающему маркер, и процессу, владеющему маркером, существует ровно один путь, вдоль которого эти процессы обмениваются сообщениями REQUEST и TOKEN.

Благодаря такой организации процессов каждый процесс имеет представление только о своих непосредственных соседях и взаимодействует только с ними, что отличает данный алгоритм от широковещательного алгоритма Сузуки-Касами. Например, на рис. 4.10 процесс P_1 поддерживает связь только с процессами P_2, P_3 и P_4 и может даже не знать о существовании процессов P_5 и P_6.

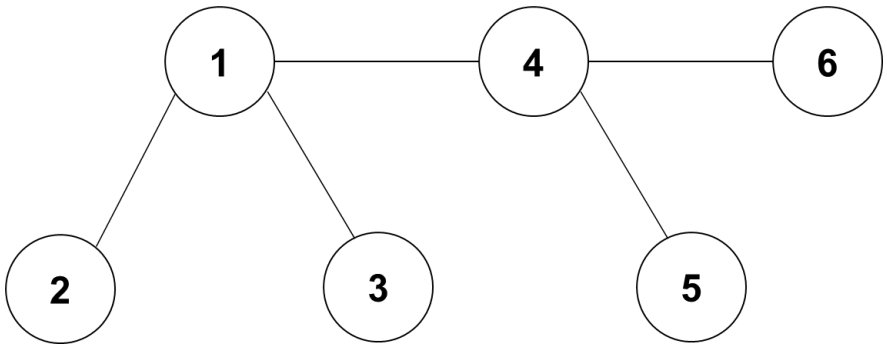


Рис. 4.10. Пример дерева процессов.

Для того чтобы каждый запрос на владение маркером смог достичь процесса, в котором находится маркер, каждый процесс в дереве поддерживает работу с переменной *Holder*, указывающей на расположение маркера относительно этого процесса. А именно, не владеющий маркером процесс P_i в своей переменной $Holder_i$ хранит идентификатор своего

соседнего процесса P_j, который, по его мнению, обладает маркером или через который проходит путь в процесс, владеющий маркером. Поэтому относительно P_i такой соседний процесс P_j будет являться корнем поддерева, в одной из вершин которого содержится маркер. Если же P_i сам владеет маркером, то переменная $Holder_i$ устанавливается в значение *self*.

Ситуация, когда относительно P_i маркер находится в поддереве, корнем которого является P_j, т.е. когда $Holder_i = j$, графически задается путем введения направления для ребра между вершинами P_i и P_j в сторону от P_i к P_j. Таким образом, объединяя информацию, хранящуюся в переменных *Holder* различных процессов, можно построить единственный направленный путь от любого процесса, не владеющего маркером, к процессу с маркером, что проиллюстрировано на рис. 4.11 для случая, когда маркер содержится в процессе P_6.

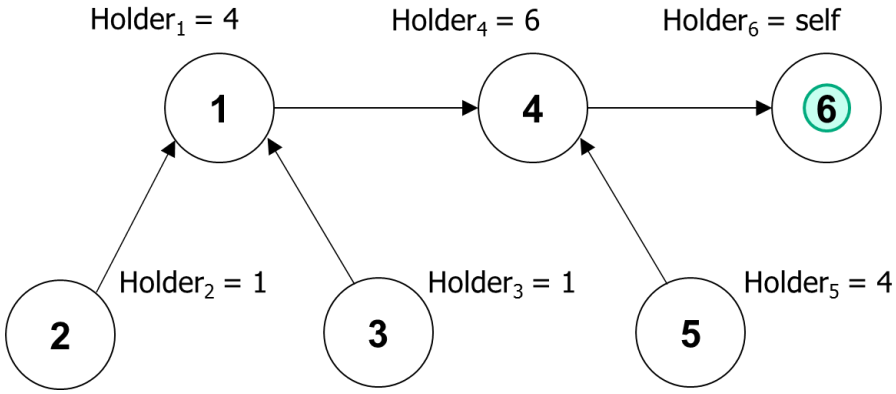


Рис. 4.11. Пример дерева процессов с ребрами, направленными в сторону процесса с маркером.

Если не владеющий маркером процесс P_i собирается войти в КС, он отправляет сообщение REQUEST с запросом на получение маркера соседнему процессу P_j, идентификатор которого содержится в переменной $Holder_i$. В свою очередь, если P_j не владеет маркером, при получении сообщения REQUEST от P_i он пересылает этот запрос дальше в сторону процесса, владеющего маркером, т.е. процессу на который указывает значение его переменной $Holder_j$. Таким образом, сообщения REQUEST последовательно проходят вдоль направленного пути от процесса, запрашивающего маркер, к процессу, обладающему маркером. К примеру, если процесс P_1 на рис. 4.11 захочет войти в КС, он отправит запрос процессу P_4, т.к. $Holder_1 = 4$. В свою очередь благодаря тому, что $Holder_4 = 6$, процесс P_4 переправит запрос процессу P_6, у которого и находится маркер.

Процесс, владеющий маркером и находящийся в состоянии выполнения вне КС, должен передать маркер одному из своих соседей, от которых он получал сообщение REQUEST. Для рассматриваемого примера,

процесс P_6 передаст маркер процессу P_4, и изменит значение своей переменной $Holder_6 = 4$. Процесс P_4 не запрашивал маркер для себя, а отправлял сообщение REQUEST "по просьбе" процесса P_1, поэтому P_4 должен переслать маркер процессу P_1, устанавливая при этом $Holder_4 = 1$. При получении маркера процесс P_1 установит $Holder_1$ в значение *self* и получит возможность войти в КС, как показано на рис. 4.12. Обратим внимание, что при пересылке маркера переменные *Holder* изменяют свои значения таким образом, что совместно эти значения *всегда* определяют направленный путь от любого процесса в системе к процессу, владеющему маркером.

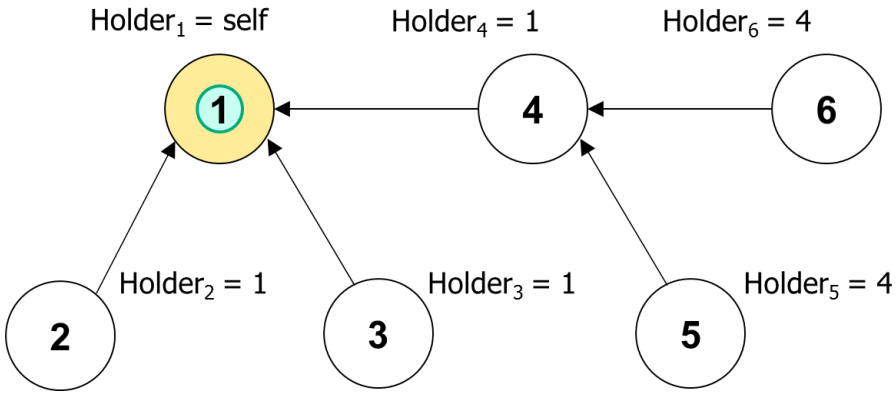


Рис. 4.12. Пример перемещения маркера.

Отметим, что описанный в общих чертах алгоритм, строго говоря, не является "полностью распределенным" в отличии, например, от алгоритма Рикарта-Агравала, в котором все процессы в равной степени принимают участие в решении о предоставления доступа к КС. Дело в том, что согласно такому определению любой "полностью распределенный" алгоритм всегда будет требовать обмен $O(N)$ сообщениями для входа в КС. В представленном же алгоритме процесс P_i просит соседний процесс P_j запрашивать маркер от имени P_i. Это позволяет процессу P_j выступать от лица всех своих соседей, тем самым уменьшая число сообщений, требуемых для работы с КС.

Для реализации алгоритма Реймонда каждый процесс P_i поддерживает работу со следующими переменными и структурами данных.

1. Переменная *Holder*. Возможные значения: *self* или идентификатор одного из непосредственных соседей процесса P_i. Указывает на расположение маркера по отношению к данному процессу.
2. Переменная *Using*. Возможные значения: *true* или *false*. Служит признаком того, выполняется ли процесс P_i в КС или нет. Очевидно, что если *Using* = *true*, то *Holder* = *self*.

3. Очередь *Q*. Возможные значения: *self* или идентификатор одного из непосредственных соседей процесса P_i. Очередь *Q* содержит идентификаторы процессов, от которых P_i получал сообщение REQUEST с запросом на обладание маркером, и которые еще не получали маркер в ответ. Значение *self* помещается в очередь *Q*, когда P_i сам собирается войти в КС, т.е. нуждается в маркере. В связи с тем, что каждый идентификатор помещается в очередь *Q* не более одного раза, максимальный размер очереди *Q* равен числу соседей процесса P_i плюс один для значения *self*.

4. Переменная *Asked*. Возможные значения: *true* или *false*. Переменная *Asked* содержит значение *true*, если не обладающий маркером процесс P_i уже отправлял сообщение REQUEST процессу, на который указывает $Holder_i$; в противном случае *Asked = false*. Переменная *Asked* предотвращает отправку излишних сообщений REQUEST, а также гарантирует отсутствие дубликатов в очереди *Q*.

Для работы с КС каждый процесс должен реализовать функцию `AssignPrivilege()`, с помощью которой осуществляется передача маркера другому процессу или использование маркера для входа в КС, и функцию `MakeRequest()`, с помощью которой отправляется запрос на получение маркера. Пример кода для этих функций приведен в Листинге 4.2 и Листинге 4.3.

Листинг 4.2. Функция работы с маркером.

```
void AssignPrivilege() {
/* Функция работы с маркером */

     if (Holder == self && !Using && !Q.empty()) {

       /* Выборка первого элемента из очереди Q */
       Holder = Q.front();
       Q.pop();
       Asked = false;

       /* Если первый элемент self, процесс входит в КС */
       /* иначе передает маркер процессу с ID в Holder */
       if (Holder == self) {
          Using = true;
          /* вход в КС */
       }
       else {
          /* отправка TOKEN процессу с ID в Holder */
       }
     }
}
```

Листинг 4.3. Функция отправки запроса на получение маркера.

```
void MakeRequest() {
/* Функция отправки запроса на получение маркера */

     if (Holder != self && !Asked && !Q.empty()) {
       /* отправка REQUEST процессу с ID в Holder */
       Asked = true;
     }
}
```

Функция работы с маркером. Отправлять маркер другому процессу или использовать его для входа в КС может только процесс, для которого одновременно выполняются следующие условия: (1) *Holder = self*, (2) *Using = false*, (3) очередь *Q* не пуста.

Если значение *self* не является первым элементом в очереди *Q*, то процесс, владеющий маркером, пересылает его соседнему процессу, идентификатор которого находится во главе очереди *Q*. При этом он удаляет этот элемент из *Q*, т.к. передача маркера по соответствующему запросу становится выполненной, и обновляет значение своей переменной *Holder* таким образом, чтобы оно указывало на нового обладателя маркера. Ситуация, когда в очереди *Q* процесса, владеющего маркером, первым элементом является *self*, возможна, например, когда этот процесс только что получил сообщение TOKEN от своего соседа. В этом случае он получает право войти в КС, удалив при этом первый элемент со значением *self* из начала очереди, и присвоив переменной *Using* значение *true*. Переменная *Asked* устанавливается в значение *false*, т.к. при передаче маркера процесс может не отправлять сообщение REQUEST процессу, на который указывает новое значение его переменной *Holder*.

Функция отправки запроса на получение маркера. Процесс, не владеющий маркером, отправляет сообщение REQUEST, если он нуждается в нем для себя или для выполнения просьбы на получение маркера от своего соседа, т.е. когда одновременно выполняются следующие условия: (1) *Holder* не равно *self*, (2) очередь *Q* не пуста, (3) *Asked = false*.

Проверка переменной *Asked* нужна для предотвращения отправки повторных сообщений REQUEST процессу, на который указывает *Holder*. Поэтому при отправке запроса REQUEST переменная *Asked* устанавливается в значение *true*. Важно отметить, что отправка сообщения REQUEST не меняет значения других переменных и структур данных, поддерживаемых процессом: значения переменных *Holder*, *Using* и очередь *Q* остаются прежними.

Алгоритм Реймонда опирается на представленные выше функции `AssignPrivilege` и `MakeRequest` и состоит из четырех частей, соответствующих четырем возможным событиям, наступающим в каждом процессе.

1. **Запрос на вход в КС.** Процесс P_i, желающий войти в КС, помещает значение *self* в свою очередь Q и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. Если P_i владеет маркером, функция `AssignPrivilege` либо предоставит возможность процессу войти в КС, либо передаст маркер другому процессу. Если маркер не содержится в P_i, функция `MakeRequest` может направить соответствующий запрос на получение маркера.
2. **Получение сообщения REQUEST.** При получении сообщения REQUEST от процесса P_j процесс P_i помещает идентификатор P_j в свою очередь Q и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. Если P_i владеет маркером, с помощью функции `AssignPrivilege` процесс P_i может отослать маркер запрашивающему процессу. Если маркер не содержится в P_i, с помощью функции `MakeRequest` P_i может передать запрос от P_j в сторону процесса с маркером.
3. **Получение сообщения TOKEN.** При получении сообщения TOKEN процесс P_i изменяет значение *Holder* на *self* и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. С помощью функции `AssignPrivilege` процесс P_i может передать маркер другому процессу или осуществить вход в КС. Если маркер будет передан другому процессу, с помощью функции `MakeRequest` процесс P_i может запросить возврат маркера себе.
4. **Выход из КС.** При выходе из КС процесс P_i изменяет значение *Using* на *false* и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. С помощью функции `AssignPrivilege` процесс P_i может передать маркер другому процессу, а выполнение функции `MakeRequest` позволит запросить возврат маркера обратно в P_i.

Обратим внимание, что процесс может войти в КС, если он обладает маркером, *и при этом* первый элемент в его очереди Q имеет значение *self*. Эта ситуация проиллюстрирована в коде функции `AssignPrivilege`, представленном в Листинге 4.2.

Благодаря организации процессов в логическое дерево возникновение конфликтных ситуаций, связанных с произвольными задержками передачи сообщений и изменением порядка их поступления в каналах, не обладающих свойством FIFO, ограничено взаимодействием двух соседних процессов. Поэтому в алгоритме Реймонда нет

необходимости использовать порядковые номера (англ. *sequence numbers*) сообщений. Алгоритм работает таким образом, что обмен сообщениями между двумя соседними процессами подчиняется определенному шаблону взаимодействия, проиллюстрированному на рис. 4.13. На рис. 4.13 предполагается, что изначально маркер находится в одной из вершин поддерева, корнем которого является процесс P_i.

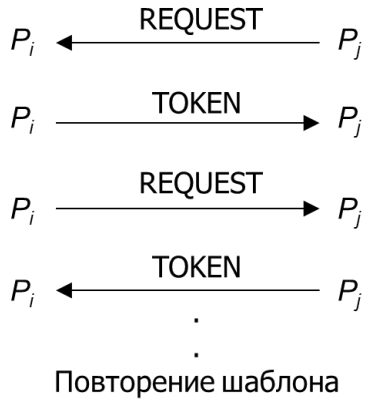


Рис. 4.13. Шаблон взаимодействия соседних процессов.

Как следует из рис. 4.13, единственно возможной ситуацией нарушения порядка поступления сообщений является ситуация, когда отправленное процессом P_i сообщение REQUEST, следующее за сообщением TOKEN, будет получено процессом P_j вперед сообщения TOKEN. На самом деле, процесс P_j в состоянии распознать такую ситуацию, т.к. согласно представленному шаблону взаимодействия, следующим ожидаемым сообщением должно быть именно сообщение TOKEN, и P_j сможет отложить обработку сообщения REQUEST до получения TOKEN. Однако это не является необходимым, т.к. такое нарушение порядка поступления сообщений не оказывает влияния на функционирование алгоритма. Действительно, если REQUEST поступает в процесс P_j раньше, чем TOKEN, то идентификатор процесса P_i будет помещен в конец очереди Q_j процесса P_j. В связи с тем, что P_j пока еще не владеет маркером, функция `AssignPrivilege` выполнена не будет. Факт того, что P_i отправил маркер P_j, означает, что очередь Q_j была не пуста, и переменная $Asked_j$ процесса P_j установлена в значение *true*, и поэтому функция `MakeRequest` также не будет выполнена, что не позволит запросу REQUEST распространиться среди других процессов. Когда сообщение TOKEN, в конце концов, доберется до процесса P_j, он войдет в КС или передаст маркер соседнему процессу, идентификатор которого находится в начале его очереди Q_j. Этим процессом не может быть процесс P_i, передающий маркер P_j, т.к. его запрос был добавлен в конец непустой очереди Q_j. Поэтому получение процессом P_j сообщения REQUEST вперед сообщения TOKEN не оказывает никакого влияния на его работу.

Начальное состояние системы и инициализация алгоритма. В начальном состоянии распределенной системы маркер должен находиться только у одного процесса. Процесс, обладающий маркером, устанавливает переменную *Holder* в значение *self* и отправляет сообщение INITIALIZE всем своим непосредственным соседям. При получении сообщения INITIALIZE процесс P_i присваивает переменной $Holder_i$ значение идентификатора процесса, от которого он получил INITIALIZE, и рассылает INITIALIZE далее всем своим соседним процессам за исключением процесса, от которого было получено сообщение INITIALIZE. Как только процесс получает сообщение INITIALIZE, он может запрашивать маркер, даже если еще не все процессы в дереве прошли процедуру инициализации. Начальное значение переменных *Using* и *Asked* равно *false* для всех процессов, очередь *Q* всех процессов пуста.

Теперь покажем, что алгоритм Реймонда обладает свойством живучести, т.е. не допускает ситуаций взаимной блокировки и голодания.

При рассмотрении возможности возникновения взаимной блокировки предположим, что допустима ситуация, когда ни один процесс не выполняется в КС, и существует один или несколько процессов, находящихся в состоянии запроса на вход в КС, но не получающих разрешения войти в нее. Такая ситуация возможна только при реализации одного из следующих сценариев:

- ни один из процессов не владеет маркером, поэтому маркер не может быть передан процессу, ожидающему его получения;
- обладающий маркером процесс не имеет информации, что другой процесс нуждается в маркере;
- сообщение TOKEN не может достигнуть процесса, запросившего доступ к КС.

Первый сценарий невозможен ввиду предположений, что каналы связи являются надежными, а процессы не выходят из строя.

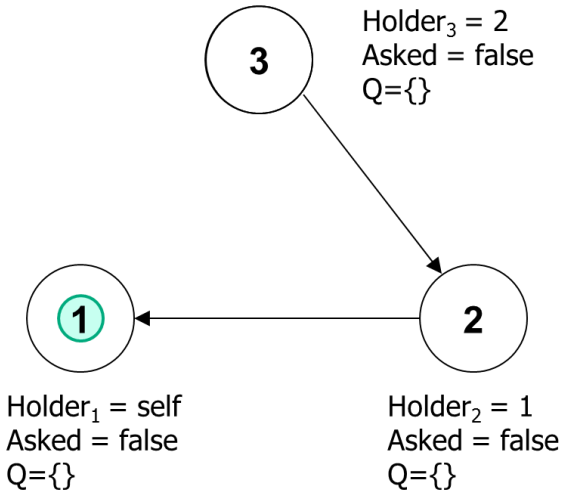
Совокупность переменных *Asked* различных процессов гарантирует, что существует последовательность сообщений REQUEST, отправленных вдоль пути, сформированному переменными *Holder*, от процесса, запросившего доступ к КС, к процессу, обладающему маркеру. Ситуация, когда сообщение TOKEN перемещается по дереву таким образом, что сообщение REQUEST никогда не достигнет процесса, в котором находится маркер, невозможна ввиду отсутствия циклов в дереве процессов. Единственной возможностью, при которой сообщение TOKEN может разминуться с сообщением REQUEST, является передача TOKEN от процесса P_i к соседнему процессу P_j в то время как REQUEST передается в обратном направлении, т.е. от P_j к P_i. Однако шаблон взаимодействия соседних процессов, представленный на рис. 4.13, предотвращает такую

ситуацию. А именно, сообщение TOKEN может быть отослано процессом P_i только после получения от P_j запроса REQUEST, на который P_i еще не отвечал. Если же P_j уже отправлял такой запрос процессу P_i, то тогда P_j не станет отправлять еще один запрос REQUEST, т.к. значение его переменной *Asked* будет равно *true*.

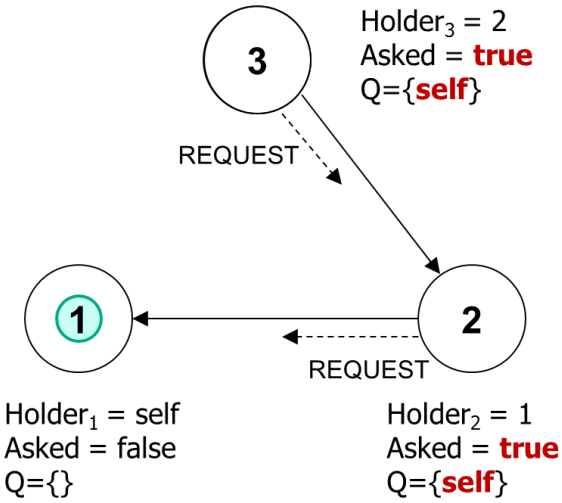
Таким образом процесс, владеющий маркером, рано или поздно получит запрос от соседнего с ним процесса, нуждающегося в маркере для себя или для выполнения просьбы на получение маркера от другого процесса. Идентификаторы процессов, отправляющих сообщения REQUEST вдоль пути, ведущего к процессу с маркером, помещаются в очереди Q промежуточных процессов таким образом, что, совместно, эти идентификаторы формируют обратный логический путь для передачи маркера к запрашивающему процессу. Поэтому, следуя этому пути, сообщение TOKEN может быть доставлено процессу, запросившему вход в КС.

Голодание в алгоритме Реймонда также невозможно, т.к. очереди Q промежуточных процессов на пути от запрашивающего процесса к процессу с маркером имеют ограниченный размер и обслуживаются по правилу FIFO. Поэтому каждый из промежуточных процессов рано или поздно (в зависимости от позиции его идентификатора в очереди Q соседнего промежуточного процесса) получит маркер и сможет передать его по обратному пути в сторону запрашивающего процесса. Таким образом, как только сообщение REQUEST от запрашивающего процесса достигнет процесса, владеющего маркером, сообщение TOKEN, в конце концов, гарантировано доберется до процесса, желающего войти в КС.

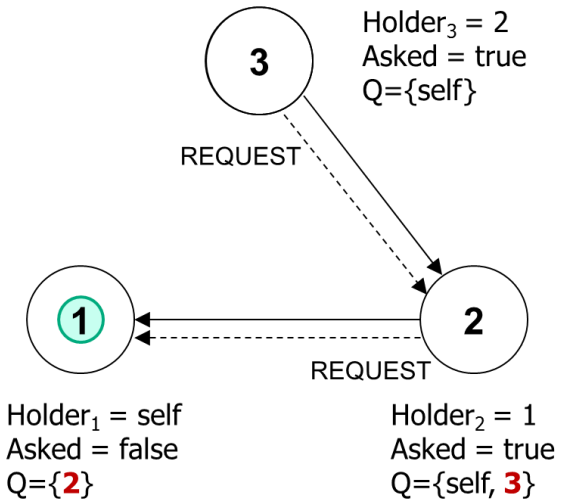
Пример работы алгоритма Реймонда приведен на рис. 4.14. В начальном состоянии системы маркером владеет процесс P_1, и все процессы находятся в состоянии выполнения вне КС, как показано на рис. 4.14а. Сетевые соединения установлены только между процессами P_1 и P_2, P_2 и P_3. Направление ребер в дереве процессов обозначено стрелками, указывающими на расположение маркера относительно каждого процесса. На рис. 4.14б, также как и в сценарии, представленном в примерах выше, процессы P_2 и P_3 запрашивают вход в КС приблизительно в одно и то же время, и каждый из них отправляет сообщение REQUEST своему соседу, идентификатор которого содержится в переменной *Holder*. При получении запроса от P_3 процесс P_2 не передает его P_1, т.к. $Asked_2 = true$, – см. рис. 4.14в. При этом сообщение REQUEST, отправленное ранее процессом P_2 процессу P_1, начинает представлять интересы и P_2, и P_3. На рис. 4.14г процесс P_1 передает маркер процессу P_2, одновременно изменяя значение своей переменной *Holder* так, чтобы оно указывало на P_2, как на нового обладателя маркера. В начале очереди Q процесса P_2 стоит *self*, поэтому при получении маркера этот процесс сможет войти в КС, как показано на рис. 4.14д.


3
Holder3 = 2
Asked = false
Q={}
1
2
Holder1 = self
Asked = false
Q={}
Holder2 = 1
Asked = false
Q={}

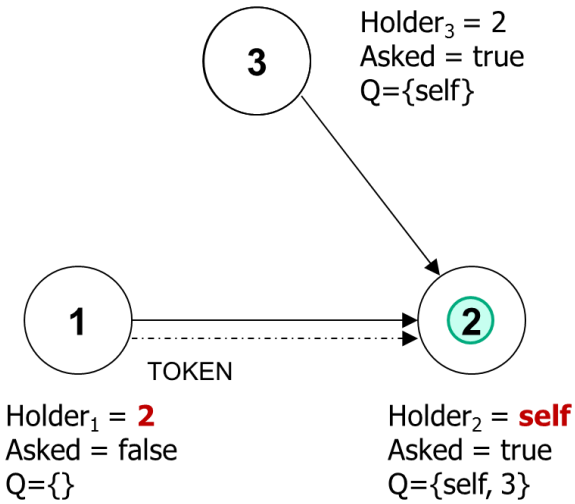
(а)


3
Holder3 = 2
Asked = true
Q={self}
REQUEST
1
2
REQUEST
Holder1 = self
Asked = false
Q={}
Holder2 = 1
Asked = true
Q={self}

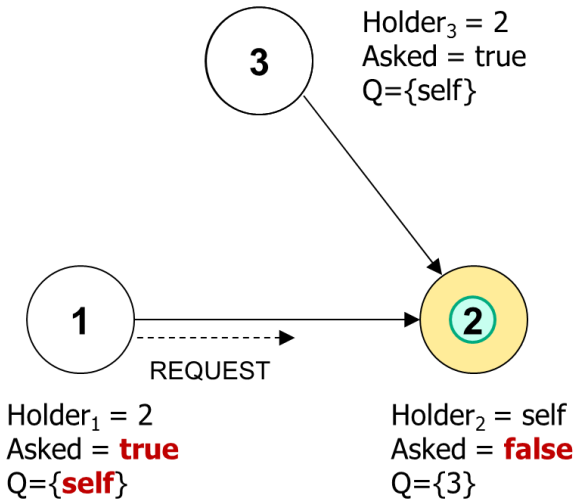
(б)


3
Holder3 = 2
Asked = true
Q={self}
REQUEST
1
2
REQUEST
Holder1 = self
Asked = false
Q={2}
Holder2 = 1
Asked = true
Q={self, 3}

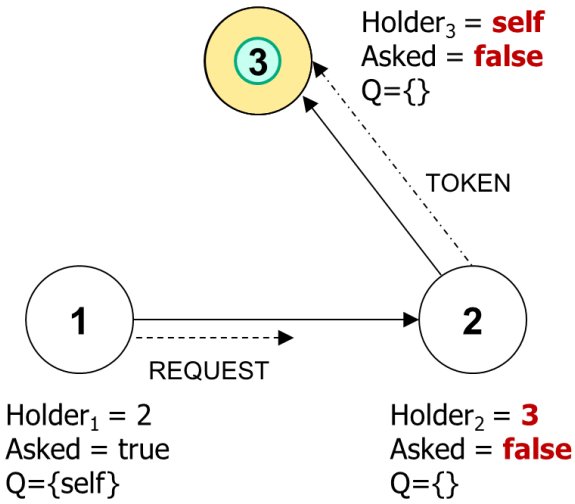
(в)


3
Holder3 = 2
Asked = true
Q={self}
1
2
TOKEN
Holder1 = 2
Asked = false
Q={}
Holder2 = self
Asked = true
Q={self, 3}

(г)


3
Holder3 = 2
Asked = true
Q={self}
1
2
REQUEST
Holder1 = 2
Asked = true
Q={self}
Holder2 = self
Asked = false
Q={3}

(д)


3
Holder3 = self
Asked = false
Q={}
TOKEN
1
2
REQUEST
Holder1 = 2
Asked = true
Q={self}
Holder2 = 3
Asked = false
Q={}

(е)

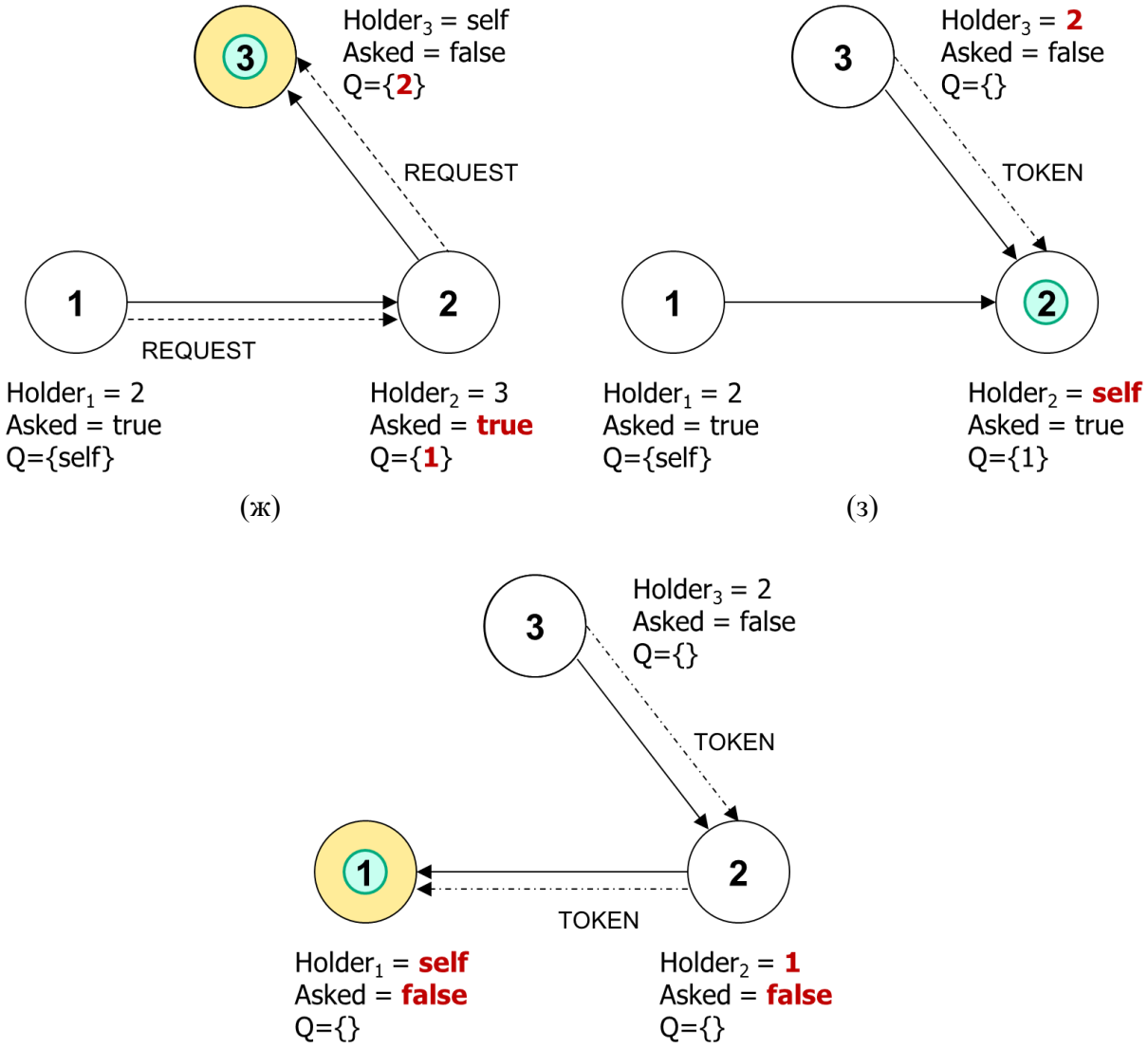


Рис. 4.14. Пример работы алгоритма Реймонда.

Далее в нашем сценарии процесс P_1 переходит в состояние запроса на вход в КС и отправляет сообщение REQUEST процессу P_2. На рис. 4.14е процесс P_2 выходит из КС и передает маркер процессу, идентификатор которого содержится во главе его очереди Q, т.е. процессу P_3. Получив маркер, процесс P_3 войдет в КС. После этого запрос процесса P_1 достигает процесса P_2, и P_2 передает запрос процессу P_3, как показано на рис. 4.14ж. Обратим внимание, что идентификаторы процессов, находящиеся в очередях Q_3 и Q_2 процессов P_3 и P_2 совместно образуют логический путь для передачи маркера из процесса P_3, владеющего маркером, в процесс P_1, запрашивающий маркер. При выходе процесса P_3 из КС сообщение TOKEN будет отправлено по этому пути: сначала в P_2 и затем в P_1, как показано на рис. 4.14з и рис. 4.14и. Также с перемещением маркера будут изменяться значения переменных *Holder* таким образом, что совместно эти

значения всегда будут указывать на текущее положение маркера. В итоге, получив маркер, процесс P_1 сможет войти в КС.

Чтобы оценить эффективность работы алгоритма Реймонда отметим, что наибольшее число сообщений, требуемых для входа в КС, составляет $2D$, где D – диаметр дерева процессов, т.е. максимальное из расстояний между парами вершин дерева. Эта ситуация возникает для случая наибольшего удаления запрашивающего процесса от процесса с маркером, и требует D сообщений REQUEST и D сообщений TOKEN для входа в КС. Поэтому самой неудачной топологией соединений между процессами является прямая линия, когда длина пути между крайними процессами составляет $N-1$ ребер. В этом случае при использовании алгоритма Реймонда может потребоваться обмен $2(N-1)$ сообщениями, что совпадает с числом сообщений, используемых в алгоритме Рикарта-Агравала, и превышает число сообщений в алгоритме Сузуки-Касами. Такая ситуация соответствует постоянной передаче маркера из одного крайнего процесса в другой. В случае, когда каждый из процессов может запросить вход в КС с одинаковой вероятностью, среднее расстояние между запрашивающим процессом и процессом с маркером составит $(N+1)/3$ ребер при условии, что маркер не находится в процессе, желающем войти в КС. Тогда среднее число сообщений будет приблизительно составлять $2N/3$, что меньше N сообщений, требуемых в алгоритме Сузуки-Касами. Если же маркер находится у запрашивающего процесса, для входа в КС не потребуется ни одного сообщения.

С точки зрения уменьшения числа сообщений, требуемых для входа в КС, предпочтительной топологией соединений между процессами является дерево с большой степенью ветвления K. В этом случае диаметр дерева и, следовательно, максимальное число сообщений будет определяться как $O(\log_{K-1} N)$. Однако важно отметить, что с ростом степени ветвления K также растет максимальный размер очереди Q, который определяется числом соседей каждого процесса.

При высокой интенсивности поступления запросов на вход в КС алгоритм Реймонда проявляет интересное свойство: с увеличением числа процессов, ожидающих маркер, число сообщений, требуемых для работы с КС, уменьшается. Связано это с тем, что сообщение REQUEST, отправленное запрашивающим процессом, в этом случае обычно не проходит весь путь к процессу с маркером, а поступает в промежуточный процесс P_i, у которого переменная $Asked_i = true$. При этом сообщение REQUEST, отправленное ранее процессом P_i в сторону $Holder_i$, начинает представлять интересы всех процессов, для которых путь в процесс с маркером проходит через P_i. Далее, при получении маркера от процесса P_j процесс P_i и все его соседи смогут воспользоваться маркером до его возврата в P_j. Другими словами, маркер будет перемещаться внутри поддерева, корнем которого является P_i, до тех пор, пока не будут

обслужены все запросы на вход в КС, появившиеся до момента поступления маркера в P_i (точнее, до поступления в P_i следующего за маркером запроса от P_j). Поэтому среднее расстояние, проходимое сообщением TOKEN между процессами, ожидающими вход в КС, будет много меньше диаметра дерева.

Если все процессы распределенной системы постоянно запрашивают вход в КС, сообщение TOKEN проходит все $N - 1$ ребер дерева ровно два раза (в прямом и обратном направлении) для предоставления доступа к КС каждому из N процессов. В связи с тем, что сообщение TOKEN отправляется в ответ на сообщение REQUEST, всего в системе будет передано $4(N - 1)$ сообщений. Поэтому при полной загрузке среднее число сообщений, требуемых для входа в КС, будет равняться $4(N - 1)/N \approx 4$.

Варианты оптимизации алгоритма Реймонда. Одним из вариантов оптимизации алгоритма Реймонда может быть допустимое в некоторых случаях совмещение запроса REQUEST вместе с сообщением TOKEN. А именно, при выходе из КС процессу может потребоваться передать маркер своему соседу с помощью функции `AssignPrivilege`, а также отправить ему запрос на возврат маркера с помощью функции `MakeRequest`. Если передачу маркера отложить до определения необходимости передачи запроса, в таких ситуациях отправку TOKEN и REQUEST можно совместить в одном сообщении, тем самым уменьшая общее количество передаваемых сообщений. При получении такого совмещенного сообщения оно обрабатывается как два независимых сообщения. Стоит отметить, что совмещенные сообщения могут быть отправлены только процессами, не являющимися листьями дерева, т.к. листья не могут требовать возврата маркера сразу же после выхода из КС. Данный подход становится эффективным при высокой интенсивности поступления запросов на вход в КС. При низкой загрузке вероятность того, что в очереди Q процесса, передающего маркер, содержатся идентификаторы других процессов, нуждающихся в маркере, невелика.

Другим вариантом оптимизации является так называемая "жадная" версия алгоритма Реймонда. Суть этого алгоритма заключается в том, что если процесс P_i запрашивает доступ к КС, он может помещать значение *self* не в конец своей очереди Q, как в стандартном алгоритме Реймонда, а в ее начало вперед идентификаторов других процессов, уже находящихся в очереди. В этом случае при получении маркера процесс P_i сможет сразу войти в КС, не передавая маркер соседним с ним процессам и не дожидаясь, пока они закончат свою работу с маркером. Как и для предыдущего случая, этот подход не влияет на работу системы при низкой загрузке, т.к. *self*, скорее всего, окажется единственным элементом в очереди Q. При высокой интенсивности поступления запросов "жадный" алгоритм позволит уменьшить *среднюю* задержку на получение маркера и,

как следствие, увеличить число входов в КС за определенный интервал времени. Однако это преимущество будет достигнуто за счет процессов, являющихся листьями дерева. Действительно, в стандартном алгоритме Реймонда задержка на получение маркера приблизительно одинакова для всех процессов в системе вне зависимости от того, являются ли они листьями дерева или нет. В случае "жадной" версии алгоритма Реймонда листья могут ожидать прихода маркера значительно дольше процессов, находящихся в вершинах ветвления, и, соответственно, получать его в пользование гораздо реже. Например, если процессы распределенной системы на рис. 4.10 постоянно запрашивают вход в КС, то процессы P_1 и P_4, находящиеся в вершинах ветвления, будут получать возможность войти в КС в три раза чаще, чем остальные процессы, и соответственно, задержка на получение маркера для них будет в три раза меньше.

Таким образом, в отличие от стандартного алгоритма "жадный" алгоритм Реймонда не является справедливым в том смысле, что он предоставляет процессам неравные шансы войти в КС. На самом деле, эффективность "жадной" версии обеспечивается напрямую за счет справедливости предоставления доступа к КС, но этот алгоритм по-прежнему гарантирует отсутствие голодания процессов. Действительно, очередь Q каждого процесса обслуживается в порядке FIFO и перед идентификатором процесса, стоящим в начале очереди Q, может быть поставлено только значение *self* и только один раз, т.к. при выходе из КС процесс должен будет передать маркер процессу с этим идентификатором. Поэтому, в конце концов, идентификатор каждого процесса в очереди Q доберется до ее начала, и, следовательно, процесс, нуждающийся в маркере, получит его.

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Столлингс В. Операционные системы. Четвертое издание. Пер. с англ. – М.: Вильямс, 2004. – 848 с.: ил.
2. Таненбаум Э., ван Стеен М. Распределенные системы. Принципы и парадигмы. – СПб: Питер, 2003. – 877 с.: ил.
3. Тель Ж. Введение в распределенные алгоритмы. Пер. с англ. – М.: МЦНМО, 2009. – 616 с.
4. Kshemkalyani A. D., Singhal M. Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, 2008.
5. Stallings W. Operating Systems: Internals and design principles. Seventh edition. Prentice Hall, 2011.
6. Tanenbaum A. S., van Steen M. Distributed Systems: Principles and Paradigms. Second edition. Pearson Prentice Hall, 2007.
7. Tel G. Introduction to Distributed Algorithms. Second edition. Cambridge University Press, 2000.

Дополнительная литература

8. Топорков В. В. Модели распределенных вычислений. – М.: Физматлит, 2004. – 320 с.
9. Хьюз К., Хьюз Т. Параллельное и распределенное программирование с использованием C++. Пер. с англ. – М.: Издательский дом "Вильямс", 2004. – 672 с.: ил.
10. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 512 с.: ил.
11. Chandy K. M., Lamport L. Distributed snapshots: determining global states of distributed systems // ACM Transactions on Computer Systems, 3(1), 1985. P.63–75.
12. Chandy K. M., Misra J. The Drinking Philosophers Problem // ACM TOPLAS, 6:4, October 1984. P.632-646.
13. Chandy K. M., Misra J. Parallel program design: a foundation. Addison-Wesley, 1988.
14. Charron-Bost B. Concerning the size of logical clocks in distributed systems // Information Processing Letters, 39, 1991. P.11–16.
15. Charron-Bost B., Tel G., Mattern F. Synchronous, asynchronous, and causally ordered communication // Distributed Computing, 9(4), 1996. P.173–191.
16. Coulouris G., Dollimore J., Kindberg T., Blair G. Distributed Systems: Concepts and Design. Fifth edition. Addison-Wesley, 2011.

17. Fidge C. Logical time in distributed computing systems // IEEE Computer, August, 1991. P.28–33.
18. Fowler J., Zwaenepoel W. Causal distributed breakpoints // Proceedings of the 10th International Conference on Distributed Computing Systems, 1990. P.134–141.
19. Garg V.K. Concurrent and distributed computing in Java. John Wiley & Sons, 2004.
20. Ghosh S. Distributed systems: an algorithmic approach. Chapman and Hall/CRC, 2007.
21. Jard C., Jourdan G.-C. Dependency tracking and filtering in distributed computations // Brief Announcements of the ACM Symposium on PODC, 1994. A full presentation appeared as IRISA Technical Report No. 851, 1994.
22. Lamport L. Time, clocks and the ordering of events in a distributed system // Communications of the ACM, 21, 1978. P.558–564.
23. Lynch N. A. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.
24. Rauber T., Rünger G. Parallel Programming For Multicore and Cluster Systems. Springer, 2010.
25. Mattern F. Virtual time and global states of distributed systems // Proceedings of the Parallel and Distributed Algorithms Conference, North-Holland, 1988. P.215–226.
26. Raymond K. Tree-based algorithm for distributed mutual exclusion // ACM Transactions on Computer Systems, 7, 1989. P.61–77.
27. Raynal M. A simple taxonomy of distributed mutual exclusion algorithms // Operating Systems Review, 25(2), 1991. P.47–50.
28. Ricart G., Agrawala A. K. An optimal algorithm for mutual exclusion in computer networks // Communications of the ACM, 24(1), 1981. P.9–17.
29. Singhal M., Kshemkalyani A. An efficient implementation of vector clocks // Information Processing Letters, 43, August, 1992. P.47–52.
30. Suzuki I., Kasami T. A distributed mutual exclusion algorithm // ACM Transactions on Computer Systems, 3(4), 1985. P.344–349.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра вычислительной техники (ВТ) Санкт-Петербургского государственного университета информационных технологий, механики и оптики является одной из авторитетнейших научно-педагогических школ России. Более 70 лет кафедра ведет подготовку высококвалифицированных специалистов в области информатики и вычислительной техники. За годы своего существования на кафедре подготовлено более 5000 высококвалифицированных специалистов, подготовлено более 230 кандидатов наук и 36 докторов наук. Преподавателями и научными сотрудниками кафедры изданы десятки монографий, учебников и учебно-методических пособий. Кафедра имеет высококвалифицированный состав преподавателей, среди которых 9 профессоров и 14 доцентов.

Традиционно основной упор в подготовке специалистов на кафедре ВТ делается на фундаментальную базовую подготовку как в рамках общепрофессиональных дисциплин, так и специальных дисциплин, охватывающих все наиболее важные разделы информатики и вычислительной техники. Студенты старших курсов могут специализироваться в более узких профессиональных областях, таких как информационно-управляющие системы, открытые информационные системы и базы данных, сети ЭВМ и корпоративные системы. В 2011 году начата подготовка бакалавров и магистров по новому направлению «Программная инженерия».

Косяков Михаил Сергеевич

Введение в распределенные вычисления

Учебное пособие

В авторской редакции
Редакционно-издательский отдел НИУ ИТМО
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ №
Тираж 250 экз.
Отпечатано на ризографе