Пролог

Распределенные системы и обработка распределенной информации получили значительное внимание в последние несколько лет, и почти каждый университет предлагает, по крайней мере, один курс по разработке распределенных алгоритмов. Существует большое число книг о принципах распределенных систем; см. например Tanenbaum [Tan88] или Sloman and Kramer [SK87], хотя они концентрируются в основном на архитектурных аспектах, а не на алгоритмах.

Было замечено, что алгоритмы – это основа любого применения компьютеров. Поэтому кажется оправданным посвятить эту книгу полностью распределенным алгоритмам. Эта книга направлена на то, чтобы представить большую часть теории распределенных алгоритмов, которые развивались в течение последних 15 лет. Эта книга может быть использована как учебник для одно- или двух-семестрового курса по распределенным алгоритмам. Преподаватель односеместрового курса может выбирать темы по своему усмотрению.

Эта книга также обеспечит полезную вспомогательную и ссылочную информацию для профессиональных инженеров и исследователей, работающих с распределенными системами.

Упражнения. Каждая глава (за исключением глав 1 и 12) оканчивается списком упражнений и маленьких проектов. Проекты обычно требуют, чтобы читатель разработал маленькое, но нетривиальное расширение или практическое решение по материалу главы, и в большинстве случаев у автора нет решения. Если читатель добьется успеха в разработке этих маленьких проектов, то мне бы хотелось иметь копию результата.

Список ответов (иногда частичных) у большинству упражнений доступен для преподавателей. Он может быть получен у автора или по анонимному ftp.

Исправления и предложения. Если читатель найдет ошибки и пропуски в этой книге, то пусть информирует автора (предпочтительно по электронной почте). Вся конструктивная критика, включая предложения по упражнения, очень приветствуется.

1 Введение: распределенные системы

Эта глава представляет причины для изучения распределенных алгоритмов, кратко описывая типы аппаратных и программных систем, для которых развивались распределенные алгоритмы. Под распределенной системой мы понимаем все компьютерные системы, где несколько компьютеров или процессоров кооперируются некоторым образом. Это определение включает глобальные компьютерные сети, локальные сети, мультипроцессорные компьютеры, в которых каждый процессор имеет свой собственный управляющий блок, а также системы со взаимодействующими процессами.

Различные типы распределенных систем и причины использования распределенных систем обсуждаются в разделе 1.1. Приводятся некоторые примеры существующих систем. Главная тема этой книги, однако, не то, как эти сис-

темы выглядят, или как они используются, но как заставить их работать. Более того, как заставить работать распределенные *алгоритмы* в этих системах.

Конечно, целиком структуру и функционирование распределенной системы нельзя полностью понять изучением только алгоритмов самих по себе. Чтобы понять такую систему полностью нужно также изучить ее архитектуру и программное обеспечение, то есть, разбиение цельной функциональности по модулям. Также, есть много важных вопросов, относящихся к свойствам языков программирования, используемых для разработки программного обеспечения распределенных систем. Эти вопросы будут обсуждаться в разделе 1.2.

Однако сейчас существует много превосходных книг по распределенным системам, касающихся архитектурных и языковых аспектов. Смотрите Tanenbaum [Tan88], Sloman and Kramer [SK87], Bal [Bal90], Coulouris [CD88], Goscinski [Gos91]. Как уже говорилось, настоящий труд делает упор на алгоритмы распределенных систем. Раздел 1.3 объясняет, почему разработка распределенных алгоритмов отличается от разработки централизованных алгоритмов, там также делается краткий обзор текущего состояния дел в исследованиях и дается описание остальной части книги.

1.1 Что такое распределенная система?

В этой главе мы будем использовать термин «распределенная система», подразумевая взаимосвязанный набор автономных компьютеров, процессов или процессоров. Компьютеры, процессы или процессоры упоминаются как узлы распределенной системы. (В последующих главах мы будем использовать более техническое понятие, см. определение 2.6.) Будучи определенными как «автономные», узлы должны быть, по крайней мере, оборудованы своим собственным блоком управления. Таким образом, параллельный компьютер с одним потоком управления и несколькими потоками данных (SIMD) не подпадает под определение распределенной системы. Чтобы быть определенными как «взаимосвязанными», узлы должны иметь возможность обмениваться информацией.

Так как процессы могут играть роль узлов системы, определение включает программные системы, построенные как набор взаимодействующих процессов, даже если они выполняются на одной аппаратной платформе. В большинстве случаев, однако, распределенная система будет, по крайней мере, содержать несколько процессоров, соединенный коммутирующей аппаратурой.

Более ограничивающие определения распределенных систем могут быть также найдены в литературе. Tanenbaum [Tan88], например, называет систему распределенной, только если существуют автономные узлы прозрачные для пользователей системы. Система распределенная в этом смысле ведет себя как виртуальная самостоятельная компьютерная система, но реализация этой прозрачности требует разработки замысловатых алгоритмов распределенного управления.

1.1.1 Мотивация

Распределенные компьютерные системы могут получить предпочтение среди ряда систем или их использования бывает просто не избежать, в силу многих причин, некоторые из которых обсуждаются ниже. Этот список не исчерпывающий. Выбор распределенной системы может быть мотивирован более чем одним аргументов приведенным ниже. И некоторые из преимуществ могут

быть получены как полезный побочный эффект при выборе других причин. Характеристики распределенных систем могут также варьироваться, в зависимости от причины их существования, но об этом мы поговорим более детально в разделах с 1.1.2 по 1.1.6.

(1) *Обмен информацией*. Необходимость обмена данными между различными компьютерами возросла в шестидесятых, когда большинство основных университетов и компаний начали пользоваться своими собственными майнфреймами. Взаимодействие между людьми из различных организаций облегчилось благодаря обмену данными между компьютерами этих организаций, и это дало рост развитию так называемых глобальных сетей (WAN). Компьютерная система соединенная в глобальную сеть обычно снабжалась всем что необходимо пользователю: резервными хранилищами данных, дисками, многими прикладными программами и принтерами.

Позже компьютеры стали меньше и дешевле, и сегодня одна организация может иметь множество компьютеров, иногда даже один компьютер на одного работника (рабочую станцию). В этом случае также требуется чтобы эти компьютеры были соединены для электронного обмена информацией между персоналом компании.

(2) *Разделение ресурсов*. Хотя с приходом более дешевых компьютеров стало возможно снабжать каждого сотрудника организации личным компьютером, это же нельзя сделать для периферии (принтеры, резервные хранилища, блоки дисков). В этом меньшем масштабе каждый компьютер может положиться на специальные серверы, которые снабжают его компиляторами и другими прикладными программами. Также, памяти любого компьютера обычно недостаточно, чтобы хранить большой набор прикладных программ, требуемых для каждого пользователя. Кроме того, компьютеры могут использовать специальные узлы для служб печати и хранения данных. Сеть, соединяющая компьютеры в масштабе предприятия называется локальной вычислительной сетью(LAN).

Причины, по которым организация устанавливает сеть небольших компьютеров, а не майнфреймы – снижение стоимости и расширяемость. Во-первых, меньшие компьютеры имеют лучше соотношение цена-производительность, чем большие компьютеры. Типичный майнфрейм может совершать операции в 50 раз быстрее, чем персональный компьютер, но иметь стоимость в 500 раз большую. Во-вторых, если мощности системы больше не достаточно, то сеть может быть расширена добавлением других машин (файловых серверов, принтеров и рабочих станций). Если мощность монолитной системы больше неудовлетворительна, остается только полная замена.

(3) *Большая надежность благодаря репликации*. Распределенные системы имеют потенциал надежности больший, чем монолитные системы благодаря свойству их частичного выхода из строя. Это значит, что некоторые узлы системы могут выйти из строя, в то время как другие по прежнему функционируют и могут взять на себя задачи испорченных компонентов. Выход из строя монолитного компьютера действует на всю систему целиком и нет возможности продолжать вычисления в

этом случае. По этой причине распределенные архитектуры представляют интерес при разработке высоко надежных компьютерных систем.

Высоко надежная система обычно состоит из двух, трех или четырех репликационных унипроцессоров, которые исполняют прикладную программу и поддерживаются механизмом голосования, чтобы отфильтровывать результаты машин. Правильное функционирование распределенной системы при наличии поврежденных компонент требует довольно сложной алгоритмической поддержки.

(4) *Большая производительность благодаря распараллеливанию*. Наличие многих процессоров в распределенной системе открывает возможность снижения дополнительного времени для интенсивной работы с помощью разделения работы среди нескольких процессоров.

Параллельные компьютеры разработаны специально для этой цели, но пользователи локальных сетей также могут получить пользу от параллелизма, перекладывая задачи на другие рабочие станции.

(5) *Упрощение разработки благодаря специализации*. Разработка компьютерной системы может быть сложной, особенно если требуется значительная функциональность. Разработка может быть зачастую упрощена разбитием системы на модули, каждый из которых отвечает за часть функциональности и коммутируется с другими модулями.

На уровне одной программы модульность достигается определением абстрактных типов данных и процедур для различных задач. Большая система может быть определена как набор кооперирующих процессов. В обоих случаях, модули могут быть исполнены в рамках одного компьютера. Но также возможно иметь локальную сеть с различными типами компьютеров: один снабжен специальным оборудованием для вычислений, другой – графическим оборудованием, третий – дисками и т.д.

1.1.2 Компьютерные сети

Под компьютерной сетью мы понимаем набор компьютеров, соединенных коммуникационными средствами, с помощью которых компьютеры могут обмениваться информацией. Этот обмен имеет место при посылке и получении сообщений. Компьютерные сети удовлетворяют нашему определению распределенных систем. В зависимости от расстояния между компьютерами и их принадлежностью, компьютерные сети называются либо *глобальными,* либо *локальными*.

Глобальная сеть обычно соединяет компьютеры, принадлежащие различным организациям (предприятия, университеты и т.д.). Физическое расстояние между узлами обычно составляет 10 километров и более. Каждый узел такой сети – это законченная компьютерная система, включающая всю периферию и значительное количество прикладного программного обеспечения. Главная задача глобальной сети – это обмен информацией между пользователями различными узлов.

Локальная сеть обычно соединяет компьютеры, принадлежащие одной организации. Физическое расстояние между узлами обычно 10 километров и

менее. Узел такой сети – это обычно рабочая станция, файловый сервер или сервер печати, т.е. относительно маленькая станция, специализирующаяся на особых функциях внутри организации. Главная задача локальной сети – это обычный обмен информацией и разделение ресурсов.

Граница между двумя типами сетей не может быть всегда четко очерчена, и обычно различие не столь важно с алгоритмической точки зрения, потому что во всех компьютерных сетях встречаются схожие проблемы. Релевантные отличия, относящиеся к развитию алгоритмов, следующие:

(1) *Параметры надежности*. В глобальных сетях вероятность, что что-то пойдет не так в течение предачи сообщения никода не может быть игнорирована. Распределенные алгоритмы для глобальных сетей обычно разрабатываются так, чтобы справляться с возможными неполадками. Локальные сети более надежные, и алгоритмы для них могут быть разработаны в предположении абсолютной надежности коммуникаций. В этом случае, однако, невероятное событие, что что-то произойдет не так может быть пропущено, что обусловит неправильную работу системы.
(2) *Время коммуникации*. Времена передачи сообщений в глобальных сетях на порядки больше, чем времена передачи в локальных сетях. В глобальных сетях время необходимое для обработки сообщения почти всегда может быть игнорировано по стравнению со временем передачи сообщения.
(3) *Гомогенность*. Даже хотя в локальных сетях не все узлы обязательно равны, обычно возможно принять единое программное обеспечение и протоколы для использования в рамках одной организации. В глобальных сетях используется множество различных протоколов, которые поднимают проблему преобразования между различными протоколами и разработки программного обеспечения, которое совместимо с различными стандартами.
(4) *Взаимное доверие*. Внутри одной организации можно доверять всем пользователям, но в глобальной сети это определенно не так. Глобальная сеть требует развития безопасных алгоритмов, защищающих узлы от аггресивных пользователей.

Раздел 1.1.3 посвящен краткому обсуждлению глобальных сетей, локальные сети обсуждаются в разделе 1.1.4.

1.1.3 Глобальные сети

Историческое развитие. Большая часть первооткрывательской работы в развитии глобальных компьютерных сетей было проделано в проектах агентства ARPA министерства обороны США. Сеть ARPANET начала работать в 1969, и соединяла в то время 4 узла. Эта сеть выросла до нескольких сотен узлов, и другие сети были установлены с использованием подобной технологии (MILNET, CYRPRESS). ARPANET содержит специальные узлы (называемые процессорами интерфейса сообщений (IMP)), которые предназначены только для обработки потока сообщений.

Когда UNIX системы стали широко использоваться, было признана необходимость информационного обмена между различными UNIX машинами, для чего была написана программа uucp (Unix-to-Unix CoPy). С помощью этой программы можно обмениваться файлами по телефонным каналам и сетям с пользователями UNIX – эта программа дала название быстрорастущим *UUCP сетям*. Также другая большая сеть, BITNET, была разработана в восьмидесятые, так как ARPANET принадлежала министерству обороны и только несколько организаций могли к ней подключаться.

Сегодня все эти сети соединены между собой с помощью узлов, которые принадлежат двум сетям (называемые шлюзами) и позволяющих обмениваться информацией узлам различных сетей. Введение унифицированного адресного пространства превратило все сети в одну виртуальную сеть, известную как Internet. Электронный адрес автора (gerard@cs.ruu.nl) обеспечивает информацию о сети, к которой подключен его департамент.

Алгоритмические проблемы и проблемы организации. Глобальные сети всегда организованы как сети типа *точка-точка*. Это означает, что коммуникация между парой узлов осуществляется при помощи механизма особенного по отношению к этим двум узлам. Такой механизм может быть телефонной линией, оптоволокном или спутниковой связью и т.д. Структура соединений в сетях точка-точка может быть хорошо изображена, если нарисовать каждый узел как окружность и связи между ними как линии, если линия коммуникация существует между этими двумя узлами, см. рис. 1.1. Говоря техническим языком, структура представляется графом, грани которого представляют собой линии коммуникации в сети. Сводка по терминологии теории графов приведена в Дополнении Б.

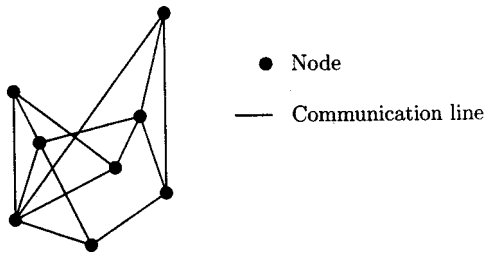


Рис. 1.1 Пример сети точка-точка

Основное назначение глобальных сетей – это обмен информацией, например, в форме электронной почты, досок объявлений, и удаленных файлов. Разработка приемлемой системы коммнуникаций для этих целей требует решения следующих алгоритмических проблем, некоторые из которых обсуждаются в Части 1 этой книги.

(1) *Надежность обмена данными по типу точка-точка* (глава 3). Два узла соединенные линией, обмениваются данными по этой линии, но они должны как-то справляться с потенциальной ненадежностью линии. Из-за атмосферных явлений, падения напряжения и других физических обстоятельств, сообщение, посланное через линию может быть получено с частично искаженным или даже утерянным.

Эти нарушения при передаче должен быть распознаны и исправлены.

Эта проблема встречается не только для двух напрямую соединенных узлов, но также для узлов, не соединенных напрямую, а связанных посредством промежуточных узлов. В этом случае проблема даже более сложна, потому что ко всему прочему сообщения могут доставляться в порядке, отличном от того, в котором они были посланы, а также сообщения могут прибывать с большим опозданием или продублированные.

(2) *Выбор путей коммуникации.* (глава 4). В сети точка-точка обычно слишком дорого обеспечивать связь между каждой парой узлов. Следовательно, некоторые пары узлов должны положиться на другие узлы для того, чтобы взаимодействовать. Проблема маршрутизации касается выбора пути (или путей) между узлами, которые хотят взаимодействовать. Алгоритм, используемый для выбора пути, связан со схемой, по которой узлы именуются, т.е. форматом адреса, который узел должен использовать, чтобы послать сообщение другому узлу. Выбор пути в промежуточных узлах производится с использованием адреса, и выбор может быть сделан эффективно, если в адресе кодируется в адресах.

(3) *Контроль перегрузок.* Пропускная способность коммутируемой сети может сильно падать, если много сообщений передается одновременно. Поэтому генерирование сообщений различными узлами должно управляться и должно зависеть от свободных мощностей сети. Некоторые методы предотвращения перегрузок обсуждаются в [Tann88, раздел 5.3].

(4) *Предотвращение тупиков.* (глава 5). Сети типа точка-точка иногда называются сетями типа *сохранить-и-передать*, потому что сообщение, которое посылается через несколько промежуточных узлов должно сохраняться в каждом из этих узлов, а затем форвардиться к следующему узлу. Так как пространство памяти, доступное для этой цели в промежуточных узлах ограничено, то память должна тщательно управляться для того, чтобы предотвратить тупиковые ситуации. В таких ситуациях существует набор сообщений, ни одно из которых не может быть отфорвардено, потому что память следующего узла в маршруте полностью занято другими сообщениями.

(5) *Безопасность.* Сети, соединяют компьютеры с различными пользователями, некоторые из которых могут попытаться злоупотребить или даже испортить системы других. Так как возможно зарегистрироваться в компьютерной системе из любой точки мира, то требуются надежные методы для аутентификации пользователей, криптографические методы, сканирование входящей информации. Криптографические методы могут быть использованы, чтобы шифровать данные для безопасности от несанкционированного чтения и чтобы ставить электронные подписи против несанкционированного написания.

1.1.4 Локальные сети

Локальная сеть используется организацией для соединения набора компьютеров, которые ей принадлежат. Обычно, основное назначение этих компьютеров заключается в разделении ресурсов (как файлов, так и аппаратной периферии) и для облегчения обмена информацией между сотрудниками. Иногда сети также используются для повышения скорости вычислений (перекладыванием задач на другие узлы) и чтобы позволить некоторым узлам быть для других запасными в случае их повреждения.

Рис. 1.2 Сеть с шинной организацией

Примеры и организация. В первой половинек 1970-х локальная сеть Ethernet была разработана Xerox. В то время как имена глобальных сетей ARPANET, BITNET, и т.д. происходят от конкретных сетей, имена локальных сетей – это обычно имена производителей. Есть одна ARPANET, одна BITNET, и одна UUCP сеть, каждая компания может установить свою собственную Ethernet, Token Ring или SNA сеть.

В отличие от глобальных сетей, ethernet организована с использованием шинной структуры, т.е. сообщение между узлами имеет место посредством единственного механизма, к которому все узлы подключены; см. рис. 1.2. Шинная организация стала повсеместной для локальных сетей, хотя могут быть различия в том как выглядит механизм или как он используется.

Устройство Ethernet разрешает передачу только одного сообщения в каждый момент времени; другие разработки, такие как токен ринг (разработанный в лаборатории Цюрих IBM), допускает *пространственное использование*, которое означает, что несколько сообщений могут передаваться через механизм комму-

никации одновременно. Шинная организация требует немного аппаратуры и поэтому дешевая, но имеет тот недостаток, что эта организация не очень хорошо масштабируется. Это означает, что существует очень жесткий потолок числа узлов, которые могут быть соединены одной шиной. Большие компании со многими компьютерами должны соединять их несколькими шинами, и использовать *мосты* для соединения шин друг с другом, создавая иерархию всей сети организации.

Не все локальные сети используют шинную организацию. IBM разработала точка-точка сетевой продукт называемый SNA для того, чтобы позволить покупателям соединять их разнообразные продукты IBM. Разработка SNA усложнялась требованием ее совместимости с почти каждым сетевым продуктом, уже предлагаемым IBM.

Алгоритмические проблемы. Внедрение локальных сетей требует решения некоторых, но не всех, проблем, рассмотренных в предыдущем подразделе по глобальным сетям. Надежный обмен данными не такая большая проблема, потому что шины обычно очень надежны и быстры. Проблема маршрутизации не встает в шинных сетях, потому что каждое назначение может быть адресовано прямо по сети. В кольцевых сетях все сообщения обычно посылаются в одном направлении вдоль кольца и удаляются либо получателем, либо отправителем, что также делает проблему маршрутизации исчерпанной. В шине нет перегрузки благодаря тому, что каждое сообщение принимается (берется с шины) немедленно после его отправки, но все равно необходимо ограничивать нагрузку от сообщений, ожидающих в узлах выхода на шину. Раз сообщения не сохраняются в промежуточных вершинах, то и не возникает тупика типа сохрани-передай. Нет необходимости в механизмах безопасности помимо той обычной защиты, предлагаемой операционной системой, если компьютерами владеет одна компания, которая доверяет своим сотрудникам.

Использование локальных сетей для распределенного выполнения прикладных программ (набора процессов, распространенных по узлам сети) требует решения следующих проблем распределенного управления, некоторые из которых обсуждаются в части 2.

(1) *Широковещание и синхронизация* (глава 6). Если информация должна быть доступна всем процессам, или все процессы должны ждать выполнения некоторого глобального условия, необходимо иметь схему передачи сообщений, которая каким-либо образом «дозванивается» до всех процессов.
(2) *Выборность* (глава 7). Некоторые задачи должны быть осуществлены точно одним процессом из множества, например, генерирование вывода или инициализация структуры данных. Если, как иногда желательно или необходимо, нет процесса предназначенного для этого заранее, то распределенный алгоритм должен выбрать одни из процессов для выполнения задачи.
(3) *Обнаружение завершения* (глава 8). Не всегда есть возможность для процессов в распределенной системе замечать напрямую, что распределенные вычисления, в которые они вовлечены, завершены. Поэтому обнаружение необходимо для того, чтобы сделать вычисляемые результаты окончательными.

(4) *Распределение ресурсов*. Узел может потребовать доступ к некоторым ресурсам, которые доступны, где-либо в сети, но не знает, где этот ресурс находится. Поддержка таблицы, которая показывает местоположение каждого ресурса не всегда адекватна, потому что число потенциальных ресурсов может быть слишком большим для этого, или ресурсы могут мигрировать от одного узла к другому. В этом случае, запрашивающий узел может опрашивать все или некоторые узлы на предмет доступности ресурса, например, используя широковещательный механизм. Алгоритмы для этой проблемы могут базироваться на волновых механизмах, описанных в главе 6, см., например Баратц и другие [BGS87].

(5) *Взаимное исключение*. Проблема взаимного исключения встает, если процессы могут полагаться на общий ресурс, который может быть использован только одним ресурсом в каждый момент времени. Таким ресурсом может быть принтер или файл, который должен быть перезаписан. Распределенному алгоритму в этом случае необходимо определить, если требуют процессы доступа одновременно, какому из них разрешить использовать ресурс первым. Также удостовериться в том, что следующий процесс начнет использовать ресурс, только после того, как предыдущий процесс закончит его использовать.

(6) *Обнаружение тупиков и их разрешение*. Если процессы должны ждать друг друга (как в случае, если они разделяют ресурсы, и также, если их вычисления полагаются на данные, обеспечиваемые другими процессами), может возникнуть циклическое ожидание, при котором не будет возможно дальнейших вычислений. Эти тупиковые ситуации должны определяться и правильные действия должны предприниматься для того, чтобы перезапустить или продолжить вычисления.

(7) *Распределенная поддержка файлов*. Когда узлы помещают запросы на чтение и запись удаленного файла, эти запросы, могут обрабатываться в произвольном порядке, и отсюда должна быть предусмотрена мера для уверенности в том, что каждый узел наблюдает целостный вид файла или файлов. Обычно это производится временным штампованием запросов, также как и информации в файлах и упорядочивание входящих запросов по их временным отметкам; см., например, [LL86].

1.1.5 Многопроцессорные компьютеры

Многопроцессорный компьютер это вычислительная система, состоящая из нескольких процессоров в маленьком масштабе, обычно внутри одной большой коробки. Этот тип компьютерной системы отличается от локальных сетей по следующему критерию. Его процессоры гомогенны, т.е. они идентичны по аппаратуре. Географический масштаб машины очень маленький, обычно порядка метра или менее. Процессоры предназначены для совместного использования в одном вычислении (либо чтобы повысить скорость, либо для повышения надежности). Если основное назначение многопроцессорного компьютера это повышение скорости вычислений, то он часто называется параллельным

компьютером. Если его основное назначение – повышение надежности, то он часто называется система репликации.

Параллельные компьютеры подразделяются на одно-командные много-поточные по данным (или SIMD) и много-командные много-поточные по данным (или MIMD) машины.

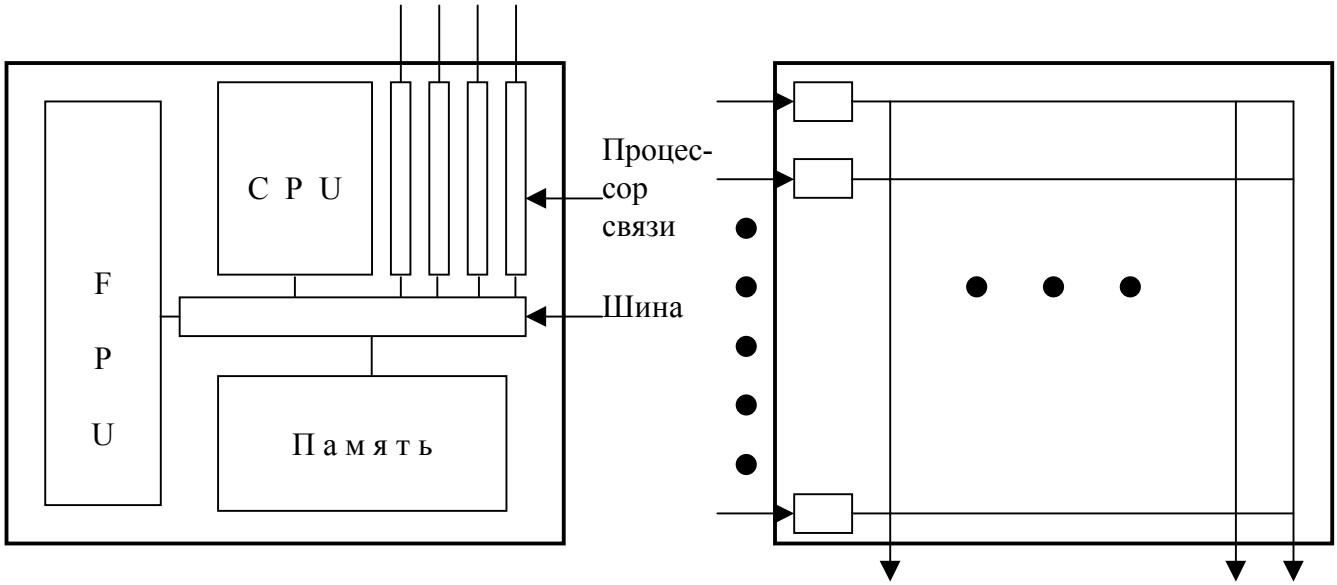


Рис. 1.3 Транспьютер и микросхема маршрутизатора

SIMD машины имеют один интерпретатор инструкций, но команды выполняются большим числом арифметических блоков. Ясно, что эти блоки имеют недостаток автономности, которая требуется в нашем определении распределенных систем, и поэтому SIMD компьютеры не будут рассматриваться в этой книге. MIMD машины состоят из нескольких независимых процессоров и они классифицируются как распределенные системы.

Процессоры обычно оборудуются специальной аппаратурой для коммуникации с другими процессорами. Коммуникация между процессорами может иметь место либо через шину, либо через соединения точка-точка. Если выбрана шинная организация, то архитектура масштабируема только до определенного уровня.

Очень популярным процессором для разработки многопроцессорных компьютеров является транспьютер, разработанный Inmos; см. рис. 1.3. Транспьютер состоит из центрального процессора (CPU), специального блока с плавающей точкой (FPU), локальной памяти, и четырех специальных процессоров. Чипы очень хорошо подходят для построения сетей степени 4 (т.е. каждый узел соединен с четырьмя другими узлами). Inmos также производит специальные чипы для коммуникации, называемые маршрутизаторами. Каждый маршрутизатор может одновременно обрабатывать трафик 32 транспьютерных соединений. Каждое входящее сообщение просматривается на предмет того, по какой связи оно может быть перенаправлено; затем оно направляется по это связи.

Другой пример параллельного компьютера это система Connection Machine CM-5, разработанная Thinking Machines Corporation [LAD92]. Каждый узел машины состоит из быстрого процессора и обрабатывающих блоков, таким образом, предлагая внутренний параллелизм в добавление параллелизму, происходящему благодаря наличию нескольких узлов. Так как каждый узел имеет

потенциальную производительность 128 миллионов операций в секунду, и одна машина может содержать 16384 узлов, полная машина может выполнять свыше 10^{12} операций в секунду. (Максимальная машина из 16384 процессоров занимает комнату 900 $м^2$ и скорее всего очень дорогая.) Узлы CM-5 соединены тремя точка-точка коммуникационными сетями. Сеть данных, с топологией толстого дерева, используется для обмена данными по технологии точка-точка между процессорами. Сеть управления, с технологией бинарного дерева, осуществляет специальные операции, такие как глобальная синхронизация и комбинирование ввода. Диагностическая сеть невидима для программиста и используется для распространения информации о вышедших из строя компонентах.. Компьютер может быть запрограммирован как в режиме SIMD, так и в (синхронном) MIMD режиме.

В параллельном компьютере вычисления поделены на подвычисления, каждое осуществляется одноим из узлов. В репликационной системе каждый узел проводит вычисление целиком, после чего результаты сравниваются для того, чтобы обнаружить и скорректировать ошибки.

Построение многопроцессорных компьютеров требует решения нескольких алгоритмических проблем, некоторые из которых подобны проблемам в компьютерных сетях. Некоторые из этих проблем обсуждаются в этой книге.

(1) *Разработка системы передачи сообщений*. Если многопроцессорный компьютер организован как сеть точка-точка, то должна быть разработана коммуникационная система. Это обладает проблемами подобными тем, которые возникают в разработке компьютерных сетей, таким как управление передачей, маршрутизация, и предотвращение тупиков и перегрузок. Решения этих проблем часто проще, чем в общем случае компьютерных сетей. Проблема маршрутизации, например, очень упрощена регулярностью сетевой топологии (например, кольцо или сетка) и надежностью узлов.

Inmos C104 маршрутизаторы используют очень простой алгоритм маршрутизации, называемый внутренней маршрутизацией, которая обсуждается в подразделе 4.4.2, он не может быть использован в сетях с произвольной топологией. Это поднимает вопрос могут ли использоваться решения для проблем, например, предотвращение тупиков, в комбинации с механизмом маршрутизации (см. проект 5.5).

(2) *Разработка виртуальной разделяемой памяти*. Многие параллельные алгоритмы разработаны для так называемой модели параллельной памяти с произвольным доступом (PRAM), в которой каждый процессор имеет доступ к разделяемой памяти. Архитектуры с памятью, которая разделяется физически, не масштабируются; здесь имеет место жесткий предел числа процессоров, которые могут быть обслужены одним чипом памяти.

Поэтому исследования направлены на архитектуры, которые имеют несколько узлов памяти, подсоединенных к процессорам через интерсеть. Такая интерсеть может быть построена, например, из траспьютеров.

(3) *Балансировка загрузки*. Вычислительная мощь параллельного компьютера эксплуатируется только, если рабочая нагрузка вычислений распределена равномерно по процессорам; концентрация работы на од-

ном узле понижает производительность до производительности одного узла. Если все шаги вычислений могут быть определены во время компиляции, то возможно распределить их статически. Более трудный случай возникает, когда блоки работы создаются динамически во время вычисления; в этом случае требуются сложные методы. Очереди задач процессоров должны регулярно сравниваться, после чего задачи должны мигрировать от одной к другой. Для обзора некоторых методов и алгоритмов для балансировки загрузки см. Гочинский [Gos91, глава 9] или Харгет и Джонсон [HJ90].

(4) *Робастость против необнаруживаемых сбоев* (часть 3). В репликационной системе должен быть механизм для преодоления сбоев в одном или нескольких процессорах. Конечно, компьютерные сети должны также продолжать их функционирование, несмотря на сбои узла, но обычно предполагается, что такой сбой может быть обнаружен другими узлами (см., например, алгоритм сетевого обмена в разделе 4.3). Предположения, при которых репликационные системы должны оставаться правильными, более строгие, т.к. процессор может производить ошибочный ответ, и то же время кооперироваться с другими при помощи протоколов как правильно работающий процессор. Должен быть внедрен механизм голосования, чтобы отфильтровывать результаты процессоров, так, что только правильные ответы передаются во все время, пока большинство процессоров работает правильно.

1.1.6 Взаимодействующие процессы

Разработка сложных программных систем может быть зачастую упрощена организацией программы как набора (последовательных) процессов, каждый с хорошо определенной, простой задачей.

Классический пример для иллюстрации этого упрощения это преобразование записей Конвея. Проблема состоит в том, чтобы читать 80 символьные записи и записывать ту же информацию в 125 символьные записи. После каждой входной записи должен вставляться дополнительный пробел, и каждая пара звездочек («**») должна заменяться на восклицательный знак («!»). Каждая выходная запись должна завершаться символов конца записи (EOR). Преобразование может быть проведено одной программой, но написание этой программы очень сложно. Все функции, т.е. замена «**» на «!», вставка пробелов, и вставка символов EOR, должны осуществляться за один цикл.

Программу лучше структурировать как два взаимодействующих процесса. Первый процесс, скажем p_1, читает входные карты и конвертирует входной поток в поток печатных символов, не разбивая на записи. Второй процесс, скажем *p2*, получает поток символов и вставляет EOR после 125 символов. Структура программы как набор двух процессов обычно предполагается для операционных систем, телефонных переключающих центров, и, как мы увидим в подразделе 1.2.1, для коммуникационных программ в компьютерных сетях.

Набор кооперирующих процессов становится причиной того, что приложение становится *локально* распределенным, но абсолютно возможно выполнять процессы на одном компьютере, в этом случае приложение не является фи-

зически распределенным. Конечно, в этом случае достигнуть физической распределенности легче именно для систем, которые логически распределены. Операционная система компьютерной системы должна управлять конкурентным выполнением процессов и обеспечить средства коммуникации и синхронизации между процессами.

Процессы, которые выполняются на одном компьютере, имеют доступ к одной физической памяти, отсюда – естественно использование этой памяти для коммуникации. Один процесс пишет в определенное место памяти, и другой процесс читает из этого места. Эта модель конкурирующих процессов была использована Дейкстрой [Dij68] и Овицким и Грайсом [OG76]. Проблемы, которые рассматривались в этом контексте, включают следующие.

(1) *Атомичность операций с памятью*. Часто предполагается, что чтение и запись одного слова памяти атомичны, т.е. чтение и запись выполняемые процессом завершается перед тем как другая операция чтения или записи начнется. Если структуры большие, больше чем одно слово обновляется, операции должны быть аккуратно синхронизированы, чтобы избежать чтения частично обновленной структуры. Это может быть осуществлено, например, применением взаимного исключения [Dij68] в структуре: пока один процесс имеет доступ к структуре, ни один другой процесс не может начать чтение или запись. Применение взаимного исключения с использованием разделяемых переменных усложнено из-за возможности нескольких процессов искать поле в этой структуре в это же время.

Условия ожидания, налагаемые доступом со взаимным исключением к разделяемым данным, могут понизить производительность процессов, например, если «быстрый» процесс должен ждать данные, в настоящее время используемые «медленным» процессом. В недавние годы внимание концентрировалось на применении разделяемых переменных, которые являются wait-free, что значит, что процесс может читать или писать данные без ожидания любых других процессов. Чтение и запись могут перекрываться, но только при тщательной проработке алгоритмов чтения и записи, которые должны обеспечить атомичность. Для обзора алгоритмов для wait-free атомичных разделяемых переменных см. Киросис и Кранакис [KK89].

(2) *Проблема производитель-потребитель*. Два процесса, один из которых пишет в разделяемый буфер и другой и которых читает из буфера, должны быть скоординированы, чтобы предупредить первый процесс от записи, когда буфер полон и второй процесс от чтения, когда буфер пуст. Проблема производитель-потребитель возникает, когда решение проблемы преобразования Конвея выработано; p_1 производит промежуточный поток символов, и p_2 потребляет его.

(3) *Сборка мусора*. Приложение, которое запрограммировано с использованием динамических структур данных может производить недоступные ячейки памяти, называемые *мусором*. Формально, приложение должно бы прерваться, когда у системы памяти кончается свободное место, для того чтобы позволить специальной программе, называемой *сборщиком мусора*, идентифицировать и вернуть недоступную память. Дейкстра и другие [DLM78] предложили сборщик мусора на-лету, ко-

торый может работать как отдельный процесс, параллельно с приложением.

Требуется сложное взаимодействие между приложением и сборщиком, т.к. приложение может модифицировать структуры указателей в памяти, в то время как сборщик решает какие ячейки являются недоступными. Алгоритм должен быть тщательно проанализирован, чтобы показать, что модификации не обусловят ошибочный возврат доступным ячеек. Алгоритм для сбора мусора на-лету с упрощенным доказательством правильности был предложен Бен-Ари [BA84].

Решения проблем, перечисленных здесь, демонстрируют, что могут быть решены очень трудные проблемы взаимодействия процессов для процессов, которые сообщаются посредством разделяемой памяти. Однако, решения часто исключительно усложнены и иногда очень незначительное перемешивание шагов различных процессов дает ошибочные результаты для решений, которые кажутся правильными на первый и даже на второй взгляд. Поэтому, операционные системы и языки программирования предлагают примитивы для более структурной организации межпроцессовых коммуникаций.

(1) *Семафоры*. Семафор [Dij68] это неотрицательная переменная, чье значение может быть прочитано и записано за одну атомичную операцию. V операция приращает ее значение, а P операция уменьшает ее значение, когда оно положительно (и подвешивает выполнение процесса на этой операции, пока значение переменной нулевое).

Семафоры – подходящее средство для применения взаимного исключения над разделяемой структурой данных: семафор инициализируется в 1, и доступ к структуре предваряется операцией P и завершается операцией V. Семафоры накладывают большую ответственность на каждый процесс за правильное использование; целостность разделяемых данных нарушается, если процесс манипулирует данными неправильно или не выполняет требуемых P и V операций.

(2) *Мониторы*. Монитор [Hoa74] состоит из структуры данных и набора процедур, которые могут выполняться над этими данными, с помощью их вызова процессами способом, использующим взаимное исключение. Т.к. к данным доступ осуществляется полностью через процедуры, объявленные в мониторе, гарантируется правильное использование данных, если монитор объявлен корректно. Монитор, таким образом, предотвращает не позволенный доступ к данным и синхронизирует доступ различных процессов.

(3) *Каналы*. Канал [Bou83] это механизм, который передает поток данных от одного процесса к другому и синхронизирует два коммутирующих процесса; это заранее запрограммированное решение проблемы производитель-потребитель.

Канал это основной механизм коммуникаций в операционной системе UNIX. Если программа $\mathbf{p_1}$ выполняет процесс p_1 преобразования Конвея и $\mathbf{p_2}$ выполняет p_2 , команда UNIX $\mathbf{p_1} \mid \mathbf{p_2}$ вызывает две программы и соединяет их каналом. Вывод $\mathbf{p_1}$ буферизируется и ста-

новится вводом $\mathbf{p_2}$; $\mathbf{p_1}$ подвешивается, когда буфер полон, и $\mathbf{p_2}$ подвешивается, когда буфер пуст.

(4) *Передача сообщений*. Некоторые языки программирования, такие как OCCAM и ADA, обеспечивают передачу сообщений, как механизм для межпроцессовой коммуникации. Проблемы синхронизации относительно легко решаются с использованием передачи сообщений; т.к. сообщение не может быть получено до его передачи, возникает временное отношение между событиями благодаря обмену сообщениями.

Передача сообщений может быть выполнена с использованием мониторов или каналов, и это естественные средства для систем коммуникации, которые используются в аппаратуре распределенных систем (без разделяемой памяти). В самом деле, языки OCCAM и ADA были разработаны с идеей использования их для физически распределенных приложений.

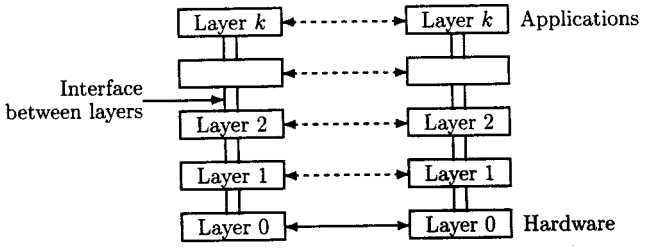


Рис 1.4 Слоеная сетевая архитектура

1.2 Архитектура и Языки

Программное обеспечение для выполнения компьютерных сетей связей очень усложнено. В этом разделе объяснено, как это программное обеспечение обычно структурируется в ациклически зависимых модулях названных уровнями (Подраздел 1.2.1). Мы обсуждаем два стандарта с сетевой архитектурой, а именно, модель МЕЖДУНАРОДНОЙ ОРГАНИЗАЦИИ ПО СТАНДАРТИЗАЦИИ Соединения Открытых систем, стандарт для глобальных сетей, и дополнительного стандарта IEEE для локальных сетей (Подразделы, 1.2.2 и 1.2.3). Также языки, используемые для программирования распределили системы, кратко обсуждены (Подраздел 1.2.4).

1.2.1 Архитектура

Сложность задач, выполняемых подсистемой связи распределенной системы требует, чтобы эта подсистема была разработана высоко структурированным способом. К этому моменту, сети всегда организовываются как совокупность модулей, каждое выполнение очень специфическая функция и основывающаяся на услугах, предлагаемых другими модулями. В сетевых организациях имеется всегда строгая иерархия между этими модулями, потому что каждый модуль исключительно использует услуги, предлагаемые предыдущим модулем. Модули названы уровнями или уровнями в контексте сетевой реализации; см.

1.4 Рисунок. Каждый уровень осуществляет часть функциональных возможностей, требуемых для реализации сети и полагается на уровень только ниже этого. Услуги, предлагаемые i уровнем i + 1 уровню точно описаны в интерфейсе i уровня и i + 1 уровня (кратко, i / (i + 1) интерфейс). При проектировании сети, в первую очередь, нужно определить число уровней и интерфейсов между последующими уровнями.

Функциональные возможности каждого уровня должны быть выполнены распределенным алгоритмом, таким, что алгоритм для i уровня решает "проблему", определенную i / (i + 1) интерфейсом, согласно "предположениям", определенным в (i — l) /i интерфейсе. Например, (i — 1) /i интерфейс может определять, что сообщения транспортируются из узла p к узлу q, но некоторые сообщения могут быть потеряны, в то время как i / (i + 1) интерфейс определяет, что сообщения передаются от p до q надежно. Алгоритмическая проблема для i уровня затем - выполнить надежное прохождение сообщения, используя ненадежное прохождение сообщения, что обычно делается с использованием подтверждения и перепередачи потерянных сообщений (см. Подраздел, 1.3.1 и Главу 3). Решение этой проблемы определяет тип сообщений, обменянных процессами i уровня и значение этих сообщений, т.е., как процессы должны реагировать на эти сообщения. Правила и соглашения, используемые в "сеансе связи" между процессами i уровня упоминаются как *layer-i протокол*. Самый низкий уровень иерархии (уровень 0 на Рисунке 1.4) - всегда аппаратный уровень. Интерфейс 0/1 описывает процедуры, которыми уровень i может передать необработанную информацию через соединяющие провода, и описание уровня непосредственно определяет то, какие типы провода используются, сколько вольт представляют единицу или ноль, и т.д. Важное наблюдение - то, что изменение в реализации уровня 0 (замена проводов другими проводами или спутниковыми подключениями) не требует, чтобы интерфейс 0/1 был изменен. Те же самые условия в более высоких уровнях: интерфейсы уровня служат экраном от реализация уровня для других уровней, и реализация может быть изменена без того, чтобы воздействовать на другие уровни. Под сетевой архитектурой мы понимаем совокупность уровней и сопровождающих описаний всех интерфейсов и протоколов. Поскольку сеть может содержать узлы, произведенные различными изготовителями, программируемые программным обеспечением, написанным различными компаниями, важно, чтобы изделия различных компаний являлись совместимыми. Важность совместимости была признана во всем мире и следовательно стандартные сетевые архитектуры были разработаны. В следующем подразделе два стандарта обсуждаются, что получило "официальное" статус, потому что они приняты влиятельными организациями (МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ, и Институт Электрических и Электронных Инженеров, IEEE). Протокол управления передачей / internet протокол (TCP/IP) - совокупность протоколов, используемых в Internet. TCP/IP - не официальный стандарт, но используется настолько широко, что стал фактическим стандартом. Семейство протоколов TCP/IP (см. Davidson [Dav88] для введения) структурирован согласно уровням OSI модели, обсужденной в следующем подразделе, но протоколы могут использоваться в глобальных сетях также как в локальных сетях.

Более высокие уровни содержат протоколы для электронной почты (простой протокол передачи почты - SMTP), передача файлов (протокол передачи файлов, FTP), и двунаправленная связь для удаленного входа в систему (Telnet).

1.2.2 Ссылочная Модель OSI

МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ установила стандарт для компьютерных изделий(программ) для работы с сетями типа тех, которые используются (главным образом) в глобальных сетях. Их стандарт для сетевой архитектуры назван Соединением открытых систем (OSI) ссылочной моделью, и будет описан кратко в этом подразделе. Потому что стандарт не полностью соответствующий для использования в локальных сетях, дополнительные стандарты IEEE для локальных сетей обсуждены в следующем подразделе. Модель ссылки OSI состоит из семи уровней, а именно физического, связи данных, сети, транспорта, сеанса, представления, и уровней прикладной программы. Ссылочная модель определяет интерфейсы между уровнями и обеспечивает, для каждого уровня, один или большее количество стандартных протоколов (распределенные алгоритмы, чтобы выполнить уровень).

Физический (1) уровень. Цель физического уровня состоит в том, чтобы передать последовательности битов по каналу связи. Поскольку имя уровня предполагает, что эта цель достигнута посредством физического подключения между двумя узлами, типа телефонной линии, волоконно-оптического подключения, или спутникового подключения. Проект уровня непосредственно - вполне вопрос для инженеров - электриков, в то время как интерфейс 1/2 определяет процедуры, которыми следующий уровень вызывает услуги физического уровня. Обслуживание физического уровня не надежно; поток битов может быть попорчен в течение передачи.

Канальный уровень (2). Цель канального уровня состоит в том, чтобы маскировать ненадежность физического уровня, то есть обеспечивать надежную связь с более высокими уровнями. Уровень связи данных только осуществляет надежное подключение между узлами, которые непосредственно связаны физической связью, потому что он сформирован непосредственно над физическим уровнем. (Связь между несмежными узлами выполнена в сетевом уровне.) Чтобы достигнуть цели, уровень делит поток битов на части фиксированной длины, названные *кадрами*. Приемник кадра может проверять(отмечать), был ли кадр получен правильно, проверяя контрольную сумму, которая является некоторой избыточной информацией, добавленной к каждому кадру. Имеется обратная связь от приемника до отправителя, чтобы сообщить отправителю относительно правильно или неправильно полученного кадра; эта обратная связь происходит посредством сообщений подтверждения.

Отправитель пошлет кадр снова, если оказалось, что он получен неправильно или полностью потерян. Общие принципы, объясненные в предыдущем параграфе могут быть усовершенствованы к ряду различных протоколов связи данных. Например, сообщение подтверждения может быть послано для кадров, которые получены (положительные подтверждения) или для кадров, которые отсутствуют из совокупности полученных кадров (отрицательные подтверждения). Окончательная ответственность за правильную передачу всех кадров может быть на отправителе или стороне приемника. Подтверждения могут быть посланы для одиночных кадров или блоков кадров, кадры могут иметь числа последовательности или не иметь, и т.д.

Сетевой уровень (3). Цель сетевого уровня состоит в том, чтобы обеспечить средства связи между всеми парами узлов, не только связанных физиче-

ским каналом. Этот уровень должен выбрать маршруты через сеть, используемую для связи между не-смежными узлами и должен управлять загрузкой движения в каждом узле и канале. Выбор маршрутов обычно основан на информации относительно сетевой топологии, содержащейся в маршрутизации таблиц, сохраненных в каждом узле. Сетевой уровень содержит алгоритмы, чтобы модифицировать таблицы маршрутизации, если топология сети изменилась (вследствие сбоя канала или восстановления). Такой сбой или восстановление обнаруживается канальным уровнем связи. Хотя канальный уровень обеспечивает надежное обслуживание у сетевого уровня, обслуживание, предлагаемое сетевым уровнем не надежно. Сообщения (названные пакетами в этом уровне) посланные от одного узла до другого могут следовать различными путями, вызывая опасность, что одно сообщение настигнет другое. Вследствие сбоев узла сообщения могут быть потеряны (узел может накрыться во время хранения сообщения), и вследствие лишних сообщений переперадач могут даже быть дублированы. Уровень может гарантировать ограниченному пакету срок службы; то есть, существует константа с такая, что каждый пакет или передается в узел адресата в течение с секунд, или теряется.

Транспортный уровень (4). Цель транспортного уровня состоит в том, чтобы маскировать ненадежность, представленную сетевым уровнем, то есть, обеспечивать надежную связь между любыми двумя узлами. Проблема была бы подобна той решенной канальным уровнем, но это еще усложнено возможностью дублирования и переупорядочения сообщений. Это делает невозможным использовать циклические числа последовательности, если ограничение на срок службы пакета не гарантируется сетевым уровнем.

Алгоритмы, используемые для управления передачи в транспортном уровне используют подобные методы для алгоритмов в канальном уровне: числа последовательности, обратная связь через подтверждения, и переперадачи.

Уровень сеанса (5). Цель уровня сеанса состоит в том, чтобы обеспечить средства для поддержания подключений между процессами в различных узлах. Подключение может быть открыто и закрыто и между открытием, и закрытием подключение может использоваться для обмена данных, используя адрес сеанса скорее, чем повторение адреса удаленного процесса с каждым сообщением. Уровень сеанса использует надежную непрерывную связь, предлагаемую транспортным уровнем, но структурирует передаваемые сообщения в сеансы. Сеанс может использоваться для передачи файла или удаленного входа в систему. Уровень сеанса может обеспечивать механизмы для восстановления, если узел терпит крах в течение сеанса и для взаимного исключения, если критические операции не могут выполняться на обоих концах одновременно.

Уровень представления (6). Цель уровня представления состоит в том, чтобы выполнить преобразование данных, где представление информации в одном узле отличается от представления в другом узле или не подходящее для передачи. Ниже этого уровня (то есть, при интерфейсе 5/6) данные находятся в передавабельной и стандартизированной форме, в то время как выше этого уровня (то есть, при интерфейсе 6/7) данные находятся в пользовательско - или компьютерно - специфической форме. Уровень выполняет сжатие данных и декомпрессию, чтобы уменьшить количество данных, переданных через более низкие уровни. Уровень выполняет шифрование данных и расшифровку, чтобы

гарантировать конфиденциальность и целостность в присутствии злонамеренных сторон, которые стремятся получать или разрушать переданные данные.

Уровень прикладной программы (7). Цель уровня прикладной программы состоит в том, чтобы выполнять конкретные требования пользователя типа передачи файла, электронной почты, информационных табло, или виртуальных терминалов. Широкое разнообразие возможных прикладных программ делает невозможным стандартизировать полные функциональные возможности этого уровня, но для некоторых из прикладных программ, перечисленных здесь, стандарты были предложены.

1.2.3 OSI Модель в локальных сетях: IEEE Стандарты

На проект ссылочной модели OSI влияют в большой степени архитектуры существующих глобальных сетей. Технология, используемая в локальных сетях налагает различные программные требования, и из-за этих требований некоторые из уровней могут почти совсем отсутствовать в локальных сетях. Если сетевая организация полагается на общую шину, общедоступную всеми узлам (см. Подраздел 1.1.4), то сетевой уровень почти пуст, потому что каждая пара узлов связана непосредственно через шину. Проект транспортного уровня очень упрощен ограниченным количеством недетерминизма представленного шиной, по сравнению с промежуточной двухточечной сетью. Напротив, канальный уровень усложнен фактом, что к той же самой физической среде обращается потенциально большое количество узлов. В ответе на эти проблемы IEEE одобрил дополнительные стандарты, покрывая только более низкие уровни OSI иерархии, для использования в локальных сетях (или, если быть более точным, во всех сетях, которые являются структурированными шиной скорее, чем двухточечными соединениями). Потому что никакой одиночный стандарт не мог бы быть достаточно общий, чтобы охватить все сети уже широко использующиеся, IEEE одобрил три различных, несовместимых стандарта, а именно МНОЖЕСТВЕННЫЙ ДОСТУП С ОПРОСОМ НЕСУЩЕЙ И РАЗРЕШЕНИЕМ КОНФЛИКТОВ, маркерную шину , и эстафетное кольцо. Канальный уровень заменен двумя подуровнями, а именно управление доступом к среде и подуровни управления логическим соединением.

Физический (1) уровень. Цель физического уровня в стандартах IEEE подобна таковому первоначального стандарта МЕЖДУНАРОДНОЙ ОРГАНИЗАЦИИ ПО СТАНДАРТИЗАЦИИ, а именно передавать последовательности битов. Фактические стандартные описания (тип монтажа и т.д.), однако, радикально различны, вследствие того, что вся связь происходит через общедоступную среду, а не через двухточечные подключения.

Medium-access-control подуровень (2a). Цель этого подуровня состоит в том, чтобы решить конфликты, которые возникают между узлами, которые хотят использовать общедоступную среду связи. Статичный подход раз и навсегда планировал бы интервалы времени, в течение которых каждому узлу позволяют использовать среду. Этот метод теряет много пропускной способности, однако, если только несколько узлов имеют данные, чтобы передавать, и все другие узлы тихи, среда остается в простое в течение времен, планируемых для тихих узлов. В шинах маркера и эстафетных кольцах доступ к среде находится по карусельному принципу: узлы циркулируют привилегию, названную маркером, среди них, и узлу, задерживающему этот маркер, позволяют использовать среду. Если узел, задерживающий маркер, не имеет никаких данных, чтобы передать,

он передает маркер к следующему узлу. В эстафетном кольце циклический порядок, в котором узлы получают их право хода, определен физической топологией подключения (который, действительно, кольцо), в то время как в шине маркера, циклический порядок определен динамически основываясь на порядке адресов узлов. В стандарте МНОЖЕСТВЕННОГО ДОСТУПА С ОПРОСОМ НЕСУЩЕЙ И РАЗРЕШЕНИЕМ КОНФЛИКТОВ узлы наблюдают, когда среда неактивна, и если так, то им позволяют послать. Если два или больше узла запускают посылку (приблизительно) одновременно, имеется проверка на пересечение, которое обнаруживается, что заставляет каждый узел прерывать передачу и пытаться снова в более позднее время.

Logical-link-control подуровень (2b). Цель этого уровня сравнима с целью канального уровня в OSI модели, а именно: управлять обменом данными между узлами. Уровень обеспечивает управление ошибками и управление потоком данных, используя методы, подобные тем использованных в OSI протоколах, а именно числа последовательности и подтверждения. Видящийся с точки зрения более высоких уровней, logical-link-control подуровень появляется подобно сетевому уровню OSI модели. Действительно, связь между любой парой узлов происходит без того, чтобы использовать промежуточные узлы, и может быть обработана непосредственно logical-link-control подуровнем. Отдельный сетевой уровень не следовало бы выполнять в локальных сетях; вместо этого, транспортный уровень сформирован непосредственно на верхней части logical-link-control подуровня.

1.2.4 Поддержка Языка

Реализация одного из программных уровней сети связей или распределенной прикладной программы требует, чтобы распределенный алгоритм, используемый в том уровне или прикладной программе был кодирован на языке программирования. На фактическое кодирование конечно высоко влияет язык и особенно примитивы, которые он предлагает. Так как в этой книге мы концентрируемся на алгоритмах и не на их кодировании как программа, наша базисная модель процессов основана на состояниях процесса и переходах состояния (см. Подраздел 2.1.2), а не на выполнении команд, принимаемых из предписанного набора. Конечно, неизбежно, чтобы там, где мы представили алгоритмы, требовалась некоторая формальная запись; запись программирования, используемая в этой книге обеспечена в Приложении A. В этом подразделе мы описываем некоторые из конструкций, которые можно наблюдать в фактических языках программирования, разработанных для распределенных систем. Мы ограничиваемся здесь кратким описанием этих конструкций; Для большего количества деталей и примеров фактических языков, которые используют различные конструкции, см., например, Bal [Bal90]. Язык для программирования распределенных прикладных программ, должен обеспечить средства, чтобы выразить параллелизм, обрабатывать взаимодействие, и недетерминизм. Параллелизм, конечно, требуется для программирования различных узлов системы таким способом, которым узлы выполнят их часть программы одновременно. Связь между узлами должна также быть поддержана в соответствии с языком программирования. Недетерминизм необходим, потому что узел должен иногда быть способен получить сообщение от различных узлов, или быть способным либо посылать, либо получать сообщение.

Параллелизм. Наиболее соответствующая степень параллелизма в распределенной прикладной программе зависит от отношения(коэффициента) между стоимостью связи и стоимостью вычисления. Меньшая степень параллелизма учитывает более быстрое выполнение, но также и требует большего количества связи, так, если связь дорога, усиление в быстродействии вычисления может быть потеряно в дополнительной стоимости связи. Параллелизм обычно выражается, определением нескольких процессов, где каждый процесс является последовательным объектом с собственным пространством состояния. Язык может или предлагать возможность статического определения совокупности процессов или позволять динамическое создание и завершение процессов. Также возможно выразить параллелизм посредством параллельных инструкций или в функциональном языке программирования. Параллелизм не всегда явен в языке; выделение разделов кода в параллельные процессы может выполняться сложным транслятором.

Связь. Связь между процессами свойственна распределенным системам: если процессы не связываются, каждый процесс функционирует в изоляции от других процессов и должен изучаться в изоляции, а не как часть распределенной системы. Когда процессы сотрудничают в вычислении, связь необходима, если один процесс нуждается в промежуточном результате, произведенном другим процессом. Также, синхронизация необходима, потому что вышеупомянутый процесс должен быть приостановлен, пока результат не доступен. Прохождение сообщения затрагивает, и связь и синхронизацию; общедоступная память затрагивает только связь: дополнительная осторожность должна быть предусмотрена для синхронизации процессов, которые сообщаются с использованием общедоступной памяти. В языках, которые обеспечивают передачу сообщения, доступны операции "посылать" и "получать". Связь происходит выполнением посылающейся операции в одном процессе (следовательно названным процессом отправителя) и получающейся операцией в другом процессе (процесс приемника). Параметры посылающей операции - адрес приемника и дополнительные данные, формирующие содержание сообщения. Эти дополнительные данные становятся доступными приемнику, когда получающая инструкция выполнена, то есть, таким образом осуществляет связь. Получающая операция может быть завершена только после того, как посылающая операция была выполнена, что и осуществляет синхронизацию. В некоторых языках получающая операция не доступна явно; вместо этого, процедура или операция активизируется неявно, когда сообщение получено. Язык может обеспечивать синхронное прохождение сообщения, когда посылающая операция завершена только после выполнения получающей операции.

Другими словами, отправитель блокирован, пока сообщение не было получено, и имеет место двухсторонняя синхронизация между результатами приемника и отправителем. Сообщения могут быть посланы двухточечно, то есть, от одного отправителя на один приемник, или широковещательно, когда то же самое сообщение получено всеми приемниками. Термин мультиприведение также используется, чтобы обратиться к сообщениям, которые посланы совокупности (не обязательно всех) процессов. Несколько более структурированный примитив связи - удаленный вызов процедуры (RPC). Чтобы связываться с процессом b, процедура a обращается к процедуре, представленной в процессе b, посылая параметры процедуры в сообщении; a приостанавливается, пока результат процедуры не будет возвращен в другом сообщении. Вариант для про-

хождения сообщения - использование общедоступной памяти для связи; один процесс пишет значение переменной, и другой процесс читает значение. Синхронизация между процессами тяжелее, чтобы ее достигнуть, потому что чтение переменной может быть использовано прежде, чем переменная была записана. При использовании примитивов синхронизации типа семафоров [Dij68] или мониторов [Hoa78], возможно выполнить передачу сообщения, в среде общедоступных переменных. И наоборот, также возможно выполнить (виртуальную) общедоступную память в передающей сообщения среде, но это очень неэффективно.

Недетерменизм. В многих точках в выполнении процесс может быть способен продолжиться различными способами. Получающая операция часто недетерминирована, потому что это позволяет получение сообщений от различных отправителей. Дополнительные способы выражать недетерменизм основаны на охраняемых командах. Охраняемая команда в наиболее общей форме - список инструкций, каждый предшествованный булевым выражением (его защитником). Процесс может продолжать выполнение с любой из инструкций, для которых соответствующая защита оценивается истиной. Защита может содержать получающую операцию, когда она оценивается истиной, если имеется сообщение, доступное, чтобы быть полученным.

1.3 Распределенные Алгоритмы

Предыдущие разделы дали причины для использования распределенных компьютерных систем и объяснили характер этих систем; потребность программировать эти системы возникает как следствие. Программирование распределенных систем должно быть основано на использовании правильных, гибких, и эффективных алгоритмов. В этом разделе обсуждается, что разработка распределенных алгоритмов - ремесло, совершенно различное по характеру от ремесла, используемого в разработке централизованных алгоритмов. Распределенные и централизованные системы отличаются по ряду существенных отношений, обрабатываемых в Подразделе 1.3.1 и иллюстрируемых в 1.3.2 Подразделе. Распределенное исследование алгоритмов следовательно развилось как независимое поле научного исследования; см. 1.3.3 Подраздел. Эта книга предназначена, чтобы представить читателю это поле исследования. Цели книги и выбора результатов, включенных в книгу установлены в Подразделе 1.3.4.

1.3.1 Распределенный против Централизованных Алгоритмов

Распределенные системы отличаются от централизованных компьютерных систем по трем существенным отношениям, которые мы теперь обсуждаем.

(1) *Недостаток знания глобального состояния*. В централизованных решениях управление алгоритмом может быть сделано основанным на наблюдениях состояния системы. Даже при том, что к всему состоянию обычно нельзя обращаться в одиночной машинной операции, программа может осматривать переменные один за другим, и принимать решение, в конце концов релевантная информация будет расценена. Никакие данные не изменяются между проверкой и решением, и это гарантирует целостность решения. Узлы в распределенной системе имеют доступ только к их собственному состоянию и не к глобальному

состоянию всей системы. Следовательно, не возможно делать решение управления основанным на глобальном состоянии. Это имеет место тот факт, что узел может получать информацию относительно состояния других узлов и базировать решения управления на этой информации. В отличие от централизованных систем, факт, что полученная информация является старой, может стать причиной получения недопустимой информации, потому что состояние другого узла, возможно, изменилось между посылкой информации состояния и решения, основанного на этом. Состояние подсистемы связи (то есть, какие сообщения находятся в транзите в некоторый момент) никогда непосредственно не наблюдается узлами. Эта информация может только быть выведена косвенно, сравнивая информацию относительно сообщений, посланных и полученных узлами. *Недостаток глобального кадра времени*. События, составляющие выполнение централизованного алгоритма полностью упорядочиваются естественным способом их временным появлением; для каждой пары событий, каждое происходит ранее или позже чем другое. Временное отношение, вызванное на событиях, составляющих выполнение распределенного алгоритма - не общее количество; Для некоторых пар событий может иметься причина для решения, что каждое происходит перед другим, но для других пар имеет место, что ни одно из событий не происходит перед другим [Lam78]. Взаимное исключение может быть достигнуто в централизованной системе требующих его, если доступ процесса p к ресурсу начинается позже чем доступ процесса q, то доступ процесса p начался после того, как доступ процесса q закончился. Действительно, все такие события (старт и окончание доступа процессов p и q) полностью упорядочиваются отношением временного предшествования; в распределенной системе они - не упорядочиваются, и та же самая стратегия не достаточна. Процессы p и q могут начать обращаться к ресурсу, в то время как начало одного не предшествует началу другой.

(3) *Недетерменизм*. Централизованная программа может описывать вычисления, поскольку они разворачиваются из некоторого ввода недвусмысленно; имея данную программу и ввод, только одиночное вычисление возможно. Напротив, выполнение распределенной системы обычно не -детерминировано, из-за возможных различий в быстродействии выполнения компонентов системы.

Рассмотрим ситуацию, где процесс сервера может получать запросы из неизвестного числа процессов пользователя. Сервер не может приостановить обработку запросов, пока все запросы не были получены, потому что не известно, сколько сообщений прибудет. Следовательно, каждый запрос должен быть обработан немедленно, и порядок обработки - порядок, в который запросы прибывают. Порядок, в котором клиентура посылает их запросы, может быть известен, но поскольку задержки передачи не известны, запросы могут прибывать в различном порядке.

Комбинация недостатка знания относительно глобального состояния, недостаток глобального кадра времени, и недетерменизм делает проект распределенных алгоритмов запутанным ремеслом, потому что три аспекта вмешиваются несколькими способами. Понятия времени и состояния очень связаны; в централизованных системах понятие времени может быть определено, рассматривая последовательность состояний, принятых системой в течение выполнения. Даже при том, что в распределенной системе глобальное состояние может быть определено, и выполнение может рассматриваться как последовательность гло-

бальных состояний (Определение 2.2), это представление имеет ограниченное использование, так как выполнение может также быть описано другими последовательностями глобальных состояний (Теорема 2.21). Те альтернативные последовательности обычно состоят из различных глобальных состояний; это придает утверждению "система, принимала это или то состояние в течение выполнения " очень сомнительное значение. Недостаток знания относительно глобального состояния мог бы компенсироваться, если было возможно предсказать это глобальное состояние из алгоритма, который выполняется. К сожалению, это не возможно из-за свойственного недетерменизма в выполнении распределенных систем.

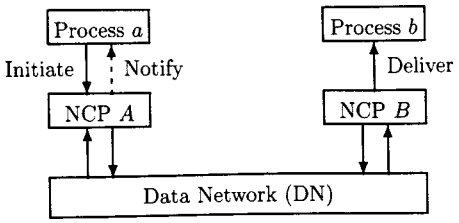


Рис. 1.5 Упрощенная сетевая архитектура

1.3.2 Пример: Связь с одиночным сообщением

Мы проиллюстрируем трудности, налагаемые недостатком знания относительно глобального состояния и недостатка глобального кадра, с помощью примера, обсужденного Beisnes [Bel76l, а именно надежный обмен информацией через ненадежную среду. Рассмотрим два процесса a и b, связанных сетью передачи данных, которая передает сообщения от одного процесса до другого. Сообщение может быть получено в произвольно длительное время после того, как оно послано, оно может также быть потеряно в целом в сети. Надежность связи увеличивается при использовании сетевых процедур управления (NCPs), через который a и b обращаются к сети. Процесс a инициализирует связь, передавая информационный модуль *m* к NCP *A*. Взаимодействие между NCPs (через сеть передачи данных, DN) должно гарантировать, что информация *m* передана в процесс b (NCP *B*), после которого a уведомляется относительно доставки (через NCP A). Структура связи изображена в Рисунке 1.5. Даже если только одиночный информационный модуль должен транспортироваться от a до b, ненадежность сети вынуждает NCP *A* и NCP B вовлекаться в сеанс связи, состоящий из нескольких сообщений. Они поддерживают информацию состояния относительно этого сеанса связи, но потому что число возможных партнеров сеанса связи для каждого процесса большое, то требуется, чтобы информация состояния была отброшена после того, как обмен сообщениями завершен. Инициализация информации состояния называется *открытие*, и ее отбрасывание называется *закрытием* сеанса связи. Заметьте, что после закрытия сеанса связи, NGP находится в точно том же самом состоянии как и перед открытием его; это называется *закрытым* состоянием. Информационный модуль *m*., говорят, *потерян*, если a получил уведомление от b, но модуль фактически не был никогда передан к b. Модуль *m*, говорят, *дублирован* если он был передан дважды. Надежные механизмы связи предотвращают и потери и дублирования. Принимается, что NCPs могут терпеть неудачу, после которой они перезапускаются в за-

крытом состоянии (действительно теряя всю информацию относительно открытого в настоящее время сеанса связи).

Никакая надежная связь не достижима. Как первое наблюдение, может быть показано, что независимо от того, как запутанно NCPs разработаны, не возможно достигнуть полностью надежной связи. Это наблюдение может быть сделано независимо от проекта сети передачи данных или NCPs и только полагается на предположение, что NCP может терять информацию относительно активного сеанса связи. Чтобы видеть это, предположим, что после того, как инициализация связи a, NCP и NCP B запускает разговор(сеанс связи), в течение которого NCP B доставляет м. b после получения сообщения M. из NCP A. Рассмотрите случай где NCP B сбоями и перезапущен в закрытом состоянии после того, как NCP послал сообщение, м. В этой ситуации, ни NCP ни NCP B не может сообщать, был ли м. уже поставлен, когда NCP B потерпел крах; NCP, потому что это не может наблюдать события в NCP B (недостаток знания относительно глобального состояния) и NCP B, потому что это потерпело крах и было перезапущено в закрытом состоянии. Независимо от того, как NCPs продолжают их разговор(сеанс связи), ошибку можно представлять. Если NCP посылает сообщение NCP B, снова и NCP B доставляет сообщение, дублирование может возникать. Если сообщение к дано без поставки, потеря может возникать. Мы теперь оценим несколько возможных проектов NCPs относительно возможности потери или дублирования сообщений. Мы пробуем разрабатывать протоколы таким способом, которым потерь избегают в любом случае.

Сеанс связи с одним сообщением. В самом простом возможном проекте, NCP *A* посылает данные, неизменные через сеть, сообщает об этом a, и закрывается, в одиночном действии после инициализации. NCP *B* всегда доставляет сообщение, которое он получает, к b и закрывается после каждой доставки. Этот протокол представляет потерю всякий раз, когда сеть отказывается доставлять сообщение, но не имеется никакой возможности введения дублирований.

Сеанс связи с двумя сообщениями. Ограниченная защита против потери сообщений предлагается добавлением подтверждений к протоколу. В нормальном сеансе связи, NCP *A* посылает сообщение данных (**данные**, m) и ждет получения сообщения подтверждения (**ack**) из NCP B. Когда это сообщение получено, NCP *A* закрывает сеанс связи. NCP B, после получения сообщения (**данные**, m), доставляет m к b, отвечает сообщением (**ack**), и закрывается. Подводя итоги, можно сказать, что свободный от ошибок сеанс связи состоит из трех событий.

1. NCP *A* send (**данные**, m)
2. NCP *B* receive (**данные**, m), deliver m., send (**ack**), close
3. NCP *A* receive (**ack**), notify, close.

Возможность потери сообщения данных вынуждает NCP *A* посылать (**данные**, m) снова, если подтверждение не получено после некоторого времени. (Из-за недостатка знания относительно глобального состояния, NCP *A* не может наблюдать, были ли (**данные**, m) потеряны, (**ack**) был потерян, или NCP *B* потерпел крах между получением (**данные**, m) и посылкой (**ack**).) К этому моменту, NCP *A* ждет получения подтверждения в течение ограниченного количества времени, и если никакое такое сообщение не получено, таймер переполняется и

происходит таймаут. Может быть легко замечено, что эта опция переперадачи представляет возможность дубликата, а именно, если не первоначальное сообщение данных, а подтверждение было потеряно, как в следующем сценарии:

1. NCP *A* send (**data,** *m)*
2. NCP *B* receive (data, m), deliver *m,* send (ack), close
3. DN (**ack**) is lost
4. NCP *A* timeout, send (**data,** *m)*
5. NCP *B* receive (data, m), deliver *m,* send (ack), close
6. NCP *A* receive (ack), notify, close

Но подтверждения представляют не только возможность дубликатов, они также терпят неудачу, чтобы уберечь против потерь, как следующий сценарий показывает. Процесс а предлагает два информационных модуля, m1 и m2, для передачи.

1. NCP *A* send (**данные,** m1)
2. NCP *B* receive (**данные**, m1), deliver m1, send (**ack**), close
3. NCP *A* timeout, send (**данные,** m1)
4. NCP *B* receive (**данные**, m1), deliver m1, send (**ack**), close
5. NCP *A* receive (**ack**), notify, close
6. NCP *A* send (**данные,** m2*)*
7. DN (**данные,** m2 *)* is lost
8. NCP *A* receive (**ack**) (step 2), notify, close

Сообщение m1 дублировано как в предыдущем сценарии, но первое подтверждение было доставлено медленно, а не потеряно, вызывая потерю более позднего информационного модуля. Медленная доставка не обнаружена из-за недостатка глобального времени. Проблема надежной связи между процессами может быть решена более легко, если принято слабое понятие глобального времени, а именно, существует верхняя граница T задержки передачи любого сообщения, посланного через сеть. Это называется *глобальным* предположением синхронизации, потому что это порождает временное отношение между событиями в различных узлах (а именно, посылка NCP *A* и получение NCP B). Получение сообщений от более ранних сеансов связи может быть предотвращено в этом протоколе закрытием сеанса связи в NCP *A* только через 2T после посылки последнего сообщения.

Сеанс связи с тремя сообщениями. Поскольку протокол с двумя сообщениями теряет или дублирует информационный модуль, когда подтверждение потеряно или отсрочено, можно рассматривать добавление третьего сообщения к сеансу связи для информирования NCP B, что NCP *A* получил подтверждение. Нормальный сеанс связи затем состоит из следующих событий.

1. NCP *A* send (**data**, m)
2. NCP *B* receive (**data**, m), deliver m, send (**ack**)
3. NCP *A* receive (**ack**), notify, send (**close**), close
4. NCP *B* receive (**close**), close

Потеря сообщения (**данные**, m) вызывает таймаут в NCP *A*, когда NCP *A* повторно передает сообщение. Потеря сообщения (**ack**) также вызывает пере-

передачу (**данные**, m), но это не ведет к дублированию, потому что NCP B имеет открытый сеанс связи и распознает сообщение, которое он уже получил.

К сожалению, протокол может все еще терять и дублировать информацию. Потому что NCP B должен быть способен закрыться даже, когда сообщение (**close**) потеряно, NCP B должен повторно передать (**ack**) сообщение, если он не получает никакого сообщения (**close**). NCP *A* отвечает, говоря, что он не имеет никакого сеанса связи (сообщение (**nocon**)), после которого NCP B закрывается. Переперадача (**ack**) может прибывать, однако, в следующем сеансе связи NCP *A* и интерпретироваться как подтверждение в том сеансе связи, вызывая тот факт, что следующий информационный модуль будет потерян, как в следующем сценарии.

1. NCP *A*	send (**data,** m1)
2. NCP *B*	receive (**data**, m1), deliver m1, send (**ack**)
3. NCP *A*	receive (**ack**), notify, send (**close**), close
4. DN	(**close**) is lost
5. NCP *A*	send (**data,** m2)
6. DN	(**data,** m2) is lost
7. NCP *B*	retransmit (**ack**) (step 2)
8. NCP *A*	receive (**ack**), notify, send (**close**), close
9. NCP *B*	receive (**close**), close

Снова проблема возникла, потому что сообщения одного сеанса связи сталкивались с другим сеансом связи. Это может быть исключено выбором пары новых чисел идентификации сеанса связи для каждого нового сеанса связи, одно для NCP *A* и одно для NCP *B*. Выбранные числа включены во все сообщения сеанса связи, и используются, чтобы проверить, что полученное сообщение действительно принадлежит текущему сеансу связи. Нормальный сеанс связи протокола с тремя сообщениями следующий.

1. NCP *A*	send (**data,** *m, x)*
2. NCP *B*	receive *(* **data,** m, *x),* deliver *m,* send (**ack,** *x, y*)
3. NCP *A*	receive (**ack**, *x, y),* notify, send (**close**, *x, y),* close
4. NCP *B*	receive (**close,** *x, y),* close

Эта модификация протокола с тремя сообщениями исключает ошибочный сеанс связи, данный ранее, потому что сообщение, полученное NCP *A* в шаге 8 не принято как подтверждение для сообщения данных, посланного в шаге 5. Однако, NCP *B* не проверяет проверку правильности (**данные**, m, x) перед доставкой m (в шаге 2), что легко ведет к дублированию информации. Если сообщение, посланное в шаге 1 отсрочено и перетранслировано, позже прибывающее сообщение (**данные**, m, x) заставляет NCP B доставлять информацию m снова. Конечно, NCP B должен также проверять правильность сообщений, которые он получает, перед доставкой данных. Мы рассматриваем модификацию сеанса связи с тремя сообщениями, в котором NCP B доставляет данные в шаге 4, а не в шаге 2. Уведомление теперь передается от NCP *A перед* доставкой от NCP B, но потому что NCP B уже получил информацию, это кажется оправданным. Должно быть обеспечено, тем не менее, что NCP B теперь доставит данные в любом случае; в частности когда сообщение (close, x, y) потеряно. NCP B повторяет сообщение (ack, x, y) , на которое NCP *A* отвечает с сообщени-

ем (nocon, x, y) , заставляя NCP B доставить и закрыться, как в следующем сценарии.

1. NCP *A* send (**data**,m,x)
2. NCP *B* receive (**data**, *m, x*), send (**ack**, *x, y*)
3. NCP *A* receive (***ack****,x,y),* notify, send (**close**, *x, y),* close
4. DN (**close**, *x, y)* is lost
5. NCP *B* timeout, retransmit (**ack**, *x, y)*
6. NCP *A* receive (**ack**, *x, y),* reply (**nocon**, *x, y)*
7. NCP *B* receive (**nocon**, x, y), deliver m, close

Оказалось, чтобы избегать потери информации NCP B должен доставлять данные, даже если NCP *A* не подтверждает, что имеет подключение с идентификаторами x и y. Это делает механизм проверки правильности бесполезным для NCP B, ведя к возможности дублирования информации как в следующем сценарии.

1. NCP *A* send (**data,** m, x)
2. NCP *A* timeout, retransmit (**data,** m, *x)*
3. NCP *B* receive (data, m, a:) (sent in step 2), send (ack, x,y1)
4. NCP *A* receive (**ack,** *x, y1),* notify, send *{* **close,** *x, y1),* close
5. NCP *B* receive **(close,** *x, yi*), deliver *m,* close
6. NCP *B* receive (**data,** *m, x)* (sent in step 1), send (**ack,** *x, y2)*
7. NCP *A* receive (**ack,** *x, y2),* reply *{* **nocon,** *x, y2)*
8. NCP *B* receive (**nocon,** *x,y2)* in reply to (**ack,** *x, y2),* deliver *m,* close

Сеанс связи с четырьмя сообщениями. Доставки информации из старых сеансов связи можно избегать при наличии NCPs, взаимно согласующих их числа идентификации сеанса связи прежде, чем любые данные будут поставлены, как в следующем сеансе связи.

1. NCP A send (**data,** *m, x)*
2. NCP B receive (**data,** *m, x),* send (**open,** *x, y)*
3. NCP A receive (**open,** *x, y),* send (**agree,** *x, y)*
4. NCP B receive (**agree**, *x, y),* deliver *m,* send (**ack**, *x, y),* close
5. NCP A receive (**ack**, *x, y),* notify, close

Возможность аварийного отказа NCP B вынуждает обработку ошибок быть такой, что дубликат может все еще происходить, даже, когда никакой NCP фактически не терпит крах. Сообщение об ошибках (nocon, x, y) послано NCP B когда сообщение (agree, x, y) получено, и никакой сеанс связи не открыт. Предположим, что NCP A не получает сообщение (ack, x, y), даже после несколько перепередач {agree, x, y) ; только сообщения (nocon, x, y) получены. Так как возможно, что NCP B потерпел крах прежде, чем он получил (agree, x, y), NCP вынужден запустить новый сеанс связи (посылая {data, m, x}) чтобы предотвратить потерю m! Но также возможно, что NCP B уже доставил m, и сообщение (ack, x, y) было потеряно, тогда появляется дубликат. Возможно изменить протокол таким образом, что NCP A уведомляет и закрывается после получения сообщения {nocon, x, y); это предотвращает дубликаты, но может представлять потерю, которая рассматривается даже менее желательной.

Сеанс связи с пятью сообщениями и сравнение. Beisnes [Bel76] дает протокол с пятью сообщениями, который не теряет информацию, и это представляет дубликаты только, если NCP фактически терпит крах. Следовательно, это - самый лучший возможный протокол, рассматриваемый в свете того наблюдения, что никакая надежная связь не является возможной, ранее в этом подразделе. Из-за чрезмерных накладных расходов (пять сообщений проходят через NCPs, чтобы передать один информационный модуль), должно быть подвергнуто сомнению, должен ли протокол с пятью сообщениями действительно быть предпочтен намного более простому протоколу с двумя сообщениями. Действительно, потому что даже протокол с пятью сообщениями может представлять дубликаты (когда сбоят NCP), уровень процесса должны иметь дело с ними так или иначе. Так получается, что протокол с двумя сообщениями, который может представлять дубликаты, но может быть сделан свободным от потерь, если идентификации сеанса связи добавлены, как мы делали для протокола с тремя сообщениями, можем также использоваться.

1.3.3 Область исследования

Имелось продолжающееся исследование в распределенных алгоритмах в течение последнего двух десятилетий, и значительный прогресс был сделаны особенно в течение 80-х. В предыдущих разделах мы указали на некоторые технические достижения, которые стимулировали исследование в распределенных алгоритмах, а именно, разработка компьютерных сетей (и глобальных и локальных) и многопроцессорные компьютеры. Первоначально исследование было очень нацелено к прикладному применению алгоритмов в глобальных сетях, но в настоящее время разработаны четкие математические модели, позволяющие прикладное применение результатов и методов к более широким классам распределенных сред. Однако, исследование поддерживает плотные связи с достижениями техники в методах связи, потому что результаты в алгоритмах часто чувствительны к изменениям в сетевой модели. Например, доступность дешевых микропроцессоров сделала возможным создать системы с многими идентичными процессорами, которые стимулировали изучение " анонимных сети " (см. Главу 9).

Имеются несколько журналов и ежегодных конференций, которые специализируются на результатах распределенных алгоритмов и распределенных вычислений. Некоторые другие журналы и конференции не специализируются исключительно по этому предмету, но тем не менее содержат много публикаций в этой области. Ежегодный симпозиум по Принципам распределенного вычисления (PoDC) организовывался каждый год начиная с 1982 до времени записи в Северной Америке, и слушания изданы Ассоциацией для Вычисления Машин. Международные Симпозиумы по распределенным алгоритмам (WDAG) были проведены в Оттаве (1985), Амстердаме (1987), Ницце (1989), Bari (1990), Delphi (1991), Хайфе (1992), Lausanne (1993), и Terschelling (1994). С 1989, симпозиумы проводились ежегодно и слушания были изданы Springer-Verlag в сериях Примечания по лекциям по информатике. Ежегодные симпозиумы на теории вычисления (SToC) и основ информатики (FoCS) покрывают все фундаментальные области информатики, и часто несут статьи об распределенном вычислении. Слушания SToC встреч изданы Ассоциацией для Вычисления Машин, и таковых FoCS встреч институтом IEEE. Журнал Параллельного и Распределенного

Вычисления (JPDC) и Распределенного Вычисления издает распределенные алгоритмы регулярно, и делает Письма по обработке информации (IPL).

1.3.4 Иерархическая структура книги

Эта книга была написана со следующими тремя целями в памяти.

(1) Сделать читателя знакомым с методами, которые могут использоваться, чтобы исследовать свойства данного распределенного алгоритма, анализировать и решать проблему которая возникает в контексте распределенных систем, или оценивать качества специфической сетевой модели.

(2) чтобы обеспечить понимание в свойственных возможностях и невозможности нескольких моделей системы. Воздействие доступности глобального кадра времени изучается в Разделе 3.2 и в Главах 11 и 14. Воздействие знания процессами их идентичности изучается в Главе 9. Воздействие требования завершения процесса изучается в Главе 8. Воздействие сбоев процесса изучается в части 3.

(3) Представлять совокупность недавнего современного состояния распределенных алгоритмов, вместе с их проверкой и анализом их сложности.

Где предмет не может обрабатываться в полных подробностях, ссылки к релевантной научной литературе даны. Материал, собранный в книге разделен в три части: Протоколы, Фундаментальные Алгоритмы, и Отказоустойчивость.

Часть 1: Протоколы. Эта часть имеет дело с протоколами связи, используемыми в реализации компьютерных сетей связи и также представляет методы, используемые в более поздних частях. В Главе 2 модель, которая будет использоваться в большинстве более поздних глав, представляется. Модель является, и достаточно общей, чтобы быть подходящей для разработки и проверки алгоритмов и достаточно плотной для доказательства результатов невозможности. Это основано на понятии систем перехода, для которых правила доказательства свойств безопасности и живости могут быть даны легко. Понятие причинной связи как частичного порядока на событиях вычисления представляется, и определены логические часы.

В Главе 3 проблема передачи сообщения между двумя узлами рассматривается. Сначала семейство протоколов для обмена пакетами над одиночной связью обеспечено, и доказательство правильности, по Schoone, дано. Также, протокол по Fletcher и Watson рассматривается, правильность которого полагается на правильное использование таймеров. Обработка этого протокола показывается, как метод проверки может применяться к протоколам, основанным на использовании таймеров. Глава 4 рассматривает проблему маршрутизации в компьютерных сетях. Она сначала представляет некоторую общую теорию относительно маршрутизации и алгоритма Toueg для вычисления маршрутизации таблиц. Также обрабатываемый - Netchange алгоритм Tajibnapis и доказательства правильности для этого алгоритма, данного Lamport. Эта глава заканчивается компактными алгоритмами маршрутизации, включая интервал и префиксную маршрутизацию. Эти алгоритмы названы компактными алгоритмами маршрутизации, потому что они требуют только маленького количества памяти в каждом узле сети. Обсуждение протоколов для компьютерных сетей заканчивается некоторыми стратегиями для ухода от тупиков с промежуточным накоплением в компьютерных сетях с коммутацией пакетов в Главе 5. стратегии основаны

при определении свободных от циклов направленных графов на буферах в узлах сети, и показано, как такой граф может быть создан, используя только скромное количество буферов в каждом узле.

Часть 2: Фундаментальные Алгоритмы. Эта часть представляет ряд алгоритмических "строительных блоков", которые используются как процедуры во многих распределенных прикладных программах, и разрабатывает теорию относительно вычислительной мощности различных сетевых предложений. Глава 6 определяет понятие " волновой алгоритм ", который является обобщенной схемой посещения всех узлов сети. Волновые алгоритмы используются, чтобы распространить информацию через сеть, синхронизировать узлы, или вычислять функцию, которая зависит от распространения информации над всеми узлами. Поскольку это соберется в более поздних главах, много проблем распределенного управления могут быть решены в соответствии с очень общими алгоритмическими схемами, в которых волновой алгоритм используется как компонент. Эта глава также определяет сложность времени распределенных алгоритмов и исследует время и сложность сообщения ряда распределенных алгоритмов поиска в глубину.

Фундаментальная проблема в распределенных системах - *выбор*: Выбор одиночного процесса, который должен запустить различаемую роль в последующем вычислении. Эта проблема изучается в Главе 7. Сначала проблема изучается для кольцевых сетей, где показано, что сложность сообщения проблемы - O (NlogN) сообщений (на кольце N процессоров). Проблема также изучается для общих сетей, и некоторые конструкции показываются, к которым алгоритмы выбора могут быть получены из волновых алгоритмов и алгоритмов обхода. Эта глава также обсуждает алгоритм для конструкции охвата дерева Gallager и другие.

Вторая фундаментальная проблема - *обнаружение завершения*, распознавание (процессами непосредственно) того, что распределенное вычисление завершено. Эта проблема изучается в Главе 8. Нижняя граница сложности решения этой проблемы доказана, и несколько алгоритмов обсуждены подробно. Глава включает некоторые классические алгоритмы (например., Dijkstra, Feijen, и Van Gasteren и Dijkstra и Scholten) и снова конструкция дана для получения алгоритмов для этой проблемы из волновых алгоритмов.

Глава 9 изучает вычислительную мощность систем, где процессы не различаются уникальными идентификаторами. Как показал Angluin, что в этом случае много вычислений не могут быть выполнены детерминированным алгоритмом. Глава представляет вероятностные алгоритмы, и мы исследуем какие проблемы, могут быть решены этими алгоритмами.

Глава 10 объясняет, как процессы системы могут вычислять глобальное "изображение", снимок состояния системы. Такой кадр полезен для определения свойств вычисления, типа того, произошел ли тупик, или как далеко вычисление прогрессировало.

В Главе 11 эффект доступности понятия глобального времени будет изучаться. Несколько степеней синхронизма будут определены, и будет показано, что полностью асинхронные системы могут моделировать полностью синхронные довольно тривиальными алгоритмами. Таким образом замечено, что предположения относительно синхронизма не влияют на совокупность функций, которые являются вычислимыми распределенной системой. Будет впоследствии

показываться, однако, что имеется влияние на сложность связи многих проблем: чем лучше синхронизм сети, тем ниже сложность алгоритмов для этих проблем.

Часть 3: Отказоустойчивость. В практических распределенных системах возможность сбоя в компоненте не может игнорироваться, и следовательно важно изучить, как хорошо алгоритм ведет себя, если компоненты терпят неудачу. Этот предмет будет обрабатываться в последней части книги; короткое введение в предмет дано в Главе 12. Отказоустойчивость асинхронных систем изучается в Главе 13. Результат Fischer и других обеспечен; показывается, что детерминированные асинхронные алгоритмы не могут справляться с даже очень скромным типом сбоя, аварийным отказом одиночного процесса. Будет также показано, что с более слабыми типами неисправностей можно иметь дело, и что некоторые задачи являются разрешимыми несмотря на сбой типа аварийного отказа. Алгоритмы Bracha и Toueg будут обеспечены: оказывается, напротив, рандомизированные асинхронные системы, способны справиться с приемлемо большим количеством сбоев. Таким образом замечено, что имеет место для надежных систем (см. Главу 9), рандомизированные алгоритмы предлагают большее количество возможностей чем детерминированные алгоритмы.

В Главе 14 отказоустойчивость синхронных алгоритмов будет изучаться. Алгоритмы Lamport и другие показали, что детерминированные синхронные алгоритмы могут допустить нетривиальные сбои. Таким образом замечено, что, в отличие от случая надежных систем (см Главу 11), синхронные системы предлагают большее количество возможностей чем асинхронные системы. Даже большее число неисправностей может допускаться, если процессы способны "подписаться" на связь к другим процессам. Следовательно, выполнение синхронизма в ненадежной системе больше усложнено, чем в надежном случае. И последний раздел Главы 14 будет посвящен этой проблеме.

Другой подход к надежности, а именно через само-стабилизацию алгоритмов, сопровождается в Главе 15. Алгоритм стабилизируется, если, независимо от начальной конфигурации, он сходится в конечном счете к предназначенному поведению. Некоторая теория относительно стабилизации алгоритмов будет разработана, и ряд алгоритмов стабилизации будет обеспечен. Эти алгоритмы включают протоколы для нескольких алгоритмов графа типа вычисления дерева поиска в глубину (как в Разделе 6.4) и вычисления таблиц маршрутизации (как в Главе 4). Также, стабилизационные алгоритмы для передачи данных (как в Главе 3) были предложены. Это может означать, что все компьютерные сети могут быть выполнены, с использованием стабилизационых алгоритмов.

Приложения. Приложение A объясняет нотацию, используемую в этой книге, чтобы представить распределенные алгоритмы. Приложение B обеспечивает некоторые фоновые основы из теории графов и терминологии графов. Книга заканчивается списком ссылок и индексом терминов.

2 Модель

В изучении распределенных алгоритмов часто используется несколько различных моделей распределенной обработки информации. Выбор определен-

ной модели обычно зависит того какая проблема распределенных вычислений изучается и какой тип алгоритма или невозможность доказательства представлена. В этой книге, хотя она и покрывает большой диапазон распределенных алгоритмов и теории о них, сделана попытка работать с одной, общей моделью, описанной в этой главе насколько это возможно.

Для того чтобы признать выводы невозможности (доказательство не существования алгоритма для определенной задачи), модель должна быть очень точной. Вывод невозможности это утверждение о *всех* возможных алгоритмах, разрешенных в системе, отсюда модель должна быть достаточно точной, чтобы описать релевантные свойства для всех допускаемых алгоритмов. Кроме того, вычислительная модель это более чем детальное описание конкретной компьютерной системы или языка программирования. Существует множество различных компьютерных систем, и мы хотим, чтобы модель была применима к *классу* схожих систем, имеющих общие основные свойства, которые делают их «распределенными». И наконец, модель должна быть приемлемо компактной, потому что хотелось бы, чтобы в доказательствах учитывались все аспекты модели. Подводя итог, можно сказать, что модель должна описывать точно и кратко релевантные аспекты *класса* компьютерных систем.

Распределенные вычисления обычно понимаются как набор дискретных событий, где каждое событие это атомарное изменение в конфигурации (состояния всей системы). В разделе 2.1 это понятие включено в определение *систем перехода*, приводящих к понятию достижимых конфигураций и конструктивному определению множества исполнений, порождаемых алгоритмом. Что делает систему «распределенной»? То, что на каждый переход влияет, и он в свою очередь оказывает влияние только на часть конфигурации, в основном на локальное состояние одного процесса. (Или на локальные состояния подмножества взаимодействующих процессов.)

Разделы 2.2 и 2.3 рассматривают следствия и свойства модели, описанной в разделе 2.1. Раздел 2.2 имеет дело с вопросом о том, как могут быть доказаны желаемые свойства данного распределенного алгоритма. В разделе 2.3 обсуждается очень важное понятие, а именно: каузальное отношение между событиями в исполнении. Это отношение вызывает отношение эквивалентности, определенное на исполнениях; вычисление это класс эквивалентности, порожденный этим отношением. Определены часы, и представлены логические часы как первый распределенный алгоритм, обсуждаемый в этой книге. И наконец, в разделе 2.4 будут обсуждаться дальнейшие допущения и нотация, не включенные в основную модель.

2.1 Системы перехода и алгоритмы

Система, чьи состояния изменяются дискретными шагами (переходами или событиями) может быть обычно удобно описана с помощью понятия *системы переходов*. В изучении распределенных алгоритмов это применимо к распределенной системе как целиком, так и к индивидуальным процессам, которые сотрудничают в рамках алгоритма. Поэтому системы перехода это очень важное понятие в изучении распределенных алгоритмов и оно определяется в подразделе 2.1.1.

В распределенных системах переходы влияют только на часть конфигурации (системного глобального состояния). Каждая конфигурация сама по себе это кортеж, и каждое состояние связано с некоторыми компонентами только из этого кортежа. Компоненты конфигурации включают состояния каждого индивидуального процесса. Для точного описания конфигураций должны подразделяться различные виды распределенных систем, в зависимости от типа коммуникаций между процессами.

Процессы в распределенной системе сообщаются либо с помощью доступа с *разделяемым переменным* либо при помощи *передачи сообщений*. Мы примем более ограниченный взгляд и рассмотрим только распределенные системы, где процессы сообщаются при помощи обмена сообщениями. Распределенные системы, где сообщение производится посредством разделяемых переменных, будут обсуждаться в главе 15. Читатель, интересующийся сообщением посредством разделяемых переменных, может проконсультироваться в поворотной статье Дейкстры [Dij68] или Овицкий и Грайс [OG76].

Сообщения в распределенных системах могут передаваться либо *синхронно,* либо *асинхронно*. Основной упор в этой книге делается на алгоритмы для систем, где сообщения передаются асинхронно. Во многих случаях синхронная передача сообщений может рассматриваться как специальный случай асинхронной передачи сообщений, как это было продемонстрировано Чаррон-Бост и др. [CBMT92]. Подраздел 2.1.2 описывает модель асинхронной передачи сообщений точно; в подразделе 2.1.3 модель адаптируется к системам, использующим синхронную передачу сообщений. В подразделе 2.1.4 кратко обсуждается справедливость.

2.1.1 Системы переходов

Система переходов состоит из множества всех возможных состояний системы, переходов («ходов»), которые система совершает в этом множестве, и подмножества состояний, в которых системе позволено стартовать. Чтобы избежать беспорядка между состояниями отдельного процесса и состояниями алгоритма целиком («глобальных состояний»), последние теперь будут называться *конфигурациями*.

Определение 2.1 Система переходов есть тройка $S = (C, \rightarrow, I)$, где C это множество конфигураций, $\rightarrow$ это бинарное отношение перехода на C, и I это подмножество C начальных конфигураций.

Отношение перехода это подмножество C x C. Вместо $(\gamma, \delta) \in \rightarrow$ будет использоваться более удобная нотация $\gamma \rightarrow \delta$.

Определение 2.2 Пусть $S = (C, \rightarrow, I)$ это система переходов. Исполнение S это есть максимальная последовательность $E = (\gamma_0, \gamma_1, \gamma_2, \ldots)$, где $\gamma_0 \in I$, и для всех $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$.

Терминальная конфигурация это конфигурация γ, для которой не существует δ такой, что $\gamma \rightarrow \delta$. Нужно помнить, что последовательность $E = (\gamma_0, \gamma_1,$

$\gamma_2,\ldots$) с $\gamma_i \rightarrow \gamma_{i+1}$ для всех i максимальна, если она либо бесконечна, либо заканчивается в терминальной конфигурации.

Определение 2.3 *Конфигурация* δ *достижима из* γ, *нотация* $\gamma \Rightarrow \delta$, *если существует последовательность* $\gamma = \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k = \delta$ *с* $\gamma_i \rightarrow \gamma_{i+1}$ *для всех* $0 \leq i < k$. *Конфигурация* δ *достижима, если она достижима из начального состояния.*

2.1.2 Системы с асинхронной передачей сообщений

Распределенная система состоит из набора *процессов* и *коммуникационной подсистемы*. Каждый процесс является системой переходов сам по себе с той лишь оговоркой, что он может взаимодействовать с коммуникационной подсистемой. Чтобы избежать путаницы между атрибутами распределенной системы как целого и атрибутов индивидуальных процессов, мы используем следующее соглашение. Термины «переход» и «конфигурация» используются для атрибутов системы целиком, и (их эквиваленты) термины «событие» и «состояние» используются для атрибутов процессов. Чтобы взаимодействовать с коммуникационной системой процесс имеет не только обычные события (упоминаемые как *внутренние события*), но также *события отправки* и *события получения*, при которых сообщения воспроизводятся и потребляются. Пусть M будет множеством возможных *сообщений,* и обозначим набор мультимножеств с элементами из M через $\mathbf{M}(M)$.

Определение 2.4 Локальный алгоритм процесса есть пятерка $(Z, I, \perp^i, \perp^s, \perp^r)$, где Z это множество состояний, I это подмножество Z начальных состояний, $\perp^i$ это отношение на $Z \times Z$, и $\perp^s$ и $\perp^r$ это отношения на $Z \times M \times Z$. Бинарное отношение $\perp$ на Z определяется как

$$c \perp d \Leftrightarrow (c, d) \in \perp^i \vee \exists m \in M((c, m, d) \in \perp^s \cup \perp^r).$$

Отношения $\perp^i$, $\perp^s$, $\perp^r$ соответствуют переходам состояния, соотносящихся с внутренними сообщениями, сообщениями отправки и сообщениями получения, соответственно. Впоследствии мы будем обозначать процессы через p, q, r, p_1, p_2 и т.д., и обозначать множество процессов системы **P**. Определение 2.4 служит как теоретическая модель для процессов; конечно, алгоритмы в этой книге не описываются только перечислением их состояний и событий, но также средствами удобного псевдокода (см. приложение А). Исполнения процесса есть исполнения системы переходов $(Z, \perp, I)$. Нас, однако, будут интересовать исполнения системы целиком, и в таком исполнении исполнения процессов координируются через коммуникационную систему. Чтобы описать координацию, мы определим распределенную систему как систему переходов, где множество конфигураций, отношение перехода, и начальные состояния строятся из соответствующих компонентов процессов.

Определение 2.5 *Распределенный алгоритм для набора* $\mathbf{P} = \{p_1, \ldots, p_N\}$ *процессов это набор локальных алгоритмов, одного для каждого процесса в* **P**.

Поведение распределенного алгоритма описывается системой переходов, как это объясняется далее. Конфигурация состоит из состояния каждого процесса и набора сообщений в процессе передачи; переходы это события процессов, которые влияют не только на состояние процесса, но также оказывают влияние (или подвергаются таковому) на набор сообщений; начальные конфигурации это конфигурации, где каждый процесс находится в начальном состоянии и набор сообщений пуст.

Определение 2.6 *Система переходов, порожденная распределенным алгоритмом для процессов $p_1, \ldots, p_N$ при асинхронной коммуникации, (где локальный алгоритм для процесса p_i это есть $(Z, I, \perp^i, \perp^s, \perp^r)$), это $S = (C, \rightarrow, I)$, где*

(1) $C = \{(c_{P1}, \ldots, c_{PN}, M) : (\forall p \in \mathbf{P} : c_p \in Z_p)$ *и* $M \in \mathbf{M}(M)\}$.

(2) $\rightarrow = (\cup_{p \in \mathbf{P}} \rightarrow_p)$, *где* $\rightarrow_p$ *это переходы соответствующие изменениям состояния процесса p;* $\rightarrow_{Pi}$ *это множество пар*

$(c_{P1}, \ldots, c_{Pi}, \ldots, c_{PN}, M_1), (c_{P1}, \ldots, c'_{Pi}, \ldots, c_{PN}, M_2),$

для которых выполняется одно из следующих трех условий:

- $(c_{Pi}, c'_{Pi}) \in \perp^i_{Pi}$ *и* $M_1 = M_2$;
- *для некоторого* $m \in M$, $(c_{Pi}, m, c'_{Pi}) \in \perp^s_{Pi}$ *и* $M_2 = M_1 \cup \{m\}$;
- *для некоторого* $m \in M$, $(c_{Pi}, m, c'_{Pi}) \in \perp^r_{Pi}$ *и* $M_1 = M_2 \cup \{m\}$.

(3) $I = \{(c_{P1}, \ldots, c_{PN}, M) : (\forall p \in \mathbf{P} : c_p \in I_p) \wedge M = \varnothing\}$.

Исполнение распределенного алгоритма это исполнение его, породившее систему переходов. События исполнения выполняются явно с помощью следующей нотации. Пары $(c, d) \in \perp^i_p$ называются (возможными) *внутренними событиями* процесса p, и тройки в $\perp^s_p$ и $\perp^r_p$ называются событиями *отправки* и событиями *получения* процесса.

- Внутреннее событие e заданное как $e = (c, d)$ процесса p называется *применимым* в конфигурации $\gamma = (c_{P1}, \ldots, c_P, \ldots, c_{PN}, M)$, если $c_p = c$. В этом случае, $e(\gamma)$ определяется как конфигурация $(c_{P1}, \ldots, d, \ldots, c_{PN}, M)$.
- Событие отправки e, заданное как $e = (c, m, d)$ процесса p называется *применимым* в конфигурации $\gamma = (c_{P1}, \ldots, c_P, \ldots, c_{PN}, M)$, если $c_p = c$. В этом случае, $e(\gamma)$ определяется как конфигурация $(c_{P1}, \ldots, d, \ldots, c_{PN}, M \cup \{m\})$.
- Событие получения e, заданное как $e = (c, m, d)$ процесса p называется *применимым* в конфигурации $\gamma = (c_{P1}, \ldots, c_P, \ldots, c_{PN}, M)$, если $c_p = c$ и $m \in M$. В этом случае, $e(\gamma)$ определяется как конфигурация $(c_{P1}, \ldots, d, \ldots, c_{PN}, M \setminus \{m\})$.

Предполагается, что для каждого сообщения существует уникальный процесс, который может получить сообщение. Этот процесс называется *назначением* сообщения.

2.1.3 Системы с синхронной передачей сообщений

Говорят, что передача сообщений синхронная, если событие отправки и соответствующее событие получения скоординированы так, чтобы сформировать отдельный переход системы. То есть, процессу не разрешается посылать сообщение, если назначение сообщения не готово принять сообщение. Следовательно, переходы системы делятся на два типа: одни соответствуют изменениям внутренних состояний, другие соответствуют скомбинированным коммуникационным событиям двух процессов.

Определение 2.7 *Система переходов, порожденная распределенным алгоритмом для процессов* $p_1, \ldots, p_N$ *при синхронной коммуникации, это* $S = (C, \rightarrow, I)$, *где*

(1) $C = \{(c_{P1}, \ldots, c_{PN}) : \forall p \in \mathbf{P} : c_p \in Z_p\}$.

(2) $\rightarrow = (\cup_{p\in \mathbf{P}} \rightarrow_p) \cup (\cup_{p,q\in \mathbf{P}:p\neq q} \rightarrow_{pq})$, *где*
- $\rightarrow_{Pi}$ *это множество пар*

$$(c_{P1}, \ldots, c_{Pi}, \ldots, c_{PN}), (c_{P1}, \ldots, c'_{Pi}, \ldots, c_{PN}),$$

для которых $(c_{Pi}, c'_{Pi}) \in \perp^i_{Pi}$;
- $\rightarrow_{PiPj}$ *это множество пар*

$$(\ldots, c_{Pi}, \ldots, c_{Pj}, \ldots), (\ldots, c'_{Pi}, \ldots, c'_{Pj}, \ldots),$$

для которых существует сообщение $m \in M$ *такое, что*

$$(c_{Pi}, m, c'_{Pi}) \in \perp^s_{Pi} \text{ и } (c_{Pj}, m, c'_{Pj}) \in \perp^r_{Pj}.$$

(3) $I = \{(c_{P1}, \ldots, c_{PN}) : (\forall p \in \mathbf{P} : c_p \in I_p)\}$.

Некоторые распределенные системы допускают гибридные формы коммуникации; процессы в них имеют коммуникационные примитивы для передачи сообщений как в синхронном так и в асинхронном стиле. Имея две модели, определенные выше, нетрудно разработать формальную модель для этого типа распределенных систем. Конфигурации такой системы включают состояния процессов и набор сообщений в процессе передачи (а именно, асинхронных сообщений). Переходы включают все типы переходов представленных в определениях 2.6 и 2.7.

Синхронизм и его влияние на алгоритмы. Уже было замечено, что во многих случаях синхронная передача сообщений может рассматриваться как специальный случай асинхронной передачи сообщений. Набор исполнений ограничен в случае синхронной передачи сообщений исполнениями, где за каждым событием отправки немедленно следует соответствующее событие приема [CBMT92]. Мы поэтому рассматриваем асинхронную передачу сообщений как

более общую модель, и будем разрабатывать алгоритмы в основном для этого общего случая.

Однако, нужно быть внимательным, когда алгоритм, разработанный для асинхронной передачи сообщений исполняется в системе с синхронной передачей сообщений. Пониженный недетерминизм в коммуникационной системе должен быть сбалансирован повышенным недетерминизмом в процессах, в противном случае результатом всего этого может стать тупик.

Мы проиллюстрируем это элементарным примером, в котором два процесса посылают друг другу некоторую информацию. В асинхронном случае, каждый процесс может сначала послать сообщение и впоследствии получает сообщение от другого процесса. Сообщения временно накапливаются в коммуникационной подсистеме между их отправкой и посылкой. В синхронном случае, такого накапливания невозможно, и если оба процесса должны послать их собственные сообщения перед тем как они могут получить сообщение, то никакой передачи вообще не будет. В синхронном случае, один из процессов должен получить сообщение перед тем как другой процесс отправит свое собственное сообщение. Нет нужды говорить, что, если оба процесса должны получить сообщение перед отправкой их собственных сообщений, опять же не будет никакой передачи.

Обмен двумя сообщениями будет иметь место в синхронном случае, только если одно из двух нижеследующих условий выполняется.

(1) Заранее определено, какой из двух процессов будет отправлять первым, и какой процесс будет первым получать. Во многих случаях невозможно сделать такой выбор заранее, потому что это потребует выполнения различных локальных алгоритмов в процессах.
(2) Процессы имеют право недетерминированного выбора либо отправлять сначала, потом принимать, либо получать сначала, потом – посылать. В каждом исполнении один из возможных порядков исполнения будет выбран для каждого процесса, т.е. симметрия нарушается коммуникационной системой.

Когда мы представляем алгоритм для асинхронной передачи сообщений и утверждаем, что алгоритм может также использоваться при синхронной передаче сообщений, добавление этого недетерминизма, который всегда возможен, предполагается неявно.

2.1.4 Справедливость

В некоторых случаях необходимо ограничить поведение системы так называемыми справедливыми исполнениями. Условия справедливости вводят исполнения, где события всегда (или бесконечно часто) применимы, но никогда не встречаются как переход (потому что исполнение продолжается с помощью других применимых событий).

Определение 2.8 *Исполнение справедливо в слабом смысле, если нет события применимого в бесконечно многих последовательных конфигурациях без*

появления в исполнении. Исполнение справедливо в сильном смысле, если нет события применимого в бесконечно многих конфигурациях без появления в исполнении.

Возможно включить условия справедливости в формальную модель явно, как это сделано Манна и Пнули [MP88]. Большинство алгоритмов, с которыми мы имеем дело в этой книге, не полагаются на эти условия; поэтому мы решили не включать их в модель, а устанавливать эти условия явно, когда они используются для конкретного алгоритма или проблемы. Также, существует спор, приемлемо ли включать предположение справедливости в модели распределенных систем. Было выдвинуто утверждение, что предположение справедливости не должны производиться, более того алгоритмы не должны разрабатываться с учетом этих предположений. Дискуссия по некоторым запутанным вопросам, относящимся к предположению справедливости может быть найдена в [Fra86].

2.2 Доказательство свойств систем перехода

Рассматривая распределенный алгоритм для некоторой проблемы, необходимо продемонстрировать, что алгоритм есть корректное решение этой проблемы. Проблема указывает, какие свойства требуемый алгоритм должен иметь; должно быть показано, что решение обладает этими свойства. Вопрос проверки распределенных алгоритмов получил значительное внимание и есть большое количество статей, обсуждающих формальные методы проверки; см. [CM88, Fra86, Kel76, MP88]. В этом разделе обсуждаются некоторые простые, но часто используемые методы для демонстрации правильности распределенных алгоритмов. Эти методы полагаются только на определение системы переходов.

Многие из требуемых свойств распределенных алгоритмов попадают в один из двух типов: требования безопасности и требования живости. Требования безопасности накладывают ограничение, что определенное свойство должно выполняться для каждого исполнения системы в *каждой* конфигурации, достигаемой в этом исполнении. Требования живости определяют, что определенное свойство должно выполняться для каждого исполнения системы в *некоторых* конфигурациях, достигаемых в этом исполнении. Эти требования могут также встречаться в ослабленной форме, например, они могут удовлетворяться с некоторой фиксированной вероятностью над множеством возможных исполнений. Другие требования к алгоритмам могут включать ограничения, которые основываются только на использовании некоторого данного знания (см. подраздел 2.4.4), что они гибки по отношен ию к нарушениям в некоторых процессах (см. часть 3), что процессы равны (см. главу 9), и т.д.

Методы проверки, описанные в этом разделе, базируются на истинности *утверждений* в конфигурациях, достигаемых в исполнении. Такие методы называются методами проверки утверждений. Утверждение это унарное отношение на множестве конфигураций, то есть, предикат, который принимает значение истина на одном подмножестве конфигураций и ложь – на другом.

2.2.1 Свойства безопасности

Свойство безопасности алгоритма это свойство в форме «Утверждение *P* истина в каждой конфигурации каждого исполнения алгоритма». Неформально это формулируется как «Утверждение *P* всегда истина». Основной метод для того, чтобы показать, что утверждение *P* всегда истина, это продемонстрировать, что *P инвариант* согласно следующим определениям. Нотация $P(\gamma)$, где γ это конфигурация, есть булево выражение, чье значение истина, если *P* выполняется в γ, и ложь в противном случае.

Определения зависят от данной системы переходов $S = (C, \rightarrow, I)$. Далее, мы будем писать $\{P\} \rightarrow \{Q\}$, чтобы обозначить, что для каждого перехода $\gamma \rightarrow \delta$ (системы *S*), если $P(\gamma)$ то $Q(\delta)$. Таким образом $\{P\} \rightarrow \{Q\}$ означает, что, если *P* выполняется перед любым переходом, то *Q* выполняется после этого перехода.

Определение 2.9 *Утверждение P инвариант системы S, если*

(1) *для всех* $\gamma \in I$, *и*
(2) $\{P\} \rightarrow \{P\}$.

Определение говорит, что инвариант выполняется в каждой начальной конфигурации, и сохраняется при каждом переходе. Из этого следует, что он сохраняется в каждой достигаемой конфигурации, как и формулируется в следующем теореме.

Теорема 2.10 *Если P это инвариант системы S, то P выполняется для каждой конфигурации каждого исполнения системы S.*

Доказательство. Пусть $E = (\gamma_0, \gamma_1, \gamma_2, ...)$ исполнение системы S. Будет показано по индукции, что $P(\gamma_i)$ выполняется для каждого i. Во-первых, $P(\gamma_0)$ выполняется, потому что $\gamma_0 \in I$ и по первому предложению определения 2.9. Во-вторых, предположим $P(\gamma_i)$ выполняется и $\gamma_i \rightarrow \gamma_{i+1}$ есть переход, который встречается в E. По второму предложению определения 2.9 $P(\gamma_{i+1})$ выполняется, что и завершает доказательство.

И наоборот, утверждение, которое выполняется в каждой конфигурации каждого исполнения, есть инвариант (см. упражнение 2.2). Отсюда не каждое свойство безопасности может быть доказано применением теоремы 2.10. В этом случае, однако, каждое утверждение, которое всегда истинно, включено в инвариант; отсюда может быть показано, применением следующей теоремы, что утверждение всегда истинно. (Нужно помнить, однако, что часто очень трудно найти подходящий инвариант Q, к которому можно применить теорему.)

Теорема 2.11 *Пусть Q будет инвариантом системы S и положим* $Q \Rightarrow P$ *(для каждого* $\gamma \in C$*). Тогда P выполняется в каждой конфигурации каждого исполнения системы S.*

Доказательство. По теореме 2.10, Q выполняется в каждой конфигурации, и так как Q включает P, то P выполняется в каждой конфигурации также.

Иногда полезно доказать сначала слабый инвариант, и впоследствии использовать его для доказательства более сильного инварианта. Как можно сделать инвариант более сильным демонстрируется в следующем определении и теореме.

Определение 2.12 *Пусть S будет системой переходов и P, Q будут утверждениями. P называется Q-производным, если*

(1) *для всех* $\gamma \in I$, $Q(\gamma) \Rightarrow P(\gamma)$; *и*
(2) $\{Q \wedge P\} \rightarrow \{Q \Rightarrow P\}$.

Теорема 2.13 *Если Q есть инвариант и P – Q-производное, то* $Q \wedge P$ *есть инвариант.*

Доказательство. Согласно определению 2.9, должно быть показано, что

(1) для всех $\gamma \in I$, $Q(\gamma) \wedge P(\gamma)$; и
(2) $\{Q \wedge P\} \rightarrow \{Q \wedge P\}$.

Т.к. Q это инвариант, $Q(\gamma)$ выполняется для всех $\gamma \in I$, и т.к. для всех $\gamma \in I$, $Q(\gamma) \Rightarrow P(\gamma)$, $P(\gamma)$ выполняется для всех $\gamma \in I$. Следовательно, $Q(\gamma) \wedge P(\gamma)$ выполняется для всех $\gamma \in I$.

Предположим $\gamma \rightarrow \delta$ и $Q(\gamma) \wedge P(\gamma)$. Т.к. Q это инвариант, $Q(\delta)$ выполняется, и т.к. $\{Q \wedge P\} \rightarrow \{Q \Rightarrow P\}$, $Q(\delta) \Rightarrow P(\delta)$, откуда $P(\delta)$ вытекает. Следовательно, $Q(\delta) \wedge P(\delta)$ выполняется.

Примеры доказательства безопасности, основывающиеся на материале данного раздела, представлены в разделе 3.1.

2.2.2 Свойства живости

Свойство живости алгоритма это свойство в форме «Утверждение P истина в некоторой конфигурации каждого исполнения алгоритма». Неформально это формулируется как «Утверждение P в конечном счете истина». Основные методы, используемые, чтобы показать, что P в конце концов истина – это *нормирующие функции* и *беступиковость* (*или правильное завершение*). Более простой метод может быть использован для алгоритмов, в которых разрешаются только исполнения с фиксированной, конечной длиной.

Пусть S будет системой переходов и P – предикат. Определим **term** как предикат, который истина во всех терминальных конфигурациях и ложь во всех нетерминальных конфигурациях. Мы сначала предположим ситуации, где исполнение достигает терминальной конфигурации. Обычно нежелательно, чтобы такая конфигурация достигалась, в то время, как «цель» P не была достигнута. Говорят, что в этом случае имеет место *тупик*. С другой стороны, завершение позволено, если цель была достигнута, в этом случае говорят о *правильном завершении*.

Определение 2.14 *Система S завершается правильно (или без тупиков), если предикат (term ⇒ P) всегда истинен в системе S.*

Нормирующие функции полагаются га математическое понятие *обоснованных множеств*. Это множество с порядком <, где нет бесконечных убывающих последовательностей.

Определение 2.15 *Частичный порядок (W, <) является обоснованным, если в нем нет бесконечной убывающей последовательности*

$$w_1 > w_2 > w_3 \ldots .$$

Примеры обоснованных множеств, которые будут использоваться в этой книге – это натуральные числа с обычным порядком, и n-кортежи натуральных чисел с лексикографическим порядком (см. раздел 4.3). Свойство, что обоснованное множество не имеет бесконечной убывающей последовательности, может использоваться, чтобы показать, что утверждение P в конечном счете истина. К этому моменту должно быть показано, что существует функция f из *C* в обоснованное множество W такая, что в каждом переходе значение f убывает или P становится истиной.

Определение 2.16 *Пусть даны система переходов S и утверждение P. Функция f из C в обоснованное множество W называется нормирующей функцией (по отношению к P), если для каждого перехода $\gamma \rightarrow \delta$, $f(\gamma) > f(\delta)$ или $P(\delta)$.*

Теорема 2.17 *Пусть даны система переходов S и утверждение P. Если S завершается правильно и нормирующая функция f (w.r.t P) существует, то P – истина в некоторой конфигурации каждого исполнения системы S.*

Доказательство. Пусть E = (γ_0, γ_1, γ_3, ...) – исполнение системы S. Если E конечно, его последняя конфигурация является терминальной, и т.к. **term** ⇒ P всегда истина в системе S, то P выполняется в этой конфигурации. Если E бесконечно, пусть E’ будет самым длинным префиксом E, который не содержит конфигураций, в которых P истина, и пусть s будет последовательностью (f(γ_0), f(γ_1), ...) для всех конфигураций γ_i, которые появляются в E’. В зависимости от выбора E’ и свойства f, s может быть убывающей последовательностью, и отсюда, по обоснованности W, s конечна. Это подразумевает также, что E’ – конечный префикс (γ_0, γ_1, ..., γ_k) исполнения E. В зависимости от выбора E’, P(γ_{k+1}) выполняется.

Если приняты свойства справедливости, то можно заключить из более слабых посылок (чем в теореме 2.17), что P в конце концов станет истиной. Значение нормирующей функции не должно уменьшаться при *каждом* переходе. Предположение справедливости может быть использовано, чтобы показать, что бесконечные исполнения содержат переходы определенного типа бесконечно часто. Затем будет достаточно показать, что f никогда не увеличивается, а уменьшается с каждым переходом этого типа.

В некоторых случаях мы будем использовать следующий результат, который есть специальный случай теоремы 2.17

Теорема 2.18 *Если S завершается правильно и есть число K такое, что каждое исполнение содержит по крайней мере K переходов, то P истина в некоторой конфигурации каждого исполнения.*

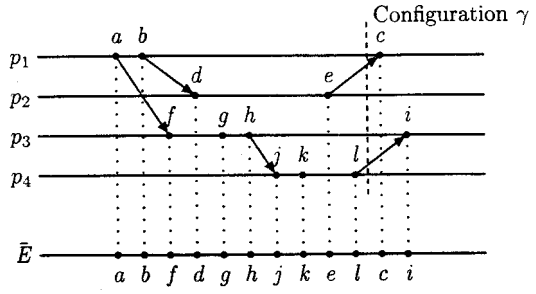


Рис. 2.1 Пример пространственно-временной диаграммы

2.3 Каузальный порядок событий и логические часы

Взгляд на исполнения как последовательности переходов естественным образом порождает понятие времени в исполнениях. Говорят, что переход a появляется раньше перехода b, если a встречается в последовательности перед b. Для исполнения $E = (\gamma_0, \gamma_1, ...)$ определим ассоциированную последовательность событий $E'=(e_0, e_1, ...)$, где e_i – это событие, при котором конфигурация изменяется из γ_i в γ_{i+1}. Заметьте, что каждое исполнение определяет *уникальную* последовательность событий этим путем. Исполнение может быть визуализировано в *пространственно-временной* диаграмме, рисунок 2.1, которой, представляет пример. В такой диаграмме, горизонтальная линия нарисована для каждого процесса, и каждое событие нарисовано точкой на линии процесса, где оно имеет место. Если сообщение m послано при событии s и получено при событии r, стрелка рисуется от s к r. Говорят, что события s и r *соответственные* в этом случае.

Как мы увидим в подразделе 2.3.1, события распределенного исполнения могут иногда быть взаимно обменены без воздействия на последующие конфигурации исполнения. Поэтому понятие времени как абсолютного порядка на событиях исполнения не приемлемо для распределенных исполнений, и вместо этого представляется понятие каузальной зависимости. Эквивалентность исполнений при переупорядочивании событий изучается в подразделе 2.3.2. Мы обсуждаем в подразделе 2.3.3 как могут быть определены часы для измерения каузальной зависимости (а не времени), и представляем логические часы Лампорта, важный пример таких часов.

2.3.1 Независимость и зависимость событий

Уже было замечено, что переходы распределенной системы влияют, и подвержены влиянию, только на часть конфигураций. Это ведет к тому наблюдению, что два последовательных события, влияя на разделенные части конфигурации, независимы и могут также появляться в обратном порядке. Для систем с асинхронной передачей сообщений, это выражается в следующей теореме.

Теорема 2.19 *Пусть γ будет конфигурацией распределенной системы (с асинхронной передачей сообщений) и пусть e_p и e_q будут событиями различных процессов p и q, применимых в γ. Тогда e_p применимо в $e_q(\gamma)$, e_q применимо в $e_p(\gamma)$, и $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$.*

Доказательство. Чтобы избежать анализа случаев, которые есть посылка, получение, или внутренние события, мы представим каждое событие однородной нотацией (c, x, y, d). Здесь c и d обозначают состояние процесса до и после события, x – набор сообщений, полученных во время события, и y – набор сообщений, посланных во течение события. Таким образом, внутренне событие (c, d) обозначается как (c,∅,∅,d), событие отправки (c, m, d) обозначается как (c, ∅, {m}, d), и событие приема (c, m, d) – (c, {m}, ∅, d). В этой нотации, событие e = (c, x, y, d) процесса p применимо в конфигурации $\gamma = (C_{p1}, ..., C_p, ..., C_{pN}, M)$, если $c_p = c$ и $x \subseteq M$. В этом случае

$$e(\gamma) = (C_{p1}, ..., d, ..., (M \setminus x) \cup y).$$

Теперь предположим $e_p = (b_p, x_p, y_p, d_p)$ и $e_q = (b_q, x_q, y_q, d_q)$ применимы в

$$\gamma = (..., c_p, ..., c_q, ..., M),$$

то есть $c_p = b_p$, $c_q = b_q$, $x_p \subseteq M$, и $x_q \subseteq M$. Важное наблюдение состоит в том, что x_p и x_q разделены, сообщение в x_p (если есть такое) имеет назначением p, в то время как сообщение в x_q (если есть такое) имеет назначением q.

Запишем $\gamma_p = e_p(\gamma)$, и запомним что

$$\gamma_p = (..., d_p, ..., c_q, ..., (M \setminus x_p) \cup y_p).$$

Так как $x_q \subseteq M$ и $x_q \cap x_p = \varnothing$, следует, что $x_q \subseteq (M \setminus x_p \cup y_p)$, и отсюда e_q применимо в γ_p. Запишем $\gamma_{pq} = e_q(\gamma_p)$, и запомним, что

$$\gamma_{pq} = (..., d_p, ..., d_q, ..., ((M \setminus x_p \cup y_p) \setminus x_q) \cup y_q).$$

С помощью симметричного аргумента может быть показано, что e_p применимо в $\gamma_q = e_q(\gamma)$. Запишем $\gamma_{qp} = e_p(\gamma_q)$, и запомним, что

$$\gamma_{qp} = (..., d_p, ..., d_q, ..., ((M \setminus x_q \cup y_q) \setminus x_p) \cup y_p).$$

Так как M – мультимножество сообщений, $x_p \subseteq M$, и $x_q \subseteq M$,

$$((M \setminus x_p \cup y_p) \setminus x_q \cup y_q) = ((M \setminus x_q \cup y_q) \setminus x_p \cup y_p),$$

и отсюда $\gamma_{pq} = \gamma_{qp}$.

Пусть e_p и e_q будут двумя событиями, которые появляются последовательно в исполнении, т.е. исполнение содержит подпоследовательность

$$..., \gamma, e_p(\gamma), e_q(e_p(\gamma)), ...$$

для некоторых γ. Посылка теоремы 2.19 применима к этим событиям за исключением следующих двух случаев.

(1) p = q; или
(2) e_p – событие отправки, и e_q - соответствующее событие получения.

В самом деле, теорема явно утверждает, что p и q должны быть различными, и если e_q получает сообщение, посланное в e_p, событие отправки не применимо в начальной конфигурации события e_p, как требуется. Таким образом, если одно из этих двух утверждений истина, события не могут появляться в обратном порядке, иначе они могут встречаться в обратном порядке и кроме того иметь результат в одной конфигурации. Запомните, что с глобальной точки зрения переходы не могут быть обменены, потому что (в нотации теоремы 2.19) переход из γ_p в γ_{pq} отличается от перехода из γ в γ_q. Однако, с точки зрения процесса эти *события* неразличимы.

Тот факт, что конкретная пара событий не может быть обменена, выражается тем, что существует *каузальное отношение* между этими двумя событиями. Это отношение может быть расширено до частичного порядка на множестве событий в исполнении, называемого *каузальный порядок* исполнения.

Определение 2.20 *Пусть E – исполнение. Отношение $\prec$, называемое каузальным порядком, на событиях исполнения есть самое малое отношение, которое удовлетворяет*

(1) *Если e и f – различные события одного процесса и e появляется перед f, то e $\prec$ f.*
(2) *Если s – событие отправки и r – соответствующее событие получения, то s $\prec$ r.*
(3) *Отношение $\prec$ транзитивно.*

Мы пишем a $\preceq$ b, чтобы обозначить (a $\prec$ b $\vee$ a = b). Так как $\preceq$ есть *частичный* порядок, могут существовать события a и b, для которых ни a $\preceq$ b ни b $\preceq$ a не выполняется. Говорят такие события *конкурирующие*, в нотации a || b. На рисунке 2.1, b || f, d || i, и т.д.

Каузальный порядок был впервые определен Лампортом [Lam78] и играет важную роль в рассуждениях, относящихся к распределенным алгоритмам. Определение $\prec$ подразумевает существование *каузальной цепочки* между каузально связанными событиями. Этим мы подразумеваем, что a $\prec$ b включает существование последовательности $a = e_0, e_1, ..., e_к = b$ такой, что каждая пара последовательных событий в цепочке удовлетворяет либо (1), либо (2) в определении 2.20. Каузальная цепочка может быть даже выбрана так, что каждая пара, удовлетворяющая (1), есть последовательная пара событий в процессе, где они встречаются, т.е., нет событий между ними. На рисунке 2.1 каузальная цепочка между событием a и событием l есть последовательность a, f, g, h, j, k, l.

2.3.2 Эквивалентность исполнений: вычисления

В этом подразделе показывается, что события исполнения могут быть переупорядочены в любом порядке, согласующимся с каузальным порядком, без воздействия на результат исполнения. Это переупорядочивание событий вызывает другую последовательность конфигураций, но это исполнение будет рассматриваться как эквивалент исходного исполнения.

Пусть f = (f_0 , f_1 , f_2 ,...) будет последовательностью событий. Эта последовательность - последовательность событий относящихся к исполнению F = (δ_0, δ_1, δ_2, ...), если для каждого i, f_i применимо в δ_i и f_i (δ_i) = δ_{i+1}. В этом случае F называется *включенным исполнением* последовательности f. Мы хотели бы, чтобы F уникально определялась последовательностью f, но это не всегда так. Если для некоторого процесса p нет события в p, включенного в f, то состояние процесса p может быть произвольным начальным состоянием. Однако, если f содержит по крайней мере одно событие из p, то первое событие в p, скажем (c, x, y, d), определяет, что начальное состояние процесса p будет c. Поэтому, если f содержит по крайней мере одно событие в каждом процессе, δ_0 уникально определено, и это определяет целое исполнение уникально.

Теперь пусть E = (γ_0, γ_1, γ_2, ...) будет исполнением с ассоциированной последовательностью событий E' = (e_0 , e_1 , e_2 , ...) и положим, что f –перестановка из E'. Это означает, что существует перестановка σ натуральных чисел (или множества {0, ..., k-1}, если E – конечное исполнение с k событиями) таких, что $f_i = e_{\sigma(i)}$. Перестановка (f_0 , f_1 , f_2 , ...) событий из E *согласующаяся* с каузальным порядком, если $f_i \{= f_j$ подразумевает i ≤ j, т.е., если нет события, которому предшествует в последовательности событие, которому оно само каузально предшествует.

Теорема 2.21 *Пусть f = (f_0 , f_1 , f_2 , ...) – перестановка событий из E, которая согласуется с каузальным порядком исполнения E. Тогда f определяет уникальное исполнение F, начинающееся в начальной конфигурации из E. F имеет столько же событий сколько и E, и если E конечно, то последняя конфигурация из F такая же как и последняя конфигурация из E.*

Доказательство. Конфигурации из F строятся одна за другой, и чтобы построить δ_{i+1} достаточно показать, что f_i применимо в δ_i. Возьмем $\delta_0 = \gamma_0$.

Предположим, что для всех j < i, f_j применимо в конфигурации δ_j и $\delta_{j+1} = f_j$ (δ_j). Пусть $\delta_i = (c_{p1}$, ..., c_{pN} , M) и пусть f_i =(c, x, y, d) будет событие в процессе p, тогда событие f_i применимо в δ_i, если c_p = c и x ⊆ M.

Чтобы показать, что c_p = c нужно различать два случая. В обоих случаях мы должны помнить, что каузальный порядок исполнения E *абсолютно* упорядочивает события в процессе p. Это подразумевает, что события в процессе p появляются в точно таком же порядке и в f и в E'.

Случай 1: f_i - первое событие в p из f, тогда c_p – это начальное состояние p. Но тогда f_i – также первое событие в p из E', что подразумевает, что c – это начальное состояние p. Следовательно, c = c_p.

Случай 2: f_i – не первое событие в p из f. Пусть последнее событие в p из f перед f_i будет $f_{i'}$ = (c', x', y', d'), тогда c_p = d'. Но тогда $f_{i'}$ также последнее событие в p перед f_i из E', что подразумевает, что c = d'. Следовательно, c = c_p.

Чтобы показать, что $x \subseteq M$ мы должны помнить, что соответствующие события приема и посылки встречаются в одном порядке и в f и в E'. Если f_i не событие посылки, то $x = \varnothing$ и $x \subseteq M$ выполняется тривиально. Если f_i – это событие посылки, пусть f_i будет соответствующим событием посылки. Так как $f_j \{ f_i$, $j < i$ выполняется, т.е., событие посылки предваряет f_i в f, следовательно, $x \subseteq M$.

Мы сейчас показали, что для каждого i, f_i применимо в δ_i, и δ_{i+1} может быть взято как $f_i(\delta_i)$. Мы должны, наконец, показать, что последние конфигурации из F и E совпадают, если E конечно. Пусть γ_k будет последней конфигурацией из E. Если E' не содержит события в p, то состояние p в γ_k равно его начальному состоянию. Так как f также не содержит события в p, то состояние p в δ_k также равно начальному состоянию, отсюда состояние p в δ_k равняется его состоянию в γ_k. Иначе, состояние p в γ_k есть состояние после последнего события в p из E'. Это также последнее событие в p из f, так что это также состояние p в δ_k.

Сообщения в процессе передачи в γ_k есть такие сообщения, для которых событию посылки нет соответствующего события получения в E'. Но так как E' и f содержат один и тот же набор событий, те же сообщения в процессе передачи в последней конфигурации из F.

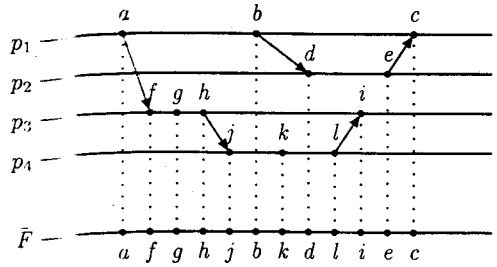


Рис. 2.2 Пространственно-временная диаграмма эквивалентная рис. 2.1

Исполнения F и E имеют один набор событий, и каузальный порядок этих событий – один и тот же для E и F. Поэтому, также, в этом случае E – это перестановка событий из F, которая согласуется с каузальным порядком исполнения F. Если применить условие теоремы 2.21, мы можем сказать, что E и F – эквивалентные исполнения, что обозначается как $E \sim F$.

Рис. 2.2 показывает временную диаграмму исполнения, эквивалентного исполнению, изображенному на рис. 2.1. Эквивалентные временные диаграммы могут быть получены с помощью «трансформаций резиновой ленты» [Mat89c]. Полагая, что временная ось процесса может быть сжата и растянута пока стрелки сообщений продолжают указывать направо, рис. 2.1 может быть деформирован до рис. 2.2.

Хотя изображенные исполнения эквивалентны и содержат одинаковый набор событий, они не могут содержать одинаковый набор конфигураций. Рис. 2.1 содержит конфигурацию (γ'), в которой сообщение, посланное в событии e и сообщение, посланное в событии l, передаются одновременно . Рис. 2.2 не со-

держит такой конфигурации, потому что сообщение, посланное в событии l, получено перед свершением события e.

Глобальный наблюдатель, кто имеет доступ к действительной последовательности событий, может различать два эквивалентных исполнения, т.е. может наблюдать либо одно, либо другое исполнение. Однако, процессы не могут различать две эквивалентных исполнения, т.к. для них невозможно решить, какое из двух исполнений имеет место. Это иллюстрируется следующим. Предположим, что мы должны решить будут ли посылаться сообщения в событии e и будут в передаче одновременно. Существует булевская переменная sim в одном из процессов, которая должна установлена в истину, если сообщения были в передаче одновременно, и ложь иначе. Таким образом, в последней конфигурации рис. 2.1 значение sim – истина, и в последней конфигурации на рис 2.2 значение – ложь. По теореме 2.21, конфигурации равны, что показывает, что требуемое присваивание sim невозможно.

Класс эквивалентности при отношении ~ полностью характеризуется набором событий и каузальным порядком на этих событиях. Классы эквивалентности называются вычислениями алгоритма.

Определение 2.22 *Вычисление распределенного алгоритма – это класс эквивалентности (при ~) исполнений алгоритма.*

Не имеет смысла говорить о конфигурациях вычисления, потому что различные исполнения вычисления могут не иметь одних и тех же конфигураций. Имеет смысл говорить о наборе событий вычисления, потому что все исполнения вычисления состоят из одного и того же набора событий. Также, каузальный порядок событий определен для вычисления. Мы будем называть вычисление конечным, если его исполнения конечны. Все исполнения вычисления начинаются в одной конфигурации и, если вычисление конечно, завершаются в одной конфигурации (теорема 2.21). Эти конфигурации называются начальными и конечными конфигурациями вычисления. Мы будем определять вычисление с помощью частично упорядоченного множества событий, принадлежащих ему.

Результат из теории частичных порядков подразумевает, что каждый порядок может встречаться для пары конкурирующих событий вычислений.

Факт 2.23 *Пусть $(X, <)$ будет частичным порядком и $a, b \in X$ удовлетворяют $b \geq a$. Существует линейное расширение $<_1$ операции $<$ такое, что $a <_1 b$.*

Следовательно, если a и b – конкурирующие события вычисления C, существуют исполнения E_a и E_b этого вычисления такие, что a имеет место раньше, чем b в E_a, и b имеет место раньше, чем a в E_b. Процессы в исполнении не имеют средств, чтобы решить, какое из двух событий произошло раньше.

Синхронная передача сообщений Версия теоремы 2.19 может быть сформулирована также для систем с синхронной передачей сообщений. В таких системах два последовательных событий независимы, если они воздействуют на различные процессы, как сформулировано в следующей теореме.

Теорема 2.24 *Пусть γ будет конфигурацией распределенной системы с синхронной передачей сообщений и пусть e_1 будет переходом процессов p и q, и e_2 будет переходом процессов r и s, отличных от p и q, такие, что и e_1 и e_2 применим в γ. Тогда e_1 применим в $e_2(\gamma)$, e_2 применим в $e_1(\gamma)$, и $e_1(e_2(\gamma)) = e_2(e_1(\gamma))$.*

Доказательство этой теоремы, которое основывается на тех же аргументах, что и доказательство теоремы 2.19, оставлено для упражнения 2.9. Понятие казуальности в синхронных системах может быть определено подобно определению 2.20. Интересующегося читателя можно отослать к [CBMT92]. Теорема 2.21 также имеет своего двойника для синхронных систем.

2.3.3 Логические часы

По аналогии с физическими часами, которые измеряют реальное время, в распределенных вычислениях часы могут быть определены, чтобы выразить каузальность. На протяжении всего этого раздела, Θ - функция, действующая из набора событий в упорядоченное множество (X, <)

Определение 2.25 *Часы есть функция Θ, действующая из событий на упорядоченное множество такое, что*

$$a \nearrow b \Rightarrow \Theta(a) < \Theta(b).$$

Далее в этом разделе обсуждаются некоторые примеры часов.

(1) *Порядок в последовательности.* В исполнении E, определенном последовательностью событий (e_0, e_1, e_2, …), множество $\Theta_g(e_i) = i$. Таким образом, каждое событие помечается своей позицией в последовательности событий.

Эта функция может использоваться глобальным наблюдателем системы, кто имеет доступ к порядку, в котором происходят события. Однако, невозможно наблюдать этот порядок *внутри* системы, или, иначе говоря, Θ_g не может быть вычислена распределенным алгоритмом. Это следствие теоремы 2.19. Предположим, что некоторый распределенный алгоритм сохраняет значение $\Theta_g(e_i) = i$ для события e_i (что удовлетворяет посылке теоремы). В эквивалентном исполнении, в котором это событие меняется со следующим событием, и следовательно имеет другое значение Θ_g, то же значение i сохраняется в процессе. Говоря другими словами, Θ_g определено для исполнений, но не для вычислений.

(2) *Часы реального времени.* Имеется возможность *расширить* модель, что является предметом обсуждения этой главы, с помощью снабжения каждого процесса аппаратными часами. Этим путем возможно записывать для каждого события реальное время, в которое оно произошло. Полученные числа удовлетворяют определению часов.

Распределенные системы с часами реального времени не удовлетворяют определению 2.6, потому что физические свойства часов синхронизируют изменения состояний в разных процессах. Время идет во всех процессах, и это порождает переходы, которые меняют

состояние (а именно, считыванием часов) всех процессов. Оказывается, что эти «глобальные переходы» ужасно меняют свойства модели. В самом деле, теорема 2.19 больше не действует, если приняты часы реального времени. Распределенные системы с часами реального времени используются на практике, однако, и они будут рассматриваться в этой книге (см. раздел 3.2) и главы 11 и 14.

var θ_p : integer **init** 0 ;

(* An internal event *)
$\theta_p := \theta_p + 1$;
Change state

(* A send event *)
$\theta_p := \theta_p + 1$;
send $\langle$ **messg**, $\theta_p \rangle$; Change state

(* A receive event *)
receive $\langle$ **messg**, $\theta \rangle$; $\theta_p := \max(\theta_p, \theta) + 1$;
Change state

Алгоритм 2.3 Логические часы Лампорта

(3) *Логические часы Лампорта.* Лампорт [Lam78] представил часовую функцию, которая приписывает событию a длину k самой длинной последовательности (e_1, …, e_k) событий, удовлетворяющей

$$e_1 \lessdot e_2 \lessdot \ldots \lessdot e_k = a$$

В самом деле, если $a \lessdot b$, эта последовательность может быть расширена, чтобы показать, что $\Theta_L(a) < \Theta_L(b)$. Значение Θ_L может быть вычислено для каждого события распределенным алгоритмом, базируясь на следующих отношениях.

(a) Если a есть внутреннее событие или событие посылки, и a' – предыдущее событие в том же процессе, то $\Theta_L(a) = \Theta_L(a') + 1$.

(b) Если a – событие получение, a' – предыдущее событие в том же процессе, и b –событие посылки, соответствующее a, то $\Theta_L(a) = \max(\Theta_L(a'), \Theta_L(b)) + 1$.

В обоих случаях $\Theta_L(a')$ предполагается нулевым, если a – первое событие в процессе.

Чтобы вычислить значения часов распределенным алгоритмом, значение часов последнего события процесса p сохраняется в переменной θ_p (инициализируемой в 0). Для того, чтобы вычислить значение часов события получения, каждое сообщение m содержит значение часов θ_m события e, при котором оно было послано. Логически часы Лампорта даны как алгоритм 2.3. Для события e в процессе p, $\Theta_L(e)$ есть значение θ_p сразу же после появления e, т.е. в момент, когда происходит изменение состояния процесса p. Оставлена для упражнения демонстрация того, что с этим определением Θ_L является часами.

Не указывается при каких условиях сообщение должно быть послано или как меняется состояние процесса. Часы –это дополнительный механизм, который может быть добавлен к любому распределенному алгоритму, чтобы упорядочивать события.

(4) *Векторные часы.* Для некоторых целей полезно иметь часы, который выражают не только каузальный порядок (как требуется по определению 2.25), но также и конкуренцию. Конкуренция выражается часами, если конкурентные события помечаются несравнимыми значениями часов, то есть, следствие в определении 2.25 заменяется на эквиваленцию, давая

$$a \nearrow b \Leftrightarrow \Theta(a) < \Theta(b). \qquad (2.1)$$

Существование конкурирующих событий подразумевает, что область таких часов (множество X) – не-полностью-упорядоченное множество.

В векторных часах Маттерна [Mat89b] $X = \mathbf{N}^N$, т.е. $\Theta_v(a)$ есть *вектор* длины N. Вектора длины n естественным образом упорядочены векторным порядком, определенным следующим образом:

$$(a_1, \ldots, a_n) \leq_v (b_1, \ldots, b_n) \Leftrightarrow \forall i\ (1 \leq i \leq n) : a_i \leq b_i. \qquad (2.2)$$

(Векторный порядок отличается от лексикографического порядка, определенного в упражнении 2.5, последний порядок абсолютен). Часы, определяемые $\Theta_v(a) = (a_1, \ldots, a_N)$, a_i – это число событий e *в процессе* p_1, для которого $e \nearrow a$. Как и часы Лампорта, эта функция может быть вычислена распределенным алгоритмом.

Чаррон-Бост [CB89] показал, что невозможно использовать более короткие векторы (с векторным порядком как в (2.2)). Если события произвольного исполнения из N процессов отображаются на вектора длины n таких, что (2.1) удовлетворяется, то $n \geq N$.

2.4 Дополнительные допущения, сложность

Определений сделанных до сих пор в этой главе достаточно, чтобы развивать оставшиеся главы. Определенная модель служит как основа для представления и проверки алгоритмов, так и для доказательств невозможности для решения распределенных проблем. В различных главах используются дополнительные допущения и нотация, если требуется. Этот раздел обсуждает некоторую терминологию, которая также общеупотребительна в литературе по распределенным алгоритмам. До сих пор, мы моделировали коммуникационную подсистему распределенной системы набором сообщений, находящихся в данный момент в процессе передачи. Далее, мы будем предполагать, что каждое сообщение может передаваться только одним процессом, называемым назначением сообщения. В общем, не обязательно чтобы каждый процесс мог посылать сообщения каждому другому процессу. Вместо этого, для каждого процесса определено подмножество других процессов (называемых *соседями* процесса), к которым он может посылать сообщения. Если процесс p может посылать сообщения процессу q, говорят, что существует *канал* от p до q. Если не утвержда-

ется обратное, предполагается, что каналы *двунаправленные*, то есть, тот же канал позволяет посылать q сообщения процессу p. Канал, который осуществляет только однонаправленный трафик от p к q, называется *однонаправленным* (или *направленным*) каналом от p до q.

Набор процессов и коммуникационная подсистема также упоминается как *сеть*. Структура коммуникационной подсистемы часто представляется как *граф* G = (V, E), в котором вершины – это процессы, и ребра между двумя процессами существуют, если и только если канал существует канал между двумя процессами. Система с однонапрвленными каналами может подобным образом представлена направленными графом. Граф распределенной системы также называется ее *сетевой топологией*.

Представление графом позволяет нам говорить о коммуникационной системе в терминах теории графов. См. дополнение Б для представления об этой терминологии. Так как сетевая топология происходит от основного влияния на существование, внешний вид, и сложность распределенных алгоритмов для многих проблем, мы включаем ниже краткое обсуждение некоторых повсеместно используемых здесь топологий. См. дополнение Б для дополнительных деталей. На протяжении этой книги, если не утверждается обратное, предполагается, что топология *связана*, то есть, существует путь между двумя вершинами.

(1) *Кольца.* N-вершинное *кольцо* – граф на вершинах от v_0 до v_{N-1} с ребрами v_0v_{N-1} (индексы – по модулю N). Кольца часто используются для распределенного управления вычислениями, потому что они просты. Также, некоторые физические сети, такие как Token Rings [Tan88, раздел 3.4], распределяют узлы в кольцо.

(2) *Деревья. Дерево* на N вершинах – это связанный граф с N –1 ребрами, он не содержит циклов. Деревья используются в распределенных вычислениях, потому что они позволяют проводить вычисление при низкой цене коммуникаций, и более того, каждый связанный граф содержит дерево, как подсеть охвата.

(3) *Звезды. Звезда* на N вершинах имеет одну специальную вершину (*центр*) и N-1 ребер, соединяющих каждую из N-1 вершин с центром. Звезды используются в централизованных вычислениях, где один процесс действует как контроллер и все другие процессы сообщаются только с этим специальным процессом. Недостатки звездной топологии это узкое место, каким может стать центр и уязвимость такой системы из-за повреждений в центре.

(4) *Клики. Клика* – это сеть, в которой ребро существует между любыми двумя вершинами.

(5) Гиперкубы. *Гиперкуб* – это граф $HC_N = (V, E)$ на $N = 2^n$ вершинах. Здесь V – множество битовых строк длины n:

$$V = \{(b_0 , ..., b_{n-1}) : b_i \in \{0, 1\}\},$$

и ребро существует между двумя вершинами b и c, тогда и только тогда, когда битовые строки b и c различаются точно на один бит. Имя гиперкуба относится к графическому представлению сети как n-размерного куба, углы которого – вершины.

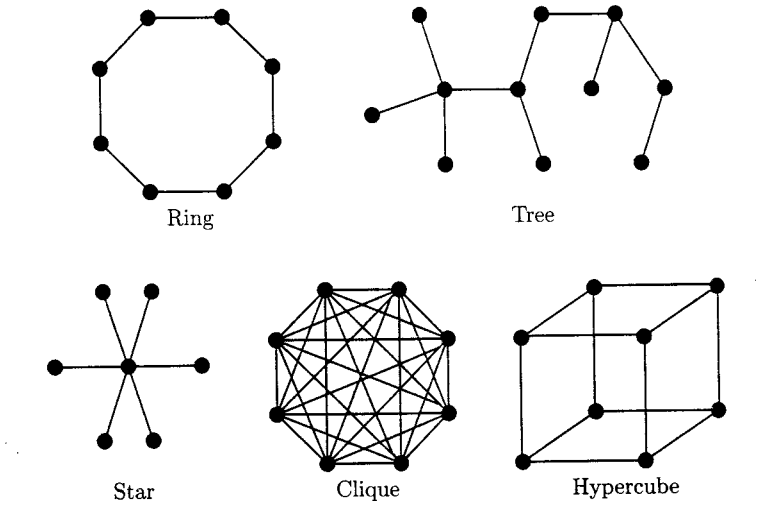


Рис. 2.4 Примеры часто используемых топологий

Примеры каждой из этих сетей приведены на рис 2.4. Топология может быть *статической* или *динамической*. Статическая топология означает, что топология остается фиксированной в течение распределенного вычисления. Динамическая топология означает, что каналы (иногда даже процессы) могут быть добавлены или удалены из системы в течение вычисления. Эти изменения в топологии могут быть также смоделированы переходами конфигураций, а именно, если состояния процесса отражают множество соседей процесса (см. главу 4).

2.4.2 Свойства каналов

Модель (как описана в подразделе 2.1.2) может быть усовершенствована при помощи представления содержимого каждого канала раздельно в конфигурации, то есть, замены множества M на набор множеств M_{pq} для каждого (однонаправленного) канала pq. Так как мы постулировали, что каждое сообщение неявно определяет свое назначение, то эта модификация не изменяет важных свойств модели. Далее обсуждаются некоторые общие допущения относительно соотношения событий приема и посылки.

(1) *Надежность*. Говорят, что канал *надежен*, когда каждое сообщение, которое посылается в канал принимается точно один раз (обеспечив назначению возможность получить сообщение). Если не утверждается обратное, всегда предполагается в этой книге, что каналы надежны. Это допущение фактически добавляет (слабое) условие справедливости. В самом деле, после того как сообщение послано, получение этого сообщения (в приемлемом для назначения состоянии) применимо.

Канал, который ненадежен, может проявлять *коммуникационные сбои*, которые могут быть нескольких типов, например, утеря, искажение, дублирование, порождение. Эти сбои могут быть представлены

переходами в модели определения 2.6, но эти переходы не соответствуют изменениям состояния процесса.

Утеря сообщения имеет место, когда сообщение посылается, но никогда не принимается. Это может быть смоделирована переходом, который удаляет сообщение из M. *Искажение* сообщение встречается, когда полученное сообщение отличается от посланного сообщения. Это может быть смоделировано переходом, который меняет одно сообщение из M. *Дублирование* сообщения появляется, если сообщение принимается более часто, чем оно посылалось. Это может быть смоделировано переходом, который копирует сообщение из M. *Порождение* сообщения, встречается, когда сообщение получено, но никогда не было послано, это моделируется переходом, который вставляет сообщение в M.

(2) Свойство fifo. Говорят, что канал является *fifo*, если он соблюдает порядок сообщений, посланных через него. То есть, если p посылает два сообщения m_1 и m_2 процессу q и отправка m_1 происходит раньше в p, чем отправка m_2, то получение m_1 происходит раньше в q, чем получение m_2. Если не утверждается обратное, fifo каналы *не* будут предполагаться в этой книге.

Fifo каналы могут быть представлены в модели определения 2.6 при помощи замены набора M на множество *очередей*, одной для каждого канала. Отправка осуществляется добавлением сообщения к концу очереди, и событие получения удаляет сообщение с головы. Когда предполагаются каналы fifo, появляется новый тип коммуникационных сбоев, а именно, переупорядочивание сообщений в канале. Это может быть смоделировано переходом, который обменивает два сообщения в очереди.

Иногда случается, что распределенный алгоритм получает пользу от свойства fifo каналов, см., например, коммуникационный протокол в разделе 3.1. Использование порядка получение сообщений снижает количество информации, которая должна транспортироваться в каждом сообщении. Во многих случаях, однако, алгоритм может быть разработан так, чтобы функционировать правильно (и эффективно) даже, если сообщения могут быть переупорядочены в канале. В общем, реализация свойства fifo расределенных систем может понизить свойственный параллелизм вычислений, т.к. это может потребовать буферизации сообщений (на стороне получателя в канале) перед тем как сообщение будет обработано. По этой причине мы не выбираем предположение свойства fifo неявно в этой книге.

Более слабое допущение было предложено Ахуджа [Ahu90]. *Выталкивающий канал* – это канал, который соблюдает порядок только сообщений, для которых это было указано отправителем. Могут быть также определены более сильные допущения. Шипер и др. [SES89] определили *каузально упорядоченную доставку сообщений*, как описывается далее. Если p_1 и p_2 посылают сообщения m_1 и m_2 процессу q в событиях e_1 и e_2 и $e_1 \{ e_2$, то q получает m_1 перед m_2 . Иерархия допущений доставки, состоящая из полного асинхронизма, каузально упорядоченной доставки, fifo, и синхронных коммуникаций, обсуждалась Чаррон-Бостом и др. [CBMT92].

(3) *Емкость канала. Емкость* – это число сообщений, которое может передаваться по каналу одновременно. Канал *полон* в каждой конфигурации, в которой он действительно содержит количество сообщений, равное его емкости. Событие посылки применимо, только если канал не полон.

Определение 2.6 моделирует каналы с неограниченной емкостью, т.е. каналы, которые никогда не наполняются. В этой книге всегда будет предполагаться, что емкость каналов не ограничена.

2.4.3 Допущения реального времени

Основное свойство представленной модели есть, конечно, ее распределенность: полная независимость событий в различных процессах, как выражает теорема 2.19. Это свойство теряется, когда предполагается кадр глобального времени и способность процессов наблюдать физическое время (устройство физических часов). В самом деле, когда некоторое реальное время истекает, это время истекает во всех процессах, и это проявится на часах каждого процесса.

Часы реального времени могут быть встроены при помощи снабжения каждого процесса переменной часов реального времени. Течение реального времени моделируется переходом, который передвигает вперед часы каждого процесса, см. раздел 3.2. Обычно, принимается ограничение на время передачи сообщения (время между отправкой и получением сообщения) вкупе с доступностью часов реального времени. Это ограничение может быть также включено в общую модель системы переходов.

Если не утверждается обратное, допущения реального времени не делаются в этой книге, т.е. мы предполагаем *полностью асинхронные* системы и алгоритмы. Допущения отсчета времени будут использованы в разделе 3.2, главе 11 и главе 14.

2.4.4 Знания процессов

Изначальные знания процесса – это термин, используемый для обращения к информации о распределенной системе, которая представляется в начальных состояниях процессов. Если алгоритму сказано полагаться на такую информацию, то предполагается, что релевантная информация правильно сохраняется в процессах, прежде чем начнется исполнение системы. Примеры таких знаний включают следующую информацию.

(1) *Топологическая информация.* Информация о топологии включает: количество процессов, диаметр графа сети, и топологию графа. Говорят, что сеть имеет *чувство направления*, если согласующаяся с направлениями разметка ребер в графе известна процессам (см. дополнение Б).
(2) *Идентичность процессов.* Во многих алгоритмах требуется, чтобы процессы имели уникальные имена (идентификаторы), и чтобы каждый процесс знал свое собственное имя изначально. Тогда предполагается, что процессы содержат переменную, которая инициализируется этим именем (т.е. различным для каждого процесса). Дальнейшие допущения могут быть сделаны касательно множества, из которого вы-

бираются имена, - что имена линейно упорядочены или что они (положительные) целые. Если не утверждается обратное, в этой книге всегда будем предполагать, что процессы имеют доступ к их идентификаторам, в этом случае система называется *именованной сетью*. Ситуации, где это не так (*анонимные сети*) будут исследованы в главе 9.

(3) *Идентификаторы соседей.* Если процессы различаются уникальным именем, то возможно предположить, что каждый процесс знает изначально имена соседей. Это допущение называется *знание соседей* и, если не утверждается обратное, не будет делаться. Имена процессов могут быть полезными для цели адресации сообщений. Имя назначения сообщения дается, когда сообщение посылается *с прямой адресацией.* Более сильные допущения состоят в том, что каждый процесс знает весь набор имен процессов. Более слабое допущение состоит в том, что процессы знают о существовании, но не знают имен своих соседей. Прямая адресация не может использоваться в этом случае, и процессы используют локальные имена для их каналов, когда хотят адресовать сообщение, что называется *непрямой адресацией.* Прямая и непрямая адресация показана на рис. 2.5. Прямая адресация использует идентификатор процесса как адрес, в то время как непрямая адресация процессов p, r и s использует различные имена (a, b и c, соответственно), чтобы адресовать сообщения в назначение q.

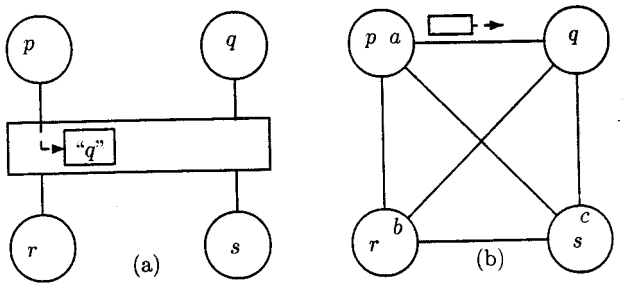


Рис. 2.5 Прямая (a) и непрямая (b) адресация

2.4.5 Сложность распределенных алгоритмов

Самое важное свойство распределенного алгоритма – его правильность: он должен удовлетворять требованиям, налагаемым проблемой, что алгоритм

3 Протоколы Связи

В этой главе обсуждаются два протокола, которые используются для надежного обмена данными между двумя вычислительными станциями. В идеальном случае, данными бы просто обменивались, посылая и получая сообщения. К сожалению, не всегда можно игнорировать возможность ошибок связи; сообщения должны транспортироваться через физическую среду, которая может терять, дублировать, переупорядочивать или искажать сообщения, передаваемые через нее. Эти ошибки должны быть обнаружены и исправлены дополнительными механизмами, выполняющимися на вычислительных станциях, которые традиционно называются протоколами.

Основная функция этих протоколов - *передача данных*, то есть, принятие информации на одной станции и получение ее на другой станции. Надежная передача данных включает повторную посылку сообщений, которые потеряны, отклонение или исправление сообщений, которые искажены, и отбрасывание дубликатов сообщений. Для выполнения этих функций протокол содержит информацию состояния, записывая, какие данные уже был посланы, какие данные считаются полученными и так далее. Необходимость использования информации состояния поднимает проблему *управления соединением*, то есть, инициализации и отбрасывания информации состояния. Инициализация называется *открытием* соединения, и отбрасывание называется *закрытием* соединения. Трудности управления соединением возникают из-за того, что сообщение может остаться в каналах связи, когда соединение закрыто. Такое сообщение могло бы быть получено, когда не существует никакого соединения или в течение более позднего соединения, и получение не должно нарушать правильную операцию текущего соединения.

Протоколы, обсуждаемые в этой главе разработаны для различных уровней в иерархии протокола, типа модели OSI (Подраздел 1.2.2). Они включены в эту книгу по различным причинам; первый протокол полностью асинхронный, в то время как второй протокол полагается на правильное использование таймеров. В обоих случаях заостряется внимание на требуемом свойстве *безопасности*, а именно на том, что приемник получит только правильные данные.

Первый протокол (Раздел 3.1) разработан для обмена данными между двумя станциями, которые имеют прямое физическое соединение (типа телефонной линии), и, следовательно, принадлежит канальному уровню модели OSI. Второй протокол (Раздел 3.2) разработан для использования двумя станциями, которые связываются через промежуточную сеть (возможно содержащую другие станции и соединяющую станции через различные пути), и этот протокол следовательно принадлежит к транспортному уровню OSI модели. Это различие отражается на функциональных возможностях, требуемых от протоколов, следующим образом.

(1) *Рассматриваемые ошибки*. Для двух протоколов будут рассматриваться различные классы ошибок передачи. Сообщения не могут пересекаться при физическом соединении, и они не могут быть продублированы; таким образом, в разделе 3.1 рассматривается только потеря сообщений (об искажении сообщений см. ниже). В сети сообщения могут передаваться различными путями, и, следовательно, пересекаться; также, из-за отказов промежуточных станций сообщения могут быть продублированы или потеряны. В Разделе 3.2 будут рассматриваться потеря, дублирование и переупорядочение сообщений.

(2) *Управление соединением*. Далее, управление соединением не будет рассматриваться для первого протокола, но будет для второго. Предполагается, что физическое соединение функционирует непрерывно в течение очень длительного времени, а не открывается и закрывается неоднократно. Для соединений с удаленными станциями это не так. Такое соединение может быть необходимо временно для обмена некоторыми данными, но обычно слишком дорого поддерживать соединение с каждой удаленной станцией неопределенно долго. Следовательно, для второго протокола будет требоваться способность открывать и закрывать соединение.

При рассмотрении первого протокола показывается, что не только механизмы, основанные на таймерах, могут обеспечить требуемые свойства безопасности протоколов передачи данных. Раздел 3.1 служит первым большим примером доказательства свойств безопасности с помощью инструментальных средств, описанных в Разделе 2.2. Многие полагают [Wat81], что правильное использование таймеров и ограничение на время, в течение которого сообщение может передаваться , необходимы для безопасного управления соединением. Таким образом, для того, чтобы доказать безопасность протоколов, нужно принимать во внимание роль таймеров в управлении соединением. Раздел 3.2 показывается, как модель распределенных систем (Определение 2.6) может быть расширена до процессов, использующих таймеры, и дает пример этого расширения.

Искажение сообщений. Естественно принять во внимание возможность того, что сообщения могут быть искажены в течение передачи. Содержание сообщения, переданного через физическое соединение, может быть повреждено из-за атмосферных шумов, плохо функционирующих модулей памяти, и т.д. Однако можно предположить, что искажение сообщения может быть обнаружено процессом-получателем, например, посредством контроля четности или более общих механизмов контрольной суммы ([Tan88, Глава 41). Получение искаженного сообщения затем обрабатывается так, как будто не было получено никакого сообщения, и таким образом, искажение сообщения фактически вызывает его потерю. По этой причине искажение не обрабатывается явно; вместо этого всегда рассматривается возможность потери сообщения.

3.1 Сбалансированный протокол скользящего окна

В этом разделе изучается симметричный протокол, который обеспечивает надежный обмен информацией в обоих направлениях. Протокол взят из [Sch91, Глава 2]. Поскольку он используется для обмена информацией между станциями, которые непосредственно соединены через линию, можно предположить, что каналы имеют дисциплину fifo. Это предположение не используется, однако, до Подраздела 3.1.3, где показано, что числа последовательности, используемые протоколом могут быть ограничены. Протокол представлен в Подразделе 3.1.1, а в Подразделе 3.1.2 доказывается его правильность.

Два процесса связи обозначаются как p и q. Предположения, требования и протокол абсолютно симметричны. Вход p состоит из информации, которую он должен послать q, и моделируется неограниченным массивом слов in_p. Выход p

состоит из информации, которую он получает от *q,* и также моделируется неограниченным массивом слов, out_p. Предполагается, что *p* имеет случайный доступ по чтению к in_p и случайный доступ по записи к out_p. Первоначально значение $out_p[i]$ не определено и представлено как *udef* для всех *i.* Вход и выход процесса *q* моделируется массивами in_q и out_q соответственно. Эти массивы нумеруются натуральными числами, т.е. они начинаются со слова с номером 0. В подразделе 3.1.3 будет показано, что произвольный доступ может быть ограничен доступом к "окну" конечной длины, передвигающемуся вдоль массива. Поэтому протокол называется протоколом «скользящего окна».
Процесс *p* содержит переменную s_p, показывающую наименьшее нумерованное слово, которое *p* все еще ожидает от *q.* Таким образом, в любой момент времени, *p* уже записал слова от $out_p[0]$ до $out_p[sp - 1]$. Значение s_p никогда не уменьшается. Аналогично *q* содержит переменную s_q. Теперь могут быть установлены требуемые свойства протокола. Свойство безопасности говорит о том, что каждый процесс передает только корректные данные; свойство живости говорит о том, что все данные когда-либо будут доставлены.

(1) *Свойство безопасности.* В каждой достижимой конфигурации протокола $out_p[0..s_p — 1] = in_q[0..Sp — 1]$ и $out_q[0..s_q — 1] = in_p[0...s_q — 1]$.

(2) *Окончательная доставка.* Для каждого целого $k \geq 0$, конфигурации с $s_p \geq k$ и
$s_q \geq k$ когда-либо достигаются.

3.1.1 Представление протокола

Протоколы передачи обычно полагаются на использование сообщений *подтверждения*. Сообщение подтверждения посылается процессом получения, чтобы сообщить отправителю о данных, которые он получил корректно. Если отправитель данных не получает подтверждение, то он предполагает, что данные (или подтверждение) потеряно, и повторно передает те же самые данные. В протоколе этого раздела, однако, не используются явные сообщения подтверждения. В этом протоколе обе станции имеют сообщения, которые нужно послать другой станции; сообщения станции служат также подтверждениями для сообщений другой станции.

Сообщения, которыми обмениваются процессы, называют *пакетами*, и они имеют форму
< **pack,** *w, i* >, где *w* - слово данных, а *i* - натуральное число (называемое *порядковым номером* пакета). Этот пакет, посылаемый процессом *p* (к *q*), передает слово = $in_p[i]$ для *q,* но также, как было отмечено, подтверждает получение некоторого количества пакетов от *q.* Процесс *p* может быть «впереди» *q* не более, чем на l_p пакетов, если мы потребуем, что пакет данных < **pack,** *w, i* >, посланный *p*, подтверждает получение слов с номерами 0.. *i*— l_p от *q*. (*q* посылает аналогичные пакеты.*)* Константы l_p и l_q неотрицательны и известны обоим процессам *p* и *q*. Использование пакета данных в качестве подтверждения имеет два последствия для протокола:

(1) Процесс *p* может послать слово $in_p[i]$ (в виде пакета < **pack,** $in_p[i]$, *i* >) только после того, как запишет все слова от $out_p[0]$ до $out_p[i — l_p]$, *т. е. ,* если $i < s_p + l_p$.

(2) Когда p принимает < **pack,** w,i>, повторная передача слов с $in_p[0]$ до $in_p[i - l_q]$ уже не нужна.

Объяснение псевдокода. После выбора модели нетрудно разработать код протокола; см. Алгоритм 3.1. Для процесса p введена переменная a_p (и a_q для q), в которой хранится самое первое слово, для которого p (или q, соответственно) еще не получил подтверждение..
В Алгоритме 3.1 действие $\mathbf{S}_p$ - посылка i-го слова процессом p, действие $\mathbf{R}_p$ - принятие слова процессом p, и действие $\mathbf{L}_p$ - потеря пакета с местом назначения p. Процесс p может послать любое слово, индекс которого попадает в указанные ранее границы. Когда сообщение принято, в первую очередь делается проверка - было ли идентичное сообщение принято ранее (на случай повторной передачи). Если нет, слово, содержащееся в нем, записывается в выход, и a_p и s_p корректируются. Также вводятся действия S_q, R_q и L_q , где p и q поменяны ролями.

```
var s_p, a_p : integer              init 0, 0 ;
   in_p   : array of word           (* Посылаемые данные *) ;
   out_p  : array of word            init udef, udef, ...',
S_p: {a_p ≤ i < S_p+l_p}
   begin send < pack,in_p[i],i> to q end
R_p: { < pack, w, i > ∈ Q_p }
   begin receive <pack, w, i> ;
      if out_p[i] = udef then
        begin out_p[i] := w ;
            a_p := max(a_p,i-l_p+1) ;
            S_p := min{j|out_p[j]= udef}
        end
      (* else игнорируем, пакет передавался повторно *)
   end

Lp: {<pack,w,i>∈ Q_p}
begin Q_p := Q_p\ {<pack,w,i>} end
```

Алгоритм 3.1 Протокол скользящего окна (для p).

Инвариант протокола. Подсистема связи представляется двумя очередями, Q_p для пакетов с адресатом p и Q_q, для пакетов с адресатом q. Заметим, что перевычисление s_p в $\mathbf{R}_p$ никогда не дает значение меньше предыдущего, поэтому s_p никогда не уменьшается. Чтобы показать, что этот алгоритм удовлетворяет данным ранее требованиям, сначала покажем, что утверждение P - инвариант. (В этом и других утверждениях i - натуральное число.)

$P \equiv \quad \forall i < s_p : out_p[i] \neq udef$ (0p)
$\wedge\ \forall i < s_q, : out_q[i] \neq udef$ (0q)
$\wedge$ < **pack,** $w, i > \in Q_p \ \Rightarrow\ w = in_q[i] \wedge (i < s_q + l_q)$ (lp)
$\wedge$ < **pack,** $w, i > \in Q_q \ \Rightarrow\ w = in_p[i] \wedge (i < s_p + l_p)$ (lq)
$\wedge\ out_p[i] \neq udef \ \Rightarrow out_p[i] = in_q[i] \wedge (a_p > i - l_q)$ (2p)
$\wedge\ out_q[i] \neq udef \ \Rightarrow out_q[i] = in_p[i] \wedge (a_q > i - l_p)$ (2q)
$\wedge\ a_p \leq s_q,$ (3p)
$\wedge\ a_q \leq s_p$ (3q)

Лемма 3.1 P - инвариант Алгоритма 3.1.

Доказательство. В любой начальной конфигурации Q_p и Q_q - пустые, для всех i, $out_p[i]$ и $out_q[i]$ равны udef, и a_p,a_p, s_p и s_q равны нулю 0; из этого следует, что P=true. Перемещения протокола рассмотрим с точки зрения сохранения значения P. Во-первых, заметим, что значения in_p и in_q, никогда не меняются.

S_p: Чтобы показать, что **S_p** сохраняет (0p), заметим, что **S_p** не увеличивает s_p и не делает ни один из $out_p[i]$ равным udef.

Чтобы показать, что **S_p** сохраняет (0q), заметим, что **S_p** не увеличивает s_q, и не делает ни один из $out_q[i]$ равным udef.

Чтобы показать, что **S_p** сохраняет (1p), заметим, что **S_p** не добавляет пакеты в Q_p и не уменьшает s_p.

Чтобы показать, что **S_p** сохраняет (lq), заметим **S_p** добавляет < **pack,** w, i > в Q_p с $w = in_p[i]$ и $i < s_p + l_p$, и не изменяет значение s_p.

Чтобы показать, что **S_p** сохраняет (2p) и (2q), заметим, что **S_p** не изменяет значения out_p, out_q, a_p, или a_q.

Чтобы показать, что **S_p** сохраняет (3p) и (3q), заметим, что **S_p** не меняет значения a_p , a_p , s_q , или s_p.

R_p: Чтобы показать, что **R_p** сохраняет (0p), заметим, что **R_p** не делает ни одно $out_p[i]$ равным udef, и если он перевычисляет s_p, то оно впоследствии также удовлетворяет (0p).

Чтобы показать, что **R_p** сохраняет (0q), заметим, что **R_p** не меняет out_q или s_q.

Чтобы показать, что **R_p** сохраняет (lp), заметим, что **R_p** не добавляет пакеты в Q_p и не уменьшает s_q.

Чтобы показать, что **R_p** сохраняет (lq), заметим, что **R_p** не добавляет пакеты в Q_q и не уменьшает s_p.

Чтобы показать, что **R_p** сохраняет (2p), заметим, что **R_p** изменяет значение $out_p[i]$ на w при принятии < **pack,** w,i>. Т.к. Q_p содержала этот пакет до того, как выполнился **R_p**, из (1p) следует, что $w = in_p[i]$. Присваивание a_p:= max (a_p, i— l_q+1) гарантирует, что $a_p > i— l_q$ сохраняется после выполнения. Чтобы показать, что **R_p** сохраняет (2q), заметим, что **R_p** не меняет значения out_q или a_q.

Чтобы показать, что **R_p** сохраняет (3p), заметим, что когда **R_p** присваивает a_p:= max(a_p, i— l_q+1) (при принятии <pack, w,i>), из (lp) следует, что $i < s_q+l_q$, следовательно $a_p \le s_q$ сохраняется после присваивания. **R_p** не меняет s_q. Чтобы показать, что **R_p** сохраняет (3q), заметим, что s_p может быть увеличен только при выполнении **R_p**.

L_p : Чтобы показать, что **L_p** сохраняет (0p), (0q), (2p), (2q), (3p), и (3q), достаточно заметить, что **L_p** не меняет состояния процессов. (lp) и (lq) сохраняются потому, что **L_p** только удаляет пакеты (а не порождает или искажает их).

Процессы **S_q**, **R_q**, и **L_q** сохраняют P , что следует из симметрии.

3.1.2 Доказательство правильности протокола

Сейчас будет продемонстрировано, что Алгоритм 3.1 гарантирует безопасную и окончательную доставку. Безопасность следует из инварианта, как показано в Теореме 3.2, а живость продемонстрировать труднее.

Теорема 3.2 Алгоритм 3.1 удовлетворяет требованию безопасной доставки.

Доказательство. Из (0p) и (2p) следует, что $out_p[0..s_p —1] = in_q[0..s_p—1]$, а из (0q) и (2q) следует $out_p[0..Sq -1] = in_p[0..Sq -1]$.

Чтобы доказать живость протокола, необходимо сделать справедливых предположений и предположение относительно l_p и l_q. Без этих предположений протокол не удовлетворяет свойству живости, что может быть показано следующим образом. Неотрицательные константы l_p и l_q еще не определены; если их выбрать равными нулю, начальная конфигурация протокола окажется тупиковой. Поэтому предполагается, что $l_p + l_q > 0$.
Конфигурация протокола может быть обозначена $\gamma = (c_p, c_q, Q_p, Q_q)$, где c_p и c_q - состояния p и q. Пусть γ будет конфигурацией, в которой применим $\mathbf{S}_p$ (для некоторого i). Пусть
$\delta = \mathbf{S}_p(\gamma) = (c_p, c_q, Q_p, (Q_q \cup \{m\}))$,
и отметим, что действие $\mathbf{L}_q$ применимо в δ. Если $\mathbf{L}_q$ удаляет m, $\mathbf{L}_q(\delta) = \gamma$. Отношение $\mathbf{L}_q(\mathbf{S}_p(\gamma)) = \gamma$ äàåò íà÷àëî íåîãðàíè÷åííûì âû÷èñëåíèÿì, â êîòîðûõ íè s_p , ни s_q не уменьшаются.
Протокол удовлетворяет требованию «окончательной доставки», если удовлетворяются два следующих справедливых предположения.
Fl. Если посылка пакета возможна в течение бесконечно долгого временно, пакет посылается бесконечно часто.
F2. Если один и тот же пакет посылается бесконечно часто, то он принимается бесконечно часто.

Предположение Fl гарантирует, что пакет посылается снова и снова, если не получено подтверждение; F2 исключает вычисления, подобные описанному выше, когда повторная передача никогда не принимается.
Ни один из двух процессов не может быть намного впереди другого: разница между s_p и s_q остается ограниченной. Поэтому протокол называется сбалансированным, а также из этого следует, что если требование окончательной доставки удовлетворяется для s_p, тогда оно также удовлетворяется для s_q, и наоборот. Понятно, что протокол не следует использовать в ситуации, когда один процесс имеет намного больше слов для пересылки, чем другой.

Ëåììà 3.3 *Из P следует* $s_p— \ l_q \le a_p \le s_q \le a_q + l_p \le s_p + l_p$.

Äîêàçàòåëüñòâî. Из (0p) и (2p) следует $s_p— \ l_q \le a_p$, из (3p) следует $a_p \le s_p$. Из (0q) и (2q) следует $s_p \le a_p + l_p$. Из (3q) следует $a_p + l_p \le s_p + l_p$.

Òåîðåìà 3.4 *Àëãîðèòì 3.1 удовлетворяет требованию окончательной доставки.*

Äîêàçàòåëüñòâî. Сначала будет продемонстрировано, что в протоколе невозможны тупики. Из инварианта следует, что один из двух процессов может по-

слать пакет, содержащий слово с номером, меньшим, чем ожидается другим процессом.

Утверждение 3.5 *Из P следует, что посылка* < **pack,** $in[s_q]$, s_q> *процессом p или посылка*
< **pack,** $in_q[s_p]$, s_p *) процессом q возможна.*

Äîêàçàòåëüñòâî. Т.к. $l_p + l_q > 0$, хотя бы одно из неравенств Ëåììы 3.3 строгое, т.е.,
$s_q < s_p + l_p \quad \vee \quad s_p < s_q + l_q$.
Из *P* также следует $a_p \le s_q$ (3p) и $a_q \le s_p$ (3q), а также следует, что
$(a_p \le s_q < s_p + l_p) \vee (a_q \le s_p < s_q + l_q)$
это значит, что $\mathbf{S}_p$ применим с $i = s_q$ или $\mathbf{S}_q$ применим с $i = s_p$.
Теперь мы можем показать, что в каждом из вычислений s_p и s_q увеличиваются бесконечно часто. Согласно Утверждению 3.5 протокол не имеет терминальных конфигураций, следовательно каждое вычисление неограниченно. Пусть *C* - вычисление, в котором s_p и s_q увеличиваются ограниченное число раз, и пусть σ_p and σ_q - максимальные значения, которые эти переменные принимают в *C.* Согласно утверждению, посылка <**pack,** $in_p[\sigma_q]$, σ_q> процессом *p* или посылка <**pack,** $in_q[\sigma_p]$, σ_p > процессом *q* применима всегда после того, как s_p, s_q, a_p и a_q достигли своих окончательных значений. Таким образом, согласно Fl, один из этих пакетов посылается бесконечно часто, и согласно F2, он принимается бесконечно часто. Но, т.к. принятие пакета с порядковым номером s_p процессом *p* приводит к увеличению s_p (и наоборот для *q),* это противоречит допущению, что ни s_p, ни s_q не увеличиваются более. Таким образом Òåîðåìà 3.4 доказана.

Мы завершаем этот подраздел кратким обсуждением предположений Fl и F2. F2-ìèíèìàëüíîå требование, которому должен удовлетворять канал, соединяющий *p* и *q,* для того, чтобы он мог передавать данные. Очевидно, если некоторое слово $in_p[i]$ никогда не проходит через канал, то невозможно достичь окончательной доставки слова. Предположение Fl обычно реализуется в протоколе с помощью условия превышения времени: если a_p не увеличилось в течение определенного промежутка времени, $in_p[a_p]$ передается опять. Как уже было отмечено во введении в эту главу, для этого протокола безопасная доставка может быть доказана без принятия во внимания проблем времени (тайминга).

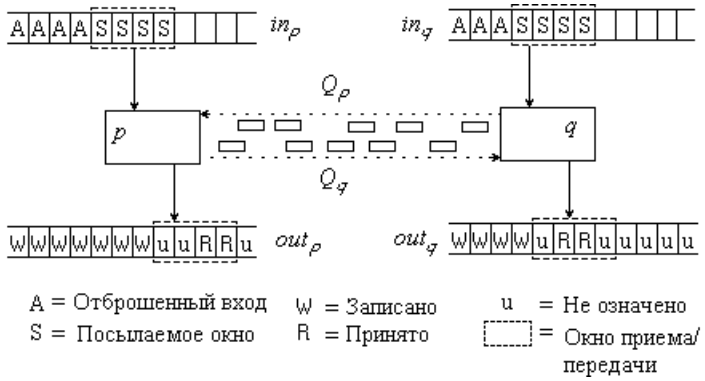
3.1.3 Обсуждение протокола

Ограничение памяти в процессах. Àëãîðèòì 3.1 не годится для реализации в компьютерной сети, т.к. в каждом процессе хранится бесконечное количество информации (массивы *in* и *out*) и т.к. он использует неограниченные порядковые номера. Сейчас будет показано, что достаточно хранить только ограниченное число слов в каждый момент времени. Пусть $L = l_p + l_q$.
Ëåììà 3.6 *Из P следует, что отправление* < **pack,** *w,i> процессом p применимо только для* $i < a_p + L$.

Äîêàçàòåëüñòâî. Сторож $\mathbf{S}_p$ требует $i < s_p + l_p$, значит согласно Ëåììе 3.3 $i < a_p + L$.

Ëåìà 3.7 *Из P следует, что если* $out_p[i] \neq udef$, *то* $i < s_p + L$.

Äîêàçàòåëüñòâî. Из (2p), $a_p > i - l_q$, значит $i < a_p + l_q$, и $i < s_p + L$ (из Ëåìы 3.3).



Ðèñóíîê 3.2 Скользящие окна протокола.

Последствия этих двух лемм отображены на Ðèñóíêå 3.2. Процессу *p* необходимо хранить только слова $in_p[a_p..s_p + l_p - 1]$ потому, что это слова, которые *p* может послать. Назовем их как *посылаемое окно p* (представлено как S на Ðèñóíêå 3.2). Каждый раз, когда a_p увеличивается, *p* отбрасывает слова, которые больше не попадают в посылаемое окно (они представлены как A на Ðèñóíêå 3.2). Каждый раз, когда s_p увеличивается, *p* считывает следующее слово из посылаемого окна от источника, который производит слова. Согласно Ëåìå 3.6, посылаемое окно процесса *p* содержит не более *L* слов.
Подобным же образом можно ограничить память для хранения процессом *p* массива out_p. Т.к. $out_p[i]$ не меняется для $i < s_p$, можно предположить, что *p* выводит эти слова окончательно и более не хранит их (они представлены как W на Ðèñóíêå 3.2). Т.к. $out_p[i] = udef$ для всех $i \geq s_p + L$, эти значения $out_p[i]$ также не нужно хранить. Подмассив $out_p[s_p..s_p + L - 1]$ назовем *принимаемое окно p*. Принимаемое окно представлено на Ðèñóíêå 3.2 как u для неозначенных слов и R для слов, которые были приняты. Только слова, которые попадают в это окно, хранятся процессом. Леммы 3.6 и 3.7 показывают, что не более *2L* слов хранятся процессом в любой момент времени.

Ограничение чисел последовательности. В заключение будет показано, что числа последовательности могут быть ограничены, если используются fifo-каналы. При использовании fifo предположения можно показать, что номер порядковый номер пакета, который получен процессом *p* всегда внутри 2L-окрестности s_p. Обратите внимание, что fifo предположение используется первый раз.

Ëåìà 3.8 *Утверждение P', определяемое как*
$P' \equiv P$
$\wedge$ <**pack,** *w, i*> *is behind* <**pack,** *w', i'*> *in* $Q_p \Rightarrow i > i' - L$ (4p)
$\wedge$ <**pack,** *w, i*> *is behind* <**pack,** *w', i'*> *in* $Q_q \Rightarrow i > i' - L$ (4q)

$\wedge$ <**pack**,w,i> $\in Q_p \Rightarrow i \geq a_p - l_p$ (5p)
$\wedge$ <**pack**,w,i> $\in Q_q \Rightarrow i \geq a_q - l_q$ (5q)

является инвариантом Àëãîðèòìà 3.1.

Äîêàçàòåëüñòâî. Т.к. уже было показано, что P - инвариант, мы можем ограничиться доказательством того, что (4p), (4q), (5p) и (5q) выполняются изначально и сохраняются при любом перемещении. Заметим, что в начальном состоянии очереди пусты, следовательно (4p), (4q), (5p) и (5q) очевидно выполняются. Сейчас покажем, что перемещения сохраняют истинность этих утверждений.

Sp: Чтобы показать, что **S**p сохраняет (4p) и (5p), заметим, что **S**p не добавляет пакетов в Q_p и не меняет a_p.
Чтобы показать, что **S**p сохраняет (5q), заметим, что если **S**p добавляет пакет <**pack,** w, i> в Q_q, то $i \geq a_p$, откуда следует, что $i \geq a_q - l_q$ (из Ëåììы 3.3).
Чтобы показать, что **S**p сохраняет (4q), заметим, что если < **pack**, w', i'> в Q_q, тогда из (lq)
$i' < s_p + l_p$, следовательно, если **S**$_p$ добавляет пакет < **pack**, w, i> с $i \geq a_p$, то из Леммы 3.3 следует $i' < a_p + L \leq i + L$.

Rp: Чтобы показать, что **R**p сохраняет (4p) and (4q), заметим, что Rp не добавляет пакеты в Q_p или Q_q.
Чтобы показать, что **R**p сохраняет (5p), заметим, что когда a_p увеличивается (при принятии <**pack,** w', i'>) до $i' - l_q + 1$, тогда для любого из оставшихся пакетов <**pack,** w, i> в Q_p мы имеем $i > i' - L$ (из 4p). Значит неравенство $i \geq a_p - l_p$ сохраняется после увеличения a_p.
Чтобы показать, что **R**p сохраняет (5q), заметим, что **R**$_p$ не меняет Q_q и a_q.

L$_p$: Действие **L**$_p$ не добавляет пакетов в Q_p или Q_q, и не меняет значения a_p или a_q; значит оно сохраняет (4p), (4q), (5p) и (5q).

Из симметрии протокола следует, что **S**$_q$, **R**$_q$ и **L**$_q$ тоже сохраняет P'.

Ëåììà 3.9 *Из P′ следует, что*
<pack, w,i>$\in$ $Q_p \Rightarrow s_p - L \leq i < s_p + L$
и
<pack, w,i>$\in$ $Q_q \Rightarrow s_q - L \leq i < s_q + L$.

Äîêàçàòåëüñòâî. Пусть <pack, w,i> $\in Q_p$. Из (lp), $i < s_q + l_q$, и из Ëåììы 3.3 $i < s_p + L$. Из (5p), $i \geq a_p — l_p$, и из Ëåììы 3.3 $i \geq s_p — L$. Утверждение относительно пакетов в Q_q доказывается так же.
Согласно Ëåììе достаточно посылать пакеты с порядковыми номерами modulo k, где
$k \geq 2L$. В самом деле, имея s_p и i mod k, p может вычислить i.

Выбор параметров. Значения констант l_p и l_q сильно влияют на эффективность протокола. Их влияние на пропускную способность протокола анализируется в [Sch91, Chapter 2]. Оптимальные значения зависят от числа системно зависимых параметров, таких как
время связи, т.е., время между двумя последовательными операциями процесса,
время задержки на обмен, т.е., среднее время на передачу пакета от p к q и получение ответа от q к p,

вероятность ошибки, вероятность того, что конкретный пакет потерян.

Протокол, чередующий бит. Интересный случай протокола скользящего окна получается, когда $L = 1$, ò.å., $l_p = 1$ и $l_q = 0$ (или наоборот). Переменные a_p и a_q, инициализируется значениями $-l_p$ и $-l_q$, а не 0. Можно показать, что $a_p + l_q = s_p$ и $a_q + l_p = s_q$ всегда выполняется, значит только одно a_p и s_p (и a_q *и* s_q) нужно хранить в протоколе. Хорошо известный *протокол, чередующий бит* [Lyn68] получается, если использование таймеров дополнительно ограничивается, чтобы гарантировать, что станции посылают сообщения в ответ.

3.2 Протокол, основанный на таймере

Теперь мы изучим роль таймеров в проектировании и проверке протоколов связи, анализируя упрощенную форму Δt-протокола Флэтчера и Уотсона (Fletcher и Watson) для сквозной передачи сообщений. Этот протокол был предложен в [FW78], но (несколько упрощенный) подход этого раздела взят из [Tel91b, Раздел 3.2]. Этот протокол обеспечивает не только механизм для передачи данных (как сбалансированный протокол скользящего окна Раздела 3.1), но также открытие и закрытие соединений. Он устойчив к потерям, дублированию и переупорядочению сообщений.

Информация о состоянии (передачи данных) протокола хранится в структуре данных, называемой *запись соединения.* (В Подразделе 3.2.1 будет показано, какая информация хранится в записи соединения). Запись соединения может быть создана и удалена для открытия и открытия соединения. Òàêèì îáðàçîì, ãîâîðÿò, ÷òî ñîåäèíåíèå îòêðûòî (на одной из станций), если существует запись соединения. Чтобы сконцентрироваться на релевантных аспектах протокола (а именно, на механизме управления соединением и роли таймеров в этом механизме), будем рассматривать упрощенную версию протокола. Более практические и эффективные расширения протокола могут быть найдены [FW78] и [Tel91b, Раздел 3.2]. В протоколе, описанном здесь, сделаны следующие упрощения.

One direction. Подразумевается, что данные передаются в одном направлении, скажем от p к *q.* Иногда будем называть *p отправителем,* а *q* - адресатом (приемником). Однако, следует отметить, что протокол использует сообщения подтверждения, которые посылаются в обратном направлении, т.е. от *q* к *p*.

Обычно данные нужно передавать в двух направлениях. Чтобы предусмотреть подобную ситуацию, дополнительно выполняется второй протокол, в котором *p* и *q* поменяны ролями. Тогда можно ввести комбинированные data/ack (данные/подтверждения) сообщения, содержащие как данные (с соответствующим порядковым номером), так и информацию, содержащуюся в пакете подтверждения протокола, основанного на таймере.

Окно приема из одного слова. Приемник не хранит пакеты данных с номером, более высоким, чем тот, который он ожидает. Только если следующий пакет, который прибудет - ожидаемый, он принимается во внимание и немедленно принимается. Более интеллектуальные версии протокола хранили бы прибывающие пакеты с более высоким порядковым номером и принимали бы их после того, как прибыли и были приняты пакеты с меньшими порядковыми номерами.

Предположения, упрощающие синхронизацию. Протокол рассмотрен с использованием минимального числа таймеров. Например, предполагается, что под-

тверждение может быть послано процессом-получателем в любое время, пока соединение открыто со стороны приемника. Также возможен случай, когда подтверждение может быть послано только в течение определенного интервала времени, но это сделало бы протокол более сложным.
Также, из описания протокола были опущены, как в Разделе 3.1, таймерные механизмы , используемые для повторной передачи пакетов данных. Включен только механизм, гарантирующий безопасность протокола.
Однословные пакеты. Отправитель может помещать только одиночное слово в каждый пакет данных. Протокол был бы более эффективным, если бы пакеты данных могли содержать блоки последовательных слов.
Протокол основан на таймере, то есть процессы имеют доступ к физическим часовым устройствам. По отношению ко времени и таймерам в системе сделаны следующие предположения.
Глобальное время. Глобальная мера времени простирается над всеми процессами системы, то есть каждое событие происходит в некоторое время. Предполагается, что каждое событие имеет продолжительность 0, и время, в которое происходит событие, не доступно процессам.
Ограниченное время жизни пакета. Время жизни пакета ограничено константой μ (*максимальное время жизни пакета).* Òàêèì îáðàçîì, если пакет посылается во время σ и принимается во время τ, то

$$\sigma < \tau < \sigma + \mu.$$

Если пакет дублируется в канале, каждая копия должна быть принята в течение промежутка времени μ после отправления оригинала (или стать потерянной).
Таймеры. Процессы не могут наблюдать абсолютное время своих действий, но они имеют доступ к таймерам. Таймер - действительная переменная процесса, чье значение непрерывно уменьшается со временем (если только ей явно не присваивают значение). Точнее, если *Xt* - таймер, мы обозначаем его значение в момент времени *t* как *Xt(t)* и если *Xt* между t_1 and t_2 не присвоено иное значение, то

$$Xt^{(t_1)} - Xt^{(t_2)} = t_2 - t_1.$$

Заметим, что эти таймеры работают так: в течение времени δ они уменьшаются точно на δ. В Подразделе 3.2.3 мы обсудим случай, когда таймеры страдают отклонением.
Входные слова для отправителя моделируются, как в Разделе 3.1, неограниченным массивом in_p. Снова этот массив не полностью хранится в *p*; *p* в каждый момент времени имеет доступ только к его части. Часть in_p, к которой *p* имеет доступ расширяется (в сторону увеличения индексов), êîãäà *p* получает следующее слово от процесса, который их генерирует. Эту операцию будем называть как *принятие* слова отправителем.
В этом разделе моделирование слов, принятых приемником, отлично от Раздела 3.1. Вместо того, чтобы записывать (бесконечный) массив, приемник передает слова процессу потребления операцией, называемой *доставка* слова. В идеале, каждое слово in_p должно быть доставлено точно один раз, и слова должны быть доставлены в правильном порядке.
Спецификация протокола, однако, слабее, и причина в том, что протокол позволяется обрабатывать каждое слово in_p только в течение ограниченного интервала времени. Не каждый протокол может гарантировать, что слово принимается за ограниченное время потому, что возможно, что все пакеты в это время поте-

ряются. Следовательно, спецификация протокола учитывает возможность *сообщенной потери*, когда протокол отправителя генерирует отчет об ошибке, указывающий, что слово возможно потеряно. (Если после этого протокол более высокого уровня предлагает это слово *p* снова, то возможно дублирование; но мы не будем касаться этой проблемы здесь.) Свойства протокола, который будет доказан в Подразделе 3.2.2:
Нет потерь. Каждое слово in_p доставляется процессом *q* или посылается отчет процессом *p* ("возможно потеряно") в течение ограниченного времени после принятия слова процессом p.
Упорядочение. Слова, доставляемые *q* принимаются в строго возрастающем порядке (так же, как они появляются в in_p).

3.2.1 Представление Протокола

Соединение в протоколе открыто, если прежде не существовало никакого соединения и если (для отправителя) принято следующее слово или (для приемника) прибывает пакет, который может быть доставлен. Таким образом, в этом протоколе, чтобы открыть соединение нет необходимости обмениваться какими-либо сообщениями управления прежде, чем могут быть посланы пакеты данных. Это делает протокол относительно эффективным для прикладных программ, где в каждом соединении передаются только несколько слов (маленькие пакеты связи). Предикат *cs* (или *cr*, соответственно) истинен, когда отправитель (или приемник, соответственно) имеет открытое соединение. Это, обычно, не явная булева переменная отправителя (или приемника, соответственно); вместо этого открытое соединение определяется существованием записи соединения. Процесс проверяет, открыто ли соединение, пытаясь найти запись соединения в списке открытых соединений.
Когда отправитель открывает новое соединение, он начинает нумеровать принятые слова с 0. Количество уже принятых слов в данном соединении обозначается *High,* и количество слов, для которых уже было получено подтверждение обозначается через *Low.* Это подразумевает (аналогично протоколу Раздела 3.1), что отправитель может передавать пакеты с порядковыми номерами в диапазоне от *Low* до *High* — 1, но есть здесь и своя особенность. Отправитель может посылать слово только в течение промежутка времени длиной *U,* начиная с того момента, когда отправитель принял слово. Для этого с каждым словом $in_p[i]$ *ассоциируется таймер Ut[i]*, он устанавливается в *U* в момент принятия, и должен быть положительным для передаваемого слова. Òàêèì îáðàçîì, посылаемое окно *p* состоит из тех слов с индексами *Low ...High* - 1, для которых ассоциированный с ними таймер положителен.

Сетевые константы:
μ : real ; (* Максимальное время жизни пакета *)
Константы протокола:
U : real ; (* Длина интервала отправки *)
R : real ; (* Значение тийм-аута приемника: $R \geq U+\mu$ *)
S : real ; (* Значение тайм-аута отправителя: $S \geq R + 2\mu$ *)
Запись соединения отправителя:
Low : integer ; (* Подтвержденные слова текущего соединения *)
High : integer ; (* Принятые слова текущего соединения *)
St : timer ; (* Таймер соединения *)

Запись соединения приемника:
Exp : integer ; (* Ожидаемый порядковый номер *)
Rt : timer ; (* Таймер соединения *)
Подсистема связи:
Mq : channel ; (* Пакеты данных для *q* *)
Mp : channel ; (* Пакеты подтверждения для *p* *)
Вспомогательные переменные:
B : integer **init** 0 ; (* Слова в предыдущем соединении *)
cr : boolean **init** false ; (* Существование соединения для приемника *)
cs : boolean **init** false ; (*Существование соединения для отправителя *)

Ðèñóíîê **3.3** Переменные протокола, основанного на таймере.

Протокол посылает пакеты данных, состоящие из: бита (бит *начала-последовательности*; его значение будет обсуждаться позже), порядкового номера и слова. Для анализа протокола каждый пакет данных содержит четвертое поле, называемое *оставшееся время жизни пакета.* Оно показывает максимальное время, в течение которого пакет еще может находиться в канале до того, как он должен быть принят или стать потерянным согласно предположению об ограниченном времени жизни. В момент отправления оставшееся время жизни пакета всегда равно μ. Пакеты подтверждения протокола состоят только из порядкового номера, ожидаемого процессом *q,* но опять для целей анализа каждое подтверждение содержит оставшееся время жизни пакета.

Ap: (* Принятие следующего слова *)
begin if not *cs* **then**
begin (* Сначала соединение открывается *)
create *(St, High, Low) ;* (* *cs* := true *)
Low := High := 0 ; *St := S*
end;
Ut[B + High] := U, High := High + 1
end

Sp: (* Отправление *i*-го слова текущего соединения *)
{ *cs* /\ *Low* ≤ *i* < *High* /\ *Ut*[B + *i*] > *0}*
begin
send <**data**, (i = *Low),i, in*$_p$[B + i],μ > ;
St:=S
end

R*p***:** (* Принятие подтверждения *)
{ *cs* /\ <**ack,** *i,* ρ >∈ *Mp* }
begin receive <**ack,** *i,* ρ > ; *Low* := max *(Low, i)* **end**

E*p*: (* Генерация сообщения об ошибке для возможно потерянного слова *)
{cs /\ *Ut[B + Low] ≤ -2*μ *-R}*
begin *error* [B + *Low] := true* ; *Low* := *Low* + 1 **end**

C*p*: (* Закрытие соединения *)
{cs /\ *St < 0* /\ *Low = High }*
begin *B := B + High* , delete *(St, High, Low)* **end**
(* *cs* := false *)

Àëãîðèòì 3.4 Протокол отправителя.

Закрытие соединения контролируется таймерами, таймером *St* для отправителя и таймером *Rt* для приемника. Ограниченный интервал посылки каждого слова и ограниченное время жизни пакета приводят к тому, что каждое слово может быть найдено в каналах только лишь в течение интервала времени длиной $\mu + U$, начиная с момента принятия слова. Это позволяет приемнику отбрасывать информацию о конкретном слове через $\mu + U$ единиц времени после принятия слова; после этого не могут появиться дубликаты, следовательно не возможна повторная доставка. Таймер *Rt* устанавливается в *R* каждый раз, когда слово доставляется, константа *R* выбирается так, чтобы удовлетворять неравенству $R \geq U + \mu$. Если следующее слово принимается в течение *R* единиц времени, то таймер *Rt* обновляется, иначе соединение закрывается. Значение таймера отправители выбирается так, чтобы невозможно было принять подтверждение при закрытом соединении; для этого, соединение поддерживается в течение по крайней мере *S* единиц времени после отправления пакета, где *S* - константа, выбираемая так, чтобы удовлетворять $S \geq R+2\mu$. Таймер *St* устанавливается в *S* каждый раз, когда посылается пакет, и соединение может быть закрыто только если $St < 0$. Если к этому времени еще остались незавершенные слова (ò.å. слова, для которых не было получено подтверждение), эти слова объявляются потерянными до закрытия соединения.

R_q: (* Принимаем пакет данных *)
{ <**data**, *s, i,w,ρ* > ∈ *Mq* }
begin receive <**data**, *s, i,w,ρ* > ;
if *cr* **then**
if *i* = *Exp* **then**
begin *Rt := R* ; *Exp := i* + 1 ; deliver *w* **end**
else if *s* = *true* **then**
begin create *(Rt, Exp)* ; (* *cr* := true *)
Rt := R ; Exp := i +1 ; deliver *w*
end
end

S_q: (* Посылаем подтверждение *)
{*cr*}
begin send <**ack**, *Exp,* μ > **end**
(* Закрытие соединения по истечении времени *Rt*, см. В действии **Time** *)

Àëãîðèòì 3.5 Протокол приемника

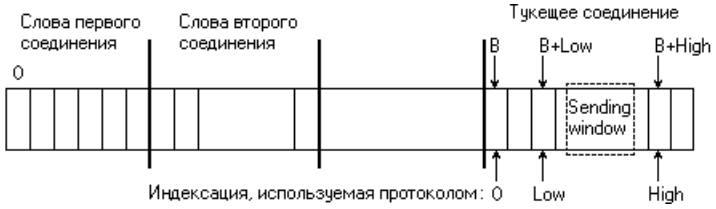
Бит начало-последовательности используется приемником, если пакет получен при закрытом соединении, чтобы решить, может ли быть открыто соединение (и доставлено слово в пакете). Отправитель устанавливает бит в true, если все предыдущие слова были подтверждены или объявлены (как возможно потерянные). Когда *q* получает пакет при уже открытом соединении, содержащееся слово доставляется тогда и только тогда, когда порядковый номер пакета равен ожидаемому порядковому номеру (хранится в Exp).
Остается обсудить значение переменной *B* в протоколе отправителя. Это вспомогательная переменная, введенная только с целью доказательства правильности протокола. Отправитель нумерует слова в каждом соединении, начиная с 0, но, чтобы различать слова в различных соединениях, все слова индексируются последовательно по возрастанию для анализа протокола. Таким образом, там, где отправитель индексирует слово как *i*, "абсолютный" номер указанного слова

$B + i$, где B - общее количество пакетов, принятых p в предыдущих соединениях. Соответствие между "внутренними" и "абсолютными" номерами слов показывается на Рисунке 3.7. В реализации протокола B не хранится, и отправитель "забывает" все слова in_p [0 .. B-1].

```
    Loss: { m ∈ M }   (* M - либо M_p, либо M_q *)
begin remove m from M end
    Dupl: { m ∈ M }   (*M - либо M_p, либо M_q *)
begin insert m in M end
    Time: (* δ > 0 *)
begin forall i do Ut[i] := Ut[i] -δ,
St := St -δ ; Rt := Rt - δ;
if Rt ≤ 0 then delete (Rt, Exp) ; (* cr := false *)
forall <.., ρ> ∈ M_p, M_q do
          begin ρ := ρ— δ;
                if ρ ≤ 0 then remove packet
          end
end
```

Àëãîðèòì 3.6 Дополнительные переходы Протокола.

Подсистема связи представляется двумя мультимножествами, M_p для пакетов с адресатом p и M_q для пакетов с адресатом q. Протокол отправителя - Алгоритм 3.4, протокол приемника - Алгоритм 3.5. Имеются дополнительные переходы системы, представленные Алгоритмом 3.6, которые не соответствуют шагам в протоколе процессов. Эти переходы представляют собой отказы канала и изменение времени. В переходах **Loss** и **Dupl** M означает или M_p, или M_q. Действие **Time** уменьшает все таймеры в системе на величину δ, это случается между двумя дискретными событиями, которые отличаются на δ единиц времени. Когда таймер приемника достигает значения 0, соединение закрывается.



Ðèñóíîê 3.7 Порядковые номера протокола.

3.2.2 Доказательство корректности протокола

Требуемые свойства протокола будут доказаны в серии лемм и теорем. Утверждение P_0, которое определено ниже, показывает, что соединение отправителя остается открытым пока в системе еще есть пакеты, и что порядковые номера этих пакетов имеют корректное значение в текущем соединении.

$P_0 \equiv \quad cs \Rightarrow St \le S$ (1)

$cr \Rightarrow 0 < Rt \le R$ (2)

$\forall i < B + High : Ut[i] \le U$ (3)

$\forall <... \rho> \in M_p, M_q : 0 < \rho \le \mu$ (4)

$< \mathbf{data},\ s,\ i,w,\ \rho > \in M_q \Rightarrow cs \wedge St \geq \rho + \mu + R$ (5)

$cr \Rightarrow cs \wedge St \geq Rt + \mu$ (6)

$< \mathbf{ack}, i,\ \rho > \in M_p \Rightarrow cs \wedge St > \rho$ (7)

$< \mathbf{data},\ s,\ i,\ w,\ \rho > \in M_q \Rightarrow (w = in_p[B + i] \wedge i < High)$ (8)

Объяснение к (3): значение *High* предполагается равным нулю во всех конфигурациях, в которых со стороны приемника нет соединения.

Ëåììà 3.10 *P_0 - инвариант протокола, основанного на таймере.*
Äîêàçàòåëüñòâî. Первоначально не соединения, нет пакетов, и $B = 0$, из чего следует, что P_0 - true.

$\mathbf{A}_p$: (1) сохраняется, т.к. *St* всегда присваивается значения *S* ($St = S$). (3) сохраняется, т.к. перед увеличением *High, Ut[B + High]* присваивается значение *U*. (5), (6) и (7) сохраняются, т.к. *St* может только увеличиваться. (8) сохраняется, т.к. *High* может только увеличиваться.

$\mathbf{S}_p$: (1) сохраняется, т.к. *St* всегда присваивается значения *S*. (4) сохраняется, т.к. каждый пакет посылается с оставшимся временем жизни равным μ. (5) сохраняется, т.к. пакет $<.., \mu>$ посылается и *St* устанавливается в *S,* и $S = R + 2\mu$. (6) и (7) сохраняется, т.к. *St* может только увеличиться в этом действии. (8) сохраняется, т.к. новый пакет удовлетворяет $w = in_p[B + i]$ и $i < High$.

$\mathbf{R}_p$: Действие $\mathbf{R}_p$ не меняет никаких переменных из P_0, и удаление пакета сохраняет (4) и (7).

$\mathbf{E}_p$: Действие $\mathbf{E}_p$ не меняет никаких переменных из P_0.

$\mathbf{C}_p$: Действие $\mathbf{C}_p$ делает равным false заключения (5), (6) и (7), но ((2), (5), (6) и (7)) применимы только когда их посылки ложны. $\mathbf{C}_p$ также меняет значение *B,* но, т.к. пакетов для передачи нет, (по (5) и (7)), (8) сохраняется.

$\mathbf{R}_q$**:** (2) сохраняется, т.к. *Rt* всегда присваивается значение *R* (если присваивается). (6) сохраняется, т.к. *Rt* устанавливается только в *R* только при принятии пакета $<\mathbf{data}, s,i,w,\rho>$, è из (4) и (5) следует $cs \wedge St \geq R + \mu$ когда это происходит.

$\mathbf{S}_q$*:* (4) сохраняется, т.к. каждый пакет посылается с оставшимся временем жизни, равным μ. (7) сохраняется, т.к. пакет $< \mathbf{ack}, i, \rho >$ посылается с $\rho = \mu$ когда *cr* истинно, так что из (2) и (6) $St > \mu$.

Loss: (4), (5), (7) и (8) сохраняются, т.к. удаление пакета может фальсифицировать только их посылку.

Dupl: (4), (5), (7) и (8) сохраняются, т.к. ввод пакета *m* применимо только если *m* уже был в канале, из чего следует, что заключение данного предложения было истинным и перед введением.

Time: (1), (2) и (3) сохраняются, т.к. *St, Rt,* и *Ut[i]* может только уменьшаться, и соединение приемника закрывается, когда *Rt* становится равным 0. (4) сохраняются, т.к. ρ может только уменьшиться, и пакет удаляется, когда его ρ-поле достигает значения 0. Заметим, что **Time** уменьшает все таймеры (включая ρ-поле пакета) на *одну и ту же* величину, значит сохраняет все утверждения вида $Xt \geq Yt + C$, где *Xt* и *Yt* - таймеры, и *C* - константа. Это показывает, что (5), (6) и (7) сохраняются.

Первое требование к протоколу в том, что каждое слово в конце концов доставляется или объявляется потерянным. Определим предикат *0k(i)* как
0k(i) $\Leftrightarrow$ *error* [*i*] = *true* $\vee$ *q* доставил in_p *[i]*.
Сейчас может быть показано, что протокол не теряет никаких слов, не объявляя об этом. Определим утверждение P_1 как

$P_1 \equiv P_0$
$\wedge \neg cs \Rightarrow \forall\ i < B$: *0k(i)* (9)
$\wedge\ cs \Rightarrow \forall\ i < B + Low$: *0k(i)* (10)
$\wedge <$**data**,*true,I,w,*$\rho > \in M_q \Rightarrow \forall i < B+I$: *0k(i)* (11)
$\wedge\ cr \Rightarrow \forall\ i < B+ Exp$: *0k(i)* (12)
$\wedge <$**ack**,*I,*$\rho > \in M_p \Rightarrow \forall\ i < B+I$: *0k(i)* (13)

Ëåìà 3.11 P_1 - *èíâàðèàíò ïðîòîêîëà, îñíîâàííîãî íà òàéìåðå.*
Äîêàçàòåëüñòâî. Сначала заметим, что как только *0k(i)* стало true для некоторого *i,* он никогда больше не становится false. Сначала нет соединения, нет пакетов, и *B* = 0, откуда следует, что P_1 выполняется.

A$_p$: Действие **A**$_p$ может открыть соединение, но при этом сохраняется (10), т.к. соединение открывается с *Low* = 0 и $\forall i < B$: *0k(i)* выполняется из (9).

S$_p$: Действие **S**$_p$ может послать пакет < **data,** *s, I, w,* $\rho >$, но т.к. *s* истинно только при *I = Low,* то это сохраняет (11) из (10).

R$_p$: Значение *Low* может быть увеличено, если принят пакет < **ack,** *I,* $\rho >$. Тем не менее, (10) сохраняется, т.к. из (13) $\forall i < B + I$: *0k(i)* выполняется, если получено это подтверждение.

E$_p$: Значение *Low* может быть увеличено, когда применяется действие **E**$_p$, но генерация сообщения об ошибке гарантирует, что (10) сохраняется.

C$_p$: Действие **C**$_p$ обращает *cs* в false, но оно применимо только если *St < 0* и *Low == High.* Из (10) следует, что $\forall i < B+ High$: *0k(i)* выполняется прежде выполнения **C**$_p$, следовательно (9) сохраняется. Посылка (10) обращается в false в этом действии, и из (5), (6) и (7) следует, что посылки (11), (12) и (13) ложны; следовательно (10), (11), (12) и (13) сохраняются.

R$_q$: Сначала рассмотрим случай, когда *q* принимает < **data**, *true,l,w,*ρ> при не существующем соединении (*cr* - false). Тогда $\forall i < B+I$: *0k(i)* из (11), и *w* доставляется в действии. Т.к.
$w = in_p[B+I]$ из (8), присваивание *Exp* := *I* + 1 сохраняет (12).

Теперь рассмотрим случай, когда *Exp* увеличивается в результате принятия
< **data,** *s,Exp,w,*ρ> при открытом соединении. Из (12), $\forall i < B + Exp$: *0k(i)* выполнялось перед принятием, и слово $w = Wp[B + Exp]$ доставляется действием, следовательно приращение *Exp* сохраняет (12).

S$_q$*:* Отправление <**ack**, *Exp,* μ> сохраняет (13) из (12).

Loss: Выполнение **Loss** может только фальсифицировать посылки предложений.

Dupl: Введение пакета *m* возможно только если посылка соответствующего предложения (и, следовательно, заключение) была истинна еще до введения.

Time: Таймеры не упоминались явно в (9)-(13). Выведение пакета или закрытие процессом *q* может только фальсифицировать посылки (11), (12) или (13).

Теперь может быть доказана первая часть спецификации протокола, но после дополнительного предположения. Без этого предположения отправитель может быть чрезвычайно ленивым в объявлении слов возможно потерянными; в Àëãîðèòìе 3.4 указано только, что это сообщение может и *не возникнуть* в промежуток времени $2\mu + R$ после окончания интервала для отправления слова, но не указано, что оно вообще должно появиться. Итак, позвольте сделать дополнительное предположение, что действие $\mathbf{E}_p$ на самом выполниться процессом *p* и в течение разумного времени, а именно прежде, чем $Ut[B + Low] = —2\mu — R—\lambda$.

Òåîðåìà 3.12 (Нет потерь) *Каждое слово in_p доставляется q или объявляется p как возможно потерянное в течение $U+2\mu+R+\lambda$ после принятия слова процессом p.*

Äîêàçàòåëüñòâî. После принятия слова $in_p[I]$, $B+High > I$ начинает выполняться. Если соединение закрывается в течение указанного периода после принятия слова $in_p[I]$, то $B > I$, и результат следует из (9). Если соединение не закрывается в этот промежуток времени и $B + Low \le I$, отчет обо всех словах из промежутка $B + Low..I$ возможен ко времени $2\mu + R$ после окончания интервала отправления $in_p[I]$. Из этого следует, что этот отчет имел место $2\mu + R + \lambda$ после окончания интервала отправления, ò.å., $U+ 2\mu + R+\lambda$ после принятия. Из этого также следует $I < B+ Low$, и, значит, слово было доставлено или объявлено (из (10)).

Чтобы установить второе требование корректности протокола, должно быть показано, что каждое принимаемое слово имеет больший индекс (в in_p), чем ранее принятое слово. Обозначим индекс самого последнего доставленного слова через *pr* (для удобства запишем, что изначально
$pr = -1$ and $Ut[-1] = -\infty$). Определим утверждение P_2 как:

$P_2 \equiv P_1$

$\wedge \langle \mathbf{data}, s, i, w, \rho \rangle \in M_q \Rightarrow Ut[B+i] > \rho - \mu$ (14)

$\wedge i_1 \le i_2 < B + High \Rightarrow Ut[i_1] \le Ut[i_2]$ (15)

$\wedge cr \Rightarrow Rt \ge Ut[pr] + \mu$ (16)

$\wedge pr < B + High \wedge (Ut[pr] > -\mu \Rightarrow cr)$ (17)

$\wedge cr \Rightarrow B + Exp = pr + 1$ (18)

Ëåììà 3.13 *P_2 - инвариант протокола, основанного на таймере.*

Äîêàçàòåëüñòâî. Изначально M_q пусто, $B + High$ равно нулю, ¬cr выполняется, и $Ut[pr] < -\mu$, откуда следуют (14)-(18).

$\mathbf{A}_p$: (15) сохраняется, т.к. каждое новое принятое слово получает значение таймера *U,* что из (3) по крайней мере равно значениям таймеров ранее принятых слов.

$\mathbf{S}_p$: (14) сохраняется, т.к. $Ut[B +i] > 0$ и пакет отправляется с $\rho=\mu$.

$\mathbf{C}_p$: (14), (16) и (18) сохраняются, т.к. из (5) è (6) их посылки ложны, когда $\mathbf{C}_p$ применимо. (15) сохраняется, т.к. *B* принимает значение *B* + *High* è таймеры не меняются. (17) сохраняется, т.к. *B* присваивается значение *B* + *High* è *pr* è *cr* не меняются.

$\mathbf{R}_q$: (16) сохраняется, т.к. когда *Rt* устанавливается в *R* (при принятии слова) $Ut[pr] \le U$ из (3), è $R \ge 2\mu + U$. (17) сохраняется, т.к. $pr < B + High$, что следует из (8), è *cr* становится true. (18) сохраняется, т.к. *Exp* устанавливается в *i* +1 è *pr* в *B* + *i,* откуда следует, что (18) становится true.

Time: (14) сохраняется, т.к. $Ut[B + i]$ è ρ уменьшаются на одно и то же число (è выведение пакета только делает ложной посылку). (15) сохраняется, т.к. $Ut[i_1]$ è $Ut[i_2]$ уменьшаются на одну и туже величину. (16) сохраняется, т.к. *cr* не становится истинным в этом действии, è *Rt* è $Ut[pr]$ уменьшаются на одну и ту же величину. (17) сохраняется, т.к. его заключение становится ложным только, если *Rt* становится ≤ 0, откуда следует (по (16)), что $Ut[pr]$ становится $< -\mu$. (18) сохраняется, т.к., если *cr* не обратился в false, *B, Exp* è *pr* не меняются.

Действия $\mathbf{R}_p$, $\mathbf{E}_p$, è $\mathbf{S}_q$, не меняют никакие переменные в (14)-(18). **Loss** è **Dupl** сохраняют (14)-(18) исходя из тех же соображений, что и в предыдущих доказательствах.

Ëåììà 3.14 *Из P_2 следует, что*
$< \mathbf{data}, s, i_1, w, \rho > \in M_q \Rightarrow (cr \vee B + i_1 > pr)$.
Äîêàçàòåëüñòâî. По (14), из $<\mathbf{data}, s, i_1, w, \rho > \in M_q$ следует $Ut[B+i_1] > \rho - \mu > -\mu$.
Если $B + i_1 \le pr$ то, т.к. $pr < B + High$ из (15), $Ut[pr] > -\mu$, так что из (17) *cr* true.

Òåîðåìà 3.15 (Упорядочение) *Слова, доставляемые q появляются в строго возрастающем порядке в массиве in_p.*
Äîêàçàòåëüñòâî. Предположим *q* получает пакет $<\mathbf{data}, s, i_1, w, \rho >$ è доставляет *w*. Если перед получением не было соединения, $B + i_1 > pr$ (по Ëåììе 3.14), так что слова *w* располагается в in_p после позиции *pr*. Если соединение было, $i_1 = Exp$, значит $B + i_1 = B + Exp = pr + 1$ из (18), откуда следует, что $w = in_p[pr+1]$.

3.2.3 Обсуждение протокола

Некоторые расширения протокола уже обсуждались во введении в этот раздел. И мы заканчиваем раздел дальнейшим обсуждением протокола и методов, представленных и используемых в этом разделе.

Качество протокола. Требования *Нет потерь* è *Упорядочение* являются свойствами безопасности, è они позволяют получить чрезвычайно простое решение, а именно протокол, который не посылает или получает никакие пакеты, и объявляет каждое слово потерянным. Само собой разумеется, что такой протокол, который не дает никакой транспортировки данных от отправителя к приемнику, не является очень "хорошим" решением.

Хорошие решения проблемы не только удовлетворяют требованиям *Нет потерь и Упорядочение*, но также объявляют потерянными как можно меньше слов. Для этой цели, протокол этого раздела может быть расширен механиз-

мом, который посылает каждое слово неоднократно (пока не конец посылки интервала), пока не получит подтверждение. Интервал посылки должен быть достаточно длинным, чтобы можно было повторить передачу некоторого слова несколько раз, и чтобы вероятность, что слово потеряется, стала очень маленькой.
На стороне приемника предусмотрен механизм, который вызывает посылку подтверждения всякий раз, когда пакет доставлен или получен при открытом соединении.

Ограниченные порядковые номера. Порядковые номера, используемые в протоколе, могут быть ограничены, если получить для протокола результат, аналогичный Ëåììe 3.9 для сбалансированного протокола скользящего окна [Tel91b, Section 3.2]. Для этого нужно предположить, что скорость принятия слов (процессом *p)* ограничена следующим образом: слово может быть принято только если первое из предыдущих слов имеет возраст по крайней мере $U + 2\mu + R$ единиц времени. Для этого нужно к действию $\mathbf{A}_p$ добавить сторож $\{(High < L) \vee (Ut[B + High - L] < -R - 2\mu)\}$.
Учитывая это предположение, можно показать, что порядковые номера принимаемых пакетов лежат в 2L-окрестности вокруг *Exp,* è порядковые номера подтверждений - в L-окрестности вокруг *High.* Следовательно, можно передавать порядковые номера modulo *2L.*
Форма действий и инвариант. Благодаря использованию утверждений, рассуждения относительно протокола связи уменьшены до (большого) манипулирования формулами. Манипулирование формулами - "безопасная" методика потому, что каждый шаг может быть проверен в очень подробно, так что возможность сделать ошибку в рассуждениях мала. Но есть риск, что читатель может потеряет идею протокола и его отношение к рассматриваемым формулам. Проблемы проектирования протокола могут быть поняты и с прагматической, и с формальной точки зрения. Fletcher и Watson [FW78] утверждают, что упраф-ляющая информация должна быть "защищена" в том смысле, что ее значение не должно измененяться потерей или дублированием пакетов; это - прагматическая точка зрения. При использовании в проверке утверждений, "значение" информации управления отражено в выборе специфических утверждений в качестве инвариантов. Выбор этих инвариантов и проектирование переходов, сохраняющих их, составляет формальную точку зрения. Действительно, как будет показано, наблюдение Fletcher и Watson может быть вновь показано в терминах "формы" формул, которые могут или не могут быть выбран как инварианты протокола, устойчивые к потере и дублированию пакетов.

Time-ε: $\{ \delta > 0 \}$
begin (* Таймеры в *p* уменьшаются на δ' *)
$\delta' := \ldots$; (* $\frac{\delta}{1+\varepsilon} \le \delta' \le \delta \times (1 + \varepsilon)$ *)
forall i do $Ut[i] := Ut[i] - \delta'$;
$St := St - \delta'$;
(*Таймеры в *q* уменьшаются на δ' *)
$\delta'' := \ldots$; (* $\frac{\delta}{1+\varepsilon} \le \delta'' \le \delta \times (1 + \varepsilon)$ *)
$Rt := Rt - \delta''$;
if $Rt < 0$ **then** delete *(Rt, Exp)* ;
(* ρ -поле передается явно *)

```
forall (..,ρ) ∈ Mp, Mq do
        begin ρ := ρ - δ,
        if ρ < 0 then remove packet
        end
end
```

Àëãîðèòì 3.8 Измененное действие **Time**.

Все инвариантные предложения P_2 относительно пакетов имеют форму
$\forall m \in M : A(m)$
è в самом деле легко видеть, что подобное предложение сохраняется при дублировании и потере пакетов. В дальнейших главах мы увидим инварианты в более общей форме, например

$$\sum_{m \in M} f(m) = K$$

èëè
условие $\Rightarrow \exists m \in M : A(m)$.
Утверждения, имеющие этй форму могут быть фальсифицированы потерей или дублированием пакетов, è следовательно не могут использоваться в дîêàçàòåëüñòâе корректности Àëãîðèòìов, которые должны допускать подобные дефекты.
Подобные же наблюдения применимы к форме инвариантов в действии **Time**. Уже было отмечено, что это действие сохраняет все утверждения формы $Xt \geq Yt + C$,
где Xt è Yt -таймеры è C -константа.

$P_1' = \ cs \Rightarrow St \leq S$	(1′)	
$\wedge\ cr \Rightarrow 0 < Rt \leq R$	(2')	
$\wedge\ \forall i < B + High : Ut[i] < U$	(3')	
$\wedge\ \forall <.., \rho > \in M_p, M_q : 0 < \rho \leq \mu$	(4')	
$\wedge$ <**data**, s, *i, w,* $\rho >\in M, \Rightarrow cs \wedge St \geq (1+\varepsilon)(\rho + \mu + (1+\varepsilon)R)$	(5')	
$\wedge\ cr \Rightarrow cs \wedge St \geq (1+\varepsilon)((i+\varepsilon)Rt+\mu)$	(6')	
$\wedge$ < **ack**, *i,* $\rho >\in M_p \Rightarrow cs \wedge St > (1+\varepsilon)\times\rho$	(7')	
$\wedge$ <**data**, *s, i, w,* $\rho > \in M_q, \Rightarrow$ (w = $in_p[B + i] \wedge i < High$)	(8')	
$\wedge\ \neg cs \Rightarrow \vee i < B: 0k(i)$		(9')
$\wedge\ cs \Rightarrow \vee i < B + Low : 0k(i)$	(10')	
$\wedge$<**data**,*true,I,w,* $\rho)\in M_q \Rightarrow \forall i<B+I: 0k(i)$	(11')	
$\wedge\ cr \Rightarrow \forall i < B + Exp : 0k(i)$	(12')	
$\wedge$ <**ack**,*I,* $\rho >\in M_p \Rightarrow \forall i <B+I: Ok(i)$	(13')	
$\wedge$ <**data**, *s, i, w,* $\rho) \in M_q \Rightarrow Ut[B+i] > (l+\varepsilon)(\rho - \mu)$	(14')	
$\wedge\ i_1 \leq i_2 < B + High \Rightarrow Ut[i_1] < Ut[i_2]$	(15')	
$\wedge$ cr $\Rightarrow Rt \geq (1+\varepsilon)((1+\varepsilon)\ Ut[pr] + (1+\varepsilon)^2\ \mu)$	(16')	
$\wedge\ pr < B + High \wedge Ut[pr] > -(1+\varepsilon)\mu \Rightarrow cr$		(17')
$\wedge\ cr \Rightarrow B + Exp = pr+1$	(18')	

Ðèñóíîê **3.9** инвариант протокола с отклонением таймеров.

Неаккуратные таймеры. Действие **Time** моделирует идеальные таймеры, которые уменьшаются точно на δ в течение δ единиц времени, на на практике таймеры страдают неточности, называемой *отклонением.* Это отклонение всегда предполагается ε-*ограниченным*, ãääå ε-известная константа, что означает, что в течение δ единиц времени таймер уменьшается на величину δ', которая

удовлетворяет $\delta/(1+\varepsilon) \le \delta' \le \delta \times (1+\varepsilon)$. (Обычно ε бывает порядка 10^{-5} или 10^{-6}.) Такое поведение таймеров моделируется действием **Time-ε**, приведенном в Àëãîðèòìå 3.8.

Было замечено, что **Time** сохраняет утверждения специальной формы $Xt \ge Yt + C$ потому, что таймеры обеих частей неравенства уменьшаются на в точности одинаковую величину, è из

$Xt \ge Yt + C$ следует $(Xt - \delta) \ge (Yt - \delta) + C$. Такое жн наблюдение может быть сделано для **Time-ε**. Для действительных чисел $Xt, Yt, \delta, \delta', \delta'', r,$ è c, удовлетворяющих $\delta > 0$ è $r > 1$, из

$$(Xt \ge r^2\ Yt + c) \wedge (\frac{\delta}{r} \le \delta' \le \delta \times r) \wedge (\frac{\delta}{r} \le \delta'' \le \delta \times r)$$

следует

$(Xt - \delta') \ge r^2(Yt - \delta'') + c$.

Следовательно, **Time-ε** сохраняет утверждение формы

$Xt \ge (1+\varepsilon)^2\ Yt + c$.

Теперь протокол может быть адаптирован к работе с отклоняющимися таймерами, если соответствующим образом изменить инварианты. Для того, чтобы другие действия тоже сохраняли измененные инварианты, константы R è S протокола должны удовлетворять

$R \ge (1+\varepsilon)((1+\varepsilon)U + (I+\varepsilon)^2)$ è $S \ge (1+\varepsilon)(2\mu + (1+\varepsilon)R)$.

Исключая измененные константы, протокол остается таким же. Его инвариант приведен на Ðèñóíêå 3.9.

Òåîðåìà 3.16 *P_2'- инвариант протокола, основанного на таймере с* ε-*ограниченным отклонением таймера. Протокол удовлетворяет требованиям Нет потерь и Упорядочение.*

Упражнения к главе 3

Раздел 3.1

Óïðàæíåíèå 3.1 *Покажите, что сбалансированный протокол скользящего окна не удовлетворяет требованию окончательной доставки, если из предположений Fl è FS, выполняется только F2.*

Óïðàæíåíèå 3.2 *Докажите, что если $L = 1$ в сбалансированном протоколе скользящего окна è a_p è a_q, инициализируются значениями $-l_q$ è $-l_p$, то всегда верно $a_p + l_q = s_p$ è $a_q + l_p = s_q$.*

Раздел 3.2

Óïðàæíåíèå 3.3 *В протоколе, основанном на таймере отправитель может объявить слово возможно потерянным, когда на самом деле оно было корректно доставлено приемником.*

(1) *Опишите выполнение протокола, при котором возникает этот феномен.*

(2)*Можно ли спроектировать протокол, в котором отправитель генерирует сообщение об ошибке в течение ограниченного промежутка времени,* тогда и только тогда, когда *слово не доставлено приемником?*

Óïðàæíåíèå 3.4 *Предположим, что из-за выхода из строя часового устройства, приемник не может закрыть соединение вовремя. Опишите работу протокола, основанного на таймере, когда слово теряется без сообщений отправителя.*

Óïðàæíåíèå 3.5 *Опишите работу протокола, основанного на таймере, в котором приемник открывает соелинение при принятии пакета с порядковым номером, большим нуля.*
Óïðàæíåíèå 3.6 *Действие* **Time-ε** *не моделирует отклонение в оставшемся времени жизни пакетов. Почему?*

Óïðàæíåíèå 3.7 *Докажите Теорему 3.16.*

Óïðàæíåíèå 3.8 *Инженер сети хочет использовать протокол, основанный на таймере, но хочет, чтобы отчет о возможно потерянных словах приходил раньше, в соответствии со следующей модификацией* $\mathbf{E}_p$.
$\mathbf{E}_p$: *(* Генерация сообщения об ошибке для возможно потерянных слов *)*
{ Ut[B + Low] < 0 *}*
begin *error[B + Low] := true ; Low := Low* + 1 **end**

Продолжает ли màêèì îáðàçîì измененный протокол удовлетворять требованиям Нет потерь и Упорядочение или должны быть сделаны какие-то изменения? Укажите преимущества и недостатки этих изменений.

4 Алгоритмы маршрутизации

Процесс (узел в компьютерной сети), вообще, не соединен непосредственно с каждым другим процессом каналом. Узел может посылать пакеты информации непосредственно только к подмножеству узлов называемых *соседями* узла. *Маршрутизация* - термин, используемый для того, чтобы описать решающую процедуру, с помощью которой узел выбирает один (или, иногда, больше) соседей для посылки пакета, продвигающегося к конечному адресату. Цель в проектировании алгоритма маршрутизации - сгенерировать (для каждого узла) процедуру принятия решения для выполнения этой функцию и предоставление гарантии для каждого пакета.

Ясно, что некоторая информация относительно топологии сети должна быть сохранена в каждом узле как рабочая основа для (локальной) решающей процедуры; мы обратимся к такой информации как *таблицы маршрутизации*. С введением этих таблиц проблема маршрутизации может быть разделена в две части.

1. *Вычисление таблицы*. Таблицы маршрутизации должны быть вычислены, когда сеть инициализирована и должна быть изменена, если топология сети изменилась.
2. *Пересылка пакета*. Пакет должен быть послан через сеть, используя таблицы маршрутизации.

Критерии для "хороших" методов маршрутизации включают следующие.

(1) *Корректность*. Алгоритм должен доставить каждый пакет, предложенный сети окончательному адресату.

(2) *Комплексность*. Алгоритм для вычисления таблиц должен использовать несколько сообщений, время, и память (хранение) насколько возможно.

(3) *Эффективность*. Алгоритм должен послать пакеты через "хорошие" пути, например, пути, которые доставляют только маленькую задержку и га-

рантируют высокую производительность всей сети. Алгоритм называется оптимальным, если он использует "самые лучшие" пути.

Другие аспекты эффективности - то, как быстро решение маршрутизации может быть сделано, как быстро пакет может быть подготовлен для передачи, и т.д., но эти аспекты получат меньшее количество внимания в этой главе.

(4) *Живучесть*. В случае топологического изменения (добавление или удаление канала или узла) алгоритм модифицирует таблицы маршрутизации для выполнения функции маршрутизации в изменяемой сети.

(5) *Адаптивность.* Алгоритм балансирует загрузку каналов и узлов, адаптируя таблицы, чтобы избежать маршрутов через каналы или узлы, которые перегружены, предпочитая каналы, и узлы с меньшей загруженностью в настоящее время .

(6) *Справедливость*. Алгоритм должен обеспечить обслуживание каждому пользователю в равной мере.

Эти критерии - иногда конфликтуют, и большинство алгоритмов выполняет хорошо только их подмножество.

Как обычно, сеть представляется как граф, где узлы графа - узлы сети, и существует ребро между двумя узлами, если они - соседи (то есть, они имеют канал связи между ними). Оптимальность алгоритма зависит от того, что называется "самым лучшим" путем в графе; существует, несколько понятий "самый лучший", каждый с собственным классом алгоритмов маршрутизации:

(1)*Минимальное количество переходов*. Стоимость использования пути измеряется как число переходов (пройденные каналы или шаги от узла до узла) пути. Минимальный переход, направляющий алгоритм, использует путь с самым маленьким возможным числом переходов.

(2)*Самый короткий путь.* Каждый канал статически назначен (неотрицательным) весом, и стоимость пути измеряется как сумма весов каналов в пути. Алгоритм с самой короткой дорожкой использует путь с самой низкой возможной стоимостью.

(3)*Минимальная задержка.* Каждый канал динамически означивается весом, в зависимости от трафика в канале. Алгоритм с минимальной задержкой неоднократно перестраивает таблицы таким способом, при котором пути с (близкой) минимальной общей задержкой выбираются всегда.

Другие понятия оптимальности могут быть полезны в специальных прикладных программах. Но не будут обсуждаться здесь.

Следующий материал обсуждается в этой главе. В Разделе 4.1 будет показано, что, по крайней мере, для маршрутизации с минимальным переходом и с самым коротким путем, можно направить все пакеты предназначенные d оптимально через дерево охватов, приложенное к d. Как следствие, отправитель пакета может игнорироваться, при расчете маршрутизации.

Раздел 4.2 описывает алгоритм, для вычисления таблицы маршрутизации для статической сети с каналами имеющими вес. Алгоритм распределенно вычисляет самый короткий путь между каждой парой узлов и в каждом исходном узле первого сосед на пути к каждому адресату. Недостаток этого алгоритма в том, что все вычисления должны быть повторены после изменения топологии сети: алгоритм не масштабируемый.

Алгоритм изменяемой сети, обсужденный в Разделе 4.3, не страдает из этого недостатка: он может адаптироваться к потере или восстановлению каналов частичным перевычислением таблиц маршрутизации. Чтобы анализ был простым, он реализован как минимальный переход, то есть число шагов принимается как стоимость пути. Возможно " изменить Netchange алгоритм, для работы с взвешенными каналами, которые могут теряться или восстанавливаться.

Алгоритмы маршрутизации Разделов 4.2 и 4.3 используют таблицы маршрутизации (в каждом узле) с записями для каждого возможного адресата. Это может слишком отяготить больших сетей из маленьких узлов. В Разделе 4.4 будут обсуждены некоторые стратегии маршрутизации, которые кодируют топологическую информацию в адресе узла, чтобы использовать более короткие таблицы маршрутизации или меньшее количество таблиц. Эти так называемые "компактные" алгоритмы маршрутизации обычно не используют оптимальные пути. Схема основой - деревом, интервальная маршрутизация, и префиксная маршрутизация также будет обсуждена.

Раздел 4.5 обсуждает иерархические методы маршрутизации. В этих методах, сеть разбита на разделы - кластеры, и различие сделано между маршрутизацией внутри кластера и маршрутизации между кластерами. Эта парадигма может использоваться, чтобы уменьшить количество решений маршрутизации.

4.1 Адресат-основанная маршрутизация

Решение маршрутизации, сделанное, когда пересылается пакет обычно основано только на *адресате* пакета (и содержании таблиц маршрутизации), и *не зависит* от первоначального отправителя (источника) пакета. Маршрутизация может игнорировать источник и использовать оптимальные пути, таковы выводы этого раздела. Выводы не зависят от выбора частного критерия оптимальности для путей. (Положим, что путь *прост*, если он содержит каждый узел только один раз, и путь - *цикл*, если первый узел равняется последнему узлу.)

(1)Стоимость посылки пакета через путь P не зависит от фактического использования пути, в частности использование ребер P в соответствии с другими сообщениями. Это предположение позволяет нам оценивать стоимость использования пути P как функцию пути; таким образом, обозначим стоимость P как $C(P) \in \Re$..

(2) Стоимость конкатенации двух путей равняется сумме стоимостей составных путей, то есть, для всякого $i = 0,..., k$

$$C(<u_0,u_1,...,u_k>)=C(<u_0,...,u_i>)+C(<u_i,...,u_k>).$$

Следовательно, стоимости пустого пути $<u_0>$(это - путь от u_0 до u_0) удовлетворяет $C(<u_0>) = 0$.

(3)Граф не содержит циклов отрицательной стоимости.

(Этот критерий удовлетворяется критерием самого короткого пути и критерием минимального перехода). Путь от u до v, называется *оптимальным*, если не существует никакой путь от u до v с более низкой стоимостью. Заметьте, что оптимальный путь не всегда единственен; могут существовать различные пути с той же самой (минимальной) стоимостью.

Лемма 4.1. *Пусть $u, v \in V$. Если путь из u в v существует в G, тогда и существует простой путь, который оптимален.*

Доказательство. Так как количество простых путей конечное число, то существует простой путь от u до v, назовем его S_o, с наименьшей стоимостью, т.е., для каждого *простого* пути P' из u в v $C(S_o) \leq C(P')$. Осталось показать что $C(So)$ нижняя граница стоимостей всех (не простого) путей
Запишем $V = \{v_1, ..., v_n\}$. Следовательно, удаляя из P циклов, включающие v_1, v_1, v_2 и т.д., покажем что для каждого пути P из u в v существует простой путь P' с $C(P') \leq C(P)$. Положим $P_o = P$, и построим для $i = 1,..., N$ путь P_i следующим образом. Если v_i входит в P_{i-1} тогда $P_i = P_{i-1}$. Иначе, запишем $P_{i-1} = \langle u_o, ..., u_k \rangle$. Пусть u_{j1} будет первым и u_{j2} будет последним вхождением v_i в P_{i-1} и положим

$$P_i = \langle u_o, \ldots, u_{j1}(=u_{j2}), u_{j2+1}, \ldots, u_k \rangle$$

по построению P_i - путь из u к v и содержит все вершины из $\{v_1, ..., v_n\}$ только единожды, следовательно P_N -простой путь из u в v. P_{i-1} состоит из P_i и цикла $Q = u_o, \ldots, u_{j2}$ следовательно $C(P_{i-1}) = C(P_i) + C(Q)$. Так как не существует циклов отрицательного веса, это предполагает $C(P_i) \leq C(P_{i-1})$ и, следовательно, $C(P_N) \leq C(P)$.
По выбору S_o, $C(S_o) \leq C(P_N)$, из которого следует $C(S_o) \leq C(P)$ []

Если G содержит циклы отрицательного веса, оптимальный путь не обязательно существует; каждый путь может быть «побежден» другим путем, который пройдет через отрицательный цикл еще раз. Для следующей теоремы, примите, что G связный (для несвязных графов, теорема может применяться к каждому связному компоненту отдельно).

Теорема4.2. *Для каждого $d \in V$ существует $T_d = (V, E_d)$ такое что $E_d \subseteq E$ и такое что для каждой вершины $v \in V$, путь из v к d в T_d - оптимальный путь от v к d в G.*

Доказательство. Пусть $V = \{ v_1, ..., v_N \}$. Мы индуктивно построим последовательность деревьев $T_i = (V_i, E_i)$ (для $i = 0, ...,N$) со следующими свойствами

(1) Каждое T_i - поддерево G, т.е., $V_i \subset V$, $E_i \subset E$, и T_i - дерево.
(2) Каждое T_i (для $i < N$) поддерево T_{i+1}.
(3) Для всех $i > 0$, $v_i \in V_i$ и $d \in V_i$.
(4) Для всех $w \in Vi$, простой путь от w к d в T, - оптимальный путь от w к d в G.

Эти свойства подразумевают, что T_N соответствует требованиям для T_d.
Конструируя последовательность деревьев, положим $V_d = \{d\}$ и $E_o = \varnothing$. Дерево T_{i+1} построим следующим образом. Выберем оптимальный простой путь $P = \langle u_o, \ldots, u_k \rangle$ от v_{i+1} к d, и пусть l будет наименьшим индексом таким, что $u_l \in T_i$ (такое l существует, потому что $u_l = d \in T_i$; возможно $l = 0$). Теперь:

$$V_i = V_i \cup \{ u_j : j < l \} \quad \text{и} \quad E_{i+1} = E_i \cup \{ (u_j, u_{j+1}) : j < 1 \}.$$

(Построение иллюстративно представлено на Рисунке 4.1.) Нетрудно видеть что T_i поддерево T_{i+1} и что $v_{i+1} \in V_{i+1}$. Чтобы увидеть что T_{i+1} дерево, заметим что по построению T_{i+1} связный, и число вершин превосходит число ребер на одно. (T_o имеет последнее свойство, на каждом шаге много вершин и ребер добавлено)

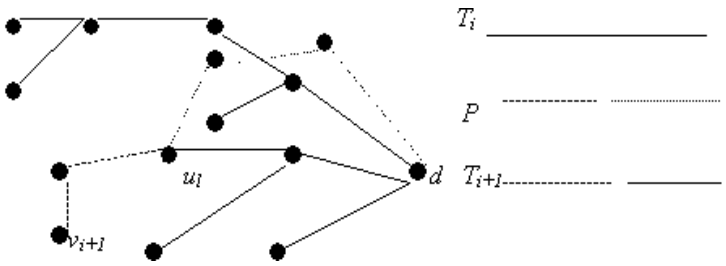


Рисунок 4.1 Построение T_{i+1}.

Осталось показать, что для всех $w \in V_{i+1}$, (уникальный) путь от w к d в T_{i+1} - оптимальный путь от w к d в G. Для вершин $w \in V_i \subset V_{i+1}$ это следует, потому что T_i поддерево T_{i+1} ; путь от w к d в T_{i+1} точно такой же, как путь в T_i, который оптимален. Теперь пусть $w = u_j, j < l$ будет вершиной в $V_{i+1} \setminus V_i$. Запишем Q для пути от u_i к d в T_i, тогда в T_{i+1} u_j соединена с d через путь $<u_j, \ldots, u_l>$ соединенный с Q, и осталось показать, что этот путь оптимальный в G. Во-первых, суффикс $P' = <u_l, \ldots, u_k>$ в P оптимальный путь от u_l до d, т.е., $C(P') = C(Q)$: оптимальность Q подразумевает что $C(P') \geq C(Q)$, и $C(Q) < C(P')$ подразумевает (добавлением стоимости пути) что путь $<u_o, \ldots, u_l>$ соединен с Q имеющий меньший путь, чем P, противоречащий оптимальности P. Теперь положим, что R из u_j к d имеет меньшую стоимость, чем путь $<u_j, \ldots, u_l>$ соединенный с Q. Тогда, по предыдущим наблюдениям, R имеет меньшую стоимость, чем суффикс $<u_j, \ldots, u_k>$ P, и это предполагает что путь $<u_o, \ldots, u_j>$ соединенный с R имеет меньшую стоимость, чем P, противоречащий оптимальности P.

Дерево охвата, приложенное к d , называется *деревом стока* для d, и дерево со свойством, данным в Теореме 4.2 , называется *оптимальным деревом стока*. Существование оптимальных деревьев стока не является компромиссом оптимальности, если только алгоритмы маршрутизации рассматриваются, для которого механизм пересылки, как в Алгоритме 4.2. В этом алгоритме, $table\_lookup_u$ локальная процедура с одним параметром, возвращая соседа u (после консультации с таблицами маршрутизации). Действительно, поскольку все пакеты для адресата d могут быть направлены оптимально используя дерево обхвата, приложенное к d, пересылка оптимальна, если, для всего $u \neq d$, $table\_lookup_u(d)$ возвращает отца u в дереве охвата T_d.
Когда механизм пересылки имеет эту форму, и никакие (дальнейшие) изменения топологии не происходят, корректность таблиц маршрутизации может быть удостоверена, используя следующий результат. Таблицы маршрутизации, как говорят, *содержат цикл* (для адресата d), если существуют узлы $u_1, \ldots, u_k$ такие, что для всех i , $u_i \neq d$, для всех $i < k$, $table\_lookup_u(d) = u_{i+1}$, и $table\_lookup_u(d) = u_{j+1}$.. Таблицы, как говорят, являются свободным от циклов, если они не содержат циклов для любого d.

(* Пакет с адресатом d был получен или сгенерирован в узле u *)
if $d=u$
then доставить «местный» пакет
else послать пакет к $table\_lookup_u\ (d)$

Алгоритм 4.2 Адресат-основанная пересылка (для узла *u).*

Лемма 4.3 *Механизм пересылки доставляет каждый пакет адресату, тогда и только тогда когда таблицы маршрутизации цикл-свободны.*

Доказательство. Если таблицы содержат цикл для некоторого адресата *d*, пакет для *d* никогда не будет доставлен, если источник - узел в цикле.
Примем, что таблицы цикл-свободны и позволяют пакету с адресатом *d* (и источник u_0) быть посланным через $u_0, u_1, u_2, \ldots$ если один встречается дважды в этой последовательности, скажем $u_i = u_j$, тогда таблицы содержат цикл, а именно $< u_i \ldots, u_j >$ противореча предположению, что таблицы являются цикл-свободными. Таким образом, каждый узел входит единожды, что подразумевает, что эта последовательность конечна, заканчивающаяся, скажем, в узле Великобританию u_k $(k < N)$. Согласно процедуре пересылки последовательность может заканчиваться только в *d* , то есть, $u_k = d$, и пакет достиг адресата за не больше чем N — 1 шагов

В некоторых алгоритмах маршрутизации случается, что таблицы не цикл-свободны в течение их вычисления. Когда такой алгоритм используется, пакет может пересекать цикл в течение вычисления таблиц, но достигает адресата не больше чем N — 1 шагов после завершения вычисления таблицы, если изменения топологии прекращаются. Если изменения топологии не прекращаются, то есть, сеть подчинена бесконечной последовательности изменений топологии, пакеты не обязательно достигают своего адресата, даже если таблицы цикл-свободны во время модификаций;

Ветвящаяся маршрутизация с минимальной задержкой. При маршрутизации через пути с минимальной задержкой требуется, и задержка канала зависит от использования (таким образом, предположение (1) в начале этого раздела не имеет силу), стоимость использования пути не может просто быть оценена как функция этого единственного пути. Кроме того, трафик на канал должен быть принят во внимание. Избегать скопления пакетов (и возникающую в результате этого задержку) на пути, обычно необходимо посылать пакеты, имеющие ту же самую пару исходный-адресат через различные пути; трафик для этой пары "распределяется" в один или большее количество узлов промежуточного звена как изображено в Рисунке 4.3. Методы маршрутизации, которые используют различные пути к одному адресату, называются много-путевыми или ветвящимися методами маршрутизации. Потому что ветвящиеся методы маршрутизация являются, обычно, очень запутанными, они не будет рассматриваться в этой главе

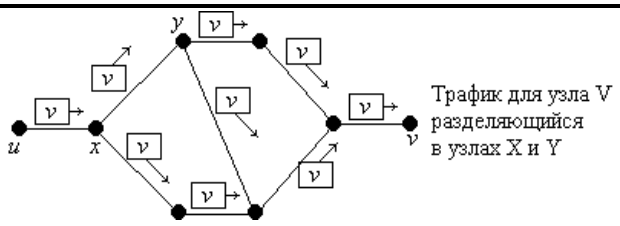


Рисунок 4.3 Пример буферизованной маршрутизации.

4.2 Проблема кротчайших путей всех пар

Этот раздел обсуждает алгоритм Toueg [Tou80a] одновременного вычисления таблицы маршрутизации для всех узлов в сети. Алгоритм вычисляет для каждой пары (*u, v*) узлов длину самый короткий пути от *u* до *v* и сохраняет первый канал такого пути в *u*. Проблема вычисления самого короткого пути между любыми двумя узлами графа известна как проблема кротчайшего пути всех пар. Распределенный алгоритм Toueg для этой проблемы основан на централизованном алгоритме Флойда-Уошалла [CLR90, Раздел 26.4]. Мы обсудим алгоритм Флойда-Уошалла в Подразделе 4.2.1, и впоследствии алгоритм Toueg в Подразделе 4.2.2. Краткое обсуждение некоторых других алгоритмов для проблемы кротчайших путей всех пар следует в Подразделе 4.2.3 ..

4.2.1 Алгоритм Флойда-Уошала

Пусть дан взвешенный граф $G = (V, E)$, где вес ребра uv дан w_{uv} . Не обязательно допускать что $w_{uv} = w_{vu}$, но допустим, что граф не содержит циклов с общим отрицательным весом. Вес пути $< u_o, ..., u_k>$ определяется как $\sum_{i=0}^{k-1} w_{u_i u_{i+1}}$.

Дистанция от *u* до *v,* обозначенная *d(u, v),* наименьший вес любого пути от *u* к *v* (∞ если нет такого пути). Проблема кротчайших путей всех пар - вычисление *d(u, v)* для каждых *u* и *v*.

Для вычисления всех расстояний, алгоритм Флойда-Уошала использует понятие *S*-путей; это пути, в которых все *промежуточные* вершины принадлежат к подмножеству *S* из *V*.

Определение 4.4. *Пусть S - подмножество V. Путь $< u_o \ ..., u_k>$ -S-путь если для любого i, $0 <i < k$, $u_j \in S$. S-расстояние от и до v, обозначенное $d^S(u. v)$, наименьший вес любого S-пути от и до v (∞ если такого пути нет).*

Алгоритм стартует рассмотрением всех ∅-путей, и увеличивая вычисления *S*-путей для больших подмножеств *S,* до тех пор пока *V*-пути будут рассмотрены. Могут быть сделаны следующие наблюдения.

Утверждение 4.5 *Для всех **u** и S, $d^S(u, u) = 0$. Более того, S-пути удовлетворяют следующим правилам для $u \neq v$.*

(1) *Существует ∅ -путь от **u** к **v** тогда и только тогда когда $uv \in E$.*

(2) *Если $uv \in E$ тогда $d^S(u, v) = w_{uv}$ иначе $d^S(u, v) = \infty$.*

(3) *Если $S' = S \cup \{w\}$ тогда простой S'-путь от **u** к **v** - S-путь от **u** к **v** или S-путь от **u** к **w** соединенные S-путем от w к v.*

(4) *Если $S' = S \cup \{w\}$ тогда $d^{S'}(u, v) = \min(d^S(u, v), d^S(u, w) + d^S(w, v))$.*

(5) *Путь от u до v существует тогда и только тогда когда V-путь от u к v существует*

(6) $d(u, v) = d^V(u, v)$,

Доказательство. Для всех *u* и *S* $d^S(u, u) \leq 0$ по причине того, что пустой путь (состоящий из 0 ребер) это *S*-путь от *u* к *u* с весом 0. Нет путей, имеющих меньший вес, потому что *G* не содержит циклов отрицательного веса, таким образом, $d^S(u, u) = 0$.

Для (1): $\varnothing$ -путь не содержит промежуточных узлов, так $\varnothing$ - путь от u к v состоит только из канала uv.

Для (2): следует непосредственно из (1).

Для (3): простой S'-путь от u к v содержит узел w единожды, или 0 раз как промежуточный. Если он не содержит w как промежуточную вершину он S-путь, иначе он - конкатенация двух S-путей, один к w и один из w.

Для(4): Это можно доказать применив Лемму 4.1 . Получим что (если S'-путь от u в v существует) это *простой* S' -путь длиной $d^{S'}(u, v)$ от u к v, такой, что $d^{S'}(u, v) = \min(d^S(u, v), d^S(u, w) + d^S(w, v))$ по (3).

Для (5): каждый S-путь - путь, и обратно.

Для (6): каждый S-путь - путь, и обратно, следовательно, оптимальный V-путь также оптимальный путь.

---

begin (* Инициализация $S = \varnothing$ и $D = \varnothing$-дистанция *)
S:=0;
forall u,v **do**
if $u = v$ **then** $D[u, v] := 0$
else
if $uv \in E$ **then** $D[u, v] := w_{uv}$
else $D[u. v] := \infty$;
(* Расширим S «центральными точками» *)
while $S \neq V$ **do**
(* Цикл инвариантен: $\forall u, v : D[u, v] = d^S(u, v)$ *)
begin выбрать w из $V \setminus S$;
(* Выполнить глобальную w-центровку *)
forall $u \in V$ ***do***
(* Выполнить локальную w-центровку u *)
forall $v \in V$ **do**
$D[u. v] := \min(D[u, v], D[u, w] + D[w, v])$;
$S := S \cup \{w\}$
end (*$\forall u, v : D[u, v] = d^S(u, v)$*)
end

Алгоритм4.4 Алгоритм Флойда-Уошалла.

Используя Утверждение 4.5 не сложно разработать алгоритм "динамического программирования" для решения проблемы кротчайших путей всех пар; смотри см. Алгоритм 4.4. Алгоритм вначале считает 0-пути, и, увеличивая, вычисляет S-пути для больших множеств S (увеличивая S "центральными" кругами), до тех пор, пока все пути не будут обсуждены.

Теорема 4.6 *Алгоритм 4.4 вычисляет расстояние между всеми парами узлов за* $\Theta(N^3)$ *шагов.*

Доказательство. Алгоритм начинает с $D[u, v] = 0$, если $u = v$, $D[u, v] = w_{uv}$,

если $uv \in E$ и $D[u, v] = \infty$ в другом случае, и $S = 0$. Следуя из Утверждения 4.5, частей (1) и (2), $\forall u, v$ имеет силу $D[u, v] = d^S(u, v)$. В центральной окружности с центральной вершиной w множество S расширено узлом w, и означивание $D[u, v]$ гарантирует (по частям (3) и (4) утверждения) что утверждение $\forall u, v : D[u, v] = d^S(u, v)$ сохранено как инвариант цикла. Программа заканчивает работу, когда $S = V$, т.е., (по частям (5) и (6) утверждения и инварианту цикла) S-расстояние эквивалентно расстоянию.
Главный цикл выполняется N раз, и содержит N^2 операций (которые могут быть выполнены параллельно или последовательно), откуда и следует временная граница данная теоремой.

```
var Su   : множество вершин;
    Du   : массив весов;
    Nbu  : массив вершин;
begin Su :=∅ ;
   forall v ∈ V do
      if v = u
         then begin Du [v] :=0 ; Nbu[v] := udef  end
      else if v ∈  Neighu
         then begin Du[v] := wuv ; Nbu[v] := v end
      else begin Du[v] := ∞ ; Nbu[v] := udef end ;
   while Su≠ V do
      begin выбрать w из V \ Su ;
            (* Все вершины должны побывать вершиной w *)
            if u == w
               then "распространить таблицу Dw"
               else  "принять таблицу Dw"
            forall v ∈ V do
               if Du[w] + Dw[v] < Du[v] then
                  begin Du[v]:= Du[w] + Dw[v] ;
                        Nbu[v] := Nb[w]
                  end;
                  Su := Su U {w}
            end
      end;
```

Алгоритм 4.5 Простой алгоритм (Для узла *u).*

4.2.2 Алгоритм кротчайшего пути.(Toueg)

Распределенный алгоритм вычисления таблиц маршрутизации бал дан Toueg [TouSOa], основанный на алгоритме Флойда-Уошалла описанном в предыдущей части. Можно проверить что алгоритм Флойда-Уошалла подходит для этих целей, т.е., что его ограничения реалистичны для распределенных систем. Наиболее важное ограничение алгоритма что граф не содержит циклов отрицательного веса. Это ограничение действительно реально для распределенных систем, где обычно каждый отдельный канал означен положительной оценкой. Даже можно дать более строгое ограничение; смотри A1 ниже. В этой части даны следующие ограничения.

A1. Каждый цикл в сети имеет положительный вес.
A2. Каждый узел в сети знает обо всех узлах (множество V).
A3. Каждый узел знает какой из узлов его сосед (хранится в $Neigh_u$ для узла *u)* и веса своих выходящих каналов.

Корректность алгоритма Toueg (Алгоритма4.6) будет более просто понять если мы сперва обсудим предварительную версию алгоритма , "простой алгоритм" (Алгоритм 4.5).

Простой алгоритм. Для достижения распределенного алгоритма переменные и операции алгоритма Флойда-Уошала распределены по узлам сети. *D[u, v]* - переменная принадлежащая узлу *u*; по соглашению, это будет выражено описанием $D_u[v]$.Операция, означивающая $D_u[v]$, должна быть выполнена узлам *u*, и когда необходимо значение переменной узла *w*, это значение должно быть послано *u*. В алгоритме Флойда-Уошала все узлы должны использовать информацию из «центрального» узла (*w* в теле цикла), который посылает эту информацию к всем узлам одновременно операцией "распространения". В заключение, алгоритм будет расширен операцией для поддержки не только длины кратчайших *S*-путей (как в переменной $D_u[v]$), но также первый канал такого пути (в переменной $Nb_u[v]$).
Утверждение что циклы сети имеет положительный вес может использоваться чтобы показать что не существует циклов в таблицах маршрутизации.

Лемма 4.7 *Пусть даны S и w и выполняется:*
(1) *для всех u :* $D_u[w] = d^S(u, w)$ *и*
(2) *если* $d^S(u, w) < \infty$ *и* $u \neq w$, *то* $Nb_u[w]$ *- первый канал кратчайшего S-пути к w.*

Тогда направленный граф $T_w = (V_w, E_w)$, *где* $(u \in V_w \Leftrightarrow D_u[w] < \infty)$ *и* $(ux \in E_w \Leftrightarrow (v \neq w \wedge Nb_u[w] = x))$ *- дерево с дугами направленными к w.*

Доказательство. Во-первых, заметим, что если $D_u[w] < \infty$ для $u \neq w$, то $Nb_u[w] \neq udef$ и $D_{Nb_u}[w] < \infty$. Таким образом для каждого узла $u \in V_w$, $u \neq w$ существует узел *x* для которого $Nb_u[w] = x$, и $x \in V_w$.
Для каждого узла $u \neq w$ в V_w существует единственное ребро в E_w, такое что число узлов в T_w превышает количество ребер на единицу и достаточно показать что T_w не содержит циклов. Так $ux \in E_w$ подразумевает что $d^S(u, w) = w_{ux} + d^S(x, w)$, существование цикла $\langle u_o, u_1, .. ., u_k \rangle$ в T_w подразумевает что

$$d^S(u_o, w) = w_{uo\,u1} + w_{u1\,u2} + \ldots + w_{uk-1\,uo}u + d^S(u_o, w),$$

т.е., $0 = w_{uo\,u1} + w_{u1\,u2} + \ldots + w_{uk-1\,uo}u$
что противоречит предположению, что каждый цикл имеет положительный вес.

Алгоритм Флойда-Уошала теперь может быть просто преобразован в Алгоритм 4.5. Каждый узел инициализирует свои собственные переменные и исполняет *N* итераций основного цикла. Этот алгоритм не является окончательным решением, и он не дан полностью, потому что мы не описали, как может бать произве-

дено (эффективно) распространение таблиц центрального узла. Пока это можно использовать как гарантированное, поскольку операция "распространить таблицу D_w'' выполняется узлом w, а операция "принять таблицу D_w'' выполняется другими узлами, и каждый узел имеет доступ к таблице D_w.
Некоторое внимание должно быть уделено операции "выбрать w из $V \setminus S''$, чтобы узлы выбирали центры в однообразном порядке. Так как все узлы знают V заранее, мы можем запросто предположить, что узлы выбираются в некотором предписанном порядке (на пример, алфавитный порядок имен узлов).
Корректность простого алгоритма доказана в следующей теореме.

Теорема 4.8 *Алгоритм 4.5 завершит свою работу в каждом узле после N итераций основного цикла. Когда алгоритм завершит свою работу в узле и* $D_u[v] = d(u, v)$, *и если путь из и в v существует то* $Nb_u[v]$ *первый канал кротчайшего пути из и в v, иначе* $Nb_u[v] = udef$.

Доказательство. Завершение и корректность $D_u[v]$ по завершении работы следует из корректности алгоритма Флойда-Уошала (теорема 4.6). Утверждение о значении $Nb_u[v]$ справедливо потому что $Nb_u[v]$ перевычисляется каждый раз когда означивается $D_u[v]$.

Усовершенствованный алгоритм. Чтобы сделать распространение в Алгоритме 4.5 эффективным, Toueg заметил, что узел u для каждого $D_u[w] = \infty$ на старте w-централизованного обхода не меняет свои таблицы в течение всего w-централизованного обхода. Если $D_u[w] = \infty$, то $D_u[w] + D_w[v] < D_u[v]$ не выполняется для каждого узда v. Следовательно, только узлы, принадлежащие T_w (в начале w-централизованного обхода) нуждаются в получении таблиц w, и операция распространения может стать более эффективной рассылая D_w только через каналы, принадлежащие дереву T_w. Таким образом, w рассылает D_w своим сыновьям в T_w и каждый узел в T_w который принимает таблицу (от своего отца в T_w) пересылает её к своим сыновьям в T_w.

```
var  Su   : множество узлов ;
     Du   : массив весов;
     Nbu  : массив узлов ;
begin
    Su := ∅  ;
    forall v ∈ V do
        if v = u
            then begin Du[v] := O ; Nbu[v] := udef end
        else if v ∈  Neighu
            then begin Du[v] := wuv ; Nbu[v] := v end
        else begin Du[v] := ∞  ; Nbu[v] := udef end ;
     while Su ≠ V do
        begin выбрать w из V \ Su ;
                (* Построение дерева Tw *)
                forall x ∈ Neighu do
                    if Nbu[w] = x then send < ys, w> to x
```

```
                    else send < nys, w > to x ;
            num_rec_u := O ; (* u должен получить |Neigh_u| сообщений *)
           while num_rec_u < |Neigh_u| do
                 begin получить < ys, w > или  < nys, w > сообщение ;
                       num_rec_u := num_rec_u + 1
                 end;
            if D_u[w] < ∞  then (* участвует в центр. обходе*)
                 begin if u ≠ w
                             then  получить <dtab,w,D> от Nb_u[w] ;
                       forall x ∈ Neigh_u do
                            if < ys, w > было послано от x
                                 then послать < dtab, w, D>) к x; ;
                       forall v ∈  V do (* локальный  w-центр *)
                            if D_u[w] + D[v] < D_u[v] then
                                 begin D_u[v] := D_u[w] + D[v] :
                                       Nb_u[v] := Nb_u[w]
                                 end
                 end;
            S_u := S_u ∪ {w}
      end
  end
```

Алгоритм 4.6 Алгоритм Toueg (для узла *u).*

В начале *w*-централизованного раунда узел *u* с $D_u[w] < \infty$ знает кто его отец (в T_w) , но не знает кто его сыновья. Поэтому каждый узел *v* должен послать сообщение к каждому своему соседу *u,* спрашивая *u* является ли *v* сыном *u* в T_w. Полный алгоритм дан как Алгоритм 4.6. Узел может участвовать в пересылке таблицы *w* когда известно что его соседи являются его сыновьями в T_w. Алгоритм использует три типа сообщений:

(1) <ys,*w*> сообщение <ys обозначение для "your son"> *u* посылает к *x*; в начале *w*-централизованного обхода если *x* отец *u* в *Tw*.

(2) <nys, *w*> сообщение <nys обозначение для "not your son"> *u* посылает *x* в начале *w*-централизованного обхода если *x* не отец *u* в T_w

(3) <dtab**,** *w, D*> сообщение посылается в течение *w*-централизованного обхода через каждое ребро T_w чтобы переслать значение D_w к каждому узлу который должен использовать это значение.

Полагая сто вес (ребра или пути) вместе с именем узла можно представить *W* битами, сложность алгоритма показана следующей теоремой.

Теорема 4.9 *Алгоритм 4.6 вычисляет для каждых* ***u*** *и* ***v*** *дистанцию от* ***u*** *к* ***v****, и, если эта дистанция конечная, первый канал. Алгоритм обменивается 0(N) сообщениями на канал,, 0(N*|E|) сообщений всего, O(N²W) бит на канал, O(N ³W) бит всего, и требуется 0(NW) бит хранения на узел.*

Доказательство. Алгоритм 4.6 выведен от Алгоритма 4.5, который корректен. Каждый канал переносит два (< ys, *w*> или < nys, *w*>) сообщений (одно в каж-

дом направлении) и не более одного <dtab, *w, D* > сообщения в *w*-централизованном обходе, который включает не более *3N* сообщений на канал. < ys, *w* > или < nys, *w* > сообщение содержит O(W) бит и <dtab, *w, D* > сообщение содержит O*(NW)* бит, что и является границей для числа бит на канал. Не более N^2 < dtab, *w,D*> сообщений и *2N - |E| (*<ys,*w*> и <nys,*w*> *)* сообщений обмена, и того всего O$(N^2 - NW + 2N - |E| - W) =$ O$(N^3 W)$ бит. Таблицы D_u и Nb_u хранящиеся в узле *u* требуют *0(NW)* бит.

В течение *w*-центализованного обхода узлу разрешено принимать и обрабатывать сообщения только данного обхода, т.е., те которые переносят параметр *w*. Если каналы удовлетворяют дисциплине FIFO тогда сообщения <ys,*w*> и <nys, *w*> прибывают первыми, по одному через каждый канал, и затем сообщение < **dtab,** *w, D* > от $Nb_u[w]$ (если узел в V_w). Таким образом возможно, аккуратно программируя, опустить параметр *w* во всех сообщениях если каналы удовлетворяют дисциплине FIFO. Если каналы не удовлетворяют дисциплине FIFO возможно что сообщение с параметром *w′* придет пока узел ожидает сообщения для обхода *w,* тогда как *w′* становится центром после *w*. В этом случае параметр используется чтобы различить сообщения для каждого централизованного обхода, и локальная буферизация (в канале и узле) должна использоваться для отсрочки выполнения *w′*-сообщения.
Toueg дал дальнейшую оптимизацию алгоритма, полагаясь на следующий результат. (Узел u_2 *потомок* u_1 если u_2 принадлежит поддереву u_1)

Лемма 4.10 *Пусть* $u_1 \neq w$, *и пусть* u_2 *потомок* u_1 *в* T_w, *в начале w-централизованного обхода, если* u_2 *изменит своё расстояние до v во время w-централизованного обхода, тогда и* u_1 *изменит своё расстояние до v в этом же обходе.*

Доказательство. Так как u_2 потомок u_1 в T_w :

$$d^S(u_2, w) = d^S(u_2, u_1) + d^S(u_1, w). \qquad (1)$$

Так как $u_1 \in S$:

$$d^S(u_2, v) \leq d^S(u_2, u_1) + d^S(u_1, v). \qquad (2)$$

Узел u_2 изменит $D_{u2}[v]$ в данном обходе тогда и только тогда когда

$$d^S(u_2, w) + d^S(w, v) < d^S(u_2, v). \qquad (3)$$

Применяя (2), и затем (1), и вычитая $d^S(u_2, u_1)$, мы получим

$$d^S(u_1, w) + d^S(w, v) < d^S(u_1, v) \qquad (4)$$

значит u_1 изменит $D_{u1}[v]$ в этом обходе.

В соответствии с этой леммой, Алгоритм 4.6 может быть модифицирован следующим образом. После получения таблицы D_w, (сообщение <**dtab,** *w,D>)* узел *u* вначале выполняет локальные *w*-централизованные операции, и затем рассылает таблицы своим сыновьям в T_w. Когда пересылка таблицы закончилась достаточно переслать те ссылки $D[v]$ для которых $D_u[v]$ изменилась в течение локальной *w*-централизованной операции. С этой модификацией таблицы маршрутизации не содержат циклов не только между централизованными обходами (как сказано в Лемме 4.7), но также в течение централизованных обходов.

4.2.3 Обсуждение и Дополнительные Алгоритмы

Представление алгоритм Toueg предоставило пример как распределенный алгоритм может быть получен непосредственным образом из последовательного алгоритма. Переменные последовательного алгоритма распределены по процессам, и любое означивание переменной *x* (в последовательном алгоритме) выполняется процессом владеющим *x*. Всякий раз когда означивающее выражение содержит ссылки на переменные из других процессов, связь между процессами потребуется для передачи значения и синхронизации процессов. Специфические свойства последовательного алгоритма могут быть использованы для минимизации числа соединений.

Алгоритм Toueg прост для понимания, имеет низкую сложность, и маршрутизирует через оптимальные пути; его главный недостаток в его плохая живучесть. Когда топология сети изменилась все вычисления должны произвестись заново. Во-первых, как ранее говорилось, однообразный выбор всеми узлами следующего центрального узла *(w)* требует чтобы множество участвующих узлов было известно заранее. Так как это в основном не известно априори, исполнение расширенного распределенного алгоритма вычисления этого множества (на пример алгоритм Финна, Алгоритм 6.9) должно предшествовать исполнению алгоритма Toueg.

Во-вторых, алгоритм Toueg основан на повторяющимися применениями *уникальности треугольника* $d(u, v) \le d(u, w) + d(w, v)$. Оценивание правой стороны (*u)* требует информацию о $d(w, v)$, и эта информация в часто *удалена*, т.е., не доступна ни в *u* ни в любой из его соседей. Зависимость от удаленных данных делает необходимым транспортирование информации к удаленным узлам, которые могут быть исследованы в алгоритме Toueg (часть распространения).

Как альтернатива, определенное ниже равенство для $d(u, v)$ может использоваться в алгоритмах для проблем кротчайших путей:

$$d(u,v) = \begin{cases} 0 & \text{если } u=v \\ w_{uw}+d(w,v) & \text{иначе} \end{cases} \qquad (4.1)$$

Два свойства этого равенства делают алгоритмы основанные на этом отличными от алгоритма Toueg.

(1) *Локальность данных.* Во время оценивания правой стороны равенством (4.1), узлу *u* необходима только информация доступная локально (именно, w_{uw}) или в соседях (именно, $d(w, v)$). Транспортирование данных между удаленными вершинами избегается.

(2) *Независимость пункта назначения.* Расстояния до *v* (именно, $d(w, v)$ где *w* сосед *u)* только нуждаются в вычислении расстояния от *u* в *v*. Таким образом, вычисление всех расстояний до фиксированного пункта назначения v_o может происходить независимо от вычисления расстояния до других узлов, и также, может бать сделано обособленно.

В завершение этой части обсуждены два алгоритма основанные на равенстве, а именно алгоритмы Мерлина-Сигалла и Чанди-Мизра. Не смотря на преимущества от локальности данных, сложность соединений этих алгоритмов не лучше алгоритма Toueg. Это из-за независимости пункта назначения введенного равенством(4.1); очевидно, использование результатов для других пунктов назна-

чения (как сделано в алгоритме Toueg) более выгодный прием чем локальность данных.
Если это не ведет к уменьшению сложности соединений, тогда каково значение локальности данных? Уверенность в удаленных данных требует повторных распространений если данные могли измениться из-за топологических изменений сети. Доведение до конца этих распространений (с возможными новыми топологическими изменениями в течение распространения) вызывает нетривиальные проблемы с дорогими решениями (см., на пример, [Gaf87]). Более того, алгоритмы основанные на равенстве 4.1 могут быть более легко адаптированы к топологическим изменениям. Оно служит примером в части 4.3, где подробно разобран такой алгоритм.

Алгоритм Мерлина-Сигалла. Алгоритм предложенный Мерлином и Сигаллом [MS79] вычисляет таблицы маршрутизации для каждого пункта назначения абсолютно обособленно; вычисления для различных пунктов назначения не оказывают влияния друг на друга. Для пункта назначения *v,* алгоритм начинает работу с дерева T_v с корнем в *v,* и повторно перевычисляет это дерево с тем чтобы оно стало оптимальным деревом стока для *v.*
Для пункта назначения *v,* каждый узел *u* содержит оценку расстояния до *v* ($D_u[v]$) и соседа через которого пакет для *u* пересылается ($Nb_u[v]$), который также является отцом *u* в T_v. В период перевычисления каждый узел *u* посылает свою оценку расстояния, $D_u[v]$, к всем соседям *исключая* $Nb_u[v]$ (в < **mydist,** *v,* $D_u[v]$ > сообщении). Если узел *u* получает от соседа *w* сообщение <mydist,*v*,*d*> и если $d+w_{uv} < D_u[v]$, *u* изменит $Nb_u[v]$ на *w* и $D_u[v]$ на $d + w_{uv}$. Период перевычисления контролируется узлом *v* и требует обмена двумя сообщениями в *W* бит на каждый канал.
В [MS79] показано что после *i* периодов перевычислений все кротчайшие пути в более чем *i* шагов будут корректно вычислены, так что после *N* раундов все кротчайшие пути будут вычислены. Кротчайшие пути в каждый пункт назначения вычисляются выполнением алгоритма независимо для каждого пункта назначения.
Теорема 4.11 *Алгоритм Мерлина и Сигалла вычисляет таблицы маршрутизации кротчайших путей с обменом* $0(N^2)$ *сообщениями на канал,* $0(N^2 W)$ *битами на канал,* $0(N^2 |E|)$ *сообщениями всего, и* $0(N^2|E|W)$ *битами всего.*

Алгоритм может также адаптироваться к изменениям топологии и весов каналов. Важное свойство алгоритма в том что в течение периода перевычислений таблицы маршрутизации не содержат циклов.

Алгоритм Чанди—Мизра. Алгоритм предложенный Чанди и Мизра [CM82] вычисляет все кротчайшие вычисляет до одного пункта назначения используя парадигму *диффузивных вычислений* (распределенные вычисления которые инициируются одним узлом, и другие узлы присоединяются только после получения сообщения).
Вычисление, для всех узлов, расстояния до узла v_o (и привилегированного исходящего канала), каждый узел *u* начинает с $D_u[v_o] = \infty$ и ждет получения сообщений. Узел v_o посылает <mydist,v_o,0> сообщение всем соседям. Когда же узел *u* получает сообщение <**maydist**,v_o,*d*> от соседа *w,* где $d + w_{uw} < D_u[v_o]$, *u* заносит значение $d+w_{uv}$ в $D_u[v_o]$ и посылает сообщение < **mydist,** v_o, $D\ D_u[v_o]$ > всем

соседям; смотри Алгоритм 4.7.

---

var $D_u[v_o]$: вес **init** ∞ ;
$Nb_u[v_o]$: узел **init** *udef* ;

Только для узла v_o :
begin $D_u[v_o] := 0$;
forall $w \in Neigh_{vo}$ **do** послать < **maydist,** v_o, *0*> к *w*
end

Обработка сообщения < **maydist,** v_o, *d*> от соседа *w* узлом *u:*
{ <mydist, v_o,*d*> $\in M_{wv}$ }
begin получить <mydist,v_o,*d*> от *w* ;
if $d + w_{uw} < D_u[v_o]$ **then**
begin $D_u[v_o] := d+w_{uw}$; $Nb_u[v_o] := w$;
forall $x \in Neigh_u$ **do** послать < maydist, v_o, $D_u[v_o]$ > к *x*
end
end

Алгоритм 4.7 Алгоритм Чанди-Мизра (для узла u).

Не трудно показать что $D_u[v_o]$ всегда верхняя граница для *d(u, vo),* т.е., $d(u,vq) \leq D_u[v_o]$ инвариант алгоритма; см. упражнение 4.3. Чтобы продемонстрировать чти алгоритм вычисляет расстояния верно, нужно показать что в конечном счете достигнется конфигурация в которой $D_u[v_o] \leq d(u, v_o)$ для каждого *u.* Мы дадим доказательство этого свойства используя предположение допущения слабой справедливости, а именно, что каждое сообщение которое посылается в конечном счете получено в каждом вычислении.

Теорема 4.12 *В каждом вычислении Алгоритма 4.7 достигнется конфигурация в которой для каждого узла u,* $D_u[v_o] \leq d(u, v_o)$.

Доказательство. Зафиксируем оптимальное дерево стока *T* для v_o и обозначим остальные узлы $v_1 \ldots v_{N-1}$ таким образом что если v_i, отец узла v_j, тогда $i < j$. Пусть *C* вычисление; можно показано индукцией по *j* что для каждого $j \leq N— 1$ достигнется конфигурация в которой, для каждого $i \leq j$, $D_{vi}[v_o] \leq d(v_i, v_o)$. Заметим что $D_{vi}[v_o]$ никогда не увеличивается в алгоритме; таким образом если $D_{vi}[v_o] \leq d(v_i, v_o)$ содержится в некоторой конфигурации то она лучшая конфигурация из всех.
Случай $j = 0$: $d(v_o, v_o) = 0$, и $D_{vo}[v_o] = 0$ после выполнения инициализационной части узлом v_o, , таким образом $D_{vo}[v_o] \leq d(v_o, v_o)$ содержится после этого выполнения.
Случай $j + 1$: Допустим что достигнется конфигурация в которой для каждого $i \leq j$, $D_{vi}[v_o] \leq d(v_i, v_o)$, и рассмотрим узел v_{j+1}. Имеется кротчайший путь v_{j+1}, v_i, ..., v_o длины $d(v_{j+1}, v_o)$ от v_{j+1} до v_o, где v_i отец v_{j+1} в *T*, отсюда $i \leq j$. Следовательно, по индукции, достигается конфигурация в которой $D_{vi}[v_o] \leq d(v_i, v_o)$. Всякий раз когда $D_{vi}[v_o]$ уменьшится, v_i посылает сообщение < mydist, v_o, $D_{vi}[v_o]$ > своим соседям, отсюда сообщение < mydist, v_o,*d*>посылается к v_{j+1} по крайней мере однажды с $d \leq d(v_i,v_o)$.

По предположению, это сообщение принимается в C узлом v_{j+1}. Алгоритм подразумевает что после получения этого сообщения хранится $D_{vi}[v_o] \leq d + w_{v_{j+1}v_i}$, и выбор i подразумевает что $d + w_{v_{j+1}v_i} \leq d(v_{j+1}, v_o)$.

Полный алгоритм также включает механизм с помощью которого узлы могут определить окончание вычисления.; сравним с замечанием об алгоритме Netchange в начале части 4.3.3. Механизм для определения завершения является вариацией алгоритма Дейкстры-Шолтена обсужденного в 8.2.1.
Алгоритм отличается от алгоритма Мерлина-Сигалла двумя вещами. Во-первых, нет "отца" узла u которому не посылаются сообщения типа <**mydist,.**,.>. Эта особенность алгоритма Мерлина и Сигалла гарантирует что таблицы всегда не содержат циклов, даже в течение вычислений и при наличии топологических изменений. Во-вторых, обмен сообщениями < mydist,.,. > не координируется в раундах, но существует отслеживание завершения, который влияет на сложность не лучшим образом.
Алгоритм может потребовать экспоненциальное количество сообщений для вычисления путей до одного пункта назначения v_o. Если стоимости всех каналов равны (т.е., рассматривается маршрутизация с минимальным количеством переходов) все кротчайшие пути к v_o вычисляются используя $O(N\ |E|)$ сообщений (O(W) бит каждое), руководствуясь следующим результатом.
Теорема 4.13 *Алгоритм Чанди и Мизра вычисляет таблицы маршрутизации с минимальным количеством шагов с помощью обменов $0(N^2)$ сообщениями (N^2W) бит на канал, и $0(N^2|E|)$ сообщений и $0(N^2\ |E|\ W)$ бит всего.*

Преимущество алгоритма Чанди и Мизра над алгоритмом Мерлина и Сигалла в его простоте, его меньшей пространственной сложности, и его меньшей временной сложности.

4.3 Алгоритм Netchange

Алгоритм Таджибнаписа Netchange [Taj77] вычисляет таблицы маршрутизации которые удовлетворяют мере "минимальное количество шагов". Алгоритм подобен алгоритму Чанди-Мизра , но содержит дополнительную информацию которая позволяет таблицам только *частично* перевычисляться после отказа или восстановления канала. Представление алгоритма в этой части придерживается Лампорта [Lam82]. Алгоритм основан на следующих предположениях.

N1. Узлы знают размер сети (N).
N2. Каналы удовлетворяют дисциплине FIFO.
N3. Узлы уведомляют об отказах и восстановлениях смежных к ним каналов.
N4. Цена пути – количество каналов в пути.

Алгоритм может управлять отказами и восстановлениями или добавлениями каналов, но положим что узел уведомляет когда смежный с ним канал отказывает или восстанавливается. Отказ и восстановление узлов не рассматривается: на самом деле отказ узла можно рассматриваться его соседями как отказ соединяющего канала. Алгоритм содержит в каждом узле u таблицу $Nb_u[v]$, дающую для каждого пункт назначения v соседа u через которого u пересылает па-

кеты для *v.* Требования алгоритмов следующие:

Rl. Если топология сети остается постоянной после конечного числа топологических изменений , тогда алгоритм завершается после конечного числа шагов.
R2. Когда алгоритм завершает свою работу таблицы $Nb_u[v]$ удовлетворяют
(a) если $v = u$ то $Nb_u[v] = local$;
(b) если путь из u в $v \neq u$ существует то $Nb_u[v] = w$, где w первый сосед u в кротчайшем пути из u в v,
(c) если нет пути из u в v тогда $Nb_u[v] = udef$.

4.3.1 Описание алгоритма

Алгоритм Таджибнаписа Netchange дан как алгоритмы 4.8 и 4.9. Шаги алгоритма будут сначала объяснены неформально, и, впоследствии правильность алгоритма будет доказана формально. Ради ясности моделирование топологических изменений упрощено по сравнению с [Lam82], примем, что уведомление об изменении обрабатывается одновременно двумя узлами задействованными изменениями. Это обозначено в Подразделе 4.3.3, как асинхронная обработка. Выбор соседа через которого пакеты для v будут посылаться основан на оценке расстояния от каждого узла до v. Предпочитаемый сосед всегда сосед с минимальной оценкой расстояния. Узел u содержит оценку $d(u, v)$ в $D_u[v]$ и оценки $d(w, v)$ в $ndis_u[w, v]$ для каждого соседа u. Оценка $D_u[v]$ вычисляется из оценок $ndis_u[w, v]$, и оценки $ndis_u[w,v]$ получены посредством коммуникаций с соседями.

Вычисление оценок $D_u[v]$ происходит следующим образом. Если $u = v$ тогда $d(u, v) = 0$ таким образом $D_u[v]$ становится 0 в этом случае. Если $u \neq v$, кротчайший путь от u в v (если такой путь существует) состоит из канала из u до сосед, присоединенного к кратчайшему пути из сосед до v, и следовательно

$$d(u, v) = 1 + \min_{w \in Neigh\ u} d(w, v).$$

Исходя из этого равенства, узел $u \neq v$ оценивает $d(u, v)$ применением этой формулы к *оценочным* значениям $d(w, v)$, найденным в таблицах $ndis_u[w, v]$. Так как всего N узлов, путь с минимальным количеством шагов имеет длину не более чем $N—1$. Узел может подозревать что такой путь не существует если оцененное расстояние равно N или больше; значение N используется для этой цели.

---

var $Neigh_u$: множество узлов ; (* Соседи u *)
D_u : массив 0.. N ; (* $D_u[v]$ - оценки $d(u, v)$ *)
Nb_u : массив узлов ; (* $Nb_u[v]$- предпочтительный сосед для v *)
$ndis_u$: массив 0.. N ; (*$ndis_u[w, v]$ - оценки $d(w. v)$ *)

Инициализация:

begin forall $w \in Neigh_u$, $v \in V$ **do** $ndis_u[w, v] := N$,
forall $v \in V$ **do**
begin $D_u[v] := N$;
$Nb_u[v] := udef$
end ;

$D_u[u] := 0$; $Nb_u[u] := local$;
forall $w \in Neigh_u$ **do** послать < mydist, *u, 0*> к *w*
end

процедура *Recompute (v):*
begin if $v = u$
then begin $D_u[v] := 0$; $Nb_u[v] := local$ **end**
else begin (* оценка расстояния до *v* *)
$d := 1 + min\{ ndis_u[w, v] : w \in Neigh_u\}$;
if $d < N$ **then**
begin $D_u[v] := d$;
$Nb_u[v] := w$ with $1 + ndis_u[w, v] = d$
end
else begin $D_u[v] := N$; $Nb_u[v] :=$ *udef* **end**
end;
if $D_u[v]$ изменилась **then**
forall $x \in Neigh_u$ **do** послать < **mydist,** *v*, $D_u[v]$> к *x*
end

Алгоритм 4.8 Алгоритм Netchange (часть I, для узла u).

Алгоритм требует чтобы узел имел оценки расстояний до *v* своих соседей. Их они получают от этих узлов послав им сообщение <mydist,.,.> следующим образом. Если узел *u* вычисляет значение *d* как оценку своего расстояния до *v* ($D_u[v] = d$), то эта информация посылается всем соседям в сообщении < mydist ,*v*,*d*>. На получение сообщения <mydist, *v, d*> от соседа *w, u* означает $ndis_u[w, v]$ значением *d*. В результате изменения $ndis_u[w, v]$ оценка *u* расстояния *d(u, v)* может измениться и следовательно оценка перевычисляется каждый раз при изменении таблицы $ndis_u$. Если оценка на самом деле изменилась то, на *d'* например, происходит соединение с соседями используя сообщение <**mydist,** *v, d'*>.

Алгоритм реагирует на отказы и восстановления каналов изменением локальных таблиц, и посылая сообщение <mydist, ., .> если оценка расстояния изменилась. Мы предположим что уведомление которое узлы получают о падении или подъеме канала (предположение N3) представлено в виде сообщений < **fail,**. > и <**repair,** . >. Канал между узлами u_1 и u_2 смоделирован двумя очередями, Q u_1 u_2 для сообщений от u_1 к u_1 и Q u_2 u_1 для сообщений из u_2 в u_1. Когда канал отказывает эти очереди удаляются из конфигурации (фактически вызывается потеря всех сообщений в обоих очередях) и узлы на обоих концах канала получают сообщение < **fail, .** > . Если канал между u_1 и u_1 отказывает, u_1 получает сообщение < **fail,** u_2 > и u_2 получает сообщение < **fail,** u_1 > . Когда канал восстанавливается (или добавляется новый канал в сети) две пустые очереди добавляются в конфигурацию и два узлы соединяются через канал получая сообщение < **repair,** . > . Если канал между u_1 и u_2 поднялся u_1 получает сообщение< **repair,** u_2 > и u_2 получает сообщение < **repair,** u_1 > .

Обработка сообщения <**mydist**,*v,d*> от соседа *w*:
{ <**mydist**,*v,d*> через очередь Q*wv*}

begin получить <**mydist**,*v,d*> от *w;*
$ndis_u[w,v] := d$; *Recompute (v)*
end

Произошел отказ канала *uw:*
begin получить < **fail,** *w*> ; $Neigh_u := Neigh_u \setminus \{w\}$,
forall $v \in V$ ***do*** *Recompute (v)*
end
Произошло восстановление канала *uw:*
begin получить < **repair,** *w*> , $Neigh_u := Neigh_u \cup \{w\}$;
forall $v \in V$ **do**
begin $ndis_u[w, v] := N;$
послать < **mydist,** $v, D_u[v]$> to *w*
end
end

Алгоритм 4.9 АЛГОРИТМ NETCHANGE (часть 2, для узла *u).*

Реакция алгоритма на отказы и восстановления выглядит следующим образом. Когда канал между *u* и *w* отказывает, *w* удаляется из $Neigh_u$ и наоборот. Оценка расстояния перевычисляется для каждого пункта назначения и, конечно, рассылается всем существующим соседям если оно изменилось. Это случай если лучший маршрут был через отказавший канал и нет другого соседа *w′* с $ndis_u[w', v] = ndis_u[w, v]$. Когда канал восстановлен(или добавлен новый канал) то *w* добавляется в $Neigh_u$, но *u* имеет теперь неоцененное расстояние *d(w, v)* (и наоборот). Новый сосед *w* немедленно информирует относительно $D_u[v]$ для всех пунктов назначения *v* (посылая сообщения<**mydist**,*v,* $D_u[v]$>). До тех пор пока *u* получает подобное сообщения от w, *u* использует *N* как оценку для *d(w,v),* т.е., он устанавливает $ndis_u[w, v]$ в *N.*

P(u, w, V)≡

$up(u, w) \Leftrightarrow w \in Neigh_u$ (1)

$\wedge\ up(u, w) \wedge$ Q_{wu} содержит сообщение <**mydis***t,v,d*> $\Rightarrow$
последнее такое сообщение удовлетворят $d = D_u[v]$ (2)

$\wedge\ up(u, w) \wedge Q_{wu}$ не содержит сообщение <**mydis***t,v,d*> $\Rightarrow$
$ndis_u[w, v] = D_w[v]$ (3)

L(u, v) ≡

$u = v \Rightarrow (D_u[v] = 0 \wedge Nb_u[v] = local)$ (4)

$\wedge\ (u \neq v) \wedge \exists w \in Neigh_u : ndis_u[w, v] < N - 1) \Rightarrow$
$(D_u[v] = 1 + \min_{w \in Neigh\ u} ndis_u[w, v] = 1 + ndis_u[Nb_u[v], v])$ (5)

$\wedge\ (u \neq v \wedge \forall w \in Neigh_u : ndis_u[w, v] \geq N — 1) \Rightarrow$
$(D_u[v] = N \wedge Nb_u[v] = udef)$ (6)

Рисунок 4.10 Инварианты *P(u, w, v)* и *L(u, v).*

Инварианты алгоритма Netchange. Мы докажем что утверждения являются

инвариантами; утверждения даны на Рисунке 4.10. Утверждение *P(u, w, v)* констатирует что если *u* закончил обработку сообщения <mydist, *v*, .> от *w* то оценка *u* расстояния *d(w,v)* эквивалентна оценке *w* расстояния *d(w, v)*. Пусть предикат *up(u, w)* истинен тогда и только тогда когда (двунаправленный) канал между *u* и *w* существует и действует. Утверждение *L(u, v)* констатирует что оценка *u* расстояния *d(u, v)* всегда согласована с локальными данными *u*, и $Nb_u[v]$ таким образом означен.
Выполнение алгоритма заканчивается когда нет больше сообщений в любом канале. Эти конфигурации не терминальны для всей системы так как вычисления в системе могут продолжится позже, начавшись отказом или восстановлением канала (на которые алгоритм должен среагировать). Мы пошлем сообщение конфигурационной *стабильности,* и определим предикат **stable** как

stable = $\forall u, w : up(u, w) \Rightarrow Q_{wu}$ не содержит сообщений <mydist,., .>.

Это предполагает что переменные $Neigh_u$ корректно отражают существующие рабочие коммуникационные каналы, т.е., что (1) существует изначально. Для доказательства инвариантности утверждений мы должны рассмотреть три типа переходов.

(1) Получение сообщения <**mydist**, .,.> . Первое выполнение результирующего кодового фрагмента, как принято, выполняется автоматически и рассматривается отдельным переходом. Обратите внимание что в данном переходе принимается сообщение и возможно множество сообщений отправляется

(2) Отказ канала и обработка сообщения < **fail**. . > узлами на обоих концах канала.

(3) Восстановление канала и обработка сообщения <**repair.** . > двумя соединенными узлами.

Лемма 4.14 *Для всех* u_o, w_o, *и* v_o, $P(u_o, w_o, v_o)$ — *инвариант.*

Доказательство. Изначально, т.е., после выполнения инициализационной процедуры каждым узлом, (1) содержится предположением. Если изначально мы имеем $\neg up(u_o, wo)$, (2) и (3) тривиально содержатся. Если изначально мы имеем $up(u_o, w_o)$, тогда $ndis_{uo}[w_o, v_o] = N$. Если $w_o = v_o$ то $D_{wo}[wo] = 0$ но сообщение < mydist, v_o, 0> в Q_{w0u0}, таким образом (2) и (3) истинны. Если $w_o \neq v_o$ то $D_{w0}[v_o] = N$ и нет сообщений в очереди, что также говорит что (2) и (3) содержаться. Мы рассмотрим три типа констатированных переходов упомянутых выше.

Тип (1). Предположим что *u* получает сообщение <**mydist**,*v,d*> от *w*. Следовательно нет топологических изменений и нет изменений в множестве *Neigh* , следовательно (1) остается истинно. Если $v \neq v_o$ то это сообщение не меняет ничего в $P(u_o, w_o, v_o)$. Если $v = v_o$, $u = u_o$, и $w = w_o$ значение $ndis_{u,}[w_o, v_o]$ может изменится. Однако, если сообщение < **mydist**, v_o, .> остается в канале значение этого сообщения продолжает удовлетворять (2), так как (2) в сохранности то и (3) также потому что посылка ложна. Если полученное сообщение было последним этого типа в канале то $d = D_{w0}[vo]$ по (2), которое подразумевает что заключение (3) становится истинным и (3) в сохранности. Посылка (2) становится ложной, таким образом (2) в сохранности. Если

$v = v_o$, $u = w_o$ (и u_o сосед u) заключение (2) или (3) может быть ложно если значение $D_{w0}[v_o]$ изменилось в следствие выполнения *Recompute(v)* в w_o. В этом случае, однако, сообщение < **mydist**, v_o, . > с новым значением посылается к u_o, которое подразумевает что посылка (3) нарушена, и заключение (2) становится истинным, таким образом (2) и (3) сохранены. Это происходит и в случае когда сообщение < **mydist**, v_o, . > добавляется в Q_{w0u0}, и это всегда удовлетворяет $d = D_{w0}[v_o]$. Если $v = v_o$ и $u \neq u_o$, w_o ничего не изменяет в $P(u_o, w_o, v_o)$.

Тип (2). Предположим что канал uw отказал. Если $u = u_o$ и $w = w_o$ этот отказ нарушил посылку (2) и (3) таким образом эти правила сохранены. (1) в безопасности потому что w_o удалился из $Neigh_{u0}$ и обратно. Нечто произойдет если $u = w_o$ и $w = u_o$. Если $u = w_o$ но $w \neq u_o$ заключение (2) или (3) может быть нарушено так как значение $D_{w0}[v_o]$ изменилось. В этом случае пересылка сообщения < **mydist**, v_o, . > узлом w_o опять нарушит посылку (3) и сделает заключение (2) истинным, следовательно (2) и (3) в безопасности. Во всех других случаях нет изменений в $P(u_o, w_o, v_o)$.

Тип (3). Предположим добавление канала uw. Если $u = u_o$ и $w = w_o$ то $up(v_o, w_o)$ истинно, но добавлением w_o в $Neigh_{u0}$ (и обратно) это защищает(1). Посылка < **mydist**, v_o, $D_{w0}[v_o]$ > узлом w_o делает заключение (2) истинным и посылку (3) ложной, таким образом $P(u_o, w_o, v_o)$ обезопасен. Во всех других случаях нет изменений в $P(u_o, w_o, v_o)$.

Лемма 4.15 *Для каждого* u_q *и* v_o, $L(u_o, v_o)$ *–инвариант.*

Доказательство. Изначально $D_{u0}[uo] = 0$ и $Nb_u[u_o] = local$. Для $v_o \neq u_o$, изначально $ndis_u[w, v_o] = N$ для всех $w \in Neigh_u$, и $D_{u0}[v_o] = N$ и $Nb_{u0}[vo] = udef$.

Тип (1). Положим что u получил сообщение < **mydist**, v,d> от w. Если $u \neq u_o$ или $v \neq v_o$ нет переменных упомянутых изменениях $L(u_o, v_o)$. Если $u = u_o$ и $v = v_o$ значение $ndis_u[w, v_o]$ меняется, но $D_{u0}[v_o]$ и $Nb_{u0}[v_o]$ перевычисляется точно так как удовлетворяется $L(v_o, v_o)$.

Тип (2). Положим что канал uw отказал. Если $u = u_o$ или $w = u_o$ то $Neigh_{u0}$ изменился, но опять $D_{u0}[v_o]$ и $Nb_{u0}[v_o]$ перевычисляются точно так как удовлетворяется $L(u_o, v_o)$.

Тип (3). Положим что добавлен канал uw . Если $u = u_o$ то изменился$Neigh_{u0}$ добавлением w, но так как u устанавливает $ndis_{u0}[w, v_o]$ в N это сохраняет $L(u_o, v_o)$.

4.3.2 Корректность алгоритма Netchange

Должны быть доказаны два требования корректности.

Теорема 4.16 *Когда достигнута стабильная конфигурация, таблицы* $Nb_u[v]$ *удовлетворяют :*

(1) *если* $u = v$ *то* $Nb_u[v] = local$;

(2) *если путь от u до* $v \neq u$ *существует то* $Nb_u[v] = w$, *где w первый сосед u на кратчайшем пути от u до v;*

(3) *если нет путь от u до v не существует то* $Nb_u[v] = udef$.

Доказательство. Когда алгоритм прекращает работу, предикат **stable** добавляется к $P(u,w,v)$ для всех u, v, и w, и это подразумевает что для всех u, v, и w

$$up(u, w) \Rightarrow ndis_u[w, v] = D_w[v]. \qquad (4.2)$$

Применив также $L(u, v)$ для всех u и v мы получим

$$D_u[v] = \begin{cases} \mathbf{0} & \text{если } u \neq v \\ 1 + \mathbf{min}_{w \in \text{Neigh u}} D_w[v] & \text{если } u \neq v \wedge \exists w \in Neigh_u : D_w[v] < N-1 \\ \mathbf{N} & \text{если } u \neq v \wedge \forall \text{w} \in Neigh_u : D_w[v] \geq N-1 \end{cases} \qquad (4.3)$$

которого достаточно для доказательства что $D_u[v] = d(u, v)$ если u и v в некотором связном компоненте сети, и $D_u[v] = N$ если u и v в различных связных компонентах.

Во-первых покажем индукцией по $d\{u, v)$ что если u и v в некотором связном компоненте то $D_u[v] \leq d(u, v)$.

Случай $d(u, v) = 0$: подразумевает $u = v$ и следовательно $D_u[v] = 0$.

Случай $d(u, v) = k + 1$: это подразумевает что существует узел $w \in Neigh_u$ с $d(w, v) = k$. По индукции $D_u[v] \leq k$, которое по (4.3) подразумевает что $D_u[v] \leq k + 1$.

Теперь покажем индуктивно по $D_u[v]$ что если $D_u[v] < N$ то существует путь между u и v и $d(u, v) \leq D_u[v]$.

Случай $D_u[v] = 0$: Формула (4.3) подразумевает что $D_u[v] = 0$ только для $u = v$, что дает пустой путь между u и v, и $d(u, v) = 0$.

Случай $D_u[v] = k + 1 < N$: Формула (4.3) подразумевает что существует узел $w \in Neigh_u$ с $D_w[v] = k$. По индукции существует путь между w и v и $d(w, v) \leq k$, что подразумевает существование пути между u и v и $d(u, v) < k + 1$.

Следовательно что если u и v в некотором связном компоненте то $D_u[v] = d(u, v)$, иначе $D_u[v] = N$. Это, Формула (4.2), и $\forall u,v : L(u, v)$ и доказывает теорему о $Nb_u[v]$. []

Докажем что стабильная ситуация в конечном счете достигается если топологические изменения прекращаются, норм-функция в отношении **stable** определена. Определим, для конфигурации алгоритма γ,

$t_i =$ (число сообщений <**mydist**, .,i>) +
(число упорядоченных пар u, v таких что $D_u[v] = i$)

и функцию f

$$f(\gamma) = (t_o, t_1, \ldots, t_N)$$

$f(\gamma)$ кортеж из $(N + 1)$ натуральных чисел, в некотором лексиграфическом порядке (обозначенном $\leq_l$) . напомним что $(\mathbf{N}, \leq_l)$ хорошо-обоснованное множество (Упражнение 2.5).

Лемма 4.17 *Обработка сообщения* < **mydist,**.,. > *уменьшает f.*

Доказательство. Пусть узел u с $D_u[v] = d_1$ получил сообщение < **mydist,** v, d_2> , и после перевычислил новое значение $D_u[v] = d$. Алгоритм подразумевает что $d < d_i + 1$.

Случай $d < d_1$: Пусть $d = d_1 + 1$ что подразумевает что t_{d2} уменьшилось на 1 (и t_{d1} следовательно), и только t_d с $d > d_1$ увеличилось. Это подразумевает что значение f уменьшилось.

Случай $d = d_1$: Не новое сообщение < **mydist**, ., .> посланное u, и влияет только на f так что t_{d2} уменьшилось на 1, т.о. значение f уменьшилось.

Случай $d > d_1$: Пусть t_{d1} уменьшилось на 1 (и t_{d2} следовательно), и только t_d с $d > d_1$ увеличилось. Это подразумевает что значение f уменьшилось.

[]

Теорема 4.18 *Если топология сети остается неизменной после конечного числа топологических изменений, то алгоритм достигнет стабильной конфигурации за конечное число шагов.*

Доказательство. Если сетевая топология остается постоянной только обрабатывая сообщения <**mydist**,.,.>которые имеют место, и, по предыдущей лемме, значение f уменьшается с каждым таким переходом. Это следует из хорошей обоснованности области f в которой может быть только конечное число таких переходов; следовательно алгоритм достигает стабильной конфигурации после конечного числа шагов. []

4.3.3 Обсуждение алгоритма

Формальные результаты правильности алгоритма, гарантирующие сходимость для исправления таблиц за конечное время после последнего топологического изменения, не очень хорошо показывают фактическое поведение алгоритма. Предикат **stable** может быть ложным практически долгое время (а именно, если топологические изменения часты) и когда **stable** ложен, ничто не известно о таблицы маршрутизации. Они могут содержать циклы или даже давать ошибочную информацию относительно достижимости узла назначения. Алгоритм поэтому может только применятся, где топологические изменения настолько редки, что время сходимости алгоритма является малым по сравнению с средним временем между топологических изменений. Тем более , потому что **stable** - глобальное свойство, и устойчивые конфигурации алгоритма неразличимы от неустойчивых для узлов. Это означает, что узел никогда не знает, отражают ли таблицы маршрутизации правильно топологию сети, и не может отсрочить отправления пакетов данных , пока устойчивая конфигурация не достигнута.

Асинхронная обработка уведомлений. Было этой части предположили что уведомления о топологических изменениях обрабатываются автоматически вместе с именением в единой транзакции. Обработка происходит на обоих сторонах удаленного или добавленного канал одновременно. Лампорт [Lam82] выполнил анализ мелких деталей и учел задержки в обработке этих уведомлений. Коммуникационный канал от w до u смоделирован объединением трех очередей.

(1) OQ_{wu}, выходная очередь w,

(2) TQ_{wu}, очередь сообщений (и пакетов данных) передаваемая

(3) IQ_{wu}, входная очередь u.

При нормальном функционировании канала, w посылает сообщение к u добавлением его в OQ_{wu}, сообщения двигаются от OQ_{wu} к TQ_{wu} и от TQ_{wu} к IQ_{wu}, и u

получает их удаляя из IQ_{wu}. Когда канал отказывает сообщения в TQ_{wu} выбрасываются и сообщения в OQ_{wu} соответственно выбрасываются раньше чем добавились к TQ_{wu}. Сообщение < **fail,** *w*> становится в конец IQ_{wu}, и когда нормальное функционирование восстановилось сообщение <**repair**,*w*> также становится в конец IQ_{wu}. предикаты *P(u, w, v)* принимают слегка более сложную форму но алгоритм остается тот же самый.

Маршрутизация по кротчайшему пути. Возможно означить вес каждого канала и модифицировать алгоритм так чтобы вычислять кратчайший путь вместо пути с минимальным количеством шагов. Процедура *Recompute* алгоритма Netchange использовать вес канала *uw* в вычислении оценки длины кратчайший путь через *w* если константу 1 заменить на w_{uw}. Константа *N* в алгоритме должна быть заменена верхней границей диаметра сети.
Довольно просто показать что когда модифицированный алгоритм достигнет стабильной конфигурации таблицы маршрутизации будут корректны и содержать оптимальный путь (все циклы в сети должны иметь положительный вес). Доказательство что алгоритм действительно достигает такого состояния требует более сложной нормфункции.
Даже возможно расширить алгоритм для работы с изменяющимися весами каналов; реакция узла *u* на изменение веса канала – перевычисление $D_u[v]$ для *v*. Алгоритм был бы практическим, однако, только в ситуации когда среднее время изменений стоимостей каналов больше времени сходимости что не реально. В этих ситуациях должна алгоритм должен предпочесть гарантию свободы от циклов в течение сходимости, на пример алгоритм Мерлина-Сигалла.

4.4 Маршрутизация с Компактными Таблицами маршрутизации

Обсужденные алгоритмы маршрутизации требуют что бы каждый узел содержал таблицу маршрутизации с отдельной ссылкой для каждого возможного пункта назначения. Когда пакет передается через сеть эти таблицы используются каждом узле пути (исключая пункта назначения). В этой части рассматриваются некоторые организации таблиц маршрутизации которые хранение и поиск механизмов маршрутизации. Как эти таблицы маршрутизации могут быть вычислены распределенными алгоритмами здесь не рассматриваются. Для простоты представления положим что сеть связная.
Стратегию уменьшения таблицы в каждом из трех механизмов маршрутизации, обсуждаемых в этой части, просто объяснить следующим образом. Если таблицы маршрутизации узла хранят выходящий канал для каждого пункта назначения отдельно, таблица маршрутизации имеет длину *N*; следовательно таблицы требуют $\mathbf{\Omega}(N)$ бит, не важно как компактно выходящий канал закодирован для каждого пункта назначения. Теперь рассмотрим перестройку таблицы: таблица содержит для каждого *канала* узла ссылку говорящую какие пункты назначения должны быть смаршрутизированны через этот канал; смотри Рисунок 4.11. Таблица теперь имеет "длину" *deg* для узел с *deg* каналами; теперь компактность зависит от того как компактно множество пунктов назначения для каждого канала может быть представлено.

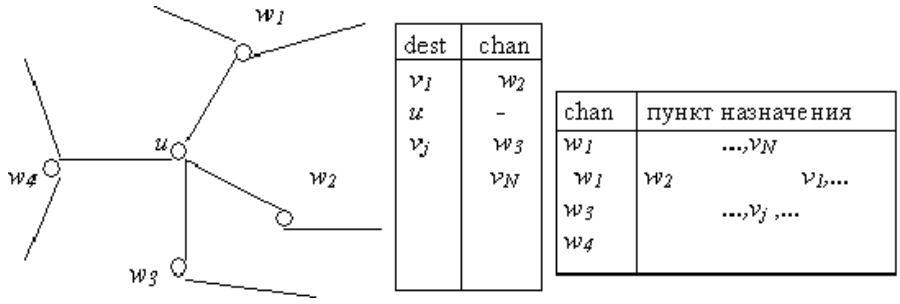


Рисунок 4.11 Уменьшение размера таблицы маршрутизации.

4.4.1 Схема разметки деревьев

Первый метод компактной маршрутизации был предложен Санторо и Кхатибом [SK85]. Метод основан на пометке узлов целыми от 0 до N— I, таким образом чтобы множество пунктов назначения для каждого канала было интервалом. Пусть $\mathbf{Z}_N$ обозначает множество {0, 1,..., N- 1}. В этой части все арифметические операции по модулю N, т.е., (N— 1) + 1≡ 0.

Определение 4.19 *Циклический интервал* [a, b) *в* $\mathbf{Z}_N$ *– множество целых определенное как*

$$[a,b)=\begin{cases} \{a, a+1,..., b-1\} & \text{если } a<b \\ \{0,\ldots, b-1, a,..., N-1\} & \text{если } a\geq b \end{cases}$$

Заметим что [a, a) = $\mathbf{Z}_N$, и для a ≠ b дополнение [a, b) – [b, a). Циклический интервал [a, b) называется *линейным* если $a < b$.

Теорема 4.20 *Узлы дерева T могут быть пронумерованы таким образом что для каждого выходящего канала каждого узла множество пунктов назначения которые маршрутизируются через данный канал есть циклический интервал.*

Доказательство. Возьмем произвольный узел *vo* за корень дерева и для каждого *w* пусть обозначим за *T[w]* поддерево *T* с корнем в w. Возможно перенумеровать узлы таким образом чтобы для каждого *w* числа для означивания узлов в *T[w]* составляли линейный интервал, на пример префиксным обходом дерева как на Рисунке 4.12. В этом порядке, *w* – первый узел дерева *T[w]* который посетится и после *w* все узлы *T[w]* посетятся перед узлами не входящими в *T[w]*; следовательно узлы в *T[w]* перенумерованы линейным интервалом [l_w, l_w +|*T{w]*|) (*1*$_w$ метка w*)*.
Через [a_w, b_w) обозначим интервал чисел означивающие узлы в *T[w]*. Сосед *w* – один из двух сыновей или отец *w*. Узел *w* передает к сыну *u* пакеты с пунктом назначения в *T[u],* т.е., узлы с числами в [a_u, b_u). Узел *w* передает своему отцу пакеты с пунктами назначения не в *T[w],* т.е., узлы с номерами в $\mathbf{Z}_N \setminus [a_w,b_w) = [b_w,a_w)$. []

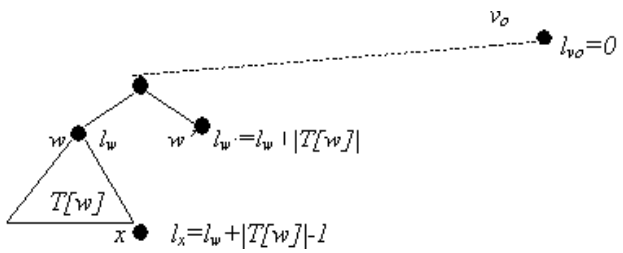


Рисунок 4.12 Префиксный обход дерева

Циклический интервал может быть представлен использование только 2 log N бит дающих начальный и конечный пункт. Так ка в этом приложении объединение разъединенных интервалов в объединении $\mathbf{Z}_N$ должно хранится, достаточно *logN* бит на интервал. Хранится только начальная точка интервала для каждого канала; конечная точка эквивалентна начальной точке следующего интервала в том же узле. Начальная точка интервала приписанного каналу *uw* в узле *u* дается как

$$\alpha_{uw} = \begin{cases} l_w & \text{если } w \text{ сын } u, \\ l_u+|T[u]| & \text{если } w \text{ отец } u \end{cases}$$

Положим что каналы узла *u* со степенью deg_u помечены $\alpha_1,...,\alpha_{deg\,u}$, где $\alpha_1 <$ $,...,< \alpha_{deg\,u}$, передающая процедура дана в Алгоритме 4.13. Метки каналов отображают $\mathbf{Z}_N$ в сегменты deg_u , каждый соответствует одному каналу; см. Рисунок 4.14. Заметим что существует (не более чем) один интервал который не линейный. Если метка отсортированы в узле, соответствующая метка находится за O(log deg_u) шагов используя бинарный поиск. Индекс *i* вычисляется по модулю deg_u, т.е., $\alpha_{deg\,u+1} == \alpha_1$.

(* пакет с адресом *d* был получен в узле *u* *)
if $d = l_u$
then доставить пакет локально
else begin выбрать α_i, т.ч. $d \in [\alpha_i, \alpha_{i+1})$;
послать пакет через канал с меткой α_i
end

Алгоритм 4.13 Интервальная передача (для узла *u).*

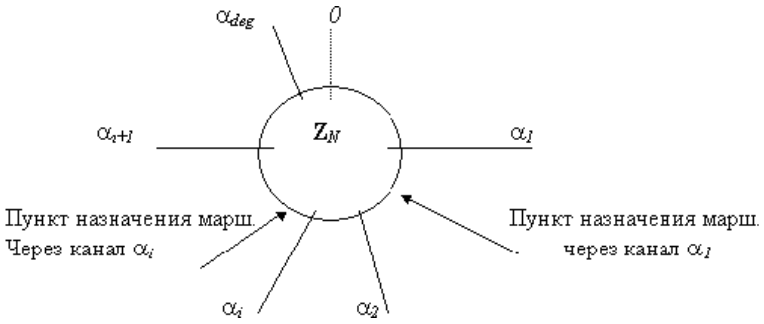


Рисунок 4.14 Часть $\mathbf{Z}_N$ в узле.

Схема разметки деревьев маршрутизирует оптимально через деревья, потому

что в дереве существует только один простой путь между каждыми двумя узлами. Схема может также использоваться если сеть не является деревом. Выбирается фиксированное дерево охвата T в сети, и схема применяется на этом дереве. Каналы не принадлежащие этому дереву никогда не используются; каждый помечен специальной меткой в таблице маршрутизации чтобы показать что нет пакетов проходящих через этот канал.
Сравним длины путей выбранное этой схемой с оптимальными путями, пусть $d_T(u, v)$ обозначает расстояние от *u* до *v* в *T* и $d_G(u, v)$ расстояние от *u* до *v* в *G*. Пусть D_G обозначает *диаметр G,* определенный как максимум для любых *u* и *v* из $d_G(u, v)$.

Лемма 4.21 *Нет общего ограничения пропорции между* $d_T(u, v)$ *и* $d_G(u, v)$.*Это имеет силу только в специальном случае измерения переходами путей.*

Доказательство. Выберем *G* как кольцо из *N* узлов, и заметим что дерево охвата *G* получается удалением одного канала, скажем *xy,* из G. Теперь $d_G(x, y) = 1$ и $d_T(x, y) = N-1$,таким образом пропорция *N—1*. Пропорция может быть гораздо больше выбором большего кольца. []

Следующая Лемма полагается на симметричность стоимостей каналов, т.е., это значит что $w_{uw} = w_{wu}$.Это подразумевает что $d_G(u, v) = d_G(v, u)$ для всех *u* и *v*.

Лемма 4.22 *T может быть выбрано таким образом чтобы для всех u и v,* $d_T(u, v) \leq 2G$.

Доказательство. Выберем *T* как оптимальное дерево стока для узла w_o (как в Теореме 4.2). Тогда

$$d_T(u, v) \leq d_T(u, w_o) + d_T(w_o, v)$$
$$= d_T(u, w_o) + d_T(v, w_o) \text{ по симметричности } w$$
$$= d_G(u, w_o) + d_G(v, w_o) \text{ по симметричности } T$$
$$= D_G + D_G \text{ по определению } D_G$$

[]

В заключении , а путь выбранный одной схемой может быть плохом если сравнить с оптимальным путём между этими же двумя узлами (Лемма 4.21), но если выбрано подходящее дерево охвата, он в s не более чем два раза хуже пути между двумя другими узлами в системе(Лемма 4.22).
Схема разметки деревьев имеет следующие недостатки.
(1) Каналы не принадлежащие *T* не используются
(2) Трафик сосредоточен в дереве,(может произойти перегрузка).
(3)Каждый отказ канала делит сеть.

4.4.2 Интервальная маршрутизация

Ван Ливен и Тэн [LT87] расширили схему разметки деревьев до сетей не являющихся деревьями таким образом что каждый канал используется для передачи пакета.

Определение 4.23 *Схема разметки деревьев (ILS)для сети это*

(1) *обозначение различными метками из* $\mathbf{Z}_N$ *узлов сети, и,*

(2) *Для каждого узла, обозначение различными метками из* $\mathbf{Z}_N$ *каналов данного узла*

Алгоритм интервальной маршрутизации предполагает что ILS дана, и пакеты передаются как в Алгоритме 4.13.

Определение 4.24 *Схема интервальной разметки применим если все пакеты передаются этим путем которым достигнут своего пункта назначения.*

Можно показать что применимая схема интервальной разметки существует для каждой связной сети G (Теорема 4.25); для произвольной связной сети, однако, схема обычно не очень эффективна. Оптимальность путей выбранных схемой интервальной маршрутизации будет изучена после существующего доказательства.

Теорема 4.25 *Для каждой связной сети G применимая схема интервальной разметки существует.*

Доказательство. Схему применимой интервальной пометка построим расширением схемы разметки деревьев Санторо и Кхатиба, применив к дереву охвата сети T . В данном дереве охвата *ребро ветви* – ребро которое не принадлежит дереву охвата. Более того, *v потомок и* если только $u \in T[v]$. Основная проблема как означить метки ребер ветвей (ребра дерева будут помечены схемой разметки деревьев), дерево охвата выбирается таким образом чтобы все ребра ветвей приняли ограниченную форму

Лемма 4.26 *Существует дерево охвата такое что между узлом потомком этого узла.*

Доказательство. Каждое дерево охвата полученное обходом в глубину через сеть имеет это свойство; смотри [Tar72] и Часть 6.4. []

В продолжении, пусть T будет зафиксированное дерево охвата поиска в глубину в G.

Определение 4.27 *Поиск в глубину ILS для G (в отношении к T) – схема разметки для которой выполняются следующие правила.*
(1) *Метки узлов означены префиксным обходом в T, т.е., узлы в поддереве T[w] помечены числами из* $[l_w, , l_w+|T[w]|)$. *Обозначим* $k_w = l_w + |T[w]|$.
(2) *Метку ребра uw в узле и обозначим* α_{uw}.
- (a) *Если uw ребро ветви то* $\alpha_{uw} = l_w$
- (b) *Если w сын и (в T) то* $\alpha_{uw} = l_w$
- (c) *Если w отец и то* $\alpha_{uw} = k_u$ *если* $k_u \neq N$ *и* ***u*** *имеет ветвь к корню. (В*

*последней ситуации, ребро ветви помечаем 0 в **и** по правилу (а),таким образом означивание метки k_u нарушило бы требование что все метки различны. Метки считаются по модулю N, т.е. $N \equiv 0$.)*

(d) *Если w отец и, и имеет ветвь к корню, и* $k_u \equiv N$, *тогда* $\alpha_{uw} = l_w$.

Все примеры поиска в глубину ILS даны на Рисунке 4.15. Заметим что все ребра ветвей помечены по правилу (2a), ребра к отцам узлов 4, 8, и 10 помечены по правилу (2c), и ребро к отцу узла 9 помечено по правилу (2d).
Теперь покажем верность схемы обхода в глубину ILS . Заметим что $v \in T[u] \Leftrightarrow l_v \in [l_u, k_u)$. Следующие три леммы относятся к ситуации когда узел *и* передает пакет с пунктом назначения *v* к узлу *w* (соседу *и)* используя Алгоритм 4.13. Это подразумевает что $l_v \in [\alpha uw, \alpha)$ для некоторой метки α в *и,* и что нет метки $\alpha' \neq \alpha_{uw}$ в узле *и* такой что $\alpha' \in [\alpha_{uw}, l_v)$.

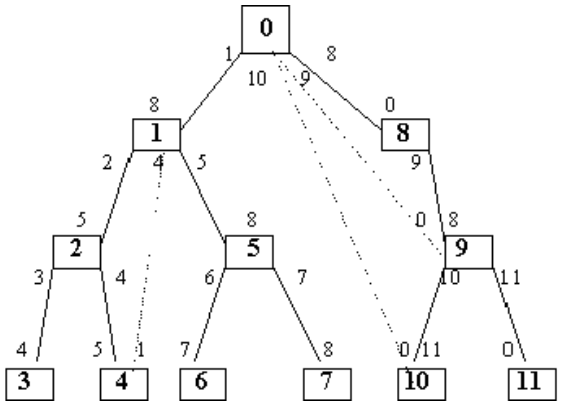


Рисунок 4.15 Интервальная разметка поиском в глубину.

Лемма 4.28 *Если* $l_u > l_v$ *то* $l_w < I_u$

Доказательство. Во-первых, рассмотрим случай $\alpha_{uw} \leq l_v$. Узел *w* не сын *и* потому что в этом случае $\alpha_{uw} = l_w > l_u > l_v$.Если u*w* ветвь то также $l_w = \alpha_{uw} < l_v < l_u$. Если *w* отец *и* то $l_w < l_u$ выполняется в любом случае. Во-вторых, рассмотрим случай когда α_{uw} – наибольшая метка ребра в *и*, и нет метки $\alpha' \leq l_v$ (т.е., l_v – нижняя граница нелинейного интервала). В этом случае ребро к отцу *и* не помечено 0, но имеет метку k_u (потому что $0 \leq l_v$, и нет метки $\alpha' \leq l_v$). Метка k_u – наибольшая метка в данном случае; ребро к сыну или ветви вниз *w′* имеет $\alpha_{uw'} = l_{w'} < k_u$, и ветвь к предком *w′* имеет $\alpha_{uw'} = l_{w'} < l_u$ Таким образом *w* – отец *и* в данном случае, что подразумевает $l_w < l_u$. []

Следующие две леммы относятся к случаю когда $l_u < l_v$. Мы выведем что каждый $v \in T[u]$ или $l_v > k_u$, и в последнем случае $k_u < N$ имеет силу так что ребро к отцу *и* помечено k_u.

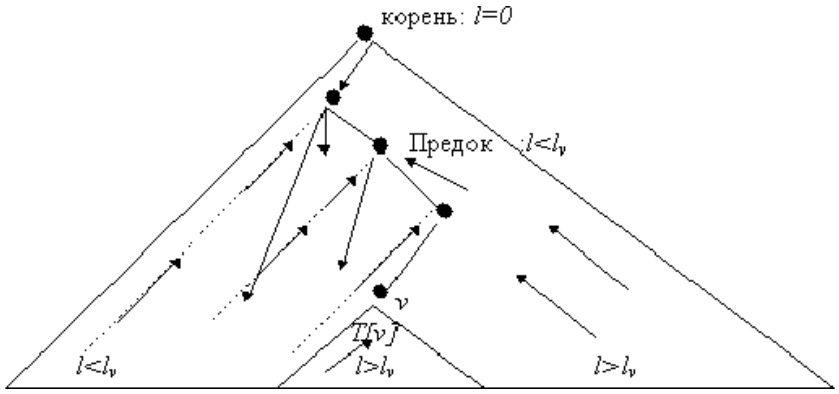


Рисунок 4.16 МАРШРУТИЗАЦИЯ ПАКЕТОВ ДЛЯ *V* В СХЕМЕ РАЗМЕТКИ ILS.

Лемма 4.29 *Если* $l_u < l_v$ *то* $l_w \le l_v$.

Доказательство. Во-первых, рассмотрим случай когда $v \in T[u]$; пусть w' сын u такой что $v \in T[w']$. Мы имеем $\alpha_{uw'} = l_{w'} \le l_v$ и это подразумевает что $\alpha_{uw'} \le \alpha_{uw} \le l_v < k_{w'}$. Мы вывели что w не является отцом u, следовательно $l_w = \alpha_{uw}$, что подразумевает $l_w < l_v$.

Во-вторых, рассмотрим случай когда $l_v \ge k_u$. В этом случае w отец u, это можно показать следующим образом. Ребро к отцу помечено k_u, и $k_u \le l_v$. Ребро к сыну w' узла u помечено $l_{w'} < k_u$, ребро к ветви вниз w' помечено $l_{w'} < k_u$, и ребро ветви вверх w' помечено $l_{w'} < l_u$. Поэтому w отец u, $l_w < l_u < l_v$. []

Нормфункция относящаяся к передаче в v может быть определена следующим образом. *Наименьший общий предок* двух узлов u и v это наименьший узел в дереве который отец u и v. Пусть lca(u, v) означает метку наименьшего общего предка u и v, и определим

$$f_v(u) = (-\text{lca}(u, v), l_u).$$

Лемма 4.30 *Если* $l_u < l_v$ *то* $f_v(w) < f_v(u)$.

Доказательство. Во-первых рассмотрим случай когда $v \in T[u]$, что подразумевает lca(u, v) = l_u. Если w' сын u такой что $v \in T[w']$, мы имеем (как в предыдущей Лемме) что $l_{w'} \le l_w < k_{w'}$, следовательно $w \in T[w']$, что подразумевает lca(u, v) $\ge l_{w'} > l_u$. Таким образом $f_v(w) < f_v(u)$.

Во-вторых, рассмотрим случай что $l_v > k_u$. Как в предыдущей лемме, w отец u, и поэтому $v \notin T[u]$, lca(w, v) = lca(u, v). Но теперь $l_w < l_u$, таким образом $f_v(w) < f_v(u)$. []

Может быть показано что каждый пакет достигает пункта назначения. Поток пакетов для v показан на Рисунке 4.16. Пусть пакет для v сгенерирован в узле u. По Лемме 4.28, метка узла уменьшается с каждым переходом, до тех пор пока, за конечное число шагов, пакет получен узлом w с $l_w \le l_v$. Каждый узел к которому пакет пересылается после w также имеет метку $\le l_v$ по лемме 4.29. За конечное число шагов пакет получает v, потому что с каждым шагом f_v уменьшается или пакет прибывает в v, по Лемме 4.30. Это завершает доказательство Теоремы 4.25. []

Эффективность интервальной маршрутизации: общий случай. Теорема 4.25 говорит что корректная ILS существует для каждой сети, но не предполагает ничего о эффективности путей выбранных схемой. Ясно что ILS с поиском в глубину используется для демонстрации *существования* схемы для каждой сети, но что они не обязательно лучшие возможные схемы. На пример, если схема с обходом в глубину применена к кольцу из N узлов, существуют узлы u и v с $d(u, v) == 2$, и схема использует $N — 2$ переходов для передачи пакета от u к v (Упражнение 4.8). Существует ILS для того же кольца что которая пересылает каждый пакет через путь с минимальным количеством шагов. (Теорема 4.34).

Определение 4.31 *ILS оптимальна если она передает все пакеты через оптимальные пути.*
ILS общительна если она передает пакет от одного узла к соседу данного узла за один шаг.
ILS линейна если интервал для передачи в каждом ребре линейный.

Мы называли ILS *с минимальным количеством шагов* (или *кротчайший путь)* если она оптимальна относительно оценки пути мерой минимальным количеством шагов (или *кратчайший путь*, соответственно). Просто показать что если схема удовлетворяет мере минимального количества шагов то схема общительна. Также легко проверить что ILS линейна тогда и только тогда когда в каждом узле u с $l_u \neq 0$ существует ребро с пометкой 0, и в узле с пометкой 0 существует ребро с меткой 0 или 1. Это показывает что для сетей в общем виде качество методов маршрутизации плохое, но для некоторых классов специальной сетевой топологии качество схемы очень неплохое. Это делает метод процессорных сетей с регулярной структурой, которые используются для реализации параллельных вычислений с виртуальной общей разделяемой памяти.
Не известно точно как, для произвольной сети, лучшие схемы интервальной разметки сравниваются с оптимальными алгоритмами маршрутизации. Некоторые нижние границы длин путей, как подразумевают оптимальные ILS не всегда существуют, было дано Ружечкой(Ruzecka).

Рисунок 4.1Т Граф-паук с тремя ногами

Теорема 4.32 [Ruz88] *Существует сеть G такая что для каждой верной ILS в G существуют узлы* ***u*** *и* ***v*** *такие что пакет от* ***u*** *к* ***v*** *доставлен только после по крайней мере $3/2D_G$ переходов.*

Известно как лучшие схемы ILC с поиском в глубину сравниваются с общими лучшими схемами ILS для этих же сетей. Упражнение 4.7 дает очень плохую схему ILS с поиском в глубину для сети которая действительно допускает опти-

мальную ILS (по Теореме 4.37), но может существовать лучшая схема ILS с поиском в глубину для такой сети.
В ситуациях когда большинство соединений происходят ммежду соседями, будучи общительными достаточные требования для ILS. Так можно показать из Рисунка 4.15 схема ILS с поиском в глубину не обязательно общительна; узел 4 передает пакеты для узла 2 через узел 1.
Это существенно для применимости метода интервальной маршрутизации, которым рассматриваются циклические интервалы. Хотя некоторые сети допустимы, и даже оптимальны, схемы с линейноинтервальной маршрутизации, не возможно маркировать каждую сеть линейными интервалами. Применимость схемы линейноинтервальной разметки была исследована Бэккером, Ван Лиуином, и Таном [BLT91].

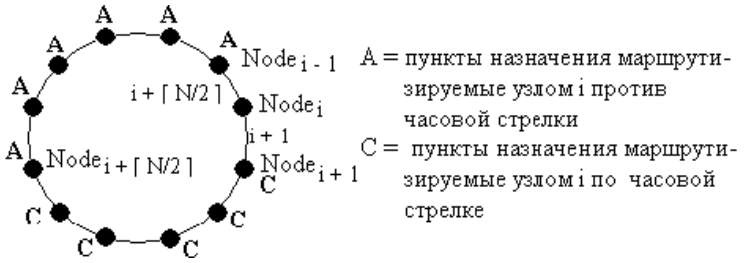


Рисунок 4.18 Оптимальная ILS для кольца

Теорема 4.33 *Существует сеть для которой нет применимой схемы линейноинтервальной разметки*

Доказательство. Рассмотрим граф-паук с тремя ногами длины 2, как нарисовано на Рисунке 4.17. Наименьшей матка (0) и наибольшей метка (6) означены два узла, и так как всего три ноги, существует(по крайней мере) одна нога которая не содержит ни меньшую ни большую метку. Пусть *x* будет первым узлом отцентра в этой ноге. Узел *x* передает пакета адресованные к 0 и 6 в центр, и единственный линейный интервал который содержит и 0 и 6 это полное множество $\mathbf{Z}_N$. Следовательно, *x* также пересылает пакеты для своих соседей через центр, и эти пакеты никогда не достигнут своих пунктов назначения. []

Бэккер, Ван Лиуин и Тан полностью описали класс сетей топологии которых допускают линейные схемы ILS кротчайших путей и представили результаты содержащие классы графических топологий которые допускают адаптацию и линейные схемы ILS с минимальным количеством шагов линейны.

Оптимальность интервальной маршрутизации: специальные топологии. Было показано что существуют оптимальные схемы интервальной разметки для некоторых классов сетей имеющих регулярную структуру. Сети таких структур используются , например, в реализации параллельных вычислений.

Теорема 4.34 [LT87] *Существует схема ILS с минимальным количеством шагов для кольца из N узлов.*

Доказательство. Метки узлов означены от 0 до *N* — I по часовой стрелке. Для

узла *i* канал по часовой стрелке означен меткой *i* +1 и канал против часовой стрелке означен (*i*+ *[N/2]*) mod *N,* см Рисунок 4.18. С этой схемой разметки узел с меткой *i* посылает пакеты для узлов *i+1, ..., (i*+ *[N/2]*) *-1* через канал по часовой стрелке и пакеты для узлов *(i* + *[N/2]*), . . . , *i — 1* через канал против часовой стрелке, что является оптимальным. []

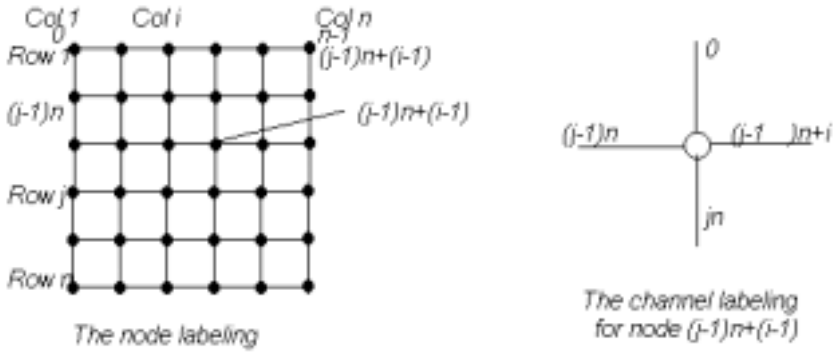


Рисунок 4.19 Оптимальная ILS для сетки *n* x n.

Так как ILS ів Доказательстве Теоремы 4.34 оптимальна , она общительна; она линейна.

Теорема 4.35 [LT87] *Существует схема ILS с минимальным количеством шагов для сетки n* x *n* .

Доказательство. Метки узлов означены по рядам в возрастающем порядке, т.е., і-ый узел в j-ом ряду помечен *(j - l)n + (i* - 1*)*. Канал вверх этого узла помечен 0, канал налево этого узла помечен *(j* - l)n, канал направо помечен (J - l)n + *i,* и канал вниз помечен *j n,* см Рисунок 4.19. Теперь легко проверить что когда узел *u* передает пакет к узлу *v,*
Случай 1: если *v* в ряду большем чем *u,* тогда *u* посылает пакет через свой канал наверх;
Случай 2: если *v* в ряду меньшем чем *u,* тогда *u* посылает пакет через свой канал вниз;
Случай 3: если *v* в том же ряду что и *u* но левее, *u* посылает пакет через свой левый канал; и
Случай 4: если *v* в том же ряду что и *u* но правее, то *u* посылает пакет через свой канал направо.
Во всех случаях , *u* посылает пакет к узлу ближайшему к *v,* что и подразумевает что выбранный путь оптимальный. []

Так как ILS в Доказательстве Теоремы 4.35 оптимальна, он общительна; то схема также линейна.

Теорема 4.36 *Существует линейная схема ILS с минимальным количеством шагов для гиперкуба.*

Теорема 4.37 [FJ88] *Существует схема ILS кротчайших путей для непланарных сетей с произвольными весами каналов.*

Интервальная маршрутизация имеет некоторые привлекательные преимущества, как следствия, над механизмами классической маршрутизации основанными на хранении привилегированных каналов отдельно для каждого пункта назначения.

(1) *Малая пространственная сложность.* Таблицы маршрутизации могут хранится в *0(deg* • log *N)* бит для узла степенью *deg.*

(2) *Эффективность вычислений таблиц маршрутизации.* Таблицы маршрутизации для схем ILC с поиском в глубину могут быть вычислены используя распределенный обход сети в глубину, который может использовать O(*E*) сообщений за время *0(N)* ; см Часть 6.4.

(3) *Оптимальность.* Метод маршрутизации способен выбирать оптимальный петь в некоторых классах сетей, см. Теоремы с 4.34 до 4.37.

Эти преимущества делают метод применимым для процессорных сетей с регулярной топологией. Транспьютеры часто используются для конструирования таких топологий, маршрутизационные чипы Инмос 104 (смотри Раздел 1.1.5) разработаны для использования интервальной маршрутизации.

К сожалению, для сетей с произвольной топологией, когда методы используют схемы ILS с поиском в глубину присутствуют несколько минусов:

(1) *Плохая живучесть.* Не возможна легкая адаптация схемы ILS с поиском в глубину при добавлении или удалении узла в сети. Дерево ILS не может долго удовлетворять требованию по которому ветвь существует только между узлом и его предком. В результате минимальное изменение топологии сети может потребовать полного перевычисления таблиц маршрутизации, включая вычисление новых адресов (меток) для каждого узла.

(2) *Не оптимальность.* Схема ILS поиска в глубину может направлять пакет через пути длиной **Ω***(N),* даже в случаях сетей с малым диаметром; см Упражнение 4.7.

(* А пакет с адресом *d* был получен или создан узлом *u* *)
if d = l_u
then обработать пакет локально
else begin α_i **:=** самая ддлинная матка канала т.что $\alpha_i < d$;
послать пакет через канал помеченый α_i
end

Алгоритм 4.20 Префиксная передача (для узла *u).*

4.4.3 Префиксная маршрутизация

Рассмотрев недостатки интервальной маршрутизации, Бэккер, Ван Лиуин, и Тан [BLT93] разработали метод маршрутизации в котором таблицы могут быть вычислены используя произвольное дерево охвата. Использование неограниченного дерева охвата может увеличить как живучесть так и эффективность. Если а канал добавлен между двумя существующими узлами то дерево охвата

остается деревом охвата а новый канал будет ветвью. Если новый узел добавляется вместе с некоторым количеством каналов соединяющих его с существующими узлами то дерево охвата расширяется используя один из каналов и новый узел. Остальные каналы становятся ветвями. Оптимальность может быть улучшена выбором дерева охвата малой глубины (как в лемме 4.22.)

Как метки узлов и каналов в префиксной маршрутизации предпочтительнее использовать строки чем целые числа, используемые в интервальной маршрутизации. Пусть Σ – алфавит; в последствии метка будет строкой над Σ, σ обозначает пустую строку, и Σ^* множество строк над Σ. Для отбора канала для передачи пакета, алгоритм рассматривает все каналы которые являются *префиксами* адреса пункта назначения. Выбирается самая длинная из этих меток, и выбранный канал используется для передачи пакета. На пример, предположим что узел имеет каналы с метками **aabb**, **abba**, **aab**, **aabc**, и **aa**, и должен переслать пакет с адресом **aabbc**. Метки каналов **aabb**, **aab**, и **aa** являются префиксами **aabbc**, и самая длинная из этих трех меток **aabb**, следовательно узел передаст пакет через канал помеченный **aabb**. Алгоритм передачи дан как Алгоритм 4.20. Мы пишем $\alpha < \beta$ для обозначения что α префикс β.

Определение 4.38 *Схема префиксной разметки (над Σ) для сети G это:*
(1) *обозначение различными строками из Σ^* узлов G; и*
(2) *Для каждого узла, означивание различными строками каналов данного узла.*

Алгоритм префиксной маршрутизация предпалагает что схема префиксной разметки (PLS) дана, и перенаправляет пакеты Алгоритмом 4.20.

Определение 4.39 *Схема префиксной разметки приемлема если все пакеты в конечном счете достигнут своих пунктов назначения.*

Теорема 4.40 *Для каждой связной сети G существует приемлемая схема PLS .*
Доказательство. Мы определим класс схем префиксной разметки и докажем, как и в Теореме 4.25, что схемы в этом классе приемлемы. Пусть T обозначает произвольное дерево охвата в G.

Определение 4.41 *Дерево T схемы PLS для G это схема префиксной разметки при которой выполняются следующие правила.*
(1) *Метка корня* – σ.
(2) *Если w сын u то* l_w *расширяет* l_u *одним символом; т.е., если* $u_1, \ldots, u_k$ *сын u в T то* $l_{ui} = l_u\alpha_i$ *где* $\alpha_1, \ldots, \alpha_k$ – *k различных символов из* Σ.
(3) *Если uw ветвь то* $\alpha_{uw} = l_w$
(4) *Если w сын u то* $\alpha_{uw} = l_w$.
(5) *Если w отец u то* $\alpha_{uw} = \sigma$ *если* ***u*** *не имеет ветви к корню: в этом случае,* $\alpha_{uw} = l_w$

В дереве PLS каждый узел исключая корень имеет канал помеченный σ, и этот канал соединяет узел с предком (отцом узла или корнем дерева). Заметим что для каждого канала uw, $\alpha_{uw} = l_w$ или $\alpha_{uw} = \sigma$. Для всех u и v, v предок у тогда и только тогда когда $l_v < l_u$.

Нужно показать что пакет никогда не "вклинивается" в узел отличный от его пункта назначения, который, каждый узел отличный от пункт назначения может перенаправить пакет используя Алгоритм 4.20.

Лемма 4.42 *Для всех узлоа* ***u*** *и* ***v****таких что* $u \neq v$ *существует канал в* ***u*** *помеченный префиксом* l_v.

Доказательство. Если u не корень T то ***u*** имеет канал помеченный σ, который является префиксом l_v. Если u корень тогда ***v*** не является корнем, и $v \in T[u]$. Если w сын ***u*** такой что $v \in T[w]$ то по построению $a_{uw} < l_v$. []

Следующие три леммы имеют отношение к ситуации когда узел u передает пакет для узла v к узлу w (соседу u) используя Алгоритм 4.20.

Лемма 4.43 *Если* $u \in T[v]$ *то* ***w*** *предок* ***u***.

Доказательство. Если $a_{uv} == \sigma$ то ***w*** предок ***u*** как упоминалось выше. Если $a_{uw} = l_w$ то, так как $a_{uw} < l_v$, также $l_w < l_v$ Это подразумевает что w предок v, и также ***u***.[]

Лемма 4.44 *Если u предок v то w предок v, ближе к v чем u.*
Доказательство. Пусть w' будет сыном u таким что $v \in T[w']$ тогда $a_{uw'} = l_w$ не пустой префикс l_v. Так как a_{uw} самый длинный префикс (в u) l_v, то $a_{uw'} < a_{uw} < l_v$, таким образом w предок v ниже u. []

Лемма 4.45 *Если* $u \notin T[v]$, *то* ***w*** *предок* ***v*** *или* $d_T(w, v) < d_T(u, v)$.

Доказательство. Если $a_{uw} = \sigma$ то ***w*** отец ***u*** или корень; отец u ближе к v чем u потому что $u \notin T[v]$, и корень – предок v. Если $a_{uw} = l_w$ то , так как $a_{uw} < l_v$, w – предок v. []

Пусть *depth* будет обозначать глубину T, т.е., число переходов в самом длинном простом пути от корня к листьям. Может быть показано каждый пакет с пункт назначения v прибудет в свой пункт назначения за не более чем 2 - *depth* переходов. Если пакет создан в предке v то v достигнется не более чем *depth* переходов по лемме 4.44. Если пакет создан в поддереве $T[v]$ тогда предок v достигнется за не более чем *depth* переходов по лемме 4.43, после которых v достигнется за другие *depth* переходов по предыдущему замечанию. (По причине того что путь содержит только предков источника в этом случае, его длина ограничена также *depth* .) Во всех других случаях предок v достигнется в пределах *depth* переходов по лемме 4.45, после которого v достигнется в пределах других *depth* переходов. (Таким образом, в этом случае длина пути ограничена 2 *depth.)* Это завершает Доказательство Теоремы 4.40
[]

Следствие 4.46 *Для каждой сети G с диаметром* D_G *(измеренным в переходах) существует схема префиксной разметки которая доставляет все пакеты за не более чем* $2D_G$ *переходов.*

Доказательство. Воспользуемся деревом PLS что качается дерева выбранного в Лемме 4.22. []

Мы включили обсуждение схему разметки деревьев с грубым анализом его пространственных требований. Как и раньше, *depth* – глубина T, и пусть *k* будет максимальным количеством сыновей дюбого узла *T.* Тогдае самая длинная метка состоит из *depth* символов, и **Σ** должен содержать (по крайней мере) *k* символов, метка может храниться в *depth* • log A бит. Таблица маршрутизации а узла с *deg* каналами хронится в *0(deg* depth* * log *k)* бит.
Несколько другая схема префиксной разметки бала предложена Бэккером. [BLT93]. Его статья также характеризует класс топологий который допускает *оптимальные* схемы префиксной разметки когда веса связей могут меняться динамически.

4.5 Иерархическая маршрутизация

Путь сокращения различных параметров стоимости метода маршрутизации - использование иерархического разделения сети и метода ассоциативной иерархической маршрутизации. Цель в большинстве случаев состоит в том, чтобы использовать факт, что многие связи в сетях компьютеров являются локальными, то есть, между узлами на относительно малых расстояниях друг от друга. Некоторые из параметров стоимости метода маршрутизации зависят от размера полной сети скорее чем длина выбранного пути, почему, мы теперь объясним
(1) *Длина адресов.* Так как каждый из *N* узлов имеет отличный от других адрес, каждый адрес состоит из по крайней мере log *N* бит; может потребоваться даже больше бит если существует информация включенная в адреса, такая как в префиксной маршрутизации.
(2) *Размер таблицы маршрутизации.* В методах маршрутизации описываемые в разделах 4.2 и 4.3, таблица содержит ссылку на каждый узел, и таким образом имеет линейный размер.
(3) *Цена табличного поиска.* Цена простого табличного поиска больше для большей таблицы маршрутизации или для больших адресов. Полное время табличного поиска для обработки простого сообщения также зависит от количества обращений к таблице.

В методе иерархической маршрутизации , сеть разделена на кластера, каждый кластер есть связное подмножество узлов. Если источник и пункт назначения пакета в одном кластере, цена передачи сообщения низка, потому что пакет маршрутизируется внутри кластера, кластер трактуется как небольшая изолированная сеть. Для метода описанного в разделе 4.5.1, в каждом кластере зафиксирован простой узел (*центр* кластера) который может делать наиболее сложные маршрутизационные решения необходимые для пересылки пакетов в другие кластера. Таким образом, большие таблицы маршрутизации и манипуляция длинными адресами необходима только в центрах. Каждый кластер сам может разделиться на подкластеры для многоуровневого деления узлов.
Не необходимо но желательно чтобы каждая коммутация между кластерами велась через центр; этот тип конструкции имеет такой недостаток что весь кластер становится уязвимым на отказ центра. Лентферт [LUST89] описал метод

иерархической маршрутизации в котором все узлы в равной степени могут посылать сообщения другим кластерам. Также метод использует только маленькие таблицы, потому что ссылки на кластеры к которым узел не принадлежит трактуется как простой узел. Овербух [ABNLP90] использует парадигму иерархической маршрутизации для конструирования класса схем маршрутизации которые всегда балансируют между эффективностью и пространственными требованиями.

4.5.1 Уменьшение количества решений маршрутизации

Все обсужденные методы маршрутизации требуют чтобы решения маршрутизации делались в каждом промежуточном узле, что подразумевает что для маршрута длиной l происходит l обращений к таблицам маршрутизации. Для стратегий минимального количества шагов l ограничено диаметром сети, но в общем, стратегии маршрутизация без циклов (такие как интервальная маршрутизация) $N—1$ – лучшая граница которая может быть достигнута. В этом разделе мы обсудим метод с помощью которого табличные поиски могут быть уменьшены.
Мы будем использовать следующую лемму, которая подразумевает существование подходящего разбиения сети на связные кластера.

Лемма 4.47 *Для каждого* $s \le N$ *существует разбиение сети на кластера* $C_1,...,$ C_m *такие что*
(1) *каждый кластер – связный подграф,*
(2) *каждый кластер содержит по крайней мере s узлов, и*
(3) *каждый кластер имеет радиус не более чем 2s.*

Доказательство. Пусть $D_1, ..., Dm$ будет максимальная коллекция разделенных связных подграфов таких что каждый D_i имеет радиус $\le s$ и содержит по крайней мере s узлов. Каждый узел не принадлежащий $\bigcup_{i=1}^{m} D_i$ соединен с одним из подмножеств путем длиной не больше чем s, иначе муть может быть добавлен как отдельный кластер. Сформируеи кластеры C_i включением каждого узла не входящего в $\bigcup_{i=1}^{m} D_i$ в кластер ближайший к нему. Расширенные кластеры остаются содержат по крайней мере s узлов каждый, они остаются связными и разделенными, и они имеют радиус не более чем 2s. []

Метод маршрутизации означивает цветом каждый пакет, и предполагается что используется только несколько цветов. Узлы теперь действуют следующим образом. В зависимости от цвета, пакет или отправляет немедленно по установленному каналу (соответствующему цвету) или вызывает более сложному решению. Это подразумевает, что узлы имеют различные протоколы для обработки пакетов.

Теорема 4.48 [LT86] *For каждой сети из N узлов существует метод маршрутизации который требует не более чем* $0(\sqrt{N})$ *решений маршрутизации для*

каждого пакета, и использует три цвета.

Доказательство. Предположим что решения (по Лемме 4.47) даны и заметим что каждый C_i содержит узел c_i такой что $d(v, c_i) \leq 2s$ для каждого $v \in C_i$ потому что C_i имеет радиус не более чем 2s. Пусть T будет поддеревом минимального размера из G соединяющее все c_i. Так как T минимально то оно содержит не более чем m листьев, следовательно оно содержит не более чем m-2 узлов разветвлений (узлы степенью большей чем 2); см Упражнение 4.9. Рассмотрим узла T как *центры* (c_i), узлы разветвлений, и узлы пути.

Метод маршрутизации сначала посылает пакет к центру c_i кластера источника (зеленая фаза), затем через T к центру c_j кластера пункта назначения (синяя фаза), и наконец внутри C_j к пункту назначения (красная фаза). Зеленая фаза использует фиксированному дерево стока для центра каждого кластера, и не решений маршрутизации. Узлы пути в T имеют два инцидентных канала, и передают каждый синий пакет через канал ыв дереве которые не принимают пакет. Узлы ветвлений и центры в T должны принимать решения маршрутизации. Для красной фазы используется стратегия кротчайшего пути внутри кластера, которая ограничивает число решений в этой фазе до $2s$. Это ограничивает число решений маршрутизации до $2m - 2 + 2s$, что не более чем $2N/s - 2 + 2s$. Выбор $s \approx \sqrt{N}$ дает ограничение $0(\sqrt{N})$. []

Теорема 4.48 устанавливает границу общего числа решений маршрутизации необходимого для обработке каждого пакета, но не полагается не любой практический алгоритм с помощью которого эти решения принимаются. Метод маршрутизации использованный в T может быть схемой маршрутизации деревьев Санторо и Кхатиба, но так же возможно применить принцип кластеризации к Т тем самым уменьшив число решений маршрутизации даже больше.

Теорема 4.49 [LT86) *Для каждой сети из N узелs и каждого положительного целого числа $f \leq \log N$ существует метод маршрутизации который требует не более чем* $O(f N^{1/f})$ *решений маршрутизации для каждого пакета, и использует* $2f + 1$ *цветов.*

Доказательство. Доказательство подобно доказательству теоремы 4.48, но вместо выбора $s \approx \sqrt{N}$ конструирование применяется рекурсивно к дереву T (оно кластер размера s). Дерево – связная сеть, по существу $< 2m$ узлов потому что узлы пути в T только перенаправляют пакеты из одного фиксированного канала в другой, и может быть игнорирован.
Кластеризация повторяется f раз. Сеть G имеет N узлов. Дерево содержит после одного уровня кластеризации не более чем N/s центров и N/s узлов ветвления, т.е., $N.(2/s)$ необходимых узлов. Если дерево полученное после i уровней кластеризации имеет m_i необходимых узлов, тогда дерево полученное после $i+1$ уровней кластеризации имеет не более чем m_i/s центров и m_i/s узлов ветвлений, т.е., m_i.(2/s) необходимых узлов. Дерево полученное после f уровней кластеризации имеет не более чем $m_f = N.(2/s)^f$ необходимых узлов.
Каждый уровень кластеризации увеличивает количество цветов на два, следовательно с f уровнями кластеризации будут использоваться $2f$+1 цветов. Не более

чем $2m_f$ решений необходимо на самом высоком уровне, и s решений необходимо нв каждом уровне кластеризации в кластере пункта назначения, отсюда количество решений маршрутизации $2m_f + fs$. Выбирая $s \approx 2N^{1/f}$ получим $m_f = O(1)$, следовательно число решений маршрутизации ограничено $fs = O(fN^{1/f})$. []

Использование приблизительно log$N$ цветов приводит к методу маршрутизации которые требуют O(log$N$) решений маршрутизации.

Упражнения к Части 4

Раздел 4.1

Упражнение 4.1 *Предположим что таблицы маршрутизации обновляются после каждого топологического изменения таким образом чтобы они были без циклов даже во время обновлений. Дает ли это гарантию что пакеты всегда обработаются даже когда сеть подвергается бесконечному числу топологических изменений ? Докажите что алгоритм маршрутизации может гарантировать доставку пакетов при продолжающихся топологических изменениях.*

Раздел 4.2

Упражнение 4.2 *Студент предложил пренебречь посылкой сообщения* $<$ **nys,** $w>$ *из алгоритма 4.6; он аргументировал это тем что узел знает своего соседа и не сын в* T_w, *если нет сообщения* $<$***ys***$,w>$ *принятого от этого соседа. Можно ли модифицировать алгоритм таким образом? Что случиться со сложностью алгоритма?*

Упражнение 4.3 *Докажите что следующее утверждение является инвариантом алгоритма Чанди-Мизра для вычисления путей до* v_o *(Алгоритм 4.7).*

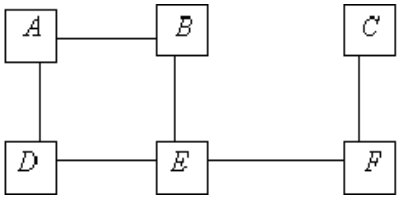
$\forall u, w$ **:** $<$***mydist***$,v_o,d> \in M_{wu} \Rightarrow d(w, v_o) \leq d$

$\wedge \forall u$ **:** $d(u, v_o) \leq D_u[vo]$

Дайте пример для которого число сообщений экспоненциально относительно числа каналов сети.

Раздел 4.3

Упражнение 4.4 *Дайте значения всех переменных терминального состояния алгоритма Netchange когда алгоритм применяется к сети со следующей топологией:*



После того как терминальное состояние достигнуто, добавляется канал между A и F. Какое сообщение F пошлет к A когда обработает уведомление $<$**repair,** $A>$ *? Какие сообщения A пошлет после получения сообщений от F?*

Раздел 4.4

Упражнение 4.5 *Дайте пример демонстрирующий что Лемма 4.22 не имеет силу для сетей с ассиметричной стоимостью каналов.*

Упражнение 4.6 *Существует ли ILS которая использует не все каналы для маршрутизации? Она применима? Она оптимальна?*

Упражнение 4.7 *Дайте граф G и дерево T поиска в глубину графа G такие что G имеет* $N = n^2$ *узлов, диаметр G и глубина T – 0(n), и существуют узлы* ***u*** *и* ***v*** *такие что пакет от* ***u*** *к* ***v*** *доставляется за N — 1 переходов схемой ILS поиска в глубину. (Граф может быть выбран таким образом что G непланарный, что предполагает (по Теореме 4.37) что G действительно имеет оптимальную ILS.)*

Упражнение 4.8 *Дайте схему ILS поиска в глубину для кольца из N узлов. Найдите узлы* ***u*** *и* ***v*** *такие что* $d(u, v) = 2$, *и схема использует N — 2 переходов для передачи пакета от* ***u*** *к* ***v****.*

Раздел 4.5

Упражнение 4.9 *Докажите что минимальность дерева T в доказательстве Теоремы 4.48 предполагает что оно имеет не более чем* ***m*** *листьев. Докажите что любое дерево с* ***m*** *листьями имеет не более чем* ***m — 2*** *узлов ветвления.*

5 Беступиковая коммутация пакетов

Сообщения (пакеты), путешествующие через сеть с коммутацией пакетов должны сохраняться в каждом узле перед тем, отправлением к следующему узлу на пути к адресату. Каждый узел сети для этой цели резервирует некоторый буфер. Поскольку количество буферного места конечно в каждом узле, могут возникнуть ситуации, когда никакой пакет не может быть послан потому, что все буферы в следующем узле заняты, как иллюстрируется Рисунком 5.1. Каждый из четырех узлов имеет буфера *B*, каждый из которых может содержать точно один пакет. Узел *s* послал *t B* пакетов с адресатом *v*, и узел *v* послал *u B* пакетов с адресатом *s*. Все буфера в *u* и *v* теперь заняты, и, следовательно, ни один из пакетов, сохраненных в *t* и *u* не может быть послан к адресату.
Ситуации, когда группа пакетов никогда не может достигнуть их адресата, потому что они все ждут буфера, в настоящее время занятого другим пакетом в группе, называются как *тупики с промежуточным накоплением*. (Другие типы тупика будут кратко рассмотрены в конце этой главы.) Важная проблема в проектировании сетей с пакетной коммутацией - что делать с тупиками с промежуточным накоплением. В этой главе мы разберем несколько методов, называемых *контроллерами*, которые могут использоваться для того, чтобы избежать возможности тупиков с промежуточным накоплением, вводя ограничения на то, когда пакет может быть сгенерирован или послан. Методы избежания тупи-

ков с промежуточным накоплением найдены в сетевом уровне модели OSI (Подраздел 1.2.2).

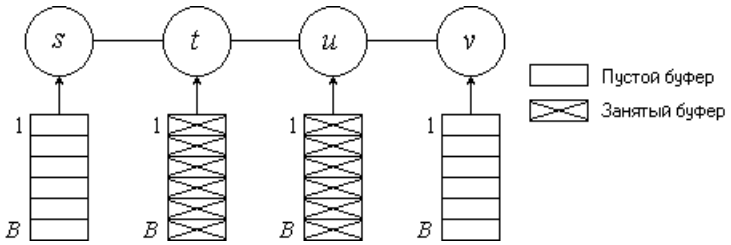


Figure 5.1 Пример тупика с промежуточным накоплением.

Два вида методов, которые мы рассмотрим, базируются на *структурированных* и *неструктурированных* буферных накопителях.

Методы, использующие структурированные буферные накопители (Раздел 5.2) идентифицируют для узла и пакета специфический буфер, который должен быть использован, если пакет генерируется или принимается. Если этот буфер занят, пакет не может быть принят. В методах, использующих неструктурные буферные накопители (Раздел 5.3) все буфера равны; метод только предписывает, может или нет пакет быть принят, но не определяет, в который буфер он должен быть помещен. Некоторые нотации и определения представляются в Разделе 5.1, и мы закончим главу обсуждением дальнейших проблем в Разделе 5.4.

5.1 Введение

Как обычно, сеть моделируется графом $G = (V, E)$; расстояние между узлами измеряется в переходах. Каждая вершина имеет B буферов для временного хранения пакетов. Множество всех буферов обозначается B, и символы b, c, b_u, и т.д., используются для обозначения буферов.

Обработка пакетов узлами описывается с помощью трех типов перемещений, которые могут происходить в сети.

Генерация. Вершина u "создает" новый пакет p (на самом деле, принимая пакет от протокола более высокого уровня) и размещает его в пустом буфере в u. Вершина u в таком случае называется *источником* пакета p.

Продвижение. Пакет p продвигается от вершины u в *пустой* буфер следующей в его маршруте вершины w (маршрут определяется, используя алгоритмы маршрутизации). В результате передвижения буфер, прежде занятый p становится пустым. Хотя контроллеры, которые мы определим, могут запретить передвижения, предполагается, что сеть всегда позволяет это движение, т.е., если контроллер его не запрещает, то оно применимо.

В системах с синхронной передачей сообщений это перемещение, как легко видно, является одиночным переходом, как в Определении 2.7. В системах с асинхронной передачей сообщений, перемещение - не один переход, как в Определении 2.6, но оно может быть выполнено, например, следующим образом. Узел u неоднократно передает p к w, но не отбрасывает пакет из буфера, пока не получено подтверждение. Когда узел w получает пакет, он решает, примет ли он пакет в одном из буферов. Если примет, пакет помещается в буфер, и посылается подтверждение u, иначе пакет просто игнорируется. Конечно, могут быть разработаны более эффективные протоколы для выполнения таких перемещений, например те, где u не передает p, пока u не уверен, что w примет p. В лю-

бом случае перемещение состоит из нескольких переходов типа упомянутых в Определении 2.6, но в целях этой главы оно будет рассматриваться как одиночный шаг.
(3) *Выведение.* Пакет *p*, занимающий буфер в вершине назначения, удаляется из буфера. Предполагается, что сеть всегда позволяет это передвижение.
Обозначим через P множество всех путей, по которым следуют пакеты. Это множество определяется алгоритмами маршрутизации (см. Главу 4); как это делается, нас здесь не интересует. Пусть *k* - количество переходов в самом длинном пути в P. Это не предполагает, что *k* равен диаметру G; *k* может превосходить диаметр, если алгоритм маршрутизации не выбирает кратчайшие пути, и *k* может быть меньше диаметра, если все коммуникации между узлами происходят на ограниченных дистанциях.
Как видно из примера, данного в начале этой главы, тупики могут возникнуть, если разрешены произвольные перемещения (исключая тривиальное ограничение, что *u* должна иметь пустой буфер, если в *u* генерируется пакет и *w* должна иметь пустой буфер, если пакет продвигается в *w).* Теперь мы определим контроллер как алгоритм, разрешающий или запрещающий различные движения в сети в соответствии со следующими требованиями.
(1) Выведение пакета (в месте его назначения) всегда позволяется.
(2) Генерация пакета в вершине, в которой все буферы пустые, всегда позволяется.
(3) Контроллер использует только локальную информацию, т.е., решение, может ли пакет быть принят в вершине *u*, зависит только от информации, известной *u* или содержащейся в пакете.
Второе требование исключает тривиальное решение избежания заблокированных пакетов (см. Определение 5.2), отказываясь принимать какие-либо пакеты в сети. Как в Главе 2, пусть Z_u обозначает множество состояний вершины *u,* и *M* - множество возможных сообщений (пакетов).
Определение 5.1 *Контроллер для сети G = (V, E)-набор пар* ***con***$=\{Gen_u, For_u\}_{u \in V}$, *где* $Gen_u \subseteq Z_u \times M$ *и* $For_u \subseteq Z_u \times M$. *Если* $c_u \in Z_u$ *- состояние u, где все буферы пусты, то для всех* $p \in M$, $(c_u, p) \in Gen_u$.
Контроллер **con** позволяет генерацию пакета *p* в вершине *u,* где состояние *u* - c_u, тогда и только тогда, когда $(c_u, p) \in Gen_u$, и позволяет продвижение пакета *p* из *u* в *w* тогда и только тогда, когда $(c_w, p) \in For_w$. Формальное определение контроллера не включает условия для выведения пакетов, потому что выведение пакета (в его месте назначения) всегда позволено. Передвижения сети под управлением контроллера **con** - это только те передвижения сети, которые разрешены **con.**
Пакет в сети в тупике, если он никогда не может достигнуть своего места назначения ни в какой последовательности передвижений.

Определение 5.2 *Дана сеть G, контроллер* **con** *для G, и конфигурация* γ *G, пакет p (возникающий в конфигурации* γ*) в тупике, если не существует последовательности передвижений под управлением* **con,** *применимой в* γ*, в которой p выводится. Конфигурация называется тупиковой, если она содержит пакеты в тупике.*

Как показывает пример на Рисунке 5.1, тупиковая ситуация существует для всех контроллеров. Задача контроллера в том, чтобы не позволить сети войти в такую конфигурацию. Начальная конфигурация сети - конфигурация, когда в сети нет пакетов.

Определение 5.3 *Контроллер беступиковый, если под управлением этого контроллера из начальной ситуации не достижима ни одна тупиковая.*

5.2 Структурированные решения

Сейчас мы обсудим класс контроллеров, полагающихся на, так называемые, буферные графы, представленные Merlin и Schweitzer [MS80a]. Идея этих буферных графов базируется на наблюдении, что (при отсутствии контроллера) тупик обусловлен ситуацией циклического ожидания. В ситуации циклического ожидания есть последовательность $p_0, ..., p_{s-1}$ пакетов, таких, что для каждого i, p_i хочет передвинуться в буфер, занятый p_{i+1} (индексы считаются modulo s). Циклическое ожидание избегается продвижением пакетов вдоль путей в ациклическом графе (буферном графе). В Подразделе 5.2.1 будут определены буферные графы и связанный с ними класс контроллеров, а также представлены два простых примера буферных графов. В подразделе 5.2.2 будет дана более запутанная конструкция буферного графа, снова с двумя примерами.

5.2.1 Буферные Графы

Пусть дана сеть G с множеством буферов B.

Определение 5.4 *Буферный граф (для G,* B*) - направленный граф BG на буферах сети, т.е., BG = (*B, $\vec{BE}$*), так, что*

(1) *BG - ациклический (не содержит прямых циклов);*

(2) *Из bc* $\in$ $\vec{BE}$ *следует, что b и c - буферы одной и той же вершины, или буферы двух вершин, соединенных каналом в G; и*

(3) *для каждого пути P* $\in$ P *существует путь в BG, чей образ (см. ниже)-P.*

Второе требование определяет отображение путей в BG на пути в G; если $b_0, b_1, ..., b_s$ - путь в BG, то, если u- вершина, в которой располагается буфер b_i, $u_0, u_1, ..., u_s$ - последовательность вершин таких, что для каждого $i < s$ либо $u_i u_{i+1} \in E$, либо $u_i = u_{i+1}$. Путь в G, который получается из этой последовательности пропусканием последовательных повторений, называется *образом* исходного пути $b_0, b_1, ..., b_s$ в BG.

Пакет не может быть помещен в произвольно выбранный буфер; он должен быть помещен в буфер, из которого он еще может достигнуть своего места назначения через путь в BG, т.е., буфер, подходящий для пакета в соответствии с определением.

Определение 5.5 *Пусть p - пакет в вершине u с пунктом назначения v. Буфер b в u подходит для p, если существует путь в BG из b в буфер c в v, чей образ - путь, которому p может следовать в G.*

Один из таких путей в BG будет называться гарантированным путем и nb(p, b) обозначает следующий буфер на гарантированном пути. Для каждого вновь сгенерированного пакета p в u Существует подходящий буфер fb(p) в u.

Здесь *fb* и *nb* - аббревиатура первого буфера (first buffer) и следующего буфера (next buffer). Заметим, что буфер *nb(p, b)* всегда подходит для p. Во всех буферных графов, используемых в этом разделе, *nb(p, b)* располагается в вершине, отличной от той, где располагается *b*. Использование "внутренних" ребер в *BG,* т.е., ребер между двумя буферами одной вершины, обсудим позже.

Контроллер буферного графа. Буферный граф *BG* может быть использован для разработки беступикового контроллера $\mathbf{bgc}_{BG}$, записывающий в каждый пакет буфер *nb(p, b)* и/или состояние вершины, где располагается *p*.

Определение 5.6 *Контроллер* $\mathbf{bgc}_{BG}$ *определяется следующим образом.*
Генерация пакета p в и позволяется тогда и только тогда, когда буфер fb(p) свободен. Сгенерированный пакет помещается в этот буфер.
Продвижение пакета p из буфера в и в буфер в w (возможно и = w) позволяется тогда и только тогда, когда nb(p, b) (в w) свободен. Если продвижение имеет место, то пакет p помещается в nb(p, b).

Theorem 5.7 *Контроллер* $\mathbf{bgc}_{BG}$ *- беступиковый контроллер.*

Доказательство. Если у вершины все буферы пусты, генерация любого пакета позволяется, откуда следует, что $\mathbf{bgc}_{BG}$ - контроллер.
Для каждого $b \in$ B*,* определим *класс буфера b* как длину самого длинного пути в *BG* , который заканчивается в *b.* Заметим, что классы буферов на пути в *BG* (а точнее, на гарантированном пути) строго возрастающие, т.е., класс буфера *nb(p, b)* больше, чем класс буфера *b.*
Т.к. контроллер позволяет размещение пакетов только в подходящих буферах и т.к. изначально нет пакетов, каждая достижимая конфигурация сети под управлением $\mathbf{bgc}_{BG}$ содержит пакеты только в подходящих буферах. С помощью индукции "сверху вниз" по классам буферов можно легко показать, что ни один буфер класса *r* не содержит в такой конфигурации тупиковых пакетов. Пусть *R* - самый большой класс буфера.

Случай $r = R$: Буфер *b* вершины *u*, имеющий самый большой из возможных классов, не имеет исходящих ребер в *BG.* Следовательно, пакет, для которого *b* - подходящий буфер, имеет пункт назначения *u* и может быть выведен, когда он попадает в буфер *b.* Значит, ни один буфер класса *R* не содержит тупиковых пакетов.

Случай $r < R$*:* Выдвинем гипотезу, что для всех r' таких, что $r < r' \leq R$, ни один буфер класса r' не содержит тупиковый пакет (в достижимой конфигурации).

Пусть γ - достижимая конфигурация с пакетом *p* в буфере *b* класса $r < R$ вершины *u.* Если *u* - место назначения *p*, то *p* может быть выведен и, следовательно, он не в тупике. Иначе, пусть *nb(p, b) = c* - следующий буфер на гарантированном пути *b,* и заметим, что класс r' буфера *c* превосходит *r*. По гипотезе индукции, *c* не содержит тупиковых пакетов, значит существует конфигурация δ, достижимая из γ, в которой *c* - пуст. Из δ *p* может продвинуться в *c*, и, по гипотезе индукции, *p* не тупиковый в результирующей конфигурации δ'. Следовательно, *p* в конфигурации γ не в тупике.

Из доказательства видно, что если гарантированный путь содержит "внутренние" ребра буферного графа (ребра между двумя буферами одной вершины), то контроллер должен позволять дополнительные передвижения, с помощью которых пакет помещается в буфер той же вершины. Обычно гарантированный путь не содержит таких ребер. Они используются только для увеличения эффективности продвижения, например, в следующей ситуации. Пакет p размещается в буфере b_1 вершины u и буфер $nb(p, b_1)$ в вершине w занят. Но существует свободный буфер b_2 в u, который подходит для p; более того, $nb(p, b_2)$ в вершине w свободен. В таком случае, пакет может быть перемещен через b_2 и $nb(p, b_2)$. Сейчас рассмотрим два примера использования буферных графов, а именно, схема адресата(destination scheme) и схема сколько-было-переходов (hops-so-far scheme).

Схема адресата. Схема адресата использует N буферов в каждой вершине u, с буфером $b_u[v]$ для каждого возможного адресата v. Вершина v называется *цель* буфера $b_u[v]$. Для этой схемы нужно предположить, что алгоритмы маршрутизации продвигают все пакеты с адресатом v по направленному дереву T_v , ориентированному по направлению к v. (На самом деле это предположение может быть ослаблено; достаточно, чтобы каналы, используемые для продвижения по направлению к v, формировали ациклический подграф G.)

Буферный граф определяется как $BG_d = (B, \vec{BE})$, где $b_u[v_1]b_w[v_2] \in \vec{BE}$ тогда и только тогда, когда $v_1 = v_2$ и uw - ребро в T_{v1}. Чтобы показать, что BG_d - ациклический, заметим, что не существует ребер между буферами с различными целями и, что буферы, с одинаковой целью v формируют дерево, изоморфное T_v. Каждый путь $P \in$ P с точкой назначения v - путь в T_v, и по построению существует путь в BG_d из буферов с целью v, чей образ - P. Этот путь выбирается в качества гарантированного. Это означает, что для пакета p с адресатом v, сгенерированного в вершине u, $fb(p) = b_u[v]$, и, если этот пакет должен продвинуться в w, то $nb(p,b) = b_w[v]$.

Определение 5.8 *Контроллер* **dest** *определяется как* $\mathbf{bgc}_{BG}$, *с fb и nb определенными как в предыдущем параграфе.*

Theorem 5.9 *Существует беступиковый контроллер для сети произвольной топологии, который использует N буферов в каждой вершине и позволяет проводить пакеты через произвольно выбранные деревья стока.*
Доказательство. **dest** - беступиковый контроллер, использующий такое количество буферов.

Как было отмечено ранее, требование маршрутизации по деревьям стока может быть ослаблено до требования того, что пакеты с одинаковыми пунктами назначения посылаются через каналы, которые формируют ациклический граф. Не достаточно, чтобы P содержало только простые пути, как это показано на примере, данном на Рисунке 5.2. Здесь пакеты из u_1 в v направляются через простой путь
$\langle u_1, w_1, u_2, \ldots, v \rangle$, и пакете из u_2 в v посылаются через простой путь
$\langle u_2, w_2, u_1, \ldots, v \rangle$.

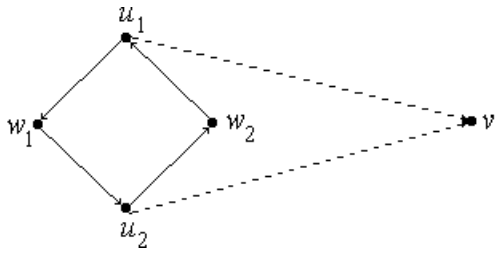


Рисунок 5.2 маршрутизация, запрещенная для контроллера **dest.**

Каждый путь в P простой; набор всех каналов, используемых для маршрутизации пакетов в v содержит цикл $\langle u_1, w_1, u_2, w_2, u_1, v \rangle$. См. Упражнение 5.2. Контроллер **dest** очень прост в использовании, но имеет недостаток - для каждой вершины требуется большое количество буферов, а именно N.

Схема сколько-было-переходов. В этой схеме вершина u содержит k +1 буфер $b_u[0], \ldots, b_u[k]$. Предполагается, что каждый пакет содержит *счетчик переходов*, показывающий, сколько переходов от источника сделал пакет.

Буферный граф определяется как $BG_h = (B, \vec{BE})$, где $b_u[i]\ b_w[j] \in \vec{BE}$ тогда и только тогда, когда $i+1 = j$ и $uw \in E$. Чтобы показать, что BG_h ациклический, заметим, что индексы буферов строго возрастают вдоль ребер BG_h. Т.к. длина каждого пути в P не более k переходов, то существует соответствующий путь в буферном графе; если $P = u_0, \ldots, u_l\ (l \le k)$, то $b_{u0}[0], b_{u1}[1], \ldots, b_{ul}[l]$-путь в BG_h с образом P. Этот гарантированный путь описывается так: $fb(p) = b_u[0]$ (для p, сгенерированного в u) и
$(p, b_u[i]) = b_w[i+1]$ для пакетов, которые должны быть продвинуты из u в w.

Определение 5.10 *Контроллер* **hsf** *определяется как* $\mathbf{bgc}_{BGh}$ *, с fb и nb определенными в предыдущем параграфе.*

Theorem 5.11 *Существует беступиковый контроллер для сетей произвольной топологии, который использует D* + 1 *буфер в каждой вершине (где D - диаметр сети), и требует, чтобы пакеты пересылались по путям с минимальным числом переходов.*

Доказательство. Использование путей с минимальным числом переходов дает $k = D$. Тогда **hsf** - беступиковый контроллер, использующий D +1 буфер в каждой вершине. (Количество буферов даже может быть меньше, если узлы, расположенные далеко друг от друга, не обмениваются пакетами.)

В схеме сколько-было-переходов буферы с индексом i используются для хранения пакетов, которые сделали i переходов. Можно спректировать двойственная схема *сколько-будет-переходов (hops-to-go)*, в которой буферы с индексом i используются для хранения пакетов, которым надо сделать еще i переходов до места назначения; см. Упражнение 5.3.

5.2.2 Ориентации G

В этом подразделе будет рассматриваться метод для построения сложных буферных графов, требующих только несколько буферов на узел,. В контроллере **hsf** индекс буфера, в котором хранился пакет, увеличивался с каждым перехо-

дом. Теперь мы замедлим рост индекса буфера (таким образом экономя на общем количестве буферов в каждом узле), предполагая увеличение индекса буфера (не путать с классом буфера) с некоторыми, но не обязательно всеми, переходами. Чтобы избежать циклов в буферном графе, каналы, которые могут быть пересечены без увеличения индекса буфера, формируют ациклический граф.

Рисунок 5.3 Граф и ациклическая ориентация.

Определение 5.12 *Ациклическая ориентация G - направленный ациклический граф, который получается ориентацией всех ребер G; см. Рисунок 5.3. Последовательность* G_1, ..., G_B *ациклических ориентаций G является покрытием из ациклических ориентаций размера B для набора* P *путей, если каждый путь* $P \in$ P *может быть записан как конкатенация B путей* P_1, ..., P_B, *где* P_i - *путь в* G_i.

Когда возможно покрытие из ациклических ориентаций размера B, может быть сконструирован контроллер, использующий только B буферов на вершину. Пакет всегда генерируется в буфере $b_u[1]$ вершины u. Пакет из буфера $b_u[i]$, который должен быть продвинут в вершину w помещается в буфер b_w [i], если дуга между u и w направлена к w в G_i , и в буфер $b_w[i + 1]$, если дуга направлена к u в G_i.

Теорема 5.13 *Если покрытие из ациклических ориентаций для* P *размера B существует, то существует беступиковый контроллер, использующий только B буферов на каждую вершину.*

Доказательство. Пусть G_1. .., G_B - покрытие, и $b_u[1]$, ..., b_u[B] - буферы вершины u. Будем писать $uw \in \bigcup E_i$, есди дуга uw направлена к w в G_i, и $wu \in \bigcup E_i$, если дуга uw направлена к u в G_i. Буферный граф определяется как $BG_a = (B, \vec{BE})$, где $b_u[i]b_w[j] \in \vec{BE}$ тогда и только тогда, когда $uw \in E$ и $(i = j \wedge uw \in \bigcup E_i)$ или $(i + 1 = j \wedge wu \in \bigcup E_i)$. Чтобы показать, что этот граф ациклический, отметим, что не существует циклов, содержащих буферы с различными индексами, потому что нет дуг от данного буфера к другому с меньшим индексом. Нет циклов с из буферов с одним и тем же индексом i , потому что эти буферы назначаются в соответствии с ациклическим графом G_i.
Оставляем читателю (см. Упражнение 5.4) продемонстрировать, что для каждого $P \in$ P существует гарантированный путь с образом P, и что такой путь описывается следующим образом:

$$fb(p) = b_u[1]$$

$$nb(p, b_u[i]) = \begin{cases} b_w[i] & if\ uw \in E_i \\ b_w[i+1] & if\ wu \in E_i \end{cases}$$

Контроллер **acoc** = **bgc**$_{BGa}$ - беступиковый контроллер, использующий B буферов в каждой вершине, что доказывает теорему.

Коммутация пакетов в кольце. Покрытия из ациклических ориентаций можно использованы при построении беступиковых контроллеров для нескольких классов сетей. Мы сначала представим контроллер для колец, использующий только три буфера на вершину. Для следующей теоремы предполагается, что веса каналов симметричны, т.е., $w_{uw} = w_{wu}$.

Теорема 5.14 *Существует беступиковый контроллер для кольцевой сети, который использует всего три буфера на вершину и позволяет маршрутизировать пакеты через самые короткие пути.*
Доказательство. Из Теоремы 5.13 достаточно дать покрытие из ациклических ориентаций размером 3 для набора путей, который включает самые короткие пути между всеми парами вершин.
Будем использовать следующую нотацию. Для вершин u и v, $d_c(u, v)$ обозначает длину пути из u в v по часовой стрелке и $d_a(u, v)$ - длина пути против часовой стрелки; $d_c(v, u) = d_a(u, v)$ и $d(u, v) = min(d_c(u, v), d_a(u, v))$ выполняется. Сумма весом всех каналов называется C (*периметр* кольца) и очевидно $d_c(u, v) + d_a(u, v) = C$ для всех u, v, *так что* $d(u, v) \leq C/2$.

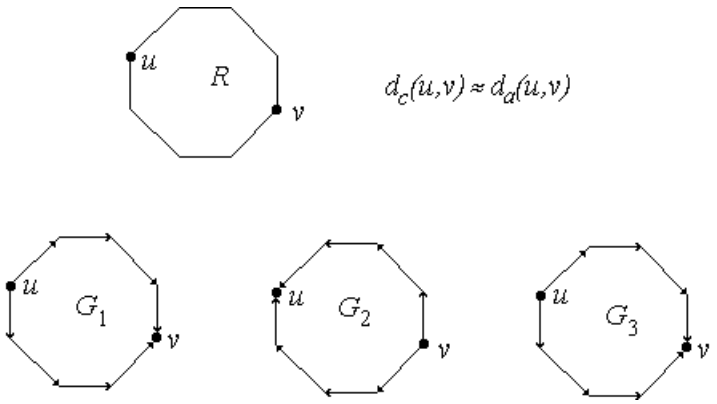


Рисунок 5.4 покрытие из ациклических ориентаций для кольца.

Во-первых рассмотрим простой случай, когда Существуют вершины u и v с $d(u, v) = C/2$. G_1 и G_3 получаются ориентацией всех ребер по направлению к v, и G_2 получается ориентацией всех ребер по направлению к u: см. Рисунок 5.4. Самый короткий путь из u в v содержится в G_1 или G_3, и наименьший путь из v в u содержится в G_2. Пусть x, y - пара вершин, отличная от пары u, v. Тогда, т.к. $d(x, y) \leq C/2$, то существует самый короткий путь P между x и y, который не содержит сразу и u, и v. Если P не сдержит ни u, ни v , то он полностью содержится либо в G_1 , либо в G_2. Если P содержит v , это конкатенация пути в G_1 и пути в G_2; если P содержит u, это конкатенация пути в G_2 и пути в G_3.

Если не существует пары *u, v* с *d(u, v) = C*/2, выберем пару, для которой *d(u, v)* как можно ближе к *C* /2. Теперь может быть показано, что если существует пара *x, y* такая, что нельзя найти самый короткий как конкатенацию путей в ориентациях покрытия, то *d(x, y)* ближе к *C* /2, чем *d(u, v).*

Пакетная коммутация в дереве. Покрытия из ациклических ориентаций могут быть использованы для построения контроллера, использующего только два буфера на вершину, для сети в виде дерева.

Теорема 5.15 *Существует беступиковый контроллер для сети в виде дерева, который использует только два буфера на вершину.*

Доказательство. Из Теоремы 5.13, достаточно дать ациклическую ориентацию для дерева, которая покрывает все простые пути. Выберем произвольную вершину *r,* и получим T_1 ориентацией всех ребер по направлению к *r* и T_2 ориентацией всех ребер от *r*; см. Рисунок 5.5.

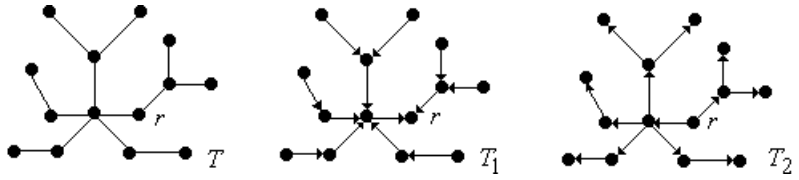


Рисунок 5.5 Покрытие из ациклических ориентаций для дерева.

Простой путь из *u* в *v* - конкатенация пути из *u* к самому нижнему общему предку, который T_1, и пути от самого меньшего общего предка к *v,* который в T_2.

5.3 Неструктурированные решения

Теперь мы обсудим класс контроллеров, предложенных Toueg и Ullman [TU81]. Эти контроллеры не предписывают, в котором буфере должен быть помещен пакет и используют только простую локальную информацию типа счетчика переходов или числа занятых буферов в узле.

5.3.1 Контроллеры с прямым и обратным счетом

Контроллер с прямым счетом (forward-count controller). Пусть (для пакета *p*) s_p -количество переходов, которые ему необходимо сделать до места назначения; конечно же $0 \le s_p \le k$ выполняется. Не всегда необходимо хранить s_p в пакете, т.к. многие алгоритмы маршрутизации хранят эту информацию в каждой вершине; см. например алгоритм Netchange Раздела 4.3. Для вершины *u,* f_u обозначает число пустых буферов в *u.* Конечно, $0 \le f_u \le B$ всегда выполняется.

Определение 5.16 *Контроллер* **FC** *(Forward-count) принимает пакет p в вершине и тогда и только тогда, когда* $s_p < f_u$.

Контроллер принимает пакет, если пустых буферов в вершине больше, чем количество переходов, которые нужно сделать пакету.

Теорема 5.17 *Если B > k, то* FC - *беступиковый контроллер.*

Доказательство. Чтобы показать, что в пустой вершине позволяется генерация пакета, заметим, что если все буферы вершины *u* пусты, $f_u = B$. Новому пакету нужно сделать не более *k* переходов, так что из $B > k$ следует, что пакет принимается.

Отсутствие тупиков **FC** будет показано методом от противного: пусть γ - достижимая конфигурация контроллера. Получим конфигурацию δ, применяя к γ максимальную последовательность из передвижений и выведения. В δ ни один пакет не может двигаться, и, т.к. γ - тупиковая конфигурация, то существует по крайней мере один пакет, оставшийся в сети в конфигурации. Пусть p - пакет в δ с минимальным расстоянием до пункта назначения, т.е., s_p - наименьшее значение для всех пакетов в δ.

Пусть u - вершина, в которой размещается p. Т.к. u не является пунктом назначения p (иначе p мог бы быть выведен), то существует сосед w вершины u , в которую нужно продвинуть p. Т. к. это передвижение не позволяется **FC**, то

$$s_p\text{-}1 \geq f_w$$

Из $s_p \leq k$ и из предположения $k < B$ следует, что $f_u < B$, что обозначает, что в вершине w располагается как минимум один пакет (в конфигурации δ). Из пакетов в w, пусть q будет последним пакетом, принятым вершиной w, и пусть f'_w обозначает количество пустых буферов в w прямо перед принятием q вершиной w. Т.к. пакет q теперь занимает один из этих f'_w буферов и (из выбора q) все пакеты, принятые вершиной w после q были выведены из w, то

$$f'_w \leq f_w + 1$$

Из принятие q вершиной w следует $s_q < f'_w$, и, комбинируя три полученных неравенства, получаем

$$s_q < f'_w \leq f_w + 1 \leq s_p,$$

что противоречит выбору p.

Контроллер с обратным счетом (backward-count controller). Контроллер, " двойственный " **FC** , получается, когда решение, принимать ли пакет, основано на количестве шагов, которые пакет уже сделал. Пусть, для пакета p, t_p - количество переходов, которые он сделал от источника. Конечно, $0 \leq t_p < k$ всегда верно.

Определение 5.18 *Контроллер* **BC** *(Backward-Count) принимает пакет p в вершине и тогда и только тогда, когда* $t_p > k—f_u$.

Доказательство того, что **BC** - беступиковый (Упражнение 5.6) очень похоже на доказательство Теоремы 5.17.

5.3.2 Контроллеры с опережающим и отстающим состоянием

Используя более детальную информацию относительно пакетов, находящихся в узле, может быть дан контроллер, который подобен контроллеру с прямым счетом, но позволяет большое количество передвижений.

Контроллер с опережающим состоянием (forward-state controller). Используем обозначение s_p как в предыдущем разделе. Для вершины u определим (как функцию состояния *u) вектор состояния* $\langle j_0, \ldots, j_k \rangle$, где j - количество пакетов p в u с $s_p = s$.

Определение 5.19 *Контроллер* FS *(Forward-State) принимает пакет p в вершине и с вектором состояния* $\langle j_0, \ldots, j_k \rangle$ *тогда и только тогда, когда*

$$\forall i, 0 \leq i \leq s_p : i < B - \sum_{s=i}^{k} j_s .$$

Теорема 5.20 *Если* $B > k$, *то* FS - *беступиковый.*

Доказательство. Оставляем читателю показать, что пустая вершина принимает все пакеты. Предположим, что существует достижимая тупиковая конфигурация γ, и получим конфигурацию δ, применяя максимальную последовательность из продвижений и выведения. Ни один пакет не может передвигаться и по крайней мере один пакет остался в δ. Выберем пакет p с минимальным значением s_p, и пусть u -вершина, в которой располагается p и w - вершина, в которую p должен продвинуться. Пусть $\langle j_0,\ldots,j_k\rangle$ - вектор состояния вершины w в δ.

Если w не содержит пакетом, то $\sum_{s=0}^{k} j_s = 0$, откуда следует, что w может принять p, что невозможно. Следовательно, w содержит по крайней мере один пакет; из пакетов в w, пусть q - пакет, расположенный ближе всего к пункту назначения, т.е., $s_q = \min\{s : j_s > 0\}$. Покажем, что $s_q < s_p$, что является противоречием. Из пакетов в w, пусть r - тот, который был принят w позже всех, конечно, $s_q \le s_r$ выполняется. Пусть $\langle j_0^{/},\ldots,j_k^{/}\rangle$ - вектор состояния w прямо перед принятием r. Из принятия r следует

$$\forall i, 0 \le i \le s_r : i < B - \sum_{s=i}^{k} j_s^{/}$$

Когда $\langle j_0^{/},\ldots,j_k^{/}\rangle$ был вектором состояния w, r был принят вершиной w. После этого пакеты могли передвигаться из w, но все пакеты, принятые в w позже, чем r были выведены (из выбора r). Из этого следует

$$j_s \le j_s^{/} \quad for \quad s \ne s_r$$
$$j_{s_r} \le j_{s_r}^{/} + 1$$

Но это означает, что

$$\forall i, 0 \le i \le s_r : i < B - \sum_{s=i}^{k} j_s + 1$$

Таким образом, принимая $i = s_q$,

$$s_q \le B - \sum_{s=s_q}^{k} j_s$$

Теперь используем факт, что p не принята w, т.е.,

$$\exists i_0, 0 \le i_0 \le s_p - 1 : i_0 \ge B - \sum_{s=i_0}^{k} j_s$$

Это дает неравенство

$s_p > s_p - 1$

$\ge \quad i_0 \qquad\qquad$ *см выше*

$\ge \quad B - \sum_{s=i_0}^{k} j_s \quad$ *см. выше*

$\ge \quad B - \sum_{s=0}^{k} j_s \quad$ *т.к.* $j_s \ge 0 \quad (u \quad i_0 \ge 0)$

$\ge \quad B - \sum_{s=s_q}^{k} j_s \quad$ *т.к.* $j_s = 0 \quad$ *для* $s < s_q$

$\ge \quad s_q$

что и является противоречием.

Контроллер с отстающим состоянием (backward-state controller.) Существует также и контроллер, "двойственный" **FS**, который использует более детальную информацию, чем контроллер **BC**, и позволяет больше передвижений.

Пусть t_p выбрано как раньше, и определим вектор состояния как $\langle i_0,\ldots,i_k\rangle$, где i_t равно количеству пакетов в вершине u, которые сделали t переходов.

Определение 5.21 *Контроллер* BS *(Backward-State) принимает пакет p в вершине и с вектором состояния* $\langle i_0,\ldots,i_k\rangle$ *тогда и только тогда, когда*

$$\forall j, t_p \le j \le k: j > \sum_{t=0}^{j} i_t - B + k$$

Доказательство того, что **BS** беступиковый очень похоже на доказательство Теоремы 5.20.

Сравнение контроллеров. Контроллер с опережающим состоянием более либеральный, чем контроллер с прямым счетом, в том плане, что он позволяет больше передвижений:

Лемма 5.22 *Каждое передвижение, позволяемое* FC *также позволяется* FS.

Доказательство. Предположим, что принятие *p* вершиной *u* позвляется контроллером **FC**. Тогда $s_p < f_u = B - \sum_{s=0}^{k} j_s$, так что для $i \le s_p$,следует $i < B - \sum_{s=i}^{k} j_s$, откуда видно, что **FS** позволяет передвижение.

D

В [TU81] было показано, что **FC** более либеральный, чем **BC**. **FS** - более либеральный, чем **BS,** и **BS** более либеральный, чем **BC**. Также было показано, что эти контроллеры самые либеральные из всех, использующих такую же информацию.

5.4 Дальнейшие проблемы

В результатах этой главы отметим, что число буферов, необходимых контроллеру, всегда играло большую роль. Пропускная способность обычно увеличивается, если доступно большое количество буферов. В неструктурированных решениях дано только нижняя граница числа буферов; большее число может использоваться без любой модификации. В структурированных решениях дополнительные буфера должны так или иначе быть вставлены в буферный граф, что может быть выполнено или *статически* или *динамически*. Если это выполнено статически, каждый буфер имеет фиксированное расположение в буферном графе, т.е., создается "более широкий" буферный граф, чем в примерах, приведенных в Разделе 5.2.

Вместо одного буфера *fb* (*p*) или *nb* (*p, b*) обычно несколько буферов определяются как возможное начало или продолжение пути через буферный граф. Если вставка дополнительных буферов выполняется динамически, то буферный граф сначала создается содержащим как можно меньшее количество буферов; буфера в графе мы называем *логическими* буферами, во время операции каждый фактический буфер (называемый *физическим* буфером) может использоваться как любой из логических буферов, но всегда должно гарантироваться, что для каждого логического буфера имеется по крайней мере один физический буферный накопитель. При такой схеме, должен быть зарезервировано только небольшое число буферов, чтобы избежать тупиков, в то время как остальная часть буферов может использоваться свободно.

В этой главе было принято, что пакеты имеют фиксированный размер: буфера одинаково большие и каждый буфер может содержать точно один пакет. Проблему может также рассматривать, предположив, что пакеты могут иметь раз-

личные размеры. Решения Раздела 5.3 были адаптированы к этому предположению Bodlaender; см. [Bod86].

5.4.1 Топологические изменения

До этого момента мы явно не рассматривали возможность топологических изменений в сети в течение путешествия пакета от источника до адресата. После возникновения такого изменения таблицы маршрутизации каждого узла будут модифицироваться, и затем пакет будет послан, используя измененные значения этих таблиц. В результате модификации таблиц, пакет может следовать за путем, которым бы не следовал, если никакие изменения не имели бы место; даже возможен случай, что окончательный маршрут пакета теперь содержит циклы. Воздействие этого на методы предотвращения тупиков, рассматриваемые в этой главе довольно не интуитивные. контроллер **Dest**, чья правильность основана на свойстве, что P содержит только простые пути, может использоваться без каких-либо модификаций. Контроллеры, которые рассматривают только верхнюю границу числа переходов, требуют дополнительных предосторожностей при использовании в этом случае.

Контроллер dest. За конечное время после последнего топологического изменения, таблицы маршрутизации сводятся к таблицам свободным от циклов. Даже не смотря на то, что ситуация циклического ожидания может существовать во время вычисления таблиц, после завершения вычислений буферный граф снова становится ациклическим, и все пакеты хранятся в подходящих буферах. Следовательно, когда таблицы маршрутизации вычислены, возникшая в результате конфигурация не содержит никаких тупиковых пакетов.

Контроллеры, подсчитывающие переходы. Рассмотрим контроллер, который полагается на предположение, что пакет должен делать не больше k переходов. Можно выбрать k достаточно большим, чтобы гарантировать с большой вероятностью, что каждый пакет достигает адресата за k переходов, даже если в течение путешествия от источника до адресата происходят некоторые топологические изменения. Для всех пакетов, которые достигают своих адресатов за k переходов, контроллеры сколько-было-переходов, с обратным счетом и с отстающим состоянием могут использоваться без какой-либо модификации. Однако, возможно, что пакет не достиг адресата после k переходов из-за неудачного образца топологических изменений и модификаций таблиц. Если дело обстоит так, пакет сохраняется в неподходящем буфере, и он будет навсегда блокирован в узле, отличном от адресата.

Такие ситуации могут быть решены только в кооперации с протоколами более высокого уровня. Самое простое решение в том, чтобы отбросить пакет. С точки зрения сквозного транспортного протокола, пакет теперь потерян; но с этой потерей можно справиться с помощью протокола транспортного уровня, как было показано в Разделе 3.2.

Отбрасывание пакетов также необходимо для выполнения предположения Раздела 3.2 о том, что максимальное время жизни пакета μ. Если пересылка пакета занимает не более μ_0 единиц времени, то из ограничения времени жизни пакета p следует ограничение $k = \mu / \mu_0$ на число переходов, которые может пройти пакет.

5.4.2 Другие виды тупиков

В этой главе рассматривались только тупики с промежуточным накоплением. Если предположения, сделанные в Разделе 5.1 имеют силу, тупики с промежуточным накоплением - единственно возможные виды тупиков. В практических сетях, однако, эти предположения не всегда выполняются и, как показали Merlin и Schweitzer [MS80b], возможны другие типы тупиков. Merlin и Schweitzer рассматривают четыре типа, а именно: тупик потомства, тупиком выпуска копии, тупик пошагового продвижения, и тупик перетрансляции, и показывают, как можно избежать эти типы тупиков расширением метода буферных графов.

Тупик потомства может возникнуть, когда пакет p может создать в сети другой пакет q, например, отчет об отказе, если произошло столкновение с испорченным каналом. Это ввело бы причинно-следственные отношение между порождением нового пакета (q) и пересылкой или выведением уже существующего (p), что нарушает предположение Раздела 5.1, что сеть всегда позволяет пересылку и выведение пакета.

Тупика потомства можно избежать, имея две копий буферного графа: одну для первоначальных сообщений и одну для вторичных сообщений. Если потомки могут снова создать следующее поколение, должны использоваться многократные уровни буферного графа.

Тупик выпуска копии может возникнуть, когда источник задерживает копию пакета, пока для пакета не получено (сквозное) подтверждение от адресата. (Сравните с протоколом основанном на таймере из Раздела 3.2, и предположите, что последовательность in_p хранится в том же самом пространстве памяти, которое используется механизмом маршрутизации для временного хранения пакетов.) Это нарушает наше предположение (в Разделе 5.1), что буфер становится пустым, когда пакет, занимающий его, продвигается.

Даны два расширения принципа буферного графа, с помощью которых можно избежать тупика выпуска копии. Первое решение полагается на предположение, что тупик выпуска копии всегда возникает из-за циклического ожидания оригинальных и подтверждающих сообщений. Решение состоит в том, чтобы обрабатывать подтверждения как потомство и хранить их в отдельной копии буферного графа. Во втором решении, которое в большинстве случаев требует меньшего количества буферов, недавно сгенерированные пакеты помещаются в специализированные исходные буфера, в которые не могут быть помещены посылаемы пакеты.

Тупик пошагового продвижения может возникнуть, когда сеть содержит узлы с ограниченной внутренней памятью, которые могут отказываться выводить сообщения, пока некоторые другие сообщения не были сгенерированы. Например, терминал телетайпа должен вывести некоторые символы прежде, чем он сможет принимать следующие символы для отображения. Это нарушает наше предположение, что пакет в адресате всегда может быть выведен.

Тупика пошагового продвижения можно избежать, делая различие между продвигаемыми пакетами и пошаговыми ответами, такими, что пакет первого типа не может быть выведен, пока пакет второго типа не был сгенерирован. Для разных типов сообщений используются различные копии буферного графа.

Тупик перетрансляции может возникнуть в сетях, где большие сообщения для передачи разделяются на более мелкие пакеты, и ни один пакет не может быть

удален из сети, пока все пакеты сообщения не достигли адресата. (Сравните с протоколом скользящего окна Раздела 3.1, где слова удаляются из out_p только если были получены все слова с меньшим индексом.) Это нарушает наше предположение, что выведение пакета в адресате всегда возможно.
Тупиков перетрансляции можно избежать, используя отдельные группы буферов для пересылки пакета и перетрансляции.

5.4.3 Лайфлок (livelock)

Из определения тупиковых пакетов (Определение 5.2) следует, что под управлением беступикового контроллера для каждого пакета существует по крайней мере одно вычисление, в котором пакет выводится. Т.к. в общем случае возможно большое количество различных вычислений, то это из этого не следует, что каждый пакет в конечном счете доходит до адресата, даже в бесконечном вычислении, как иллюстрируется Рисунок 5.6. Предположим, что *u* посылает в *v* бесконечный поток пакетов, и каждый раз, как только буфер в *w* становится пустым, принимается следующий пакет из *u*. Узел *s* имеет пакет для *t*, который не заходит в тупик, потому что каждый раз, когда буфер в *w* становится пустым, имеется возможное продолжение вычисления, в котором пакет принимается вершиной *w* и посылается к *t*. Это возможно, но не обязательно, и пакет может остаться в *s* навсегда. Ситуация этого вида называется лайфлок.
Контроллер, обсуждаемый в этой главе, может быть расширен так, чтобы вообще избежать лайфлоков.

Определение 5.23 *Дана сеть, контроллер* **con,** *и конфигурация γ, пакет p заблокирован (лайфлок), если существует бесконечная последовательность передвижений, применимых в γ, в результате которых p не выводится. Конфигурация γ - конфигурация лайфлока, если она содержит заблокированные пакеты.*

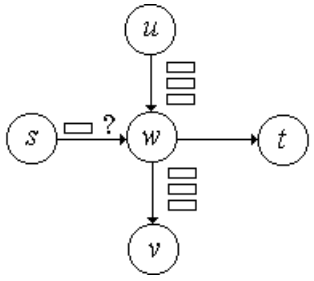


Рисунок 5.6 Пример лайфлока.

Контроллер свободен от лайфлока, если ни одна конфигурация лайфлока не достижима из конфигурации, в которой нет пакетов.
В оставшейся части этого подраздела будет доказана свобода контроллера буферных графов от лайфлока; в конце будут упомянуты расширения неструктурированных решений.

Контроллер буферного графа. Можно продемонстрировать, что контроллеры Раздела 5.2 лайфлок-свободны без каких-либо модификаций, если их передвижения в бесконечной последовательности удовлетворяют ряду справедливых предположений. Fl и F2 - сильные предположеня и F3 - слабое предположение.
Fl. Если порождение пакета *p* предпринимается непрерывно, то каждое бесконечное вычисление, в котором *fb (p)* свободен в бесконечно большом количестве конфигураций, содержит порождение *p*.

F2. Если в конфигурации γ пакет *p* должен быть послан от *u* до *w*, то каждое бесконечное вычисление, начинающееся в γ, в котором *nb (p, b)* является свободным в бесконечно большом количестве конфигураций, содержит пересылку *p*.
F3. Если пакет *p* находится в своем пункте назначения в конфигурации γ, то каждое бесконечное вычисление, начинающееся в γ, содержит выведение *p*.

Лемма 5.24 *Если справедливые предположения F2 и F3 удовлетворяются в каждом бесконечном вычислении* **bgc**, *то каждый буфер свободен бесконечно часто.*
Доказательство. Доказывать будем индукцией сверху вниз по классам буферов. Как в доказательстве Теоремы 5.7, пусть *R* - наибольший буферный класс. Напомним, что в конфигурациях, достижимых под управлением **bgc,** все пакеты располагаются в подходящих буферах.
Случай *r = R:* Буфер класса *R* не имеет исходящих ребер, и, следовательно, пакет из такого буфера достигает пункта назначения. Следовательно, по предположению F3, после каждой конфигурации, в которой такой буфер занят, будет конфигурация, в которой буфер свободен. Отсюда следует, что он пуст в бесконечно большом количестве конфигураций.
Случай *r < R:* Пусть γ - конфигурация, в которой буфер *b* класса *r < R* занят пакетом *p*. Если *p* достиг своего пункта назначения, то позже будет конфигурация, в которой *b* будет пустым из F3. Иначе, *p* должен быть продвинут в буфер *nb(p, b)* класса *r′ > r.* По индукции, в каждом бесконечном вычислении, начинающемся в γ, этот буфер пуст бесконечно часто. Из этого следует (по F2), что *p* будет передан и, следовательно, будет конфигурация γ, после которой *b* будет пуст.

Теорема 5.25 *Если справедливые предположения F1, F2 и F3 удовлетворяются, то в каждом бесконечном вычислении каждый пакет, предлагаемый сети, будет выведен из своего пункта назначения.*
Доказательство. По Лемме 5.24 все буферы пусты в бесконечно большом количестве конфигураций. Значит (по Fl) каждый паке, который продолжает предлагаться сети, будет сгенерирован. По F2 он будет продвигаться, пока не достигнет своего пункта назначения.
Легко предположить, что механизм недетерминированного выбора в распределенной системе гарантирует, что эти три предположения удовлетворяются. Альтернативно, предположения могут быть введены добавлением к контроллеру механизма, который гарантирует, что, когда буфер становится пустым, более старым пакетам позволяется входить с большими приоритетами.

Неструктурированные решения. Контроллеры Раздела 5.3 нужно модифицировать, чтобы они стали лайфлок-свободными. Это может быть показано моделированием бесконечного вычисления, в котором непрерывный поток пакетов заставляет контроллер запрещать пересылку некоторого пакета. Toueg [TouSOb] представляет такое вычисление (для **FC**) и представляет модификацию **FS** (подобную представленной здесь для контроллеров буферных графов), которая является лайфлок-свободной.

Упражнения к Главе 5

Раздел 5.1

Упражнение 5.1 *Покажите, что не существует беступикового контроллера, который использует только один буфер на вершину и позволяет каждой вершине посылать пакеты по крайней мере одной другой вершине.*

Раздел 5.2

Упражнение 5.2 *Покажите, что* **dest** *не является беступиковым, если маршрутизация пакетов осуществляется как на Рисунке 5.2.*

Упражнение 5.3 *(Схема сколько-будет-переходов) Дайте буферный граф и функции fb и nb для контроллера, который использует буфер $b_u[i]$ для хранения пакетов, которым нужно сделать еще i переходов по направлению к своим пунктам назначения. Каков буферный класс $b_u[i]$? Нужно ли хранить счетчик переходов в каждом пакете?*

Упражнение 5.4 *Закончите доказательство того, что граф BG_a (определенный в доказательстве Теоремы 5.13)- в самом деле буферный граф, т.е., для каждого пути $P \in$ P существует гарантированный путь с образом P. Покажите, что, как утверждалось, fb и nb в самом деле описывают путь в BG_a.*

Проект 5.5 *Докажите, что существует беступиковый контроллер, для коммутации пакетов в гиперкубе, который использует всего два буфера на вершину и позволяет маршрутизировать пакеты через минимальные пути. Можно ли получить набор используемых путей посредством алгоритма интервальной маршрутизации (Подраздел 4.4.2)? Можно ли использовать схему линейной интервальной разметки?*

Раздел 5.3

Упражнение 5.6 *Докажите, что* **BC** *и* **BS -** *беступиковые контроллеры.*

Упражнение 5.7 *Докажите, каждое передвижение, позволяемое* **BC,** *также позволяется* **FC.**

Часть 2 Фундаментальные алгоритмы

6 Волновые алгоритмы и алгоритмы обхода

При разработке распределенных алгоритмов для различных приложений часто в качестве подзадач возникает несколько общих проблем для сетей процессов. Эти элементарные задачи включают в себя широковещательную рассылку информации - broadcasting (например, посылка начального или заключительного сообщения), достижение глобальной синхронизации между процессами, запуск выполнения некоторого события в каждом процессе, или вычисление функции, входные данные которой распределены между процессами. Эти задачи всегда решаются путем пересылки сообщений согласно некоторой, зависящей от топологии схеме, которая гарантирует участие всех процессов. Эти задачи настолько фундаментальны, что можно дать решения более сложных задач, таких как выбор (Глава 7), обнаружение завершения (Глава 8), или взаимное ис-

ключение, в которых связь между процессами осуществляется только через эти схемы передачи сообщений.

Важность схем передачи сообщений, называемых далее волновыми алгоритмами, оправдывает их рассмотрение отдельно от конкретного прикладного алгоритма, в который они могут быть включены. В этой главе формально определяются волновые алгоритмы (Подраздел 6.1.1) и доказываются некоторые общие результаты о них (Подраздел 6.1.2). Замечание о том, что те же самые алгоритмы могут использоваться для всех основных задач, изложенных выше, т.е. широковещание, синхронизация и вычисление глобальных функций, будет формализовано (Подразделы 6.1.3-5). В Разделе 6.2 представлены некоторые широко используемые волновые алгоритмы. В Разделе 6.3 рассматриваются алгоритмы обхода; это волновые алгоритмы, в которых все события вычисления алгоритма *совершенно* упорядочены каузальным отношением. В Разделе 6.4 представлены несколько алгоритмов для распределенного поиска в глубину.

Несмотря на то, что волновые алгоритмы обычно используются как подзадачи в более сложных алгоритмах, их полезно рассматривать отдельно. Во-первых, введение новых понятий облегчает последующее рассмотрение более сложных алгоритмов, т.к. свойства их подзадач уже изучены. Во-вторых, определенные задачи в распределенных вычислениях могут быть решены с помощью универсальных конструкций, в качестве параметров которых могут использоваться конкретные волновые алгоритмы. Тот же метод может использоваться для получения алгоритмов для различных сетевых топологий или для различных предположений о начальном знании процессов. Эта глава основана на [Tel91b, Раздел 4.2], где понятие волновых алгоритмов изучается под названием *общие алгоритмы*.

6.1 Определение и использование волновых алгоритмов

В пределах этой главы считается, если не указано обратное, что сетевая топология фиксирована (не происходит топологических перемен), не ориентирована (каждый канал передает сообщения в обоих направлениях) и связна (существует путь между любыми двумя процессами). Множество всех процессов обозначено через P, а множество каналов - через E. Как и в предыдущих главах, предполагается, что система использует асинхронную передачу сообщений и не существует понятия глобального или реального времени. Алгоритмы из этой главы также могут быть использованы в случае синхронной передачи сообщений (возможно с некоторыми изменениями во избежание тупиков) или с часами глобального времени, если они доступны. Однако некоторые более общие теоремы в этих случаях неверны; см. Упражнение 6.1.

6.1.1 Определение волновых алгоритмов

Как отмечалось в Главе 2, распределенные алгоритмы обычно допускают большой набор возможных вычислений благодаря недетерминированности как в процессах, так и в подсистеме передачи. Вычисление - это набор событий, частично упорядоченных отношением каузального (причинно-следственного) предшествования $\underline{\pi}$; см. Определение 2.20. Количество событий в вычислении C

обозначается через |C|, а подмножество событий, происходящих в процессе p, обозначается через C_p. Считается, что существует особый тип внутренних событий, называемый *decide (принять решение)*; в алгоритмах этой главы такое событие представляется просто утверждением *decide*. Волновой алгоритм обменивается конечным числом сообщений и затем принимает решение, которое каузально зависит от некоторого события в каждом процессе.

Определение 6.1 *Волновой алгоритм - это распределенный алгоритм, который удовлетворяет следующим трем требованиям*:

a) **Завершение.** Каждое вычисление конечно:

$$\forall\, C : |C| < \infty$$

b) **Принятие решения.** Каждое вычисление содержит хотя бы одно событие decide:

$$\forall\, C : \exists\, e \in C : e \text{ - } \textit{событие decide}.$$

c) **Зависимость.** В каждом вычислении каждому событию decide каузально предшествует какое-либо событие в каждом процессе:

$$\forall\, C : \forall\, e \in C : (\, e \text{ - } \textit{событие decide} \Rightarrow\ \forall\, q \in P\ \exists\, f \in C_q : f \underline{\pi}\, e).$$

Вычисление волнового алгоритма называется *волной*. Кроме того, в вычислении алгоритма различаются *инициаторы*, также называемые *стартерами*, и *не-инициаторы*, называемые *последователями*. Процесс является инициатором, если он начинает выполнение своего локального алгоритма самопроизвольно, т.е. при выполнении некоторого условия, внутреннего по отношению к процессу. Не-инициатор включается в алгоритм только когда прибывает сообщение и вызывает выполнение локального алгоритма процесса. Начальное событие инициатора - внутреннее событие или событие посылки сообщения, начальное событие не-инициатора - событие получения сообщения.

Существует множество волновых алгоритмов, так как они могут различаться во многих отношениях. Для обоснования большого количества алгоритмов в этой главе и в качестве помощи в выборе одного из них для конкретной цели здесь приведен список аспектов, которые отличают волновые алгоритмы друг от друга; см. также Таблицу 6.19.

1) *Централизация*. Алгоритм называется *централизованным*, если в каждом вычислении должен быть ровно один инициатор, и *децентрализованным*, если алгоритм может быть запущен произвольным подмножеством процессов. Централизованные алгоритмы также называют алгоритмами *одного источника*, а децентрализованные - алгоритмами *многих источников.* Как видно из Таблицы 6.20, централизация существенно влияет на сложность волновых алгоритмов.
2) *Топология.* Алгоритм может быть разработан для конкретной топологии, такой как кольцо, дерево, клика и т.д.; см. Подраздел 2.4.1 и Раздел B.2.
3) *Начальное знание*. Алгоритм может предполагать доступность различных типов начального знания в процессах; см. Подраздел 2.4.4. Примеры требуемых заранее знаний:

   (a) *Идентификация процессов*. Каждому процессу изначально известно свое собственное уникальное имя.

   (b) *Идентификация соседей*. Каждому процессу изначально известны имена его соседей.

(c) *Чувство направления* (sense of direction). См. Раздел B.3.

4) *Число решений.* Во всех волновых алгоритмах этой главы в каждом процессе происходит не более одного решения. Количество процессов, которые выполняют событие *decide*, может быть различным; в некоторых алгоритмах решение принимает только один процесс, в других - все процессы. В древовидном алгоритме (Подраздел 6.2.2) решают ровно два процесса.
5) *Сложность*. Меры сложности, рассматриваемые в этой главе, это количество передаваемых сообщений (message complexity), количество передаваемых бит (bit complexity) и время, необходимое для одного вычисления (определено в Разделе 6.4). См. также Подраздел 2.4.5.

Каждый волновой алгоритм в этой главе будет дан вместе с используемыми переменными и, в случае необходимости, с информацией, передаваемой в сообщениях. Большинство этих алгоритмов посылают «пустые сообщения», без какой-либо реальной информации: сообщения передают причинную связь, а не информацию. Алгоритмы 6.9, 6.11, 6.12 и 6.18 используют сообщения для передачи нетривиальной информации. Алгоритмы 6.15 и 6.16/6.17 используют различные типы сообщений; при этом требуется, чтобы каждое сообщение содержало 1-2 бита для указания типа сообщения.

Обычно при применении волновых алгоритмов в сообщение могут быть включены дополнительные переменные и другая информация. Многие приложения используют одновременное или последовательное распространение нескольких волн; в этом случае в сообщение должна быть включена информация о волне, которой оно принадлежит. Кроме того, процесс может хранить дополнительные переменные для управления волной, или волнами, в которых он в настоящее время активен.

Важный подкласс волновых алгоритмов составляют централизованные волновые алгоритмы, обладающие двумя дополнительными качествами: инициатор является единственным процессом, который принимает решение; и все события совершенно упорядочены каузальными отношениями. Такие волновые алгоритмы называются *алгоритмами обхода* и рассматриваются в Разделе 6.3.

6.1.2 Элементарные результаты о волновых алгоритмах

В этом подразделе доказываются некоторые леммы, которые помогают лучше понять структуру волновых вычислений, и приведены две тривиальные нижние границы сложности сообщений волновых алгоритмов.

Структурные свойства волн. Во-первых, каждому событию в вычислении предшествует событие в инициаторе.

Лемма 6.2 *Для любого события* $e \in C$ *существует инициатор* p *и событие* f *в* C_p *такое, что* $f \underline{\pi} e$.

Доказательство. Выберем в качестве f минимальный элемент в предыстории e, т.е. такой, что $f \underline{\pi} e$ и не существует $f' \pi f$. Такое f существует, поскольку предыстория каждого события конечна. Остается показать, что процесс p, в котором находится f, является инициатором. Для начала, заметим, что f - это первое событие p, иначе более раннее событие p предшествовало бы f. Первое со-

бытие не-инициатора - это событие получения сообщения, которому предшествовало бы соответствующее событие посылки сообщения, что противоречит минимальности f. Следовательно, p является инициатором.

Волна с одним инициатором определяет остовное дерево сети, где для каждого не-инициатора выбирается канал, через который будет получено первое сообщение.

Лемма 6.3 *Пусть* C - *волна с одним инициатором* p*; и пусть для каждого не-инициатора* q father_q - *это сосед* q, *от которого* q *получает сообщение в своем первом событии. Тогда граф* $T = (P, E_T)$, *где* $E_T = \{qr: q \neq p \ \& \ r = \text{father}_q\}$ - *остовное дерево, направленное к* p.

Доказательство. Т.к. количество вершин T превышает количество ребер на 1, достаточно показать, что T не содержит циклов. Это выполняется, т.к. если e_q - первое событие в q, из того, что $qr \in E_T$ следует, что $e_r \underline{\pi} e_q$, а $\underline{\pi}$ - отношение частичного порядка.

В качестве события f в пункте (3) Определения 6.1 может быть выбрано событие посылки сообщения всеми процессами q, кроме того, где происходит событие *decide*.

Лемма 6.4 *Пусть* C - *волна, а* $d_p \in C$ - *событие decide в процессе* p. *Тогда*

$$\forall\, q \neq p: \exists\, f \in C_q: (\, f \underline{\pi} d_p \ \& \ f \text{ - } \textit{событие} \text{ send})$$

Доказательство. Т.к. C - это волна, существует событие $f \in C_q$, которое каузально предшествует d_p; выберем в качестве f последнее событие C_q, которое предшествует d_p. Чтобы показать, что f - событие *send*, отметим, что из определения каузальности (Определение 2.20) следует, что существует последовательность (каузальная цепочка)

$$f = e_0, e_1, ..., e_k = d_p,$$

такая, что для любого $i < k$, e_i и e_{i+1} - либо последовательные события в одном процессе, либо пара соответствующих событий send-receive. Т.к. f - последнее событие в q, которое предшествует d_p, e_1 происходит в процессе, отличном от q, следовательно f - событие *send*.

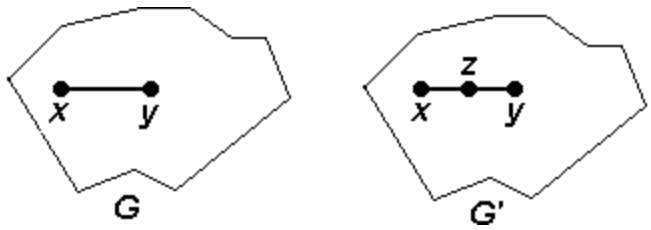


Рис.6.1 Включение процесса в неиспользуемый канал.

Нижние границы сложности волн. Из леммы 6.4 непосредственно следует, что нижняя граница количества сообщений, которые передаются в волне,

равна N-1. Если событие *decide* происходит в единственном инициаторе волны (что выполняется всегда в случае алгоритмов обхода), граница равна N сообщениям, а волновые алгоритмы для сетей произвольной топологии используют не менее |E| сообщений.

Теорема 6.5 *Пусть* C - *волна с одним инициатором* p, *причем событие decide* d_p *происходит в* p. *Тогда в* C *передается не менее* N *сообщений.*

Доказательство. По лемме 6.2 каждому событию в C предшествует событие в p, и, используя каузальную последовательность, как в доказательстве леммы 6.4, нетрудно показать, что в p происходит хотя бы одно событие *send*. По лемме 6.4 событие *send* также происходит во всех других процессах, откуда количество посылаемых сообщений составляет не меньше N.

Теорема 6.6 *Пусть* A - *волновой алгоритм для сетей произвольной топологии без начального знания об идентификации соседей. Тогда* A *передает не менее* |E| *сообщений в каждом вычислении.*

Доказательство. Допустим, A содержит вычисление C, в котором передается менее |E| сообщений; тогда существует канал xy, по которому в C не передаются сообщения; см. Рис.6.1. Рассмотрим сеть G′, полученную путем включения одного узла z в канал между x и y. Т.к. узлы не имеют знания о соседях, начальное состояние x и y в G′ совпадает с их начальным состоянием в G. Это верно и для всех остальных узлов G. Следовательно, все события C могут быть применены в том же порядке, начиная с исходной конфигурации G′, но теперь событию *decide* не предшествует событие в z.

В Главе 7 будет доказана улучшенная нижняя граница количества сообщений децентрализованных волновых алгоритмов для колец и сетей произвольной топологии без знания о соседях; см. Заключение 7.14 и 7.16.

6.1.3 Распространение информации с обратной связью

В этом подразделе будет продемонстрировано применение волновых алгоритмов для случая, когда некоторая информация должна быть передана всем процессам и определенные процессы должны быть оповещены о завершении передачи. Эта задача распространения информации с обратной связью (PIF - propagation of information with feedback) формулируется следующим образом [Seg83]. Формируется подмножество процессов из тех, кому известно сообщение M (одно и то же для всех процессов), которое должно быть распространено, т.е. все процессы должны принять M. Определенные процессы должны быть *оповещены* о завершении передачи; т.е. должно быть выполнено специальное событие *notify*, причем оно может быть выполнено только когда все процессы уже получили сообщение M. Алгоритм должен использовать конечное количество сообщений.

Оповещение в PIF-алгоритме можно рассматривать как событие *decide*.

Теорема 6.7 *Каждый* PIF-*алгоритм является волновым алгоритмом.*

Доказательство. Пусть P - PIF-алгоритм. Из формулировки задачи каждое вычисление P должно быть конечным и в каждом вычислении должно происходить событие оповещения (*decide*). Если в некотором вычислении P проис-

ходит оповещение d_p, которому не предшествует никакое событие в процессе q, тогда из Теоремы 2.21 и Аксиомы 2.23 следует, что существует выполнение P, в котором оповещение происходит *до того*, как q принимает какое-либо сообщение, что противоречит требованиям.

Мы должны иметь в виду, что теорема 2.21 выполняется только для систем с асинхронной передачей сообщений; см. Упражнение 6.1.

Теорема 6.8 *Любой волновой алгоритм может использоваться как* PIF-*алгоритм.*

Доказательство. Пусть A - волновой алгоритм. Чтобы использовать A как PIF-алгоритм, возьмем в качестве процессов, изначально знающих M, стартеры (инициаторы) A. Информация M добавляется к каждому сообщению A. Это возможно, поскольку по построению стартеры A знают M изначально, а последователи не посылают сообщений, пока не получат хотя бы одно сообщение, т.е. пока не получат M. Событию *decide* в волне предшествуют события в каждом процессе; следовательно, когда оно происходит, каждый процесс знает M, и событие *decide* можно считать требуемым событием *notify* в PIF-алгоритме.

Построенный PIF-алгоритм имеет такую же сложность сообщений, как и алгоритм A и обладает всеми другими качествами A, описанными в Подразделе 6.1.1, кроме битовой сложности. Битовая сложность может быть уменьшена путем добавления M только к первому сообщению, пересылаемому через каждый канал. Если w - количество бит в M, битовая сложность полученного алгоритма превышает битовую сложность A на w*|E|.

6.1.4 Синхронизация

В этом разделе будет рассмотрено применение волновых алгоритмов для случаев, когда должна быть достигнута глобальная синхронизация процессов. Задача синхронизации (SYN) формулируется следующим образом [Fin79]. В каждом процессе q должно быть выполнено событие a_q, и в некоторых процессах должны быть выполнены события b_p, причем все события a_q должны быть выполнены по времени раньше, чем будет выполнено какое-либо событие b_p. Алгоритм должен использовать конечное количество сообщений.

В SYN-алгоритмах события b_p будут рассматриваться как события *decide*.

Теорема 6.9 *Каждый* SYN-*алгоритм является волновым алгоритмом.*

Доказательство. Пусть S - SYN-алгоритм. Из формулировки задачи каждое вычисление S должно быть конечным и в каждом вычислении должно происходить событие b_p (*decide*). Если в некотором вычислении S происходит событие b_p, которому каузально не предшествует a_q, тогда (по Теореме 2.21 и Аксиоме 2.23) существует выполнение S, в котором b_p происходит ранее a_q.

Теорема 6.10 *Любой волновой алгоритм может использоваться как* SYN-*алгоритм.*

Доказательство. Пусть A - волновой алгоритм. Чтобы преобразовать A в SYN-алгоритм, потребуем, чтобы каждый процесс q выполнял a_q до того, как q пошлет сообщение в A или примет решение в A. Событие b_p происходит после

события *decide* в p. Из леммы 6.4, каждому событию *decide* каузально предшествует a_q для любого q.

Построенный SYN-алгоритм имеет такую же сложность сообщений, как и A, и обладает всеми другими свойствами A, описанными в Подразделе 6.1.1.

6.1.5 Вычисление функций инфимума

В этой главе будет продемонстрировано применение волновых алгоритмов для вычисления функций, значения которых зависят от входов каждого процесса. В качестве представителей таких функций будут рассмотрены алгоритмы, вычисляющие *инфимум* по всем входам, которые должны быть извлечены из частично упорядоченного множества.

Если $(X, \leq)$ - частичный порядок, то c называют *инфимумом* a и b, если $c \leq a$, $c \leq b$, и $\forall\, d : (d \leq a \,\&\, d \leq b \Rightarrow d \leq c)$. Допустим, что X таково, что инфимум всегда существует; в этом случае инфимум является единственным и обозначается как $a \wedge b$. Т.к. $\wedge$ - бинарный оператор, коммутативный ($a \wedge b = b \wedge a$) и ассоциативный (т.е. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$), операция может быть обобщена на конечные множества:

$$\inf \{ j_1, ..., j_k\} = j_1 \wedge ... \wedge j_k .$$

Задача вычисления инфимума формулируется следующим образом. Каждый процесс q содержит вход j_q, являющийся элементом частично упорядоченного множества X. Потребуем, чтобы определенные процессы вычисляли значение $\inf \{j_q : q \in P\}$ и чтобы эти процессы знали, когда вычисление завершается. Они записывают результат вычисления в переменную *out* и после этого не могут изменять ее значение.

Событие *write*, которое заносит значение в *out*, рассматривается в INF-алгоритме как событие *decide*.

Теорема 6.11 *Каждый* INF-*алгоритм является волновым алгоритмом.*

Доказательство. Пусть I - INF-алгоритм. Предположим, что существует вычисление C алгоритма I с начальной конфигурацией Υ, в котором p записывает значение J в out_p и этому событию *write* не предшествует никакое событие в q. Рассмотрим начальную конфигурацию Υ', которая аналогична Υ за исключением того, что q имеет другой вход j_q', выбранный так, что $j_q' \not< J$. Так как никакое применение входа q не предшествует каузально событию *write* процесса p в C, все события C, предшествующие событию *write*, применимы в том же порядке, начиная с Υ'. В полученном вычислении p записывает ошибочный результат J, так же как в C.

Теорема 6.12 *Любой волновой алгоритм может быть использован для вычисления инфимума.*

Доказательство. Допустим, что дан волновой алгоритм A. Назначим каждому процессу q дополнительную переменную v_q, которой придадим начальное значение j_q. Во время волны эти переменные переприсваиваются следующим образом. Всякий раз, когда процесс q посылает сообщение, текущее значение v_q включается в сообщение. Всякий раз, когда процесс q получает со-

общение со значением v, v_q присваивается значение $v_q \wedge v$. Когда в процессе p происходит событие *decide*, текущее значение v_p заносится в out_p.

Теперь нужно показать, что в результат заносится правильное значение. Обозначим правильный ответ через J, т.е. $J = \inf \{ j_q : q \in P \}$. Для события a в процессе q обозначим через $v^{(a)}$ значение v_q сразу после выполнения a. Т.к. начальное значение v_q равно j_q, и в течение волны оно только уменьшается, неравенство $v^{(a)} \leq j_q$ сохраняется для каждого события a в q. Из присваивания v следует, что для событий a и b, $a \underline{\pi} b \Rightarrow v^{(a)} \geq v^{(b)}$. Кроме того, т.к. v всегда вычисляется как инфимум двух уже существующих величин, неравенство $J \leq v$ выполняется для всех величин в течение волны. Таким образом, если d - событие *decide* в p, значение $v^{(d)}$ удовлетворяет неравенству $J \leq v^{(d)}$ и, т.к. событию d предшествует хотя бы одно событие в каждом процессе q, $v^{(d)} \leq j_q$ для всех q. Отсюда следует, что $J = v^{(d)}$.

Построенный INF-алгоритм обладает всеми свойствами A, кроме битовой сложности, поскольку к каждому сообщению A добавляется элемент X. Понятие функции инфимума может показаться довольно абстрактным, но фактически многие функции могут быть выражены через функцию инфимума, как показано в [Tel91b, Теорема 4.1.1.2].

Аксиома 6.13 (Теорема об инфимуме) *Если* * - *бинарный оператор на множестве* X*, причем он:*

(1) коммутативен, т.е. a * b = b * a,

(2) ассоциативен, т.е. (a * b) * c = a * (b * c), *и*

(3) идемпотентен, т.е. a * a = a

то существует отношение частичного порядка ≤ *на* X *такое, что* * - *функция инфимума.*

Среди операторов, удовлетворяющих этим трем критериям - логическая конъюнкция и дизъюнкция, минимум и максимум целых чисел, наибольший общий делитель и наименьшее общее кратное целых чисел, пересечение и объединение множеств.

Заключение 6.14 &, ∨, min, max, *НОД, НОК,* ∩ *и* ∪ *величин, локальных по отношению к процессам, могут быть вычислены за одну волну.*

Вычисление операторов, которые являются коммутативными и ассоциативными, но не идемпотентны, рассматривается в Подразделе 6.5.2.

6.2 Волновые алгоритмы

В следующих трех разделах будет представлено несколько волновых алгоритмов и алгоритмов обхода. Все тексты алгоритмов даны для процесса p.

6.2.1 Кольцевой алгоритм

В этом разделе будет приведен алгоритм для кольцевой сети. Этот же алгоритм может быть применен для Гамильтоновых сетей, где один фиксированный Гамильтонов цикл проходит через все процессы. Предположим, что для каждого процесса p задан сосед $Next_p$ такой, что все каналы, выбранные таким образом, составляют Гамильтонов цикл.

Алгоритм является централизованным; инициатор посылает сообщение <**tok**> (называемое маркером) вдоль цикла, каждый процесс передает его дальше и когда оно возвращается к инициатору, инициатор принимает решение; см. Алгоритм 6.2.

Для инициатора:
begin send <**tok**> to $Next_p$; receive <**tok**>; *decide* **end**
Для не-инициатора:
begin receive <**tok**>; send <**tok**> to $Next_p$ **end**

Алгоритм 6.2 Кольцевой алгоритм.

Теорема 6.15 *Кольцевой алгоритм (Алгоритм 6.2) является волновым алгоритмом.*

Доказательство. Обозначим инициатор через p_0. Так как каждый процесс посылает не более одного сообщения, алгоритм передает в целом не больше N сообщений.

За ограниченное количество шагов алгоритм достигает заключительной конфигурации. В этой конфигурации p_0 уже переслал маркер, т.е. выполнил оператор *send* в своей программе. Кроме того, ни одно сообщение <**tok**> не передается ни по одному каналу, иначе оно может быть получено и конфигурация не будет заключительной. Также, ни один процесс, кроме p_0, не «задерживает» маркер (т.е. получил, но не передал дальше <**tok**>), иначе процесс может послать <**tok**> и конфигурация не будет конечной. Следовательно, (1) p_0 отправил маркер, (2) для любого p, пославшего маркер, $Next_p$ получил маркер, и (3) каждый $p \neq p_0$, получивший маркер, отправил маркер. Из этого и свойства Next следует, что каждый процесс отправил и получил маркер. Т.к. p_0 получил маркер и конфигурация конечна, p_0 выполнил оператор *decide*.

Получение и отправка <**tok**> каждым процессом $p \neq p_0$ предшествует получению маркера процессом p_0, следовательно, условие зависимости выполнено.

6.2.2 Древовидный алгоритм

В этом разделе представлен алгоритм для древовидной сети. Этот же алгоритм может быть использован для сетей произвольной топологии, если доступно остовное дерево сети. Предполагается, что алгоритм инициируют все листья дерева. Каждый процесс в алгоритме посылает ровно одно сообщение. Ес-

ли процесс получил сообщение по всем инцидентным каналам, кроме одного (это условие изначально выполняется для листьев), процесс отправляет сообщение по оставшемуся каналу. Если процесс получил сообщения через все инцидентные каналы, он принимает решение; см. Алгоритм 6.3.

---

var $rec_p[q]$ for each $q \in Neigh_p$: boolean **init** *false* ;
(* $rec_p[q]$ = *true*, если p получил сообщение от q *)

begin **while** # {q : $rec_p[q]$ is *false*} > 1 **do**
begin receive <**tok**> from q ; $rec_p[q]$:= *true* **end** ;
(* Теперь остался один q_0, для которого $rec_p[q_0]$ = *false* *)
send <**tok**> to q_0 with $rec_p[q_0]$ is *false* ;
x : receive <**tok**> from q_0 ; $rec_p[q_0]$:= *true* ;
decide
(* Сообщить другим процессам о решении:
forall $q \in Neigh_p$, $q \neq q_0$ **do** send <**tok**> to q *)
end

Алгоритм 6.3 Древовидный алгоритм.

Чтобы показать, что этот алгоритм является волновым, введем некоторые обозначения. Пусть f_{pq} - событие, где p посылает сообщение q, а g_{pq} - событие, где q получает сообщение от p. Через T_{pq} обозначим подмножество процессов, которые достижимы из p без прохождения по дуге pq (процессы на стороне p дуги pq); см. Рис.6.4.

Из связности сети следует, что (см. Рис.6.4)

$$T_{pq} = \bigcup_{r \in Neigh_p \setminus \{q\}} T_{rp} \cup \{p\} \quad \text{и} \quad P = \{p\} \cup \bigcup_{r \in Neigh_p} T_{rp}$$

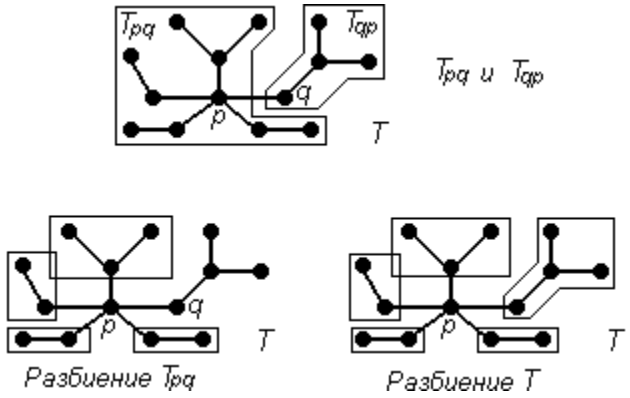


Рис. 6.4 Поддеревья T_{pq}.

Оператор **forall** в комментариях в Алгоритме 6.3 будет обсуждаться в конце этого подраздела; в следующей теореме речь идет об алгоритме без этого оператора.

Теорема 6.16 *Древовидный алгоритм (Алгоритм 6.3) является волновым алгоритмом.*

Доказательство. Т.к. каждый процесс посылает не более одного сообщения, в целом алгоритм использует не более N сообщений. Отсюда следует, что алгоритм достигает заключительной конфигурации Υ за конечное число шагов; мы покажем, что в Υ хотя бы один процесс выполняет событие *decide*.

Пусть F - количество битов *rec* со значением *false* в Υ, а K - количество процессов, которые уже послали сообщения в Υ. Т.к. в Υ не передается ни одно сообщение (иначе Υ не была бы заключительной), F = (2N-2) - K; общее число битов *rec* равно 2N-2, а K из них равны *true*.

Предположим, что ни один процесс в Υ не принял решения. N-K процессов, которые еще не послали сообщение в Υ, содержат хотя бы по два бита *rec*, равных *false*; иначе они бы могли послать сообщение, что противоречит тому, что Υ - заключительная конфигурация. K процессов, которые послали сообщение в Υ, содержат хотя бы один бит *rec*, равный *false*; иначе они могли бы принять решение, что противоречит тому, что Υ - заключительная конфигурация. Итак, $F \geq 2(N-K) + K$, а из $(2N-2) - K \geq 2(N-K) + K$ следует, что $-2 \geq 0$; мы пришли к противоречию, следовательно, хотя бы один процесс в Υ принимает решение. См. Упражнение 6.5.

Наконец, нужно показать, что решению предшествует событие в каждом процессе. Пусть f_{pq} - событие, где p посылает сообщение q, а g_{pq} - событие, где q получает сообщение от p. Применяя индукцию по событиям получения сообщений, можно доказать, что $\forall\, s \in T_{pq}\ \exists\, e \in C_s$: $e \underline{\pi} g_{pq}$.

Предположим, что это выполняется для всех событий получения сообщений, предшествующих g_{pq}. Из того, что событию g_{pq} предшествует f_{pq} (в процессе p), и из алгоритма p следует, что для всех $r \in Neigh_p$ при $r \neq q$, g_{rp} предшествует f_{pq}. Из гипотезы индукции следует, что для всех таких r и для всех $s \in T_{rp}$ существует событие $e \in C_s$, где $e \underline{\pi} g_{rp}$, следовательно, $e \underline{\pi} g_{pq}$.

Решению d_p в p предшествуют g_{rp} для всех $r \in Neigh_p$, откуда следует, что $\forall\, s \in P\ \exists\, e \in C_s : e \underline{\pi} d_p$.

Читатель может смоделировать вычисление алгоритма на небольшом дереве (например, см. дерево на Рис.6.4) и самостоятельно убедиться в справедливости следующих замечаний. В Алгоритме 6.3 существует два процесса, которые получают сообщения через все свои каналы и принимают решение; все остальные тем временем ожидают сообщения с счетчиком команд, установленным на x, в заключительной конфигурации. Если к программе добавить оператор **forall** (в скобках комментария в Алгоритме 6.3), то все процессы принимают решение и в конечной конфигурации каждый процесс находится в конечном состоянии. Модифицированная программа использует 2N-2 сообщений.

6.2.3 Эхо-алгоритм

Эхо-алгоритм - это централизованный волновой алгоритм для сетей произвольной топологии. Впервые он был представлен Чангом [Chang; Cha82] и

поэтому иногда называется эхо-алгоритмом Чанга. Более эффективная версия, которая и представлена здесь, была предложена Сегаллом [Segall; Seg83].

Алгоритм распространяет сообщения <**tok**> по всем процессам, таким образом определяя остовное дерево, как определено в Лемме 6.3. Маркеры «отражаются» обратно через ребра этого дерева аналогично потоку сообщений в древовидном алгоритме. Алгоритм обозначен как Алгоритм 6.5.

Инициатор посылает сообщения всем своим соседям. После получения первого сообщения не-инициатор пересылает сообщения всем своим соседям, кроме того, от которого было получено сообщение. Когда не-инициатор получает сообщения от всех своих соседей, эхо пересылается родителю (father). Когда инициатор получает сообщения от всех своих соседей, он принимает решение.

---

var rec_p : integer **init** 0 ; (* Счетчик полученных сообщений *)
$father_p$: P **init** *udef* ;

Для инициатора:

begin forall $q \in Neigh_p$ **do** send <**tok**> to q ;
while $rec_p < \#\, Neigh_p$ **do**
begin receive <**tok**> ; $rec_p := rec_p + 1$ **end** ;
decide
end ;

Для не-инициатора:

begin receive <**tok**> from neighbor q ; $father_p := q$; $rec_p := rec_p + 1$;
forall $q \in Neigh_p,\ q \neq father_p$ **do** send <**tok**> to q ;
while $rec_p < \#\, Neigh_p$ **do**
begin receive <**tok**> ; $rec_p := rec_p + 1$ **end** ;
send <**tok**> to $father_p$
end

Алгоритм 6.5 Эхо-алгоритм.

Теорема 6.17 *Эхо-алгоритм (Алгоритм 6.5) является волновым алгоритмом.*

Доказательство. Т.к. каждый процесс посылает не более одного сообщения по каждому инцидентному каналу, количество сообщений, пересылаемых за каждое вычисление, конечно. Пусть Υ - конечная конфигурация, достигаемая в вычислении C с инициатором p_0.

Для этой конфигурации определим (подобно определению в лемме 6.3) граф $T = (P, E_T)$, где $pq \in E_T \Leftrightarrow father_p = q$. Чтобы показать, что этот граф является деревом, нужно показать, что количество ребер на единицу меньше, чем количество вершин (Лемма 6.3 утверждает, что T - дерево, но предполагается, что алгоритм является волновым, что нам еще нужно доказать). Отметим, что каждый процесс, участвующий в C, посылает сообщения всем своим соседям, кроме соседа, от которого он получил первое сообщение (если процесс - не-инициатор). Отсюда следует, что все его соседи получают хотя бы одно сообщение в C и также участвуют в C. Из этого следует, что $father_p \neq$ *udef* для всех

$p \neq p_0$. Что T не содержит циклов, можно показать, как в доказательстве Леммы 6.3.

В корне дерева находится p_0; обозначим через T_p множество вершин в поддереве p. Ребра сети, не принадлежащие T, называются *листовыми* ребрами (frond edges). В Υ каждый процесс p, по крайней мере, послал сообщения всем своим соседям, кроме родителя $father_p$, следовательно, каждое листовое ребро передавало в C сообщения в обоих направлениях. Пусть f_p - событие, в котором p посылает сообщение своему родителю (если в C это происходит), а g_p - событие, в котором родитель p получает сообщение от p (если это происходит). С помощью индукции по вершинам дерева можно показать, что

(1) C содержит событие f_p для любого $p \neq p_0$;

(2) для всех $s \in T_p$ существует событие $e \in C_s$ такое, что $e \underline{\pi} g_p$.

Мы рассмотрим следующие два случая.

p - лист. p получил в C сообщение от своего родителя и от всех других соседей (т.к. все остальные каналы - листовые). Таким образом, посылка <**tok**> родителю p была возможна, и, т.к. Υ - конечная конфигурация, это произошло. T_p содержит только p, и, очевидно, $f_p \underline{\pi} g_p$.

p - не лист. p получил в C сообщение от своего родителя и через все листовые ребра. По индукции, C содержит $f_{p'}$ для каждой дочерней вершины p' вершины p, и, т.к. Υ - конечная конфигурация, C также содержит $g_{p'}$. Следовательно, посылка <**tok**> родителю p была возможна, и, т.к. Υ - конечная конфигурация, это произошло. T_p состоит из объединения $T_{p'}$ по всем дочерним вершинам p и из самого p. С помощью индукции можно показать, что в каждом процессе этого множества существует событие, предшествующее g_p.

Отсюда следует, также, что p_0 получил сообщение от каждого соседа и выполнил событие *decide*, которому предшествуют события в каждом процессе.

Остовное дерево, которое строится в вычислении Алгоритма 6.5, иногда используют в последовательно выполняемых алгоритмах. (Например, алгоритм Мерлина-Сегалла (Merlin-Segall) для вычисления таблиц кратчайших маршрутов предполагает, что изначально дано остовное дерево с корнем в v_0; см. Подраздел 4.2.3. Начальное остовное дерево может быть вычислено с использованием эхо-алгоритма). В последней конфигурации алгоритма каждый процесс (кроме p_0) запомнил, какой сосед в дереве является его родителем, но не запомнил дочерних вершин. В алгоритме одинаковые сообщения принимаются от родителя, через листовые ребра, и от дочерних вершин. Если требуется знание дочерних вершин в дереве, алгоритм может быть слегка изменен, так чтобы отправлять родителю сообщения, отличные от остальных (в последней операции отправления сообщения для не-инициаторов). Дочерними вершинами процесса тогда являются те соседи, от которых были получены эти сообщения.

6.2.4 Алгоритм опроса

В сетях с топологией клика между каждой парой процессов существует канал. Процесс может определить, получил ли он сообщение от каждого соседа. В *алгоритме опроса*, обозначенном как Алгоритм 6.6, инициатор запрашивает у

каждого соседа ответ на сообщение и принимает решение после получения всех ответных сообщений.

Теорема 6.18 *Алгоритм опроса (Алгоритм 6.6) является волновым алгоритмом.*

Доказательство. Алгоритм пересылает по два сообщения через каждый канал, смежный с инициатором. Каждый сосед инициатора отвечает только один раз на первоначальный опрос, следовательно, инициатор получает N-1 ответ. Этого достаточно, чтобы принять решение, следовательно, инициатор принимает решение и ему предшествует событие в каждом процессе.

Опрос может быть использован и в сети с топологией звезда, в которой инициатор находится в центре.

---

var rec_p : integer **init** 0 ; (* только для инициатора *)

Для инициатора:

```
begin forall q ∈ Neigh_p do send <tok> to q ;
        while rec_p < # Neigh_p do
              begin receive <tok> ; rec_p := rec_p + 1 end ;
        decide
end ;
```

Для не-инициатора:

```
begin receive <tok> from q ; send <tok> to q end
```

Алгоритм 6.6 Алгоритм опроса.

6.2.5 Фазовый алгоритм

В этом разделе будет представлен фазовый алгоритм, который является децентрализованным алгоритмом для сетей с произвольной топологией. Алгоритм дан в [Tel91b, Раздел 4.2.3]. Алгоритм может использоваться как волновой для ориентированных сетей.

Алгоритм требует, чтобы процессам был известен диаметр сети, обозначенный в тексте алгоритма как D. Алгоритм остается корректным (хотя и менее эффективным), если процессы вместо D используют константу $D' > D$. Таким образом, для применения алгоритма необязательно точно знать диаметр сети; достаточно, если известна верхняя граница диаметра (например, N-1). Все процессы должны использовать *одну и ту же* константу D'. Пелег [Peleg; Pel90] дополнил алгоритм таким образом, чтобы диаметр вычислялся во время выполнения, но это расширение требует уникальной идентификации.

Общий случай. Алгоритм может использоваться в *ориентированных* сетях произвольной топологии, где каналы могут передавать сообщения только в одном направлении. В этом случае, соседи p являются *соседями по входу* (процессы, которые могут посылать сообщения p) и *соседями по выходу* (процессы,

которым p может посылать сообщения). Соседи по входу p содержатся в множестве In_p, а соседи по выходу - в множестве Out_p.

В фазовом алгоритме каждый процесс посылает ровно D сообщений каждому соседу по выходу. Только после того, как i сообщений было получено от каждого соседа по входу, (i+1)-ое сообщение посылается каждому соседу по выходу; см. алгоритм 6.7.

```
cons D          : integer      = диаметр сети ;
 var rec_p[q]   : 0..D         init 0, для каждого q ∈ In_p ;
                  (* Количество сообщений, полученных от q *)
     Sent_p     : 0..D         init 0 ;
                  (* Количество сообщений, посланных каждому соседу по выходу *)

begin if p - инициатор then
        begin forall r ∈ Out_p do send <tok> to r ;
              Sent_p := Sent_p + 1
        end ;
      while min_q Rec_p[q] < D do
            begin receive <tok> (от соседа q_0) ;
                  Rec_p[q_0] := Rec_p[q_0] + 1 ;
                  if min_q Rec_p[q] ≥ Sent_p and Sent_p < D then
                     begin forall r ∈ Out_p do send <tok> to r ;
                           Sent_p := Sent_p + 1
                     end
            end ;
      decide
end
```

Алгоритм 6.7 Фазовый алгоритм.

Действительно, из текста алгоритма очевидно, что через каждый канал проходит не более D сообщений (ниже показано, что через каждый канал проходит не менее D сообщений). Если существует ребро pq, то $f_{pq}^{(i)}$ - i-е событие, в котором p передает сообщение q, а $g_{pq}^{(i)}$ - i-е событие, в котором q получает сообщение от p. Если канал между p и q удовлетворяет дисциплине FIFO, эти события соответствуют друг другу и неравенство $f_{pq}^{(i)} \underline{\pi} g_{pq}^{(i)}$ выполняется. Каузальные отношения между $f_{pq}^{(i)}$ и $g_{pq}^{(i)}$ сохраняются и в случае, если канал не является FIFO, что доказывается в следующей лемме.

Лемма 6.19 *Неравенство* $f_{pq}^{(i)} \underline{\pi} g_{pq}^{(i)}$ *выполняется, даже если канал не является каналом* FIFO.

Доказательство. Определим m_h следующим образом: $f_{pq}^{(mh)}$ - событие отправления сообщения, соответствующее $g_{pq}^{(h)}$, т.е. в своем h-м событии получения q получает m_h-е сообщение p. Из определения каузальности $f_{pq}^{(mh)} \underline{\pi} g_{pq}^{(h)}$.

Т.к. каждое сообщение в C получают только один раз, все m_h различны, откуда следует, что хотя бы одно из чисел m_1, ..., m_i больше или равно i. Выберем $j \le i$ так, чтобы $m_j \ge i$. Тогда $f_{pq}^{(i)} \underline{\pi} f_{pq}^{(mj)} \underline{\pi} g_{pq}^{(j)} \underline{\pi} g_{pq}^{(i)}$.

Теорема 6.20 *Фазовый алгоритм (Алгоритм 6.7) является волновым алгоритмом.*

Доказательство. Т.к. каждый процесс посылает не более D сообщений по каждому каналу, алгоритм завершается за конечное число шагов. Пусть Υ - заключительная конфигурация вычисления C алгоритма, и предположим, что в C существует, по крайней мере, один инициатор (их может быть больше).

Чтобы продемонстрировать, что в Υ каждый процесс принял решение, покажем сначала, что каждый процесс хотя бы один раз послал сообщения. Т.к. в Υ по каналам не передается ни одно сообщение, для каждого канала qp $Rec_p[q] = Sent_{pq}$. Также, т.к. каждый процесс посылает сообщения, как только получит сообщение сам, $Rec_p[q] > 0 \Rightarrow Sent_p > 0$. Из предположения, что существует хотя бы один инициатор p_0, для которого $Sent_{p0} > 0$, следует, что $Sent_p > 0$ для каждого p.

Впоследствии будет показано, что каждый процесс принял решение. Пусть p - процесс с минимальным значением переменной Sent в Υ, т.е. для всех q $Sent_q \ge Sent_p$ в Υ. В частности, это выполняется, если q - сосед по входу p, и из $Rec_p[q] = Sent_q$ следует, что $\min_q Rec_p[q] \ge Sent_p$. Но отсюда следует, что $Sent_p = D$; иначе p послал бы дополнительные сообщения, когда он получил последнее сообщение. Следовательно, $Sent_p = D$ для всех p, и $Rec_p[q] = D$ для всех qp, откуда действительно следует, что каждый процесс принял решение.

Остается показать, что каждому решению предшествует событие в каждом процессе. Если $P = p_0, p_1, ..., p_l$ ($l \le D$) - маршрут в сети, тогда, по Лемме 6.19,

$$f_{p_i p_{i+1}}^{(i+1)} \ \pi \ g_{p_i p_{i+1}}^{(i+1)}$$

для $0 \le i < l$ и, по алгоритму,

$$g_{p_i p_{i+1}}^{(i+1)} \ \pi \ f_{p_{i+1} p_{i+2}}^{(i+2)}$$

для $0 \le i < l - 1$. Следовательно, $f_{p_0 p_1}^{(1)} \ \pi \ g_{p_{l-1} p_l}^{(l)}$. Т.к. диаметр сети равен D, для любых q и p существует маршрут $q = p_0, p_1, ..., p_l = p$ длины не более D. Таким образом, для любого q существует $l \le D$ и сосед по входу r процесса p, такие, что $f_{qq'}^{(1)} \ \pi \ g_{rp}^{(l)}$; на основании алгоритма, $g_{rp}^{(l)}$ предшествует d_p.

Алгоритм пересылает D сообщений через каждый канал, что приводит в сложности сообщений, равной |E|*D. Однако нужно заметить, что |E| обозначает количество *направленных* каналов. Если алгоритм используется для неориентированной сети, каждый канал считается за два направленных канала, и сложность сообщений равна 2|E|*D.

---

var rec_p : 0..N - 1 **init** 0 ;
(* Количество полученных сообщений *)

```
    Sent_p     : 0..1        init  0 ;
                  (* Количество сообщений, посланных каждому соседу *)

begin  if  p - инициатор  then
         begin  forall  r ∈ Neigh_p  do  send <tok>  to r ;
                Sent_p := Sent_p + 1
         end ;
       while  Rec_p < # Neigh_p  do
            begin  receive  <tok> ;
                   Rec_p := Rec_p + 1 ;
                   if  Sent_p = 0  then
                      begin  forall  r ∈ Neigh_p do  send <tok>  to r ;
                             Sent_p := Sent_p + 1
                      end
            end ;
       decide
end
```

Алгоритм 6.8 Фазовый алгоритм для клики.

Фазовый алгоритм для клики. Если сеть имеет топологию клика, ее диаметр равен 1; в этом случае от каждого соседа должно быть получено ровно одно сообщение, и для каждого процесса достаточно посчитать общее количество полученных сообщений вместо того, чтобы считать сообщения от каждого соседа по входу отдельно; см. Алгоритм 6.8. Сложность сообщений в этом случае равна N(N-1) и алгоритм использует только O(log N) бит оперативной памяти.

6.2.6 Алгоритм Финна

Алгоритм Финна [Fin79] - еще один волновой алгоритм, который можно использовать в ориентированных сетях произвольной топологии. Он не требует того, чтобы диаметр сети был известен заранее, но подразумевает наличие уникальных идентификаторов процессов. В сообщениях передаются множества идентификаторов процессов, что приводит к довольно высокой битовой сложности алгоритма.

Процесс p содержит два множества идентификаторов процессов, Inc_p и $NInc_p$. Неформально говоря, Inc_p - это множество процессов q таких, что событие в q предшествует последнему произошедшему событию в p, а $NInc_p$ - множество процессов q таких, что для всех соседей r процесса q событие в r предшествует последнему произошедшему событию в p. Эта зависимость поддерживается следующим образом. Изначально $Inc_p = \{p\}$, а $NInc_p = \varnothing$. Каждый раз, когда одно из множеств пополняется, процесс p посылает сообщение, включая в него Inc_p и $NInc_p$. Когда p получает сообщение, включающее множества Inc и NInc, полученные идентификаторы включаются в версии этих множеств в процессе p. Когда p получит сообщения от всех соседей по входу, p включается в $NInc_p$. Когда два множества становятся равны, p принимает решение; см. Алго-

ритм 6.9. Из неформального смысла двух множеств следует, что для каждого процесса q такого, что событие в q предшествует d_p, выполняется следующее: для каждого соседа r процесса q событие в r также предшествует d_p, откуда следует зависимость алгоритма.

В доказательстве корректности демонстрируется, что это выполняется для каждого p, и что из равенства двух множеств следует, что решению предшествует событие в каждом процессе.

Теорема 6.21 *Алгоритм Финна (Алгоритм 6.9) является волновым алгоритмом.*

Доказательство. Заметим, что два множества, поддерживаемые каждым процессом, могут только расширяться. Т.к. размер двух множеств в сумме составляет не менее 1 в первом сообщении, посылаемом по каждому каналу, и не более 2N в последнем сообщении, то общее количество сообщений ограничено 2N*|E|.

Пусть C - вычисление, в котором существует хотя бы один инициатор, и пусть Υ - заключительная конфигурация. Можно показать, как в доказательстве Теоремы 6.20, что если процесс p отправил сообщения хотя бы один раз (каждому соседу), а q - сосед p по выходу, то q тоже отправил сообщения хотя бы один раз. Отсюда следует, что каждый процесс переслал хотя бы одно сообщение (через каждый канал).

```
var  Inc_p     : set of processes        init {p} ;
     NInc_p    : set of processes        init ∅ ;
     rec_p[q]  : boolean  for q ∈ In_p   init false ;
                 (* признак того, получил ли p сообщение от q  *)

begin  if  p - инициатор  then
           forall  r ∈ Out_p  do  send <sets, Inc_p, NInc_p>  to r ;
       while  Inc_p ≠ NInc_p  do
              begin  receive <sets, Inc, NInc>  from q_0 ;
                     Inc_p := Inc_p ∪ Inc ; NInc_p := NInc_p ∪ NInc ;
                     rec_p[q_0] := true ;
                     if  ∀q ∈ In_p : rec_p[q]  then  NInc_p := NInc_p ∪ {p} ;
                     if  Inc_p  или  NInc_p  изменились  then
                        forall  r ∈ Out_p  do  send <sets, Inc_p, NInc_p>  to r
              end ;
       decide
end
```

Алгоритм 6.9 Алгоритм Финна.

Сейчас мы покажем, что в Υ каждый процесс принял решение. Во-первых, если существует ребро pq, то $Inc_p \subseteq Inc_q$ в Υ. Действительно, после последнего изменения Inc_p процесс p посылает сообщение <**sets**, Inc_p, $NInc_p$>, и после его получения в q выполняется $Inc_q := Inc_q \cup Inc_p$. Из сильной связности сети

следует, что $Inc_p = Inc_q$ для всех p и q. Т.к. выполняется $p \in Inc_p$ и каждое множество Inc содержит только идентификаторы процессов, для каждого p $Inc_p = P$.

Во-вторых, подобным же образом может быть показано, что $NInc_p = Ninc_q$ для любых p и q. Т.к. каждый процесс отправил хотя бы одно сообщение по каждому каналу, для каждого процесса p выполняется: $\forall q \in In_p : rec_p[q]$, и следовательно, для каждого p выполняется: $p \in NInc_p$. Множества NInc содержат только идентификаторы процессов, откуда следует, что $NInc_p = P$ для каждого p. Из $Inc_p = P$ и $NInc_p = P$ следует, что $Inc_p = NInc_p$, следовательно, каждый процесс p в Υ принял решение.

Теперь нужно показать, что решению d_p в процессе p предшествуют события в каждом процессе. Для события e в процессе p обозначим через $Inc^{(e)}$ (или, соответственно, $NInc^{(e)}$) значение Inc_p ($NInc_p$) сразу после выполнения e (сравните с доказательством Теоремы 6.12). Следующие два утверждения формализуют неформальные описания множеств в начале этого раздела.

Утверждение 6.22 *Если существует событие* $e \in C_q : e \underline{\pi} f$, *то* $q \in Inc^{(f)}$.

Доказательство. Как в доказательстве Теоремы 6.12, можно показать, что $e \underline{\pi} f \Rightarrow Inc^{(e)} \subseteq Inc^{(f)}$, а при $e \in C_q \Rightarrow q \in Inc^{(e)}$, что и требовалось доказать.

Утверждение 6.23 *Если* $q \in NInc^{(f)}$, *тогда для всех* $r \in In_q$ *существует событие* $e \in C_r : e \underline{\pi} f$.

Доказательство. Пусть a_q - внутреннее событие q, в котором впервые в q выполняется присваивание $NInc_q := NInc_q \cup \{q\}$. Событие a_q - единственное событие с $q \in NInc^{(aq)}$, которому не предшествует никакое другое событие a′, удовлетворяющее условию $q \in NInc^{(a')}$; таким образом, $q \in NInc^{(f)} \Rightarrow a_q \underline{\pi} f$.

Из алгоритма следует, что для любого $r \in In_q$ существует событие $e \in C_r$, предшествующее a_q. Отсюда следует результат.

Процесс p принимает решение только когда $Inc_p = NInc_p$; можно записать, что $Inc^{(dp)} = NInc^{(dp)}$. В этом случае

(1) $p \in Inc^{(dp)}$; и

(2) из $q \in Inc^{(dp)}$ следует, что $q \in NInc^{(dp)}$, откуда следует, что $In_q \subseteq Inc^{(dp)}$.

Из сильной связности сети следует требуемый результат: $Inc^{(dp)} = P$.

6.3 Алгоритмы обхода

В этом разделе будет представлен особый класс волновых алгоритмов, а именно, волновые алгоритмы, в которых все события волны совершенно упорядочены каузальным отношением, и в котором последнее событие происходит в том же процессе, где и первое.

Определение 6.24 *Алгоритмом обхода называется алгоритм, обладающий следующими тремя свойствами.*

(1) В каждом вычислении один инициатор, который начинает выполнение алгоритма, посылая ровно одно сообщение.

(2) Процесс, получая сообщение, либо посылает одно сообщение дальше, либо принимает решение.

Из первых двух свойств следует, что в каждом конечном вычислении решение принимает ровно один процесс. Говорят, что алгоритм *завершается* в этом процессе.

(3) Алгоритм завершается в инициаторе и к тому времени, когда это происходит, каждый процесс посылает сообщение хотя бы один раз.

В каждой достижимой конфигурации алгоритма обхода либо передается ровно одно сообщение, либо ровно один процесс получил сообщение и еще не послал ответное сообщение. С более абстрактной точки зрения, сообщения в вычислении, взятые вместе, можно рассматривать как единый объект (*маркер*), который передается от процесса к процессу и, таким образом, «посещает» все процессы. В Разделе 7.4 алгоритмы обхода используются для построения алгоритмов выбора и для этого важно знать не только общее количество переходов маркера в одной волне, но и сколько переходов необходимо для того, чтобы посетить первые x процессов.

Определение 6.25 *Алгоритм называется алгоритмом* f*-обхода (для класса сетей), если*

(1) он является алгоритмом обхода (для этого класса); и

(2) в каждом вычислении после f(x) *переходов маркера посещено не менее* min (N, x+1) *процессов.*

Кольцевой алгоритм (Алгоритм 6.2) является алгоритмом обхода, и, поскольку x+1 процесс получил маркер после x шагов (для $x < N$), а все процессы получат его после N шагов, это алгоритм x-обхода для кольцевой сети.

6.3.1 Обход клик

Клику можно обойти путем *последовательного опроса*; алгоритм очень похож на Алгоритм 6.6, но за один раз опрашивается только один сосед инициатора. Только когда получен ответ от одного соседа, опрашивается следующий; см. Алгоритм 6.10.

---

var rec_p : integer **init** 0 ; (* только для инициатора *)
Для инициатора:
(* обозначим $Neigh_p = \{q_1, q_2, .., q_{N-1}\}$ *)
begin while $rec_p < \#\, Neigh_p$ **do**
begin send <**tok**> to q_{rec_p+1} ;
receive <**tok**>; $rec_p := rec_p + 1$
end ;
decide
end
Для не-инициатора:
begin receive <**tok**> from q ; send <**tok**> to q **end**

Алгоритм 6.10 Последовательный алгоритм опроса.

Теорема 6.26 *Последовательный алгоритм опроса (Алгоритм 6.10) является алгоритмом* 2x*-обхода для клик.*

Доказательство. Легко заметить, что к тому времени, когда алгоритм завершается, каждый процесс послал инициатору ответ. (2x-1)-е сообщение - это опрос для q_x, а (2x)-е - это его ответ. Следовательно, когда было передано 2x сообщений, был посещен x+1 процесс p, q_1, ..., q_x.

6.3.2 Обход торов

Тором n×n называется граф G = (V,E), где

$$V = \mathbf{Z}_n \times \mathbf{Z}_n = \{ (i, j) : 0 \le i, j < n\} \text{ и}$$

$$E = \{(i, j)(i', j') : (i = i' \ \& \ j = j' \pm 1) \vee (i = i' \pm 1 \ \& \ j = j')\};$$

см. Раздел B.2.4. (Сложение и вычитание здесь по модулю n.) Предполагается, что тор обладает чувством направления (см. Раздел B.3), т.е. в вершине (i, j) канал к (i, j+1) имеет метку *Up*, канал к (i, j-1) - метку *Down*, канал к (i+1, j) - метку *Right*, и канал к (i-1, j) - метку *Left*. Координатная пара (i, j) удобна для определения топологии сети и ее чувства направления, но мы предполагаем, что процессы не знают этих координат; топологическое знание ограничено метками каналов.

Для инициатора (выполняется один раз):
send <**num**, 1> to *Up*
Для каждого процесса при получении маркера <**num**,k>:
begin if $k=n^2$ **then** *decide*
else if n|k **then** send <**num**,k+1> to *Up*
else send <**num**,k+1> to *Right*
end

Алгоритм 6.11 Алгоритм обхода для торов.

Тор является Гамильтоновым графом, т.е. в торе (произвольного размера) существует Гамильтонов цикл и маркер передается по циклу с использованием Алгоритма 6.11. После k-го перехода маркера он пересылается вверх, если n|k (k делится на n), иначе он передается направо.

Теорема 6.27 *Алгоритм для тора (Алгоритм 6.11) является алгоритмом* x*-обхода для торов.*

Доказательство. Как легко заметить из алгоритма, решение принимается после того, как маркер был передан n^2 раз. Если маркер переходит от процесса p к процессу q с помощью U переходов вверх и R переходов вправо, то p = q тогда и только тогда, когда (n|U & n|R). Обозначим через p_0 инициатор, а через p_k - процесс, который получает маркер <**num**, k>.

Из n^2 переходов маркера n - переходы вверх, а оставшиеся n^2-n - переходы вправо. Т.к. и n, и n^2-n кратны n, то $p_{n^2} = p_0$, следовательно, алгоритм завершается в инициаторе.

Далее будет показано, что все процессы с p_0 по p_{n^2-1} различны; т.к. всего n^2 процессов, то отсюда следует, что каждый процесс был пройден. Предположим, что $p_a = p_b$ для $0 \le a < b < n^2$. Между p_a и p_b маркер сделал несколько переходов вверх и вправо, и т.к. $p_a = p_b$, количество переходов кратно n. Изучив алгоритм, можно увидеть, что отсюда следует, что

$\#\{k : a \le k < b \ \& \ n|k\}$ кратно n, и

$\#\{k : a \le k < b \ \& \ n \text{ не } |k\}$ кратно n.

Размеры двух множеств в сумме составляют b-a, откуда следует, что n|(b-a). Обозначим (b-a) = l*n, тогда множество $\{k: a \le k < b\}$ содержит l кратных n. Отсюда следует, что n|l, а значит $n^2|(b-a)$, что приводит к противоречию.

Т.к. все процессы с p_0 по p_{n^2-1} различны, после x переходов маркера будет посещен x+1 процесс.

6.3.3 Обход гиперкубов

N-мерным гиперкубом называется граф G = (V,E), где

$V = \{ (b_0,...,b_{n-1}) : b_i = 0, 1\}$ и

$E = \{ (b_0,...,b_{n-1}),(c_0,...,c_{n-1}) :$ b и c отличаются на 1 бит$\}$;

см. Подраздел В.2.5. Предполагается, что гиперкуб обладает чувством направления (см. Раздел В.3), т.е. канал между вершинами b и c, где b и c различаются битом с номером i, помечается «i» в обеих вершинах. Предполагается, что метки вершин не известны процессам; их топологическое знание ограничено метками каналов.

Как и тор, гиперкуб является Гамильтоновым графом, и Гамильтонов цикл обходится с использованием Алгоритма 6.12. Доказательство корректности алгоритма похоже на доказательство для Алгоритма 6.11.

Для инициатора (выполняется один раз):
send <**num**, 1> по каналу n -1
Для каждого процесса при получении маркера <**num**,k>:
begin if $k=2^n$ **then** *decide*
else begin пусть l - наибольший номер : $2^l|k$;
send <**num**,k+1> по каналу *l*
end
end

Алгоритм 6.12 Алгоритм обхода для гиперкубов.

Теорема 6.28 *Алгоритм для гиперкуба (Алгоритм 6.12) является алгоритмом* x-*обхода для гиперкуба.*

Доказательство. Из алгоритма видно, что решение принимается после 2^n пересылок маркера. Пусть p_0 - инициатор, а p_k - процесс, который получает

маркер <**num**,k>. Для любого $k < 2^n$, обозначения p_k и p_{k+1} отличаются на 1 бит, индекс которого обозначим как l(k), удовлетворяющий следующему условию:

$$l(k) = \begin{cases} n-1; & if\ k = 0 \\ наибольшее\ l\ c\ \ 2^l | k; & if\ k > 0 \end{cases}$$

Т.к. для любого $i < n$ существует равное количество $k \in \{0, ..., 2^n\}$ с $l(k) = i$, то $p_0 = p_{2^n}$ и решение принимается в инициаторе. Аналогично доказательству Теоремы 6.27, можно показать, что из $p_a = p_b$ следует, что $2^n | (b-a)$, откуда следует, что все $p_0, ..., p_{2^n-1}$ различны.

Из всего этого следует, что, когда происходит завершение, все процессы пройдены, и после x переходов маркера будет посещен x+1 процесс.

6.3.4 Обход связных сетей

Алгоритм обхода для произвольных связных сетей был дан Тарри в 1895 [Tarry; T1895]. Алгоритм сформулирован в следующих двух правилах; см. Алгоритм 6.13.

R1. Процесс никогда не передает маркер дважды по одному и тому же каналу.

R2. Не-инициатор передает маркер своему *родителю* (соседу, от которого он впервые получил маркер), только если невозможна передача по другим каналам, в соответствии с правилом R1.

```
var used_p[q]   : boolean   init false для всех q ∈ Neigh_p ;
                            (* Признак того, отправил ли p сообщение q *)
    father_p    : process   init udef ;

Для инициатора (выполняется один раз):
    begin  father_p := p ;  выбор  q ∈ Neigh_p ;
           used_p[q] := true ;  send <tok>  to q ;
    end

Для каждого процесса при получении <tok> от q_0:
    begin  if  father_p = udef  then  father_p := q_0 ;
           if  ∀q ∈ Neigh_p : used_p[q]
             then  decide
             else  if  ∃q ∈ Neigh_p : (q ≠ father_p  &  ¬used_p[q])
                     then  begin  выбор  q ∈ Neigh_p \ {father_p}  c ¬used_p[q] ;
                                  used_p[q] := true ;  send <tok>  to q
                           end
                     else  begin  used_p[father_p] := true ;
                                  send <tok>  to father_p
                           end
    end
```

Алгоритм 6.13 Алгоритм обхода Тарри.

Теорема 6.29 *Алгоритм Тарри (Алгоритм 6.13) является алгоритмом обхода.*

Доказательство. Т.к. маркер передается не более одного раза в обоих направлениях через каждый канал, всего он передается не более 2|E| раз до завершения алгоритма. Т.к. каждый процесс передает маркер через каждый канал не более одного раза, то каждый процесс получает маркер через каждый канал не более одного раза. Каждый раз, когда маркер захватывается не-инициатором p, получается, что процесс p получил маркер на один раз больше, чем послал его. Отсюда следует, что количество каналов, инцидентных p, превышает количество каналов, использованных p, по крайней мере, на 1. Таким образом, p не принимает решение, а передает маркер дальше. Следовательно, решение принимается в инициаторе.

Далее, за 3 шага будет доказано, что когда алгоритм завершается, каждый процесс передал маркер.

(1) *Все каналы, инцидентные инициатору, были пройдены один раз в обоих направлениях.* Инициатором маркер был послан по всем каналам, иначе алгоритм не завершился бы. Инициатор получил маркер ровно столько же раз, сколько отправил его; т.к. инициатор получал маркер каждый раз через другой канал, то маркер пересылался через каждый канал по одному разу.

(2) *Для каждого посещаемого процесса* p *все каналы, инцидентные* p *были пройдены один раз в каждом направлении.* Предположив, что это не так, выберем в качестве p первый встретившийся процесс, для которого это не выполняется, и заметим, что из пункта (1) p не является инициатором. Из выбора p, все каналы, инцидентные $father_p$ были пройдены один раз в обоих направлениях, откуда следует, что p переслал маркер своему родителю. Следовательно, p использовал все инцидентные каналы, чтобы переслать маркер; но, т.к. маркер в конце остается в инициаторе, p получил маркер ровно столько же раз, сколько отправил его, т.е. p получил маркер по одному разу через каждый инцидентный канал. Мы пришли к противоречию.

(3) *Все процессы были посещены и каждый канал был пройден по одному разу в обоих направлениях.* Если есть непосещенные процессы, то существуют соседи p и q такие, что p был посещен, а q не был. Это противоречит тому, что все каналы p были пройдены в обоих направлениях. Следовательно, из пункта (2), все процессы были посещены и все каналы пройдены один раз в обоих направлениях.

Каждое вычисление алгоритма Тарри определяет остовное дерево сети, как показано в Лемме 6.3. В корне дерева находится инициатор, а каждый не-инициатор p в конце вычисления занес своего родителя в дереве в переменную $father_p$. Желательно, чтобы каждый процесс также знал (в конце вычисления), какие из его соседей являются его сыновьями в дереве. Этого можно достигнуть, посылая родителю $father_p$ специальное сообщение.

6.4 Временная сложность: поиск в глубину

Процессы в алгоритме Тарри достаточно свободны в выборе соседа, которому передается маркер, в результате чего возникает большой класс остовных деревьев. В этом разделе будут обсуждаться алгоритмы, которые вычисляют остовные деревья с дополнительным свойством: каждое листовое ребро соединяет две вершины, одна из которых является предком другой. Листовое ребро - это ребро, не принадлежащее остовному дереву. В данном корневом остовном дереве T сети G для каждого p T[p] обозначает множество процессов в поддереве p, а A[p] обозначает предков p, т.е. вершины на пути в T от корня до p. Заметим, что $q \in T[p] \Leftrightarrow p \in A[q]$.

Определение 6.30 *Остовное дерево* T *сети* G *называется деревом поиска в глубину, если для каждого листового ребра* pq $q \in T[p] \vee q \in A[p]$.

Деревья поиска в глубину, или DFS-деревья (DFS - Depth-First Search), используются во многих алгоритмах на графах, таких как проверка планарности, проверка двусвязности, и для построения интервальных схем маркировки (см. Подраздел 4.4.2). В Разделе 6.4.1 будет показано, что незначительная модификация алгоритма Тарри (а именно, ограничение свободы выбора процессов) позволяет алгоритму вычислять деревья поиска в глубину. Получившийся алгоритм будем называть *классическим алгоритмом поиска в глубину*. В Подразделе 6.4.2 будут представлены два алгоритма, которые вычисляют деревья поиска в глубину за меньшее *время*, чем классический алгоритм. Понятие временной сложности распределенных алгоритмов будет определено ниже. В Подразделе 6.4.3 будет представлен алгоритм поиска в глубину для сетей с начальным знанием идентификации соседей. В этом случае Теорема 6.6 неприменима, и фактически может быть дан алгоритм, использующий только 2N-2 сообщений.

Временная сложность распределенных алгоритмов. Исключительно в целях анализа распределенных алгоритмов, будут сделаны некоторые идеализированные временные предположения. Корректность алгоритмов никак не зависит от этих предположений.

Определение 6.31 *Временная сложность распределенного алгоритма - это максимальное время, требуемое на вычисление алгоритма при следующих предположениях:*

T1. *Процесс может выполнить любое конечное количество событий за нулевое время.*

T2. *Промежуток времени между отправлением и получением сообщения - не более одной единицы времени.*

Временные сложности всех волновых алгоритмов этой главы изложены в Таблице 6.19. Проверка величин из этой таблицы, не доказанных в этой главе, оставлена читателю в качестве упражнения. Альтернативные определения временной сложности обсуждаются в Подразделе 6.5.3.

Лемма 6.32 *Для алгоритмов обхода временная сложность равна сложности сообщений.*

Доказательство. Сообщения пересылаются последовательно, и каждое может занять одну единицу времени.

6.4.1 Распределенный поиск в глубину

Классический алгоритм поиска в глубину получается, когда свобода выбора соседа для передачи маркера в алгоритме Тарри ограничивается следующим, третьим правилом; см. Алгоритм 6.14.

R3. Когда процесс получает маркер, он отправляет его обратно через тот же самый канал, если это позволяется правилами R1 и R2.

Теорема 6.33 *Классический алгоритм поиска в глубину (Алгоритм 6.14) вычисляет остовное дерево поиска в глубину, используя 2*|E| *сообщений и* 2|E| *единиц времени.*

Доказательство. Т.к. алгоритм реализует алгоритм Тарри, это алгоритм обхода и он вычисляет остовное дерево T. Уже было показано, что каждый канал передает два сообщения (по одному в обоих направлениях), что доказывает оценку сложности сообщений, а оценка для временной сложности следует из того, что 2|E| сообщений передаются одно за другим, причем каждое занимает не более одной единицы времени. Остается показать, что из правила R3 следует, что получающееся дерево - дерево поиска в глубину.

Во-первых, из R3 следует, что за первым проходом по листовому ребру немедленно следует второй, в обратном направлении. Пусть pq - листовое ребро и p первым использует ребро. Когда q получает маркер от p, q уже посещен (иначе q присвоит $father_q$ p и ребро не будет листовым) и $used_p[q]$ = false (т.к. по предположению p первый из двух процессов использовал ребро). Следовательно, по правилу R3, q немедленно отправляет маркер обратно p.

Теперь можно показать, что если pq - листовое ребро, используемое сначала p, то $q \in A[p]$. Рассмотрим путь, пройденный маркером, пока он не был переслан через pq. Т.к. pq - листовое ребро, q был посещен до того, как маркер попал в q через это ребро:

..., q, ..., p, q

Получим из этого пути возможно более короткий путь, заменив все комбинации r_1, r_2, r_1, где r_1r_2 - листовое ребро, на r_1. Из предыдущих наблюдений, все листовые ребра теперь убраны, откуда следует, что получившийся путь - это путь в T, состоящий только из ребер, пройденных до первого прохождения pq. Если q не является предком p, то отсюда следует, что ребро от q до $father_q$ было пройдено до того, как было пройдено ребро qp, что противоречит правилу R2 алгоритма.

---

var $used_p[q]$: boolean **init** *false* для всех $q \in Neigh_p$;
(* Признак того, отправил ли p сообщение q *)
$father_p$: process **init** *udef* ;

Для инициатора (выполняется один раз):
begin $father_p$:= p ; выбор $q \in Neigh_p$;
$used_p[q]$:= *true* ; send <**tok**> to q ;

```
    end

Для каждого процесса при получении <tok> от q0:
    begin  if  father_p = udef  then  father_p := q0 ;
           if  ∀q ∈ Neigh_p : used_p[q]
             then  decide
             else  if  ∃q ∈ Neigh_p : (q ≠ father_p  &  ¬used_p[q])
                     then  begin  if  father_p ≠ q0  &  ¬used_p[q0]
                                    then q := q0
                                    else  выбор  q ∈ Neigh_p \ {father_p}
                                                   с ¬used_p[q] ;
                                  used_p[q] := true ;  send <tok>  to q
                           end
                     else  begin  used_p[father_p] := true ;
                                  send <tok>  to father_p
                           end
    end
```

Алгоритм 6.14 Классический алгоритм поиска в глубину.

Сложность сообщений классического распределенного поиска в глубину равна *2*|E|, по Теореме 6.6 это наилучшая сложность (за исключением множителя 2), если идентификация соседей не известна изначально. Временная сложность также составляет 2|E|, что по Лемме 6.32, является наилучшей сложностью для алгоритмов обхода в этом случае. Распределенная версия поиска в глубину была впервые представлена Cheung [Che83].

В Подразделе 6.4.2 будут рассмотрены два алгоритма, которые строят дерево поиска в глубину в сети без знания идентификации соседей за O(N) единиц времени. Следовательно, эти алгоритмы не являются алгоритмами обхода. В Подразделе 6.4.3 знание о соседях будет использовано для получения алгоритма с временной сложностью и сложностью сообщений O(N).

6.4.2 Алгоритмы поиска в глубину за линейное время

Причина высокой временной сложности классического алгоритма поиска в глубину состоит в том, что все ребра, как принадлежащие дереву, так и листовые, обходятся последовательно. Сообщение-маркер <**tok**> проходит по всем листовым ребрам и немедленно возвращается обратно, как показано в доказательстве Теоремы 6.33. Все решения с меньшей временной сложностью основаны на том, что маркер проходит только по ребрам дерева. Очевидно, на это потребуется линейное время, т.к. существует только N-1 ребро дерева.

Решение Авербаха. В алгоритм включается механизм, который предотвращает передачу маркера через листовое ребро. В алгоритме Авербаха [Awerbuch; Awe85b] гарантируется, что каждый процесс в момент, когда он должен передать маркер, знает, какие из его соседей уже были пройдены. Затем процесс выбирает непройденного соседа, или, если такого не существует, посылает маркер своему родителю.

Когда процесс p впервые посещается маркером (для инициатора это происходит в начале алгоритма), p сообщает об этом всем соседям r, кроме его родителя, посылая сообщения <**vis**>. Передача маркера откладывается, пока p не получит сообщения <**ack**> от всех соседей. При этом гарантируется, что каждый сосед r процесса p в момент, когда p передает маркер, знает, что p был посещен. Когда позже маркер прибывает в r, r не передаст маркер p, если только p не его родитель; см. Алгоритм 6.15.

Из-за передачи сообщений <**vis**> в большинстве случаев $used_p[father_p]$ = true, даже если p еще не послал маркер своему родителю. Поэтому в алгоритме должно быть явно запрограммировано, что только инициатор может принимать решения; а не-инициатор p, для которого $used_p[q]$ = true для всех соседей q, передает маркер своему родителю.

Теорема 6.34 *Алгоритм Авербаха (Алгоритм 6.15) вычисляет дерево поиска в глубину за* 4N-2 *единиц времени и использует* 4|E| *сообщений.*

Доказательство. По существу, маркер передается по тем же самым каналам, как и в Алгоритме 6.14, за исключением того, что пропускается передача по листовым каналам. Т.к. передача по листовым каналам не влияет на конечное значение $father_p$ для любого процесса p, то в результате всегда получается дерево, которое может получиться и в Алгоритме 6.14.

Маркер последовательно проходит дважды через каждый из N-1 каналов дерева, на что тратится 2N-2 единиц времени. В каждой вершине маркер простаивает перед тем, как быть переданным, не более одного раза из-за обмена сообщениями <**vis**>/<**ack**>, что приводит к задержке не более чем на 2 единицы времени в каждой вершине.

Каждое листовое ребро передает два сообщения <**vis**> и два сообщения <**ack**>. Каждое ребро в дереве передает два сообщения <**tok**>, одно <**vis**> (посылаемое от родителя потомку), и одно <**ack**> (от потомка родителю). Следовательно, передается 4|E| сообщений.

---

var $used_p[q]$: boolean **init** *false* для всех $q \in Neigh_p$;
(* Признак того, отправил ли p сообщение q *)
$father_p$: process **init** *udef* ;

Для инициатора (выполняется один раз):

begin $father_p := p$; выбор $q \in Neigh_p$;
forall $r \in Neigh_p$ **do** send <**vis**> to r ;
forall $r \in Neigh_p$ **do** receive <**ack**> from r ;
$used_p[q] :=$ true ; send <**tok**> to q ;
end

Для каждого процесса при получении <**tok**> от q_0:

begin **if** $father_p = udef$ **then**
begin $father_p := q_0$;
forall $r \in Neigh_p \setminus \{father_p\}$ **do** send <**vis**> to r ;
forall $r \in Neigh_p \setminus \{father_p\}$ **do** receive <**ack**> from r ;
end ;

```
if p - инициатор and ∀q ∈ Neigh_p : used_p[q]
  then decide
  else if ∃q ∈ Neigh_p : (q ≠ father_p & ¬used_p[q])
          then begin if father_p ≠ q_0 & ¬used_p[q_0]
                       then q := q_0
                       else выбор q ∈ Neigh_p \ {father_p}
                                  с ¬used_p[q] ;
                     used_p[q] := true ; send <tok> to q
               end
          else begin used_p[father_p] := true ;
                     send <tok> to father_p
               end
end
```

Для каждого процесса при получении <**vis**> от q_0:
begin $used_p[q_0]$:= *true* ; send <**ack**> to q_0 **end**

Алгоритм 6.15 Алгоритм поиска в глубину Авербаха.

Передачу сообщения <**vis**> соседу, которому процесс передает маркер, можно опустить. Это усовершенствование (не выполняемое в Алгоритме 6.15) сберегает по два сообщения для каждого ребра дерева и, следовательно, уменьшает сложность сообщений на 2N-2 сообщения.

Решение Сайдона. Алгоритм Сайдона [Cidon; Cid88] улучшает временную сложность алгоритма Авербаха, не посылая сообщения <**ack**>. В модификации Сайдона маркер передается немедленно, т.е. без задержки на 2 единицы времени, внесенной в алгоритм Авербаха ожиданием подтверждения. Этот же алгоритм был предложен Лакшмананом и др. [Lakshmanan; LMT87]. В алгоритме Сайдона может возникнуть следующая ситуация. Процесс p получил маркер и послал сообщение <**vis**> своему соседу r. Маркер позже попадает в r, но в момент, когда r получает маркер, сообщение <**vis**> процесса p еще не достигло r. В этом случае r может передать маркер p, фактически посылая его через листовое ребро. (Заметим, что сообщения <**ack**> в алгоритме Авербаха предотвращают возникновение такой ситуации.)

Чтобы обойти эту ситуацию процесс p запоминает (в переменной mrs_p), какому соседу он переслал маркер в последний раз. Когда маркер проходит только по ребрам дерева, p получает его в следующий раз от того же соседа mrs_p. В ситуации, описанной выше, p получает сообщение <**tok**> от другого соседа, а именно, от r; в этом случае *маркер игнорируется*, но p помечает ребро rp, как пройденное, как если бы от r было получено сообщение <**vis**>. Процесс r получает сообщение <**vis**> от p после пересылки маркера p, т.е. r получает сообщение <**vis**> от соседа mrs_r. В этом случае r действует так, как если бы он еще не послал маркер p; r выбирает следующего соседа и передает маркер; см. Алгоритм 6.16/6.17.

Теорема 6.35 *Алгоритм Сайдона (Алгоритм 6.16/6.17) вычисляет* DFS-*дерево за* 2N-2 *единиц времени, используя* 4|E| *сообщений.*

Доказательство. Маршрут маркера подобен маршруту в Алгоритме 6.14. Прохождение по листовым ребрам либо пропускается (как в Алгоритме 6.15), либо в ответ на маркер через листовое ребро посылается сообщение <**vis**>. В последнем случае, процесс, получая сообщение <**vis**>, продолжает передавать маркер, как если бы маркер был возвращен через листовое ребро немедленно.

Время между двумя успешными передачами маркера по дереву ограничено одной единицей времени. Если маркер послали по ребру дерева процессу p в момент времени t, то в момент t все сообщения <**vis**> ранее посещенных соседей q процесса p были посланы, и, следовательно, эти сообщения прибудут не позднее момента времени t+1. Итак, хотя p мог несколько раз послать маркер по листовым ребрам до t+1, не позднее t+1 p восстановился после всех этих ошибок и передал маркер по ребру, принадлежащему дереву. Т.к. должны быть пройдены 2N-2 ребра дерева, алгоритм завершается за 2N-2 единиц времени.

Через каждый канал передается не более двух сообщений <**vis**> и двух <**tok**>, откуда граница сложности сообщений равна 4|E|.

---

var $used_p[q]$: boolean **init** *false* для всех $q \in Neigh_p$;
$father_p$: process **init** *udef* ;
mrs_p : process **init** *udef* ;

Для инициатора (выполняется один раз):
begin $father_p := p$; выбор $q \in Neigh_p$;
forall $r \in Neigh_p$ **do** send <**vis**> to r ;
$used_p[q] :=$ true ; $mrs_p := q$; send <**tok**> to q ;
end

Для каждого процесса при получении <**vis**> от q_0:
begin $used_p[q_0] :=$ true ;
if $q_0 = mrs_p$ **then** (* интерпретировать, как <**tok**> *)
передать сообщение <**tok**> как при получении <**tok**>
end

Алгоритм 6.16 Алгоритм поиска в глубину Сайдона (Часть 1).

Для каждого процесса при получении <**tok**> от q_0:
begin **if** $mrs_p \neq$ *udef* and $mrs_p \neq q_0$
(* это листовое ребро, интерпретируем как сообщение <**vis**>*)
then $used_p[q_0] :=$ true
else (* действовать, как в предыдущем алгоритме *)
begin **if** $father_p =$ *udef* **then**
begin $father_p := q_0$;
forall $r \in Neigh_p \setminus \{father_p\}$
do send <**vis**> to r ;

```
                    end ;
                  if p - инициатор and ∀q ∈ Neigh_p : used_p[q]
                     then decide
                     else if ∃q ∈ Neigh_p : (q ≠ father_p & ¬used_p[q])
                             then begin if father_p ≠ q_0 & ¬used_p[q_0]
                                          then q := q_0
                                          else выбор q ∈ Neigh_p\ {father_p}
                                                  с ¬used_p[q] ;
                                        used_p[q] := true ; mrs_p := q ;
                                        send <tok> to q
                                  end
                             else begin used_p[father_p] := true ;
                                        send <tok> to father_p
                                  end
            end
end
```

Алгоритм 6.17 Алгоритм поиска в глубину Сайдона (Часть 2).

Во многих случаях этот алгоритм будет пересылать меньше сообщений, чем алгоритм Авербаха. Оценка количества сообщений в алгоритме Сайдона предполагает наихудший случай, а именно, когда маркер пересылается через каждое листовое ребро в обоих направлениях. Можно ожидать, что сообщения <**vis**> помогут избежать многих нежелательных пересылок, тогда через каждый канал будет передано только два или три сообщения.

Сайдон замечает, что хотя алгоритм может передать маркер в уже посещенную вершину, он обладает лучшей временной сложностью (и сложностью сообщений), чем Алгоритм 6.15, который предотвращает такие нежелательные передачи. Это означает, что на восстановление после ненужных действий может быть затрачено меньше времени и сообщений, чем на их предотвращение. Сайдон оставляет открытым вопрос о том, существует ли DFS-алгоритм, который достигает сложности сообщений классического алгоритма, т.е. 2|E|, и который затрачивает O(N) единиц времени.

6.4.3 Поиск в глубину со знанием соседей

Если процессам известны идентификаторы их соседей, проход листовых ребер можно предотвратить, включив в маркер список посещенных процессов. Процесс p, получая маркер с включенным в него списком L, не передает маркер процессам из L. Переменная $used_p[q]$ не нужна, т.к. если p ранее передал маркер q, то q ∈ L; см. Алгоритм 6.18.

Теорема 6.36 DFS-*алгоритм со знанием соседей является алгоритмом обхода и вычисляет дерево поиска в глубину, используя* 2N-2 *сообщений за* 2N-2 *единиц времени.*

У этого алгоритма высокая битовая сложность; если w - количество бит, необходимых для представления одного идентификатора, список L может занять до Nw бит; см. Упражнение 6.14.

var $father_p$: process **init** *udef* ;

Для инициатора (выполняется один раз):

```
begin  father_p := p ;  выбор  q ∈ Neigh_p ;
       send <tlist, {p}>  to q
end
```

Для каждого процесса при получении <**tlist**, L> от q_0:

```
begin  if  father_p = udef  then  father_p := q_0 ;
       if  ∃q ∈ Neigh_p \ L
          then  begin  выбор  q ∈ Neigh_p \ L ;
                       send < tlist, L∪{p} >  to q
                end
          else  if  p - инициатор
                    then  decide
                    else  send < tlist, L∪{p} > to father_p
end
```

Алгоритм 6.18 Алгоритм поиска в глубину со знанием соседей.

6.5 Остальные вопросы

6.5.1 Обзор волновых алгоритмов

В Таблице 6.19 дан список волновых алгоритмов, рассмотренных в этой главе. В столбце «Номер» дана нумерация алгоритмов в главе; в столбце «C/D» отмечено, является ли алгоритм централизованным (C) или децентрализованным (D); столбец «T» определяет, является ли алгоритм алгоритмом обхода; в столбце «Сообщения» дана сложность сообщений; в столбце «Время» дана временная сложность. В этих столбцах N - количество процессов, |E| - количество каналов, D - диаметр сети (в переходах).

Алгоритм	Номер	Топология	C/D	T	Сообщения	Время
Раздел 6.2: Общие алгоритмы						
Кольцевой	6.2	кольцо	C	да	N	N
Древовидный	6.3	дерево	D	нет	N	O(D)
Эхо-алгоритм	6.5	произвольная	C	нет	2|E|	O(N)
Алгоритм опроса	6.6	клика	C	нет	2N-2	2
Фазовый	6.7	произвольная	D	нет	2D|E|	2D
Фазовый на кликах	6.8	клика	D	нет	N(N-1)	2
Финна	6.9	произвольная	D	нет	≤4N|E|	O(D)
Раздел 6.3: Алгоритмы обхода						

Последовательный опрос	6.10	клика	C	да	2N-2	2N-2
Для торов	6.11	тор	C	да	N	N
Для гиперкубов	6.12	гиперкуб	C	да	N	N
Тарри	6.13	произвольная	C	да	2|E|	2|E|
Раздел 6.4: Алгоритмы поиска в глубину						
Классический	6.14	произвольная	C	да	2|E|	2|E|
Авербаха	6.15	произвольная	C	нет	4|E|	4N-2
Сайдона	6.16 /6.17	произвольная	C	нет	4|E|	2N-2
Со знанием соседей	6.18	произвольная	C	да	2N-2	2N-2

Замечание: фазовый алгоритм (6.7) и алгоритм Финна (6.9) подходят для ориентированных сетей.

Таблица 6.19 Волновые алгоритмы этой главы.

Сложность распространения волн в сетях большинства топологий значительно зависит от того, централизованный алгоритм или нет. В Таблице 6.20 приведена сложность сообщений централизованных и децентрализованных волновых алгоритмов для колец, произвольных сетей и деревьев. Таким же образом можно проанализировать зависимость сложности от других параметров, таких как знание соседей или чувство направления (Раздел В.3).

Топология	C/D	Сложность	Ссылка
Кольцо	C	N	Алгоритм 6.2
	D	O(N logN)	Алгоритм 7.7
Произвольная	C	2|E|	Алгоритм 6.5
	D	O(N logN + |E|)	Раздел 7.3
Дерево	C	2(N-1)	Алгоритм 6.5
	D	O(N)	Алгоритм 6.3

Таблица 6.20 Влияние централизации на сложность сообщений.

6.5.2 Вычисление сумм

В Подразделе 6.1.5 было показано, что за одну волну можно вычислить инфимум по входам всех процессов. Вычисление инфимума может быть использовано для вычисления коммутативного, ассоциативного и идемпотентного оператора, обобщенного на входы, такого как минимум, максимум, и др. (см. Заключение 6.14). Большое количество функций не вычислимо таким образом,

среди них - сумма по всем входам, т.к. оператор суммирования не идемпотентен. Суммирование входов может быть использовано для подсчета процессов с определенным свойством (путем присваивания входу 1, если процесс обладает свойством, и 0 в противном случае), и результаты этого подраздела могут быть распространены на другие операторы, являющиеся коммутативными и ассоциативными, такие как произведение целых чисел или объединение мультимножеств.

Оказывается, не существует общего метода вычисления сумм с использованием волнового алгоритма, но в некоторых случаях вычисление суммы возможно. Например, в случае алгоритма обхода, или когда процессы имеют идентификаторы, или когда алгоритм порождает остовное дерево, которое может быть использовано для вычисления.

Невозможность существования общего метода. Невозможно дать общий метод вычисления сумм с использованием произвольного волнового алгоритма, подобного методу, использованному в Теореме 6.12 для вычисления инфимумов. Это может быть показано следующим образом. Существует волновой алгоритм для класса сетей, включающего все неориентированные анонимные (anonymous) сети диаметра два, а именно, фазовый алгоритм (с параметром D=2). Не существует алгоритма, который может вычислить сумму по всем входам, и который является правильным для всех неориентированных анонимных (anonymous) сетей диаметра два. Этот класс сетей включает две сети, изображенные на Рис.6.21. Если предположить, что каждый процесс имеет вход 1, ответ будет 6 для левой сети и 8 - для правой. Воспользовавшись технологией, представленной в Главе 9, можно показать, что любой алгоритм даст для обеих сетей один и тот же результат, следовательно, будет работать неправильно. Детальное доказательство оставлено читателю в Упражнении 9.7.

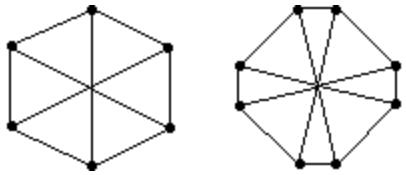


Рис.6.21 Две сети диаметра два и степени три.

Вычисление сумм с помощью алгоритма обхода. Если A - алгоритм обхода, сумма по всем входам может быть вычислена следующим образом. Процесс p содержит переменную j_p, инициализированную значением входа p. Маркер содержит дополнительное поле s. Всякий раз, когда p передает маркер, p выполняет следующее:

$$s := s + j_p \; ; \; j_p := 0$$

и затем можно показать, что в любое время для каждого ранее пройденного процесса p $j_p = 0$ и s равно сумме входов всех пройденных процессов. Следовательно, когда алгоритм завершается, s равно сумме по всем входам.

Вычисление суммы с использованием остовного дерева. Некоторые волновые алгоритмы предоставляют для каждого события принятия решения d_p в процессе p остовное дерево с корнем в p, по которому сообщения передаются по направлению к p. Фактически, каждое вычисление любого волнового алго-

ритма содержит такие остовные деревья. Однако, может возникнуть ситуация, когда процесс q посылает несколько сообщений и не знает, какие из его исходящих ребер принадлежат к такому дереву. Если процессы знают, какое исходящее ребро является их родителем в таком дереве, дерево можно использовать для вычисления сумм. Каждый процесс посылает своему родителю в дереве сумму всех входов его поддерева.

Этот принцип может быть применен для древовидного алгоритма, эхо-алгоритма и фазового алгоритма для клик. Древовидный алгоритм легко может быть изменен так, чтобы включать сумму входов T_{pq} в сообщение, посылаемое от p к q. Процесс, который принимает решение, вычисляет окончательный результат, складывая величины из двух сообщений, которые встречаются на одном ребре. Фазовый алгоритм изменяется так, чтобы в каждом сообщении от q к p пересылался вход q. Процесс p складывает все полученные величины и свой собственный вход, и результат является правильным ответом, когда p принимает решение. В эхо-алгоритме входы могут суммироваться с использованием остовного дерева T, построенного явным образом во время вычисления; см. Упражнение 6.15.

Вычисление суммы с использованием идентификации. Предположим, что каждый процесс имеет уникальный идентификатор. Сумма по всем входам может быть вычислена следующим образом. Каждый процесс помечает свой вход идентификатором, формируя пару (p, j_p); заметим, что никакие два процесса не формируют одинаковые пары. Алгоритм гарантирует, что, когда процесс принимает решение, он знает каждый отдельный вход; $S = \{(p, j_p): p \in P\}$ - объединение по всем p множеств $S_p = \{(p, j_p)\}$, которое может быть вычислено за одну волну. Требуемый результат вычисляется с помощью локальных операций на этом множестве.

Это решение требует доступности уникальных идентификаторов для каждого процесса, что значительно увеличивает битовую сложность. Каждое сообщение волнового алгоритма включает в себя подмножество S, которое занимает N*w бит, если для представления идентификатора и входа требуется w бит; см. Упражнение 6.16.

6.5.3 Альтернативные определения временной сложности

Временную сложность распределенного алгоритма можно определить несколькими способами. В этой книге при рассмотрении временной сложности всегда имеется в виду Определение 6.31, но здесь обсуждаются другие возможные определения.

Определение, основанное на более строгих временных предположениях. Время, потребляемое распределенными вычислениями, можно оценить, используя более строгие временные предположения в системе.

Определение 6.37 *Единичная сложность алгоритма (one-time complexity) - это максимальное время вычисления алгоритма при следующих предположениях.*

O1. *Процесс может выполнить любое конечное количество событий за нулевое время.*

O2. *Промежуток времени между отправлением и получением сообщения - ровно одна единица времени.*

Сравним это определение с Определением 6.31 и заметим, что предположение O1 совпадает с T1. Т.к. время передачи сообщения, принятое в T2, меньше или равно времени, принятому в O2, можно подумать, что единичная сложность всегда больше или равна временной сложности. Далее можно подумать, что каждое вычисление при предположении T2 выполняется не дольше, чем при O2, и, следовательно, вычисление с максимальным временем также займет при T2 не больше времени, чем при O2. Упущение этого аргумента в том, что отклонения во времени передачи сообщения, допустимые при T2, порождают большой класс возможных вычислений, включая вычисления с плохим временем. Это иллюстрируется ниже для эхо-алгоритма.

Фактически, верно обратное: временная сложность алгоритма больше или равна единичной сложности этого алгоритма. Любое вычисление, допустимое при предположениях O1 и O2, также допустимо при T1 и T2 и занимает при этом такое же количество времени. Следовательно, наихудшее поведение алгоритма при O1 и O2 включено в Определение 6.31 и является нижней границей временной сложности.

Теорема 6.38 *Единичная сложность эхо-алгоритма равна* O(D). *Временная сложность эхо-алгоритма равна* Θ(N), *даже в сетях с диаметром* 1.

Доказательство. Для анализа единичной сложности сделаем предположения O1 и O2. Процесс на расстоянии d переходов от инициатора получает первое сообщение <**tok**> *ровно через* d единиц времени после начала вычисления и имеет глубину d в возникающем в результате дереве T. (Это можно доказать индукцией по d.) Пусть D_T - глубина T; тогда $D_T \le D$ и процесс глубины d в T посылает сообщение <**tok**> своему родителю не позднее $(2D_T + 1) - d$ единиц времени после начала вычисления. (Это можно показать обратной индукцией по d.) Отсюда следует, что инициатор принимает решение не позднее $2D_T + 1$ единиц времени после начала вычисления.

Для анализа временной сложности сделаем предположения T1 и T2. Процесс на расстоянии d переходов от инициатора получает первое сообщение <**tok**> *не позднее* d единиц времени после начала вычисления. (Это можно доказать индукцией по d.) Предположим, что остовное дерево построено через F единиц времени после начала вычисления, тогда $F \le D$. В этом случае глубина остовного дерева D_T необязательно ограничена диаметром (как будет показано в вычислении ниже), но т.к. всего N процессов, глубина ограничена N-1. Процесс глубины d в T посылает сообщение <**tok**> своему родителю не позднее $(F+1)+(D_T-d)$ единиц времени после начала вычисления. (Это можно показать обратной индукцией по d.) Отсюда следует, что инициатор принимает решение не позднее $(F+1)+D_T$ единиц времени после начала вычисления, т.е. O(N).

Чтобы показать, что Ω(N) - нижняя граница временной сложности, построим на клике из N процессов вычисление, которое затрачивает время N. Зафиксируем в клике остовное дерево глубины N-1 (на самом деле, линейную цепочку вершин). Предположим, что все сообщения <**tok**>, посланные вниз по ребрам дерева, будут получены спустя 1/N единиц времени после их отправления, а сообщения <**tok**> через листовые ребра будут получены спустя одну еди-

ницу времени. Эти задержки допустимы, согласно предположению T2, и в этом вычислении дерево полностью формируется в течение одной единицы времени, но имеет глубину N-1. Допустим теперь, что все сообщения, посылаемые вверх по ребрам дерева также испытывают задержку в одну единицу времени; в этом случае решение принимается ровно через N единиц времени с начала вычисления.

Можно спорить о том, какое из двух определений предпочтительнее при обсуждении сложности распределенного алгоритма. Недостаток единичной сложности в том, что некоторые вычисления не учитываются, хотя они и допускаются алгоритмом. Среди игнорируемых вычислений могут быть такие, которые потребляют очень много времени. Предположения в Определении 6.31 не исключают ни одного вычисления; определение только устанавливает меру времени для каждого вычисления. Недостаток временной сложности в том, что результат определен вычислениями (как в доказательстве Теоремы 6.38), что хотя и возможно, но считается крайне маловероятным. Действительно, в этом вычислении одно сообщение «обгоняется» цепочкой из N-1 последовательно передаваемого сообщения.

В качестве компромисса между двумя определениями можно рассмотреть α-*временную сложность*, которая определяется согласно предположению, что задержка каждого сообщения - величина между α и 1 (α - константа ≤ 1). К сожалению, этот компромисс обладает недостатками обоих определений. Читатель может попытаться показать, что α-временная сложность эхо-алгоритма равна $O(\min(N,D/\alpha))$.

Наиболее точная оценка временной сложности получается, когда можно задать распределение вероятностей задержек сообщений, откуда может быть вычислено ожидаемое время вычисления алгоритма. У этого варианта есть два основных недостатка. Во-первых, анализ алгоритма слишком зависит от системы, т.к. в каждой распределенной системе распределение задержек сообщений различно. Во-вторых, в большинстве случаев анализ слишком сложен для выполнения.

Определение, основанное на цепочках сообщений. Затраты времени на распределенное вычисление могут быть определены с использованием структурных свойств вычисления, а не идеализированных временных предположений. Пусть C - вычисление.

Определение 6.39 *Цепочка сообщений в* C *- это последовательность сообщений* $m_1, m_2, ..., m_k$ *такая, что для любого* i $(0 \leq i \leq k)$ *получение* m_i *каузально предшествует отправлению* m_{i+1}.

Цепочечная сложность распределенного алгоритма (chain-time complexity) - это длина самой длинной цепочки сообщений во всех вычислениях алгоритма.

Это определение, как и Определение 6.31, рассматривает всевозможные выполнения алгоритма для определения его временной сложности, но определяет другую меру сложности для вычислений. Рассмотрим ситуацию (происходящую в вычислении, определенном в доказательстве теоремы 6.38), когда одно сообщение «обгоняется» цепочкой из k сообщений. Временная сложность этого (под)вычисления равна 1, в то время, как цепочечная сложность того же самого

(под)вычисления равна k. В системах, где гарантируется существование верхней границы задержек сообщений (как предполагается в определении временной сложности), временная сложность является правильным выбором. В системах, где большинство сообщений доставляется со «средней» задержкой, но небольшая часть сообщений может испытывать гораздо большую задержку, лучше выбрать цепочечную сложность.

Упражнения к Главе 6

Раздел 6.1

Упражнение 6.1 *Приведите пример* PIF-*алгоритма для систем с* синхронной *передачей сообщений, который не позволяет вычислять функцию инфимума (см. Теоремы 6.7 и 6.12). Пример может подходить только для конкретной топологии.*

Упражнение 6.2 *В частичном порядке* $(X, \leq)$ *элемент* b *называется дном, если для всех* $c \in X$, $b \leq c$.

В доказательстве Теоремы 6.11 используется то, что частичный порядок $(X,\leq)$ *не содержит дна. Где именно*?

Можете ли вы привести алгоритм, который вычисляет функцию инфимума в частичном порядке с дном и не является волновым алгоритмом?

Упражнение 6.3 *Приведите два частичных порядка на натуральных числах, для которых функция инфимума является (1) наибольшим общим делителем, и (2) наименьшим общим кратным (the least common ancestor).*

Приведите частичные порядки на наборах подмножеств области U, *для которых функция инфимума является (1) пересечением множеств, и (2) объединением множеств.*

Упражнение 6.4 *Докажите теорему об инфимуме (Теорему 6.13).*

Раздел 6.2

Упражнение 6.5 *Покажите, что в каждом вычислении древовидного алгоритма (Алгоритм 6.3) решение принимают ровно два процесса.*

Упражнение 6.6 *Используя эхо-алгоритм (Алгоритм 6.5), составьте алгоритм, который вычисляет префиксную схему маркировки (см. Подраздел 4.4.3) для произвольной сети с использованием* $2|E|$ *сообщений и* $O(N)$ *единиц времени.*

Можете ли вы привести алгоритм, вычисляющий схему маркировки за время $O(D)$? (D - *диаметр сети.*)

Упражнение 6.7 *Покажите, что соотношение в Лемме 6.19 выполняется, если сообщение потерялось в канале* pq, *но не выполняется, если сообщения могут дублироваться. Какой шаг доказательства не действует, если сообщения могут дублироваться*?

Упражнение 6.8 *Примените построение в Теореме 6.12 к фазовому алгоритму так, чтобы получить алгоритм, вычисляющий максимум по целочисленным входам всех процессов.*

Каковы сложность сообщений, временная и битовая сложность вашего алгоритма?

Упражнение 6.9 *Предположим, вы хотите использовать волновой алгоритм в сети, где может произойти* дублирование *сообщений.*

(1) *Какие изменения должны быть сделаны в эхо-алгоритме*?

(2) *Какие изменения должны быть сделаны в алгоритме Финна*?

Раздел 6.3

Упражнение 6.10 *Полный двудольный граф - это граф* $G = (V,E)$, *где* $V = V_1 \cup V_2$ *при* $V_1 \cap V_2 = \varnothing$ *и* $E = V_1 \times V_2$.

*Приведите алгоритм 2*x*-обхода для полных двудольных сетей.*

Упражнение 6.11 *Докажите или опровергните: Обход гиперкуба без чувства направления требует* $\Theta(N \log N)$ *сообщений.*

Раздел 6.4

Упражнение 6.12 *Приведите пример вычисления алгоритма Тарри, в котором в результате получается не* DFS*-дерево.*

Упражнение 6.13 *Составьте алгоритм, который вычисляет интервальные схемы маркировки поиска в глубину (см. Подраздел 4.4.2) для произвольных связных сетей.*

Может ли это быть сделано за $O(N)$ *единиц времени*? *Может ли это быть сделано с использованием* $O(N)$ *сообщений*?

Упражнение 6.14 *Предположим, что алгоритм поиска в глубину со знанием соседей используется в системе, где каждый процесс знает не только идентификаторы своих соседей, но и множество идентификаторов всех процессов* (P). *Покажите, что в этом случае достаточно сообщений, состоящих из* N *бит.*

Раздел 6.5

Упражнение 6.15 *Адаптируйте эхо-алгоритм (Алгоритм 6.5) для вычисления суммы по входам всех процессов.*

Упражнение 6.16 *Предположим, что процессы в сетях, изображенных на Рис.6.21, имеют уникальные идентификаторы, и каждый процесс имеет целочисленный вход. Смоделируйте на обеих сетях вычисление фазового алгоритма, вычисляя множество* $S = \{(p, j_p): p \in P\}$ *и сумму по входам.*

Упражнение 6.17 *Какова цепочечная сложность фазового алгоритма для клик (Алгоритм 6.8)* ?

7 Алгоритмы выбора

В этой главе будут обсуждаться проблемы *выбора*, также называемого *нахождением лидера*. Задача выбора впервые была изложена ЛеЛанном [LeLann; LeL77], который предложил и первое решение; см. Подраздел 7.2.1.

Задача начинается в конфигурации, где все процессы находятся в одинаковом состоянии, и приходит в конфигурацию, где ровно один процесс находится в состоянии *лидера* (leader), а все остальные - в состоянии *проигравших* (lost).

Выбор среди процессов нужно проводить, если должен быть выполнен централизованный алгоритм и не существует заранее известного кандидата на роль инициатора алгоритма. Например, в случае процедуры инициализации системы, которая должна быть выполнена в начале или после сбоя системы. Т.к. множество активных процессов может быть неизвестно заранее, невозможно назначить один процесс раз и навсегда на роль лидера.

Существует большое количество результатов о задаче выбора (как алгоритмы, так и более общие теоремы). Результаты для включения в эту главу выбирались по следующим критериям.

(1) Синхронные системы, анонимные процессы, и отказоустойчивые алгоритмы обсуждаются в других главах. В этой главе всегда предполагается, что процессы и каналы надежны, система полностью асинхронна, и процессы различаются уникальными идентификаторами.

(2) Мы сосредоточим внимание на результатах, касающихся сложности сообщений. Алгоритмы с улучшенной временной сложностью или результаты, предполагающие компромисс между временной сложностью и сложностью сообщений, не обсуждаются.

(3) Мы будем уделять внимание порядку величины сложности сообщений, и не будем рассматривать результаты, вносящие в сложность только постоянный множитель.

(4) Т.к. результаты Кораха и др. (Раздел 7.4) подразумевают существование O(N logN)-алгоритмов для нескольких классов сетей, алгоритм для клики с этой сложностью не будет рассматриваться отдельно.

7.1 Введение

Задача выбора требует, чтобы из конфигурации, где все процессы находятся в одинаковом состоянии, система пришла в конфигурацию, где ровно один процесс находится в особом состоянии *лидер* (leader), а все остальные процессы - в состоянии *проигравших* (lost). Процесс, находящийся в состоянии *лидер* в конце вычисления, называется *лидером* и говорят, что он выбран алгоритмом.

Определение 7.1 *Алгоритм выбора - это алгоритм, удовлетворяющий следующим требованиям.*

(1) Каждый процесс имеет один и тот же локальный алгоритм.

(2) Алгоритм является децентрализованным, т.е. вычисление может быть начато произвольным непустым подмножеством процессов.

(3) *Алгоритм достигает заключительной конфигурации в каждом вычислении, и в каждой достижимой заключительной конфигурации существует ровно один процесс в состоянии лидера, а все остальные процессы - в состоянии проигравших.*

Иногда последнее требование ослабляется и требуется только, чтобы ровно один процесс находился в состоянии *лидера*. В этом случае выбранный процесс знает, что он победил, но проигравшие (еще) не знают, что они проиграли. Если дан алгоритм, удовлетворяющий этим ослабленным действиям, то его можно легко расширить, добавив инициируемую лидером рассылку сообщений всем процессам, при которой все процессы информируются о результатах выбора. В некоторых алгоритмах этой главы это дополнительное оповещение опущено.

Во всех алгоритмах этой главы процесс p имеет переменную $state_p$ с возможными значениями *leader* (лидер) и *lost* (проигравший). Иногда мы будем предполагать, что $state_p$ имеет значение *sleep* (спящий), когда p еще не выполнил ни одного шага алгоритма, и значение *cand* (кандидат), если p вступил в вычисление, но еще не знает, победил он или проиграл. Некоторые алгоритмы используют дополнительные состояния, такие как *active*, *passive* и др., которые будут указаны в самом алгоритме.

7.1.1 Предположения, используемые в этой главе

Рассмотрим предположения, при которых задача выбора изучалась в этой главе.

(1) *Система полностью асинхронна.* Предполагается, что процессам недоступны общие часы, и что время передачи сообщения может быть произвольно долгим или коротким.

Оказывается, что предположение о синхронной передаче сообщений (т.е. когда отправление и получение сообщения считается единой передачей) незначительно влияет на результаты, полученные для задачи выбора. Читатель может сам убедиться, что алгоритмы, данные в этой главе, могут применяться в системах с синхронной передачей сообщений, и что полученные нижние границы также применимы в этом случае.

Предположение о существовании глобального времени, также как и предположение о том, что процессам доступно реальное время и что задержка сообщений ограничена, имеют важное влияние на решения задачи выбора.

(2) *Каждый процесс идентифицируется уникальным именем, своим* идентификатором, *который известен процессу изначально.* Для простоты предполагается, что идентификатор процесса p - просто p. Идентификаторы извлекаются из совершенно упорядоченного множества P, т.е. для идентификаторов определено отношение ≤. Количество бит, представляющих идентификатор, равно w.

Важность уникальных идентификаторов в задаче выбора состоит в том, что они могут использоваться не только для адресации сообщений, но и

для нарушения симметрии между процессами. При разработке алгоритма выбора можно, например, потребовать, что процесс с наименьшим (или наоборот, с наибольшим) идентификатором должен победить. Тогда задача состоит в поиске наименьшего идентификатора с помощью децентрализованного алгоритма. В этом случае задачу выбора называют задачей *поиска экстремума.*

Хотя некоторые из алгоритмов, обсуждаемых в этой главе, изначально были изложены для нахождения наибольшего процесса, мы излагаем большинство алгоритмов для выбора наименьшего процесса. Во всех случаях алгоритм для выбора наибольшего процесса можно получить, изменив порядок сравнения идентификаторов.

(3) *Некоторые результаты этой главы относятся к алгоритмам сравнения.* Алгоритмы сравнения - это алгоритмы, которые используют сравнение как единственную операцию над идентификаторами. Как мы увидим, все алгоритмы, представленные в этой главе, являются алгоритмами сравнения. Всякий раз, когда дается оценка нижней границы, мы явно отмечаем, касается ли она алгоритмов сравнения.

Было показано (например, Бодлендером [Bodlaender, Bod91b] для случая кольцевых сетей), что в асинхронных сетях произвольные алгоритмы не достигают лучшей сложности, чем алгоритмы сравнения. Это не так в случае синхронных систем, как будет показано в Главе 11; в этих системах произвольные алгоритмы могут достигать лучшей сложности, чем алгоритмы сравнения.

(4) *Каждое сообщение может содержать* O(w) *бит.* Каждое сообщение может содержать не более постоянного числа идентификаторов процессов. Это предположение сделано для того, чтобы позволить справедливое сравнение сложности сообщений различных алгоритмов.

7.1.2 Выбор и волны

Уже было замечено, что идентификаторы процессов могут использоваться для нарушения симметрии между процессами. Можно разработать алгоритм выбора так, чтобы выбирался процесс с наименьшим идентификатором. Согласно результатам в Подразделе 6.1.5, наименьший идентификатор может быть вычислен за одну волну. Это означает, что выбор можно провести, выполняя волну, в которой вычисляется наименьший идентификатор, после чего процесс с этим идентификатором становится лидером. Т.к. алгоритм выбора должен быть децентрализованным, этот принцип может быть применен только к децентрализованным волновым алгоритмам (см. Таблицу 6.19).

Выбор с помощью древовидного алгоритма. Если топология сети - дерево или доступно остовное дерево сети, выбор можно провести с помощью древовидного алгоритма (Подраздел 6.2.2). В древовидном алгоритме требуется, чтобы хотя бы все листья были инициаторами алгоритма. Чтобы получить развитие алгоритма в случае, когда некоторые процессы также являются инициаторами, добавляется фаза *wake-up*. Процессы, которые хотят начать выбор, рассылают сообщение <**wakeup**> всем процессам. Логическая переменная *ws* исполь-

зуется, чтобы каждый процесс послал сообщения <**wakeup**> не более одного раза, а переменная *wr* используется для подсчета количества сообщений <**wakeup**>, полученных процессом. Когда процесс получит сообщение <**wakeup**> через каждый канал, он начинает выполнять Алгоритм 6.3, который расширен (как в Теореме 6.12) таким образом, чтобы вычислять наименьший идентификатор и чтобы каждый процесс принимал решение. Когда процесс принимает решение, он знает идентификатор лидера; если этот идентификатор совпадает с идентификатором процесса, он становится *лидером*, а если нет - *проигравшим*; см. Алгоритм 7.1.

---

var ws_p : boolean **init** *false* ;
wr_p : integer **init** 0 ;
$rec_p[q]$: boolean для всех $q \in Neigh_p$ **init** *false* ;
v_p : P **init** p ;
$state_p$: (*sleep*, *leader*, *lost*) **init** *sleep* ;

begin if p - инициатор **then**
begin $ws_p := true$;
forall $q \in Neigh_p$ **do** send <**wakeup**> to q
end ;
while $wr_p < \# Neigh_p$ **do**
begin receive <**wakeup**> ; $wr_p := wr_p + 1$;
if not ws_p **then**
begin $ws_p := true$;
forall $q \in Neigh_p$ **do** send <**wakeup**> to q
end
end ;
(* Начало древовидного алгоритма *)
while $\# \{q : \neg rec_p[q]\} > 1$ **do**
begin receive <**tok**,r> from q ; $rec_p[q] := true$;
$v_p := \min(v_p, r)$
end ;
send <**tok**,v_p> to q_0 with $\neg rec_p[q_0]$;
receive <**tok**,r> from q_0 ;
$v_p := \min(v_p, r)$; (* *decide* с ответом v_p *)
if $v_p = p$ **then** $state_p := leader$ **else** $state_p := lost$;
forall $q \in Neigh_p, q \neq q_0$ **do** send <**tok**,v_p> to q
end

Алгоритм 7.1 Алгоритм выборов для деревьев.

Теорема 7.2 *Алгоритм 7.1 решает задачу выбора на деревьях, используя* O(N) *сообщений и* O(D) *единиц времени.*

Доказательство. Когда хотя бы один процесс инициирует выполнение алгоритма, все процессы посылают сообщения <**wakeup**> всем своим соседям, и каждый процесс начинает выполнение древовидного алгоритма после получения сообщения <**wakeup**> от каждого соседа. Все процессы завершают древовидный алгоритм с одним и тем же значением v, а именно, наименьшим иден-

тификатором процесса. Единственный процесс с этим идентификатором закончит выполнение в состоянии *лидер*, а все остальные процессы - в состоянии *проигравший.*

Через каждый канал пересылается по два сообщения <**wakeup**> и по два сообщения <**tok**,r>, откуда сложность сообщений равна 4N-4. В течение D единиц времени после того, как первый процесс начал алгоритм, каждый процесс послал сообщения <**wakeup**>, следовательно, в течение D+1 единиц времени каждый процесс начал волну. Легко заметить, что первое решение принимается не позднее, чем через D единиц времени после начала волны, а последнее решение принимается не позднее D единиц времени после первого, откуда полное время равно 3D+1. Более тщательный анализ показывает, что алгоритм всегда завершается за 2D единиц времени, но доказательство этого оставлено читателю; см. Упражнение 7.2.

Если порядок сообщений в канале может быть изменен (т.е. канал - не FIFO), процесс может получить сообщение <**tok**,r> от соседа *прежде чем* он получил сообщение <**wakeup**> от этого соседа. В этом случае сообщение <**tok**,r> может быть временно сохранено или обработано как сообщения <**tok**,r>, прибывающие позднее.

Количество сообщений может быть уменьшено с помощью двух модификаций. Во-первых, можно устроить так, чтобы не-инициатор не посылал сообщение <**wakeup**> процессу, от которого он получил первое сообщение <**wakeup**>. Во-вторых, сообщение <**wakeup**>, посылаемое листом, может быть объединено с сообщением <**tok**,r>, посылаемым этим листом. С этими изменениями количество сообщений, требуемое алгоритмом, уменьшается до 3N-4+k, где k - количество нелистовых стартеров [Tel91b, с.139].

Выбор с помощью фазового алгоритма. Фазовый алгоритм можно использовать для выбора, позволив ему вычислять наименьший идентификатор за одну волну, как в Теореме 6.12.

Теорема 7.3 *С помощью фазового алгоритма (Алгоритм 6.7) можно провести выбор в произвольных сетях, используя* O(D*|E|) *сообщений и* O(D) *единиц времени.*

Алгоритм Пелега [Peleg; Pel90] основан на фазовом алгоритме; он использует O(D*|E|) сообщений и O(D) времени, но не требует знания D, т.к. включает в себя вычисление диаметра.

Выбор с помощью алгоритма Финна. Алгоритм Финна (Алгоритм 6.9) не требует, чтобы диаметр сети был известен заранее. Длина O(N*|E|) сообщений, используемых в алгоритме Финна, гораздо больше, чем допускаемая предположениями в этой главе. Следовательно, каждое сообщение в алгоритме Финна должно считаться за O(N) сообщений, откуда сложность сообщений составляет $O(N^2|E|)$.

7.2 Кольцевые сети

В этом разделе рассматриваются некоторые алгоритмы выбора для *однонаправленных* колец. Задача выбора в контексте кольцевых сетей была впервые изложена ЛеЛанном [LeLann; LeL77], который также дал решение со сложностью сообщений $O(N^2)$. Это решение было улучшено Чангом (Chang) и Робертсом (Roberts) [CR79], которые привели алгоритм с наихудшей сложностью $O(N^2)$, но со средней сложностью только O(N logN). Решения ЛеЛанна и Чанга-Робертса обсуждаются в Подразделе 7.2.1. Вопрос о существовании алгоритма с наихудшей сложностью O(N logN) оставался открытым до 1980 г., когда такой алгоритм был приведен Hirschberg и Sinclair [HS80]. В отличие от более ранних решений, в решении Hirschberg-Sinclair требуется, чтобы каналы были двунаправленными. Предполагалось, что нижняя граница для однонаправленных колец равна $\Omega(N^2)$, но Petersen [Pet82] и Dolev, Klawe и Rodeh [DKR82] независимо друг от друга предложили решение, составляющее O(N log N) для однонаправленного кольца. Это решение рассматривается в Подразделе 7.2.2.

Алгоритмы были дополнены соответствующими нижними границами примерно в то же время. Нижняя граница для наихудшего случая для двунаправленных колец, равная $\approx 0.34N \log N$ сообщений, была доказана Бодлендером [Bodlaender; Bod88]. Pachl, Korach и Rotem [PKR84] доказали нижние границы в $\Omega(N \log N)$ для средней сложности, как для двунаправленных так и для однонаправленных колец. Их результаты по нижним границам будут рассмотрены в Подразделе 7.2.3.

7.2.1 Алгоритмы ЛеЛанна и Чанга-Робертса

В алгоритме ЛеЛанна [LeL77] каждый инициатор вычисляет список идентификаторов всех инициаторов, после чего выбирается инициатор с наименьшим идентификатором. Каждый инициатор посылает маркер, содержащий его идентификатор, по кольцу, и этот маркер передается всеми процессами. Предполагается, что каналы подчиняются дисциплине FIFO, и что инициатор должен сгенерировать свой маркер до того, как он получит маркер другого инициатора. (Когда процесс получает маркер, он после этого не инициирует алгоритм.) Когда инициатор p получает свой собственный маркер, маркеры всех инициаторов прошли через p, и p выбирается лишь в том случае, если p - наименьший среди инициаторов; см. Алгоритм 7.2.

```
var  List_p     : set of P     init  {p} ;
     state_p ;

begin  if  p - инициатор  then
         begin  state_p := cand ;  send <tok,p>  to Next_p ;  receive <tok,q> ;
                while  q ≠ p  do
                       begin   List_p := List_p ∪ {q} ;
                               send <tok,q>  to Next_p ;  receive <tok,q> ;
                       end ;
                if  p = min (List_p)  then state_p := leader
```

```
                                    else state_p := lost
          end
        else repeat receive <tok,q> ; send <tok,q> to Next_p ;
                    if state_p = sleep then state_p := lost
             until false
end
```

Алгоритм 7.2 Алгоритм выбора ЛеЛанна.

Теорема 7.4 *Алгоритм ЛеЛанна (Алгоритм 7.2) решает задачу выбора для колец, используя* $O(N^2)$ *сообщений и* $O(N)$ *единиц времени.*

Доказательство. Так как порядок маркеров в кольце сохраняется (из предположения о каналах FIFO), и инициатор q отправляет <**tok**,q> *до того как* получит <**tok**,p>, то инициатор p получает <**tok**,q> прежде, чем вернется <**tok**,p>. Отсюда следует, что каждый инициатор p заканчивается со списком $List_p$, совпадающим с множеством всех инициаторов, и единственным выбираемым процессом становится инициатор с наименьшим идентификатором. Всего получается не больше N маркеров и каждый делает N шагов, что приводит к сложности сообщений в $O(N^2)$. Не позднее чем через N-1 единицу времени после того, как первый инициатор отправил свой маркер, это сделали все инициаторы. Каждый инициатор получает свой маркер обратно не позднее, чем через N единиц времени с момента генерации этого маркера. Отсюда следует, что алгоритм завершается в течение 2N-1 единиц времени.

Все не-инициаторы приходят в состояние *проигравший*, но навсегда остаются в ожидании сообщений <**tok**,r>. Ожидание может быть прервано, если лидер посылает по кольцу специальный маркер, чтобы объявить об окончании выбора.

Алгоритм Чанга-Робертса [CR79] улучшает алгоритм ЛеЛанна, устраняя из кольца маркеры тех процессов, для которых очевидно, что они проиграют выборы. Т.е. инициатор p удаляет из кольца маркер <**tok**,q>, если q > p. Инициатор p становится *проигравшим*, когда получает маркер с идентификатором q < p, или лидером, когда он получает маркер с идентификатором p; см. Алгоритм 7.3.

```
var state_p ;

begin if p - инициатор then
        begin state_p := cand ; send <tok,p> to Next_p ;
              repeat receive <tok,q> ;
                     if q = p then state_p := leader
                                else if q < p then
                                       begin if state_p = cand then state_p := lost ;
                                             send <tok,q> to Next_p
                     end
              until state_p = leader
        end
      else repeat receive <tok,q> ; send <tok,q> to Next_p ;
                  if state_p = sleep then state_p := lost
```

until *false*

end

(* Только лидер завершает выполнение программы. Он передает сообщение всем процессам, чтобы сообщить им идентификатор лидера и завершить их *)

Алгоритм 7.3 Алгоритм выбора Чанга-Робертса.

Теорема 7.5 *Алгоритм Чанга-Робертса (Алгоритм 7.3) решает задачу выбора для колец, используя* $\Theta(N^2)$ *сообщений в наихудшем случае и* $O(N)$ *единиц времени.*

Доказательство. Пусть p_0 - инициатор с наименьшим идентификатором. Все процессы являются либо не-инициаторами, либо инициаторами с идентификаторами большими p_0, поэтому все процессы передают дальше маркер <**tok**,p_0>, отправленный p_0. Следовательно, p_0 получает свой маркер обратно и становится выбранным.

Не-инициаторы не могут быть выбраны, т.к. все они приходят в состояние *проигравший* самое позднее, когда через них передается маркер p_0. Инициатор p с $p > p_0$ не может быть выбран; p_0 не передаст дальше маркер <**tok**,p>, поэтому p никогда не получит свой собственный маркер. Такой инициатор p приходит в состояние *проигравший* самое позднее, когда через него передается маркер <**tok**,p_0>. Таким образом доказано, что алгоритм решает задачу выбора.

Рис.7.4 Наихудший случай для алгоритма Чанга-Робертса.

Всего используется не более N различных маркеров и каждый маркер делает не более N переходов, что подтверждает границу сложности сообщений $O(N^2)$. Чтобы показать, что в самом деле можно использовать $\Omega(N^2)$ сообщений, рассмотрим начальную конфигурацию, где все идентификаторы расположены в возрастающем порядке вдоль кольца (см. Рис. 7.4) и каждый процесс является инициатором. Маркер каждого процесса удаляется из кольца процессом 0, таким образом маркер процесса i совершает N-i переходов, откуда следует, что количество пересылок сообщений равно $\sum_{i=0}^{N-1}(N-i)=\frac{1}{2}N(N+1)$.

Алгоритм Чанга-Робертса не улучшает алгоритм ЛеЛанна в отношении временной сложности или наихудшего случая сложности сообщений. Улучшение касается только *среднего* случая, где усреднение ведется по всевозможным расположениям идентификаторов вдоль кольца.

Теорема 7.6 *Алгоритм Чанга-Робертса в среднем случае, когда все процессы являются инициаторами, требует только* O(N logN) *пересылок сообщений.*

Доказательство. (Это доказательство основано на предложении Friedemann Mattern.)

Предположив, что все процессы являются инициаторами, вычислим среднее количество пересылок маркера по всем круговым расположениям N различных идентификаторов. Рассмотрим фиксированное множество из N идентификаторов, и пусть s будет наименьшим идентификатором. Существует (N-1)! различных круговых расположений идентификаторов; в данном круговом расположении пусть p_i - идентификатор, находящийся за i шагов до s; см. Рис. 7.5.

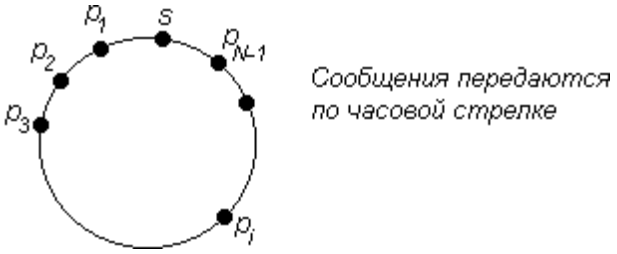


Рис.7.5 Расположение идентификаторов на кольце.

Чтобы вычислить суммарное количество пересылок маркера по всем расположениям, вычислим сначала суммарное количество пересылок маркера <**tok**,p_i> по всем расположениям, а потом просуммируем по i. Маркер <**tok**,s> при любом расположении передается N раз, следовательно, он пересылается всего N(N-1)! раз. Маркер <**tok**,p_i> передается не более i раз, так как он будет удален из кольца, если достигнет s. Пусть $A_{i,k}$ - количество циклических расположений, при которых <**tok**,p_i> передается ровно k раз. Тогда суммарное число пересылок <**tok**,p_i> равно $\sum_{k=1}^{i}(k \cdot A_{i,k})$.

Маркер <**tok**,p_i> передается ровно i раз, если p_i является наименьшим из идентификаторов от p_1 до p_i, что имеет место в (1/i)*(N-1)! расположениях; итак

$$A_{i,i} = \frac{1}{i}(N-1)!$$

Маркер <**tok**,p_i> передается *не менее* k раз (здесь k ≤ i), если за процессом p_i следует k-1 процесс с идентификаторами, большими p_i. Количество расположений, в которых p_i - наименьший из k идентификаторов p_{i-k+1}, ..., p_i, составляет 1/k часть всех расположений, т.е. (1/k)*(N-1)!. Теперь, для k<i <**tok**,p_i> передается *ровно* k раз, если он передается не менее, но и не более k раз, т.е. ≥∈k раз, но не ≥∈k+1 раз. В результате количество расположений, где это выполняется, равно $\frac{1}{k}(N-1)! - \frac{1}{k+1}(N-1)!$, т.е. $A_{i,k} = \frac{1}{k(k+1)}(N-1)!$ (для k < i).

Общее количество передач <**tok**,p_i> во всех расположениях равно:

$$\sum_{k=1}^{i-1} k \cdot \left(\frac{1}{k(k+1)} (N-1)! \right) + i \cdot \frac{1}{i} (N-1)!,$$

что равняется $(\sum_{k=1}^{i} 1/k) \cdot (N-1)!$. Сумма $(\sum_{k=1}^{i} 1/k)$ известна как i-е *гармоническое число*, обозначаемое $\mathbf{H}_i$. В качестве Упражнения 7.3 оставлено доказательство тождества $\sum_{i=1}^{m} \mathrm{H}_i = (m+1)\mathrm{H}_m - m$.

Далее мы суммируем по i количество передач маркера, чтобы получить общее количество передач (исключая передачи <**tok**,s>) во всех расположениях. Оно равно

$$\sum_{i=1}^{N-1} \left[\mathrm{H}_i \cdot (N-1)! \right] = (N \cdot \mathrm{H}_{N-1} - (N-1)) \cdot (N-1)!.$$

Добавляя N(N-1)! передач маркера для <**tok**,s>, мы получаем общее количество передач, равное

$$(N \cdot \mathrm{H}_{N-1} + 1) \cdot (N-1)! = (N \cdot \mathrm{H}_N) \cdot (N-1)!.$$

Т.к. это число выведено для (N-1)! различных расположений, среднее по всем расположениям, очевидно, равно $N \cdot \mathbf{H}_N$, что составляет $\approx 0.69N \log N$ (см. Упр.7.4).

7.2.2 Алгоритм Petersen / Dolev-Klawe-Rodeh

Алгоритм Чанга-Робертса достигает сложности сообщений O(N logN) в среднем, но не в наихудшем случае. Алгоритм со сложностью O(N logN) в наихудшем случае был дан Франклином [Franklin; Fra82], но этот алгоритм требует, чтобы каналы были двунаправленными. Petersen [Pet82] и Dolev, Klawe, Rodeh [DKR82] независимо разработали очень похожий алгоритм для однонаправленных колец, решающий задачу с использованием только O(N logN) сообщений в наихудшем случае. Алгоритм требует, чтобы каналы подчинялись дисциплине FIFO.

Сначала алгоритм вычисляет наименьший идентификатор и сообщает его каждому процессу, затем процесс с этим идентификатором становится лидером, а все остальные терпят поражение. Алгоритм легче понять, если представить, что он выполняется *идентификаторами*, а не процессами. Изначально каждый идентификатор *активен*, но на каждом круге некоторые идентификаторы становятся *пассивными*, как будет показано позднее. При обходе круга *активный* идентификатор сравнивает себя с двумя соседними *активными* идентификаторами по часовой стрелке и против нее. Если он является локальным минимумом, он остается в круге, иначе он становится *пассивным*. Т.к. все идентификаторы различны, идентификатор рядом с локальным минимумом сам не является локальным минимумом, откуда следует, что не менее половины идентификаторов выбывают из круга при каждом обходе. Следовательно, после не более чем logN кругов остается только один *активный* идентификатор, который и является победителем.

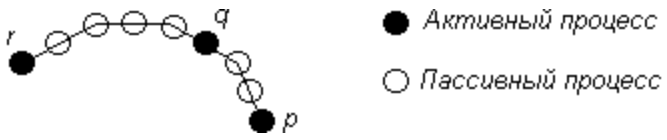


Рис.7.6 Процесс p получает текущие идентификаторы q и r.

Этот принцип может быть непосредственно реализован в двунаправленных сетях, как это сделано в алгоритме Франклина [Fra82]. В ориентированных кольцах сообщения можно посылать только по часовой стрелке, что затрудняет получение соседнего активного идентификатора в этом направлении; см. Рис. 7.6. Идентификатор q нужно сравнить с r и p; идентификатор r можно послать q, но идентификатор p нужно было бы передавать против направления каналов. Чтобы сравнить q и с r, и с p, идентификатор q передается (в направлении кольца) процессу, который имеет идентификатор p, а r передается не только процессу с идентификатором q, но и дальше, процессу с идентификатором p. Если q является единственным *активным* идентификатором в начале обхода круга, первый идентификатор, который q встречает при обходе, равен q (т.е. в этом случае p = q). Когда это происходит, идентификатор q выигрывает выборы.

Алгоритм для процессов в однонаправленном кольце обозначен как Алгоритм 7.7. Процесс p является *активным* в круге, если он в начале круга имеет *активный* идентификатор ci_p. Иначе p является *пассивным* и просто пропускает через себя все получаемые сообщения. *Активный* процесс посылает свой текущий идентификатор следующему *активному* процессу, и получает текущий идентификатор предыдущего *активного* процесса, используя сообщения <**one**,•>. Полученный идентификатор сохраняется (в переменной acn_p), и если он не выбывает из круга, он будет текущим идентификатором p в следующем круге. Чтобы определить, остается ли идентификатор acn_p в круге, его сравнивают с ci_p и активным идентификатором, полученным в сообщении <**two**,•>. Процесс p посылает сообщение <**two**,acn_p>, чтобы следующий активный процесс мог провести такое же сравнение. Исключение возникает, когда $acn_p = ci_p$; в этом случае остался один активный идентификатор и об этом сообщается всем процессам в сообщении <**smal**,acn_p>.

```
var ci_p      : P    init  p ;      (* Текущий идентификатор p *)
    acn_p     : P    init  udef ;  (* Идентификатор соседа против часовой стрелки
*)
    win_p     : P    init  udef ;  (* Идентификатор победителя *)
    state_p   : (active, passive, leader, lost)   init  active ;

begin  if  p - инициатор  then state_p := active  else  state_p := passive ;
       while  win_p = udef  do
          begin  if  state_p = active  then
                       begin   send <one,ci_p> ;  receive <one,q> ;  acn_p := q ;
                               if  acn_p = ci_p  then    (* acn_p - минимум *)
                                  begin  send <smal,acn_p> ;  win_p := acn_p ;
                                         receive <smal,q>
                                  end
```

```
                    else  (* acn_p - текущий идентификатор соседа *)
                       begin  send <two,acn_p> ;  receive <two,q> ;
                              if acn_p < ci_p  and  acn_p < q
                                then  ci_p := acn_p
                                else  state_p := passive
                       end
                 end
              else  (* state_p = passive *)
                 begin  receive <one,q> ;  send <one,q> ;
                        receive m ;  send m ;
                        (* m - либо <two,q>, либо <smal,q> *)
                        if  m - <smal,q>  then  win_p := q
                 end
          end ;
       if  p = win_p  then  state_p := leader  else  state_p := lost
end
```

Алгоритм 7.7 Алгоритм Petersen / Dolev-Klawe-Rodeh.

Теорема 7.7 *Алгоритм 7.7 решает задачу выбора для однонаправленных сетей с использованием* O(N logN) *сообщений.*

Доказательство. Будем говорить, что процесс находится на i-м круге, когда он выполняет основной цикл в i-й раз. Обходы круга не синхронизированы глобально; возможно, что в различных частях кольца один процесс на несколько кругов впереди другого. Но, т.к. каждый процесс отправляет и получает в каждом круге ровно по два сообщения и каналы подчиняются дисциплине FIFO, то сообщение всегда будет получено в том же круге, в каком оно было послано. На первом круге все инициаторы *активны* и все имеют различные «текущие идентификаторы».

Утверждение 7.8 *Если круг* i *начинается с* k (k>1) *активными процессами, и все процессы имеют различные* ci, *то в круге остаются не меньше* 1 *и не больше* k/2 *процессов. В конце круга снова все текущие идентификаторы активных процессов различны и включают наименьший идентификатор.*

Доказательство. Путем обмена сообщениями <**one**,q>, которые пропускаются *пассивными* процессами, каждый *активный* процесс получает текущий идентификатор своего *активного* соседа против часовой стрелки, который всегда отличается от его собственного идентификатора. Далее, каждый *активный* процесс продолжает обход круга, передавая сообщения <**two**,q>, благодаря которым каждый активный процесс получает текущий идентификатор своего второго *активного* соседа против часовой стрелки. Теперь все *активные* процессы имеют различные значения acn, откуда следует, что в конце круга все оставшиеся в круге идентификаторы различны. По крайней мере, остается идентификатор, который был наименьшим в начале круга, т.е. остается хотя бы один процесс. Идентификатор рядом с локальным минимумом не является локальным минимумом, откуда следует, что количество оставшихся в круге не превышает k/2.

Из Утверждения 7.8 следует, что существует круг с номером $\leq \lfloor \log N \rfloor + 1$, который начинается ровно с одним активным идентификатором, а именно, с наименьшим среди идентификаторов инициаторов.

Утверждение 7.9 *Если круг начинается ровно с одним активным процессом* p *с текущим идентификатором* ci_p, *то алгоритм завершается после этого круга с* $win_q = ci_p$ *для всех* q.

Доказательство. Сообщение <**one**,ci_p> пропускается всеми процессами и, в конце концов, его получает p. Процесс p обнаруживает, что $acn_p = ci_p$ и посылает по кольцу сообщение <**smal**,acn_p>, вследствие чего все процессы выходят из основного цикла с $win_p = acn_p$.

Алгоритм завершается в каждом процессе и все процессы согласовывают идентификатор лидера (в переменной win_p); этот процесс находится в состоянии *лидер*, а остальные - в состоянии *проигравший*.

Всего происходит не более $\lfloor \log N \rfloor + 1$ обходов круга, в каждом из которых передается ровно 2N сообщений, что доказывает, что сложность сообщений ограничена 2N logN + O(N). Теорема 7.7 доказана.

Dolev и др. удалось улучшить свой алгоритм до 1.5N logN, после чего Petersen получил алгоритм, использующий только 1.44N logN сообщений. Этот алгоритм снова был улучшен Dolev и др. до 1.356N logN. Верхняя граница в 1.356N logN считалась наилучшей для выбора на кольцах более 10 лет, но была улучшена до 1.271N logN Higham и Przytycka [HP93].

7.2.3 Вывод нижней границы

В этом подразделе будет доказана нижняя граница сложности выбора на однонаправленных кольцах. Т.к. выбор можно провести за одно выполнение децентрализованного волнового алгоритма, нижняя граница сложности децентрализованных волновых алгоритмов для колец будет получена как заключение.

Результат получен Pachl, Korach и Rotem [PKR84] при следующих предположениях.

(1) Граница доказывается для алгоритмов, вычисляющих наименьший идентификатор. Если существует лидер, наименьший идентификатор может быть вычислен с помощью N сообщений, а если наименьший идентификатор известен хотя бы одному процессу, процесс с этим идентификатором может быть выбран опять же за N сообщений. Следовательно, сложность задач выбора и вычисления наименьшего идентификатора различаются не более чем на N сообщений.

(2) Кольцо является однонаправленным.

(3) Процессам не известен размер кольца.

(4) Предполагается, что каналы FIFO. Это предположение не ослабляет результат, потому что сложность не-FIFO алгоритмов не лучше сложности FIFO алгоритмов.

(5) Предполагается, что все процессы являются инициаторами. Это предположение не ослабляет результат, потому что оно описывает ситуацию, возможную для каждого децентрализованного алгоритма.

(6) Предполагается, что алгоритмы управляются сообщениями; т.е. после отправления сообщений при инициализации алгоритма, процесс посылает сообщения в дальнейшем только после получения очередного сообщения. Т.к. рассматриваются *асинхронные* системы, общие алгоритмы не достигают лучшей сложности, чем алгоритмы, управляемые сообщениями. Действительно, если A - асинхронный алгоритм, то управляемый сообщениями алгоритм B может быть построен следующим образом. После инициализации и после получения любого сообщения B посылает максимальное количество сообщений, которое можно послать в A, не получая при этом сообщений, и только затем получает следующее сообщение. Алгоритм B не только управляется сообщениями, но кроме того, каждое вычисление B является возможным вычислением A (возможно, при довольно пессимистическом распределении задержек передачи сообщений).

Три последних предположения устраняют недетерминизм системы. При этих предположениях каждое вычисление, начинающееся с данной начальной конфигурации, содержит одно и то же множество событий.

В этом разделе через $s = (s_1, ..., s_N)$, t и т.п. обозначаются последовательности различных идентификаторов процессов. Множество всех таких последовательностей обозначено через D, т.е. $D = \{(s_1, ..., s_k): s_i \in P \text{ и } i \neq j \Rightarrow s_i \neq s_j\}$. Длина последовательности s обозначается через len(s), а конкатенация последовательностей s и t обозначается st. *Циклическим сдвигом* s называется последовательность $s's''$, где $s = s''s'$; она имеет вид $s_i, ..., s_N, s_1, ..., s_{i-1}$. Через CS(s) (cyclic shift - циклический сдвиг) обозначено множество циклических сдвигов s, и естественно $|CS(s)| = len(s)$.

Говорят, что кольцо *помечено* последовательностью $(s_1, ..., s_N)$, если идентификаторы процессов с s_1 по s_N расположены на кольце (размера N) в таком порядке. Кольцо, помеченное s также называют *s*-кольцом. Если t - циклический сдвиг s, то *t*-кольцо совпадает с *s*-кольцом.

С каждым сообщением, посылаемым в алгоритме, свяжем последовательность идентификаторов процессов, называемую *следом* (trace) сообщения. Если сообщение m было послано процессом p до того, как p получил какое-либо сообщение, след m равен (p). Если m было послано процессом p после того, как он получил сообщение со следом $s = (s_1, ..., s_k)$, тогда след m равен $(s_1, ..., s_k, p)$. Сообщение со следом s называется *s*-сообщением. Нижняя граница будет выведена из свойств множества всех следов сообщений, которые могут быть посланы алгоритмом.

Пусть E - подмножество D. Множество E *полно* (exhaustive), если

(1) E префиксно замкнуто, т.е. $tu \in E \Rightarrow t \in E$; и

(2) E циклически покрывает D, т.е. $\forall s \in D: CS(s) \cap E \neq \varnothing$.

Далее будет показано, что множество всех следов алгоритма полно. Для того, чтобы вывести из этого факта нижнюю границу сложности алгоритма, оп-

ределены две меры множества E. Последовательность t является *последовательной* цепочкой идентификаторов в s-кольце, если t - префикс какого-либо $r \in CS(s)$. Обозначим через M(s,E) количество последовательностей в E, которые удовлетворяют этому условию в s-кольце, а через $M_k(s,E)$ - количество таких цепочек длины k;

$M(s,E) = |\{ t \in E : t$ - префикс некоторого $r \in CS(s) \}|$ и

$M_k(s,E) = |\{ t \in E : t$ - префикс некоторого $r \in CS(s)$ и $len(t) = k\}|$.

В дальнейшем, допустим, что A - алгоритм, который вычисляет наименьший идентификатор, а E_A - множество последовательностей s таких, что s-сообщение посылается, когда алгоритм A выполняется на *s*-кольце.

Лемма 7.10 *Если последовательности* t *и* u *содержат подстроку* s *и* s*-сообщение посылается, когда алгоритм A выполняется на t-кольце, то s-сообщение также посылается, когда A выполняется на u-кольце.*

Доказательство. Посылка процессом s_k s-сообщения, где $s = (s_1, ..., s_k)$, каузально зависит только от процессов с s_1 по s_k. Их начальное состояние в *u*-кольце совпадает с состоянием в *t*-кольце (напоминаем, что размер кольца неизвестен), и следовательно совокупность событий, предшествующих посылке сообщения, также выполнима и в *u*-кольце.

Лемма 7.11 E_A - *полное множество.*

Доказательство. Чтобы показать, что E_A циклически замкнуто, заметим, что если A посылает *s*-сообщение при выполнении на *s*-кольце, тогда для любого префикса t последовательности s A сначала посылает *t*-сообщение на *s*-кольце. По Лемме 7.10 A посылает *t*-сообщение на *t*-кольце, следовательно $t \in E_A$.

Чтобы показать, что E_A циклически покрывает D, рассмотрим вычисление A на *s*-кольце. Хотя бы один процесс выбирает наименьший идентификатор, откуда следует (аналогично доказательству Теоремы 6.11), что этот процесс получил сообщение со следом длины len(s). Этот след является циклическим сдвигом s и принадлежит E.

Лемма 7.12 *В вычислении на* s*-кольце алгоритм* A *посылает не менее* $M(s,E_A)$ *сообщений.*

Доказательство. Пусть $t \in E_A$ - префикс циклического сдвига r последовательности s. Из определения E_A, A посылает t-сообщение в вычислении на t-кольце, а следовательно также и на r-кольце, которое совпадает с s-кольцом. Отсюда, для каждого t из $\{t \in E:$ t - префикс некоторого $r \in CS(s)\}$ в вычислении на s-кольце посылается хотя бы одно t-сообщение, что доказывает, что количество сообщений в таком вычислении составляет не менее M(s,E).

Для конечного множества I идентификаторов процессов обозначим через Per(I) множество всех перестановок I. Обозначим через $ave_A(I)$ среднее количество сообщений, используемых A во всех кольцах, помеченных идентификаторами из I, а через $wor_A(I)$ - количество сообщений в наихудшем случае. Из предыдущей леммы следует, что если I содержит N элементов, то

(1) $ave_A(I) \geq \frac{1}{N!} \sum_{s \in Per(I)} M(s, E_A)$; и

(2) $wor_A(I) \geq \max_{s \in Per(I)} M(s, E_A)$.

Теперь нижнюю границу можно вывести путем анализа произвольных полных множеств.

Теорема 7.13 *Средняя сложность однонаправленного алгоритма поиска наименьшего идентификатора составляет не менее* $N*\mathbf{H}_N$.

Доказательство. Усредняя по всем начальным конфигурациям, помеченным множеством I, мы находим

$$ave_A(I) \geq \frac{1}{N!} \sum_{s \in Per(I)} M(s, E_A)$$

$$= \frac{1}{N!} \sum_{s \in Per(I)} \sum_{k=1}^{N} M_k(s, E_A) = \frac{1}{N!} \sum_{k=1}^{N} \sum_{s \in Per(I)} M_k(s, E_A)$$

Зафиксируем k и отметим, что для любого $s \in$ Per(I) существует N префиксов циклических сдвигов s длины k. N! перестановок в Per(I) увеличивают количество таких префиксов до N*N!. Их можно сгруппировать в N*N!/k групп, каждая из которых содержит по k циклических сдвигов одной последовательности. Т.к. E_A циклически покрывает D, E_A пересекает каждую группу, следовательно $\sum_{s \in Per(I)} M_k(s, E_A) \geq \frac{N \cdot N!}{k}$.

Отсюда следует $ave_A(I) \geq \frac{1}{N!} \sum_{k=1}^{N} \frac{N \cdot N!}{k} = N \cdot \mathrm{H}_N$.

Этот результат означает, что алгоритм Чанга-Робертса оптимален, когда рассматривается средний случай. Сложность в наихудшем случае больше или равна сложности в среднем случае, откуда следует, что наилучшая достижимая сложность для наихудшего случая находится между $N*\mathbf{H}_N \approx 0.69N \log N$ и $\approx 0.356N \log N$.

Доказательство, данное в этом разделе, в значительной степени полагается на предположения о том, что кольцо однонаправленное и его размер неизвестен. Нижняя граница, равная $0.5N*\mathbf{H}_N$ была доказана Bodlaender [Bod88] для средней сложности алгоритмов выбора на *двунаправленных* кольцах, где размер кольца неизвестен. Чтобы устранить недетерминизм из двунаправленного кольца, рассматриваются вычисления, в которых каждый процесс начинается в одно и то же время и все сообщения имеют одинаковую задержку передачи. Для случая, когда размер кольца известен, Bodlaender [Bod91a] вывел нижнюю границу, равную $0.5N \log N$ для однонаправленных колец и $(1/4-\varepsilon)N*\mathbf{H}_N$ для двунаправленных колец (обе границы для среднего случая).

В итоге оказывается, что сложность выбора на кольце не чувствительна практически ко всем предположениям. Независимо от того, известен или нет размер кольца, однонаправленное оно или двунаправленное, рассматривается ли

средний или наихудший случай, - в любом случае сложность составляет Θ(N logN). Существенно важно, что кольцо асинхронно; для сетей, где доступно глобальное время, сложность сообщений ниже, как будет показано в Главе 11.

Т.к. лидер может быть выбран за одно выполнение децентрализованного волнового алгоритма, из нижней границы для выбора следует нижняя граница для волновых алгоритмов.

Заключение 7.14 *Любой децентрализованный волновой алгоритм для кольцевых сетей передает не менее* Ω(N logN) *сообщений, как в среднем, так и в наихудшем случае.*

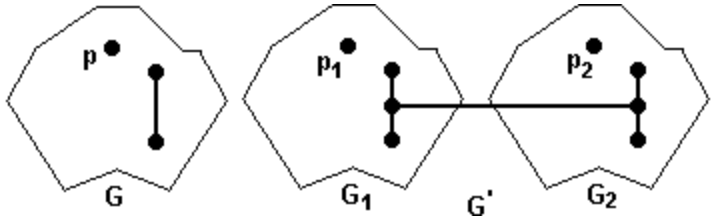


Рис.7.8

7.3 Произвольные Сети

Теперь изучим проблему выбора для сетей произвольной, неизвестной топологии без знания о соседях. Нижняя граница $\Omega(N \log N + |E|)$ сообщений будет показана ниже. Доказательство объединяет идею Теоремы 6.6 и результаты предыдущего подраздела. В Подразделе 7.3.1 будет представлен простой алгоритм, который имеет низкую сложность по времени, но высокую сложность по сообщениям в худшем случае. В Подразделе 7.3.2 будет представлен оптимальный алгоритм для худшего случая.

Теорема 7.15 Любой сравнительный алгоритма выбора для произвольных сетей имеет (в худшем и среднем случае) сложность по сообщения по крайней мере $\Omega(N \log N + |E|)$.

Рисунок 7.8 вычисление с двумя ЛИДЕРАМИ.

Доказательство. Граница $\Omega(N \log N + |E|)$ является нижней, потому что произвольные сети включают кольца, для которых нижняя граница $\Omega(N \log N)$. Чтобы видеть, что $|E|$ сообщений является нижней границей, даже в лучшем из всех вычислений, предположим что, алгоритм выбора имеет вычисление C на сети G, в котором обменивается менее чем $|E|$ сообщений ; см. Рисунок 7.8. Построим сеть G ', соединяя две копии G одним ребром между узлами, связанными ребром , которое не используется в C. Тождественные части сети имеют тот же самый относительный порядок как и в G. Вычисление C может моделироваться одновременно в обеих частях G ', выдавая вычисление, в котором два процесса станут избранными. □

Заключение 7.16 Децентрализованный волновой алгоритм для произвольных

сетей без знания о соседях имеет сложность по сообщения по крайней мере $\Omega(N\log N + |E|)$.

7.3.1 Вырождение и Быстрый Алгоритм

Алгоритм для выбора лидера может быть получен из произвольного централизованного волнового алгоритма применением преобразования называемого *вырождением*. В полученном алгоритме выбора каждый инициатор начинает отдельную волну; все сообщения волны, начатой процессом p должны быть помечены идентификатором p, чтобы отличить их от сообщений различных волн. Алгоритм гарантирует, что, независимо от того, сколько волн начато, только одна волна будет бежать к решению, а именно, волна самого маленького инициатора. Все другие волны будут прерваны прежде, чем решение может иметь место.

Для волнового алгоритма A, алгоритм выбора *Ex(A)* следующий. В каждый момент времени каждый процесс активен не более чем в одной волне ; эта волна - *текущая активная волна*, обозначенная *caw* , с начальным значением udef. Инициаторы выбора действуют, как будто они начинают волну и присваивают *caw* их собственный идентификатор . Если сообщение некоторой волны, скажем волны, которую начал q, достигает p, p обрабатывает сообщение следующим образом.

```
var caw_p    : P          init udef ; (* текущая активная волна *)
    rec_p    : integer    init 0 ;    (* число полученных ⟨tok, caw_p ⟩ *)
    father_p : P          init udef ; (* отец в волне caw_p *)
    lrec_p   : integer    init 0 ;    (* число полученных ⟨ldr, . ⟩ *)
    win_p    : P          init udef; (* идентификатор лидера*)
begin if p is initiator then
        begin caw_p := p;
              forall q ∈ Neigh_p do send ⟨ tok, p⟩ to q
        end;
      while lrec_p < #Neigh_p do
        begin receive msg from q ;
              if msg = ⟨ ldr, r ⟩ then
                begin if lrec_p = 0 then
                            forall q ∈. Neigh_p do send ⟨ ldr, r ⟩ to q ;
                      lrec_p := lrec_p + 1 ; win_p := r
                end
              else (* сообщение ⟨ tok, r⟩ *)
                begin if r < caw_p then (* Переинициализируем алгоритм*)
                            begin caw_p := r , rec_p := 0 , father_p :== q ;
                                  forall s∈ Neigh_p , s ≠ q
                                        do send ⟨ tok, r ⟩ to s
                            end;
                      if r = caw_p then
                        begin rec_p := rec_p + 1 ;
```

```
                    if rec_p = #Neigh_p then
                      if caw_p = p
                        then forall s ∈ Neigh_p do send ⟨ ldr, p ⟩ to s
                        else send ⟨ tok, cawp⟩ to father_p
                  end
              (* если r > caw_p  сообщение игнорируется*)
          end
      end;
    if win_p = p then state_p :== leader else state_p :== lost
end
```

Алгоритм 7.9 Вырождение примененное к алгоритму эха.

Если q> caw_p, сообщение просто игнорируется, эффективно приводя волну q к неудаче. Если с q = caw_p, с сообщением поступают в соответствии с волновым алгоритмом. Если q < caw_p или caw_p = udef, p присоединяется к выполнению волны q, повторно присваивая переменным их начальные значения и присваивая caw_p значение q. Когда волна, начатая q выполняет событие решения (в большинстве волновых алгоритмов, это решение всегда имеет место в q), q будет избран. Если волновой алгоритм такой, что решающий узел не обязательно равен инициатору, то решающий узел информирует инициатора через дерево охватов(остовное дерево) как определено в Lemma 6.3. При этом требуется не более N - 1 сообщений; мы игнорируем их в следующей теореме.

Теорема 7.17. Если A - централизованный волновой алгоритм, использующий M сообщений на одну волну, алгоритм *Ex(A)*, выбирает лидера использую не более NM сообщений.

Доказательство. Пусть p_0 самый маленький инициатор. К волне, начатой p_0 немедленно присоединяются все процессы, которые получают сообщение этой волны, и каждый процесс заканчивает эту волну, потому что нет волны с меньшим идентификатором, для которой процесс прервал бы выполнение волны p_0. Следовательно, волна p_0 бежит к завершению, решение будет иметь место, и p_0 становится лидером.
Если p не инициатор, никакая волна с идентификатором p не начнется, следовательно p не станет лидером. Если $p \neq p_0$ - инициатор, волна с идентификатором p будет начата, но решению в этой волне будет предшествовать событие посылки от p_0 (для этой волны) , или имееть место в p_0 (Lemma 6.4). Так как p_0 никогда не выполняет событие посылки или внутреннее событие волны с идентификатором p, такое решение не имеет место, и p не избран.
Не более N волн начаты, и каждая волна использует по крайней мере M сообщений, что приводит к полной сложности к NM. □

Более тонким вопросом является оценка сложности по времени алгоритма *Ex(A)*. Во многих случаях это будет величина того же порядока , что и сложность по времени алгоритма A, но в некоторых неудачных случаях, может слу-

читься, что инициатор с самым маленьким идентификатором начинает волну очень поздно. В общем случае можно показать сложность по времени O (Nt) (где t - сложность по времени волнового алгоритма), потому что в пределах t единиц времени после того, как инициатор p начинает волну, волна p приходит к решению или начинается другая волна.
Если вырождение применяется к кольцевому алгоритму, получаем алгоритм Chang-Poberts; см. Упражнение 7.9. Алгоритм 7.9 является алгоритмом выбора полученным из алгоритма эха. Чтобы упростить описание, принимается что udef > q для всех q∈ P. При исследовании кода, читатель должен обратить внимание, что после получения сообщения ⟨**tok**, r⟩ с r < caw_p, оператор *If* с условием r = caw_p также выполняется, вследствие более раннего присваивания caw_p. Когда выбирается процесс p (получает ⟨**tok,** p⟩ от каждого соседа), p посылает сообщение ⟨**ldr,** p⟩ всем процессам, сообщая им, что p - лидер и заставляя их закончить алгоритм.

7.3.2 Алгоритм Gallager-Humblet-Spira

Проблема выбора в произвольных сетях тесно связана с проблемой вычисления дерева охватов с децентрализованным алгоритмом, как выдно из следующего рассуждения. Пусть C_E сложность по сообщениям проблемы выбора и C_T сложность вычисления дерева охватов. Теорема 7.2 подразумевает, что $C_E \leq C_T + O(N)$, и если лидер доступен, дерево охватов, может быть вычислено используя $2|E|$ сообщений в алгоритме эха, который подразумевает что $C_T \leq C_E + 2|E|$. Нижняя граница C_E (теорема 7.15) подразумевает, что две проблемы имеют одинаковый порядок сдожности, а именно, что они требуют по крайней мере $\Omega(N \log N + E)$ сообщений.
Этот подраздел представляет Gallager-Humblet-Spira (GHS), алгоритм для вычисления (минимального) дерева охватов, используя $2|E| + 5N \log N$ сообщений. Это показывает, что C_E и C_T величины порядка $\theta(N \log N + E)$. Этот алгоритм был опубликован в [GHS83]. Алгоритм может быть легко изменен (как будет показано в конце этого подраздела) чтобы выбрать лидера в ходе вычисления, так, чтобы отдельный выбор как показано в выше не был необходим.
GHS алгоритм полагается на следующие предположения.
(1) Каждое ребро e имеет уникальный вес w (e). Предположим здесь, что w (e) - реальное число, но целые числа также возможны как веса ребер.
Если уникальные веса ребер не доступны априоре, каждому краю можно давать вес, который сформирован из меньшего из двух первых идентификаторов узлов, связанных с ребром. Вычисление веса края таким образом требует, чтобы узел знал идентификаторы соседей, что требует дополнительно $2|E|$ сообщений при инициализации алгоритма.
(2) Все узлы первоначально находятся в спящем состоянии и просыпаются прежде, чем они начинают выполнение алгоритма. Некоторые узлы просыпаются спонтанно (если выполнение алгоритма вызвано обстоятельствами, встречающимися в этих узлах), другие могут получать сообщение алгоритма, в то время как они все еще спят. В последнем случае узел получающий сообщение сначала выполняет локалбную процедуру

инициализации, а затем обрабатывает сообщение.

Минимальное дерево охватов. Пусть G = (V, E) взвешенный граф, где w {e) обозначает вес ребра e. Вес дерева охватов T графа G равняется сумме весов N-1 ребер, содержащихся в T, и T называется минимальным деревом охватов, или MST, (иногда минимальным по весу охватывающим деревом) если никакое дерево не имеет меньший вес чем T. В этом подразделе предполагаем, что каждое ребро имеет уникальный вес, то есть, различные ребра имеют различные веса, и это - известный факт что в этом случае имеется уникальное минимальное дерево охватов.

Утверждение 7.18 Если все веса ребер различны, то существует только одно MST.

Доказательство. Предположим обратное, т.е. что T1 и T2 (где T1 ≠ T2) - минимальные деревья охватов. Пусть e ребро с самым маленьким весом, который находится в одном из деревьев, но не в обоих; такой край существует потому что T1 ≠ T2. Предположим, без потери общности, что e находится в T1, но не в T2. Граф T2 ∪ {e} содержит цикл, и поскольку T1 не содержит никакой цикл, по крайней мере одно ребро цикла, скажем *e′,* не принадлежит T1. Выбор e подразумевает что w (e) < w (e '), но тогда дерево T2 ∪ {e} \ {e '} имеет меньший вес чем T2, который противоречит тому, что T2 - MST. □

Утверждение 7.18 - важное средство распределенного построения минимального дерева охватов, потому что не нужно делать выбор(распределенно) из множества законных ответов. Напротив каждый узел, который локально выбирает ребра, которые принадлежат любому минимальному дереву охватов таким образом, вносит вклад в строительство глобально уникального MST.
Все алгоритмы, для вычисления минимальное дерево охватов основаны на понятии фрагмента, который является поддеревом MST. Ребро e - исходящее ребро фрагмента F, если один конец e находится в F, и другой - нет. Алгоритмы начинают с фрагментов, состоящих из единственного узла и увеличивают фрагменты, пока MST не полон, полагаясь на следующее наблюдение.

Утверждение 7.19 Если F - фрагмент и e - наименьшее по весу исходящее ребро F, то F ∪ {e} - фрагмент

Доказательство. Предположите, что F ∪ {e} - не часть MST; тогда e формирует цикл с некоторыми ребрами MST, и одно из ребер MST в этом цикле, скажем f, - исходящее ребро F. Из выбора e - w (e) < w (f), но тогда удаляя f из MST и вставляя e получим дерево с меньшим весом чем MST, что является противоречием. □

Известные последовательные алгоритмы для вычисления MST - алгоритмы Prim и Kruskal. Алгоритм Prim [CLR90, Раздел 24.2] начинается с одного фрагмента и увеличивает его на каждом шаге включая исходящее ребротекущего фрагмента с наименьшим весом. Алгоритм Kruskal [CLR90, Раздел 24.2] начинается с множества фрагментов, состоящих из одного узла, и сливает фрагменты, до-

бавляя исходящее ребро некоторого фрагмента с наименьшим весом . Т.к. алгоритм Kruskal позволяет нескольким фрагментам действовать независимо, он более подходит для выполнения в распределенном алгоритме.

7.3.3 Глобальное Описание GHS Алгоритма.

Сначала мы опишем как алгоритм работает глобальным способом, то есть, с точки зрения фрагмента. Тогда мы опишем локальный алгоритм, который должен выполнить каждый узел, чтобы получить это глобальное преобразование фрагментов.
Вычисление GHS алгоритма происходит согласно следующим шагам.
(1) Множество фрагментов поддерживается таким, что объединение всех фрагментов содержит все вершины.
(2) Первоначально это множество содержит каждый узел как фрагмент из одного узла.
(3) Узлы во фрагменте сотрудничают, чтобы найти исходящее ребро фрагмента с минимальным весом .
(4) Когда исходящее ребро фрагмента с наименьшим весом известно, фрагмент объединяется с другим фрагментом добавлением исходящего ребра, вместе с другим фрагментом.
(5) Алгоритм заканчивается, когда остается только один фрагмент.
Эффективное выполнение этих шагов требует представления некоторого примечания и механизмов.
(1*) Имя фрагмента*. Чтобы определить исходящее ребро с наименьшим весом , нужно видеть,является ли ребро исходящим или ведет к узлу в том же самом фрагменте. Для этого каждый фрагмент будет иметь имя, которое будет известно процессам в этом фрагменте. Процесс проверяет является ли ребро внутренним или исходящим сравненивая имена фрагментов.
(2) *Объединение больших и маленьких фрагментов*. Когда объединяются два фрагмента, имя фрагмента изменяется по крайней мере в одном из фрагментов, что требует произвести изменения в каждом узле по крайней мере одного из двух фрагментов. Чтобы это изменение было эффективным, стратегия объединения основана на идеи, согласно которой меньший из двух фрагментов объединяется в больший из двух, принимая имя фрагмента большего фрагмента.
(3) *Уровни фрагментов*. Небольшое размышление показывает, что решение, кто из двух фрагментов является большим, не должно зависить от числа узлов в двух фрагментах. Для этого необходимо изменять размер фрагмента в каждом процессе, и большего и меньшего фрагментов, таким образом портя желательную свойство, что изменение необходимо только в меньшем. Вместо этого, каждому фрагменту назначен уровень, который является 0 для начального фрагмента с одним узлам. Это позволяется, что фрагмент F1 объединяется во фрагмент F2 с более высоким уровнем, после чего новый фрагмент F1 $\cup$ F2 имеет уровень F2. Новый фрагмент также имеет имя фрагмента F2, так что никакие изменения не для узлов в F2 не требуются. Такое объединение также возможно для двух фрагментов одинакового уровня; в этом случае новый фрагмент имеет новое имя, и уровень - на единицу выше чем уровень объединяющихся фрагментов. Новое имя фрагмента - вес ребра, которым объединены два фрагмента,

и этот ребро называется основным ребром нового фрагмента. Два узла, связанные основным ребром называются основными узлами.

Lemma 7.20. Если эти правила объединения выполняются, процесс изменяет имя фрагмента, или уровень не более *N log N раз*.

Доказательство. Уровень процесса никогда не уменьшается, и только, когда он увеличивается процесс изменяет имя)фрагмента. Фрагмент уровня L содержит по крайней мере 2^L процессов, так что максимальный уровень - logN, что означает, что каждый индивидуальный процесс увеличивает уровень фрагментов не более чем log N раз. Следовательно, полное общее число изменений имени фрагмента и уровня ограничено величиной N log N. □

Резюме стратегии объединения. Фрагмент F с именем FN и уровнем L обозначаем как F = (FN, L); пусть e_F обозначает исходящее ребро с наименьшим весом F.

Правило A. Если e_F ведет к фрагменту F ' = (FN ', L ') с L < L ', F объединяется в F ', после чего новый фрагмент имеет имя FN ' и уровень L '. Эти новые значения посылаются всем процессам в F.
Правило B. Если e_F ведет к фрагменту F ' = (FN ', L ') с L = L ' и e_F ' = e_F, два фрагмента объединяются в новый фрагмент с уровнем L + 1 и называют w (ep). Эти новые значения послаются всем процессам в F и F '.
Правило C. Во всех других случаях (то есть, L > L ' или L = L ' и $e_{F'} \neq e_F$) фрагмент F, должен ждать, пока применится правило A или B .

---

var $state_p$: *(sleep, find, found);*
$stach_p\ [q]$: (*basic, branch, reject)* for each $q \in Neigh_p$;
$name_p$, $bestwt_p$: real ;
$level_p$: integer;
$testch_p$, $bestch_p$, $father_p$: $Neigh_p$;
rec_p : integer;

(1) Как первое действие каждого процесса, алгоритм должен быть инициализирован:
begin пусть *pq* канал процесса *p* с наименьшим весом ;
$stach_p\ [q]$:= *branch* ; $level_p$:= 0 ;
$state_p$:= *found* ; rec_p := *0* ;
send ⟨ **connect,** 0 ⟩ to *q*
end

(2) При получении ⟨ **connect,** L ⟩ от *q:*
begin if *L* < $level_p$ **then** (* Объединение по правилу A *)
begin $stach_p[q]$:= *branch;*
send ⟨ **initiate**, $level_p$,$name_p$,$state_p$ ⟩ *to q*
end
else if $stach_p[q]$ = *basic*

then (* Правило C *) обработать сообщение позже
else (* Правило B *) send ⟨ **initiate**, $level_p + 1$, $\omega(pq)$, *find* ⟩ to *q*
end

(3) При получении ⟨**initiate**, L,F,S⟩ от *q:*
begin $level_p := L$; $name_p := F$; $state_p := S$, $father_p := q$;
$bestch_p := udef$; $bestwt_p := \infty$;
forall $r \in Neigh_p : stach_p[r] = branch \wedge r \neq q$ **do**
send ⟨ **initiate,** *L, F, S*⟩ to r ;
if $state_p = find$ **then begin** $rec_p := 0$; *test* **end**
end

Алгоритм 7.10 gallager-humblet-spira алгоритм (часть 1).

7.3.4 Детальное описания GHS алгоритма

Узел и статус связи. Узел p обслуживает переменные как показано в Алгоритме 7.10, включая статус канала stach p [q] для каждого канала pq. Этот статус - *branch*, если ребра, как известно, принадлежит MST, *reject*, если известно, что оно не принадлежит MST, и *basic*, если ребро еще не использовалось. Связь во фрагменте для определения исходящего ребра с наименьшим весом происходит через ребра *branch* во фрагменте. Для процесса p во фрагменте, *отецом* является ребро ведущее к основному ребру фрагмента. Состояние узла p, *state* p, - *find*, если p в настоящее время участвует в поиске исходящего ребра фрагмента с наименьшим весом и *found* в противном случае. Алгоритм дается как алгоритмы 7.10/7.11/7.12 . Иногда обработка сообщения должна быть отсрочена, пока локальное условие не удовлетворено.

(4) **procedure** *test:*
begin if ∃q ∈ e $Neigh_p : stach_p[q] = basic$ **then**
begin $testch_p := q$ with $stach_p[q] = basic$ and ω(*pq)* minimal;
send ⟨**test**, $level_p$, $name_p$ ⟩ to $testch_p$
end
else begin $testch_p := udef$; *report* **end**
end
(5)При получении ⟨ **test**, *L, F* ⟩ *от q:*
begin if $L > level_p$ then (* Отвечающий должен подождать! *)
обработать сообщение позже
else if $F = name_p$ **then** (* внутреннее ребро *)
begin if $stach_p[q] = basic$ **then** $stach_p[q] := reject$;
if q ≠ $testch_p$
then send ⟨ **reject** ⟩ to *q*
else *test*
end else send ⟨ **accept** ⟩ to *q*
end
(6) При получении ⟨ **accept⟩** от *q:*
begin $testch_p := udef$;

```
        if ω(pq) < bestwt_p
          then begin bestwt_p := ω(pq) ; bestch_p := q end ;
        report
    end
```
(7) Пр получении ⟨ **reject** ⟩ от *q:*
```
    begin if stach_p [q] = basic then stach_p [q] := reject ;
        test
    end
```

Алгоритм 7.11 THE GALLAGER-HUMBLET-SPIRA АЛГОРИТМ (ЧАСТЬ 2).

Принимается, что в этом случае сообщение сохраняется, и позже восстанавливается и с ним обращаются, как будто оно было получено только что. Если процесс получает сообщение, в то время как он все еще в состоянии *sleep*, алгоритм инициализируется в том узле (выполняя действие (1)) прежде, чем сообщение обработано.

Нахождение исходящго ребра с наименьшим весом. Узлы во фрагменте сотрудничают, чтобы найти исходящее ребро фрагмента с наименьшим весом, и когда ребро найдено через него посылается сообщение ⟨ **connect,** *L* ⟩ ; L - уровень фрагмента. Если фрагмент состоит из единственного узла, как имеет место после инициализации этого узла, требуемое ребро - просто ребро с наименьшим весом смежное с этим узлом.Смотри (1). Сообщение A ⟨ **connect,** 0 ⟩ посылается через это ребро.

---

(8) **procedure** *report:*
```
    begin if rec_p = #{q : stach_p [q] = branch ∧ q ≠ father_p }
                and testch_p = udef then
            begin state_p := found ; send ⟨ report, bestwt_p ) to father_p end
    end
```

(9) При получении ⟨ **report**, ω ⟩ от g:
```
    begin if q≠ father_p
            then (* reply for initiate message *)
                begin if ω < bestwt_p then
                        begin bestwt_p := ω ; bestch_p := q end ;
                    rec_p := rec_p + 1 ; report
                end
              else (* pq является основным ребром *)
                if state_p = find
                  then обработать это сообщение позже
                  else if ω > bestwt_p then changeroot
                                    else if ω = bestwt_p = ∞ then stop
     end
```

(10) **procedure** *changeroot;*
```
     begin if stach_p [bestch_p] = branch
```

then send ⟨ **changeroot** ⟩ to $bestch_p$
else begin send ⟨ **connect,** $level_p$ ⟩ to $bestch_p$;
$stach_p [bestch_p] := branch$
end
end

(11) При получении ⟨changeroot⟩:
begin *changeroot* **end**

Алгоритм 7.12 gallager-humblet-spira алгоритм (часть 3).

Затем рассмотрите случай, когда новый фрагмент сформирован, объединяя два фрагмента, соединяя их ребром e = pq. Если два объединенных фрагмента имели одинаковый уровень, L, p и q пошлют сообщение ⟨ **connect,** 1⟩ через e, и получат в ответ сообщение ⟨ **connect,** *L*⟩ , в то время как статус e - *branch*, см. действие (2). Ребро pq становится основным ребром фрагмента, p и q обменивают сообщением ⟨**initiate,** *L* + 1, *N, S* ⟩, присваивая новый уровень и имя фрагменту. Имя - w (pq), и статус *find* приводит к тому, что каждый процесс начинает искать исходящее ребро с наименьшим весом; см. действие (3). Сообщение ⟨ **initiate,** *L* + 1, *N, S*⟩ посылается каждому узлу в новом фрагменте. Если уровень p был меньше чем уровень q, p пошлет сообщение ⟨ **connect,** *L* ⟩ через e, и получит сообщение (**initiate**, *L' , N, S* ⟩ в ответ от q; см. действие (2). В этом случае, L ' и N - текущий уровень и имя фрагмента q, а имя и уровень узлов на стороне q ребра не изменяется. На стороне p ребра сообщение инициализации отправляется к всем узлам (см. действие (3)), заставляя каждый процесс изменять имя фрагмента и уровень. Если q в настоящее время ищет исходящее ребро с наименьшим весом (S = *find*) процессы во фрагменте p присоединяются к поиску с помощью вызова *test*.
Каждый процесс во фрагменте осуществляет поиск по все его ребрам (если такие существуют, см. (4), (5), (6), и (7)) для того, что определить имеются ли ребра выходящие из фрагмента, и если такие есть, выбирает наименьшее по весу. Исходящее ребро с наименьшим весом сообщается всем поддеревьям, с помощью сообщения ⟨**report**, ω⟩; см. (8). Узел p подсчитывает число сообщений ⟨**report**, ω⟩, которые получает, используя переменную recp, которой присваивается значение 0 при начале поиска (см. (3)) и увеличивается на единицу при получении сообщения ⟨**report**, ω⟩; см. (9). Каждый процесс посылает сообщение ⟨**report**, ω⟩ отцу, когда он получает такое сообщение от каждого из своих сыновей и заканчивает локальный поиск исходящего ребра.
Сообщения ⟨**report**, ω⟩ посылаются по направлению к основному ребру каждым процессом, и сообщения двух основных узлов пересекаются на ребре; оба получают сообщение от их отца; см. (9). Каждый основной узел ждет, пока он не пошлет сообщение ⟨**report**, ω⟩ сам пока он обрабатывает сообщение другого процесса. Когда два сообщения ⟨**report**, ω⟩ основных узлов пересеклись, основные узлы знают вес наименьшего исходящего ребра. Алгоритм закончился бы в этом точке, если никакое исходящее ребро не было бы передано (оба сообщения передают значения ∞).
Если исходящее ребро было передано, лучшее ребро находится следуя указателям bestch в каждом узле, начиная с основного узла той стороны, с которой бы-

ло передано лучшее ребро. Сообщение ⟨ **connect,** *L* ⟩ должно быть послано через это ребро, и все указатели отца во фрагменте должны указать в этом направлении; это выполняется с помощью посылки сообщения ⟨ **changeroot** ⟩. Основной узел, на чьей стороне расположено исходящее ребро с наименьшим весом, посылает сообщение ⟨ **changeroot** ⟩, которое посылается через дерево к исходящему ребру с наименьшим весом; см. (10) и (11). Когда сообщение ⟨ **changeroot** ⟩ достигает узла инцидентнорго исходящему ребру с наименьшим весом , этот узел посылает сообщение⟨**connect** ,L⟩ через исходящее ребро с наименьшим весом.

Проверка граней. Для нахождения наименьшего исходящего ребра узел p осматривает основные ребра одно за другим в порядке увеличения веса; см. (4). Локальный поиск ребра заканчивается когда либо не остается ребер(все грани - *reject* или *branch*), см. (4), либо один край идентифицирован как исходящий; см. (6). Из-за порядка, в котором p осматривает грани, если p опознает одно ребро как исходящее, оно должно иметь наименьший вес.
Для осметра ребра pq, p посылает сообщение ⟨ **test**, $level_p$, $name_p$ ⟩ к q и ждет ответ, который может сообщениями ⟨ **reject** ⟩, ⟨ **accept** ⟩ или ⟨ **test**, *L, F* ⟩ . Сообщение ⟨**reject** ⟩, посылается процессом q (см. (5)) если q обнаруживает, что имя фрагмента p, как в сообщении test, совпадает с именем фрагмента q; узел q также отклоняет ребро в этом случае. При получении сообщения ⟨ **reject** ⟩ p отклоняет ребро pq и продолжает локальный поиск; см. (7). Сообщение ⟨ **reject** ⟩ пропускается, если ребро pq только что использовалось q также, чтобы послать сообщение ⟨ **test**,*L,F*⟩; в этом случае сообщение ⟨ **test**, *L, F* ⟩ от q служит как ответ на сообщение p; см. (5). Если имя фрагмента q отличается от p, посылается сообщение ⟨ **accept** ⟩. По получении этого сообщения p заканчивает локальный поиск исходящих ребер ребром pq как лучшим локальным выбором; см. (6). Обработка сообщения ⟨ **test**, *L, F* ⟩ p отсрочена если L> levelp. Причина - то, что p и q может фактически принадлежать одному и тому же фрагменту, но сообщение ⟨**initiate,** *L, F, S* ⟩ еще не достиг p. Узел p мог бы ошибочно отвечать q сообщением ⟨ **accept** ⟩ .

Объединение фрагментов. После того как исходящее ребро с наименьшим весом фрагмента F = (*name, level*) было определено, сообщение ⟨**connect**, *level*⟩ посылается через это ребро, и получается узлом, принадлежащим к фрагменту F ' - = *(name', level')*. Назовем процесс, посылающий сообщение ⟨**connect**, *level*⟩ p и процесс, получающий его q. Узел q ранее послал сообщение ⟨**accept**⟩ к p в ответ на сообщение ⟨ **test**, *level, name*⟩, потому что поиск лучшего исходящегоребра во фрагменте p закончился. Ожидание, организованное перед ответом на сообщения test (см. (5)) дает $level' \leq level$.
Согласно правилам объединения, обсужденным ранее, ответ ⟨**connect**, *level*⟩ на сообщение ⟨**initiate,** *L, F, S* ⟩ имеет местов двух случаях.
Случай А: если $level' > level$, фрагмент p поглощается; узлам в этом фрагменте сообщается новое имя фрагмента и уровень с помощью сообщения ⟨ **initiate,** *level', name', S* ⟩, которое отправляется всем узлам во фрагменте F. Полный поглощенный фрагмент F становится поддеревом q в дереве охватов фрагмента F ' и если q в настоящее время занят в поиске лучшего исходящего ребра фрагмента F', все процессы в F должны участвовать. Вот почему q включает состояние

(*find* или *found*) в сообщение ⟨ **initiate,** *level', name', S* ⟩.
Случай B: если два фрагмента имеют один и тот же уровень и лучшее исходящее ребро фрагмента F ' также pq, новый фрагмент формируется с уровнем наимбольшим из двух и именем - вес ребра pq: см. (2). Этот случай происходит, если два уровня равны, и сообщение connect получено через ребро branch : заметьте, что статус ребра становится *branch*, если сообщение connect послано через него.

Если ни один из этих двух случаев не происходит, фрагмент F должен ждать, пока q посылает сообщение ⟨**connect**, L⟩, или уровень фрагмента q увеличился достаточно, чтобы делать Случай применимым.

Правильность и сложность. Из детального описания алгоритма должно быть ясно, что ребро через которое фрагмент посылает сообщение ⟨**connect**, L⟩ является действительно исходящим ребром фрагмента с наименьшим весом. Вместе с Суждением 7.19 это означает, что MST вычислен правильно, если каждый фрагмент действительно посылает такое сообщение и присоединяется к другому фрагменту, несмотря на ожидание, вызванного алгоритмом. Наиболее сложное сообщение содержит вес одного ребра, один уровень (до logN) и постоянное числа бит, чтобы указать тип сообщения и состояние узла.
Теорема 7.21 *Gallager-Humblet-Spira алгоритм (7.11/7.12 7.10/ Алгоритма) вычисляет минимальное дерево охватов, используя не более 5 N log N + 2 $|E|$ сообщений.*
Доказательство. Тупик потенциально возникает в ситуациях, где узлы или фрагменты должны ждать, пока некоторое условие не происходит в другом узле или фрагменте. Ожидание, вставляное для сообщения ⟨ **report,** ω⟩ на основном ребре не ведет к тупику, потому что каждый основной узел в конечном счете получает сообщения от всех сыновей (если фрагмент в целом не ждет другой фрагмент), после чего сообщение будет обработано.
Рассмотрите случай когда сообщение фрагмента F1 = (level1, name1) достигает узла фрагмента F2 = (level2, name2). Сообщение (**connect,** $level_1$) должно ждать, если level1 ≥ level2 и сообщение (**connect,** $level_2$) не было послано через то же самое ребро фрагментом F2, см. (2). Сообщение(test, $level_1$, $narne_1$) должно ждать, если level1 > level2, см. (5). Во всех случаях, где F1 ждет F2, верно одно из следующих утверждений.
(1) $level_1 > level_2$,
(2) $level_1 = level_2 \wedge \omega(e_{F1}) > \omega(e_{F2})$;
(3) $level_1 = level_2 \wedge \omega(e_{F1}) = \omega(e_{F2})$ и F_2 все еще ищет исходящее ребро с наименьшим весом. (Т.к. eF1 - исходящее ребро F2, не возможно чтобы w (eF2) > w (eF1).)
Таким образом никакой тупиковый цикл не может возникнуть.
Каждое ребро отклоняется не более одного раза, и это требует двух сообщений, который ограничивает число сообщений reject и сообщений test как следствий отклонений к $2|E|$. На любом уровне, узел получает не более одного сообщения initiate и accept , и посылает не более одного сообщения report, и одно changeroot или connect сообщение, и одно test сообщение, не ведущее к отклонению. На нулевом уровнени одно сообщение accept не получается и не одно сообщение report или test не посылается. На высшем уровне каждый узел только

посылает сообщение report и получает одно сообщение initiate. Общее количество сообщений поэтому ограничено $2|E| + 5N \log N$. □

7.3.5 Обсуждения и Варианты GHS Алгоритма

Gallager-Humblet-Spira алгоритм - один из наиболее сложных волновых алгоритмов, требует только локальное знание и имеет оптимальную сложность по сообщениям. Алгоритм может легко быть расширен так, чтобы он выбрал лидера, используя только на два больше сообщений. Алгоритм заканчивает в двух узлах, а именно основных узлах последнего фрагмента (охватывающего полную сеть). Вместо выполнения остановки, основные узлы обменивают их идентификаторами, и меньший из них становится лидером.

Было опубликовано множество разновидностей и родственных алгоритмов. GHS алгоритм может требовать время $\Omega(N^2)$, если некоторые узлы начинают алгоритм очень поздно. Если используется дополнительная процедура пробуждения (требующая не более $|E|$ сообщений) сложность алгоритма по времени $5N \log N$; см. Упражнение 7.11. Awerbuch [Awe87] показал, что сложность алгоритма по времени может быть улучшена од 0 (N), при сохранение оптимального порядка сложности по сообщениям ,то есть 0 ($|E| + N \log N$).

Afek и другие [ALSY90] приспособили алгоритм, для вычисления леса охвата с благоприятными свойствами, а именно, что диаметр каждого дерева и количество деревьев - 0 ($N^{1/2}$). Их алгоритм распределенно вычисляет кластеризацию сети как показано в Lemma 4.47 и дерево охвата и центр каждого кластера.

Читатель может спрасить, могут ли произвольные деревя охватов быть построены более эффективно чем минимальные деревя охватов, но Теорема 7.15 также дает низкую границу $\Omega(N \log N + |E|)$ на построение произвольных деревьев охватов. Johansen и другие [JJN ^ 87] дают алгоритм для вычисления произвольного дерева охватов, который использует $3N \log N + 2|E| + 0(N)$ сообщений, таким образом улучшая GHS алгоритме на постоянный множитель, если сеть редка. Barllan и Zernik [BIZ89] представили алгоритм, который вычисляет случайные деревья охватов, где каждое возможное дерево охватов выбрано с равной вероятностью. Алгоритм - рандомизирован и использует ожидаемое число сообщений, которое находится в границах между 0 ($N \log N + |E|$)) и 0 (N^3), в зависимости от топологии сети.

В то время как строительство произвольных и минимальных деревьев охватов имеет равную сложность в произвольных сетях, это не так в кликах. Korach, Moran и Zaks [KMZ85] показали, что строительство минимального дерева охватов в взвешенной клике требует обмена $\Omega(N^2)$ сообщениями. Этот результат указывает, что знание топологии не помогает уменьшать сложность обнаружения MST ниже границы из Теоремы 7.15. Произвольное дерево охватов клики может быть построено в 0 (N log N) сообщения, как мы покажем в следующем разделе; см. также [KMZ84].

7.4 Алгоритм Korach-Kutten-Moran

Много результатов были получены для проблемы выбора, не только для случая кольцевых сетей и произвольных сетей, но также и для случая другой специализированной топологии, типа сетей клик, и т.д. В нескольких случаях лучшие известные алгоритмы имеют сложность по сообщениям $0(N \log N)$ и в некоторых

случаях этот результат достигает $\Omega(NlogN)$. Korach, Kutten, и Moran [KKM90] показали, что имеется тесная связь между сетеми выбора и обхода. Их главный результат - общее строительство эффективного алгоритма выбора для класса сетей, учитывая алгоритм обхода для этого класса. Они показывают, что когда строительство снабжено лучшим алгоритмом обхода, известным для класса сетей, результирующий алгоритм благоприятно сравним с лучшим алгоритмом выбора, известным для того класса в большинстве случаев. Дело обстоит не так для сложности по времени; Сложность времени алгоритма равняется сложности по сообщениям, и в некоторых случаях известны другие алгоритмы с той же самой сложностью по сообщениям и более низкой сложностью времени.

7.4.1 Модульное Строительство

Korach-Kutten-Moran алгоритм использует идеи преобразования вырождения (Подраздел 7.3.1) и идеи Peterson/Dolev-Klawe-Rodeh алгоритма (Подраздел 7.2.2). Подобно преобразованию вырождения инициаторы выбора начинают обход сети с маркера, помеченного их идентификатором. Если обход заканчивается (разрешается), инициатор обхода становится избранным; алгоритм подразумевает, что это случается для точно одного обхода. В этом подразделе алгоритм описан для случая, где каналы удовлетворяют fifo предположение, но, поддерживая немного больше информации в каждом маркере и в каждом процессе алгоритм, может быть приспособлен к не - fifo случай; см. [KKM90].
Чтобы иметь дело с ситуацией больше чем одного инициатора, алгоритм работает на уровнях, которые могут быть сравнены с раундами Peterson/Dolev-Klawe-Rodeh алгоритма. Если по крайней мере два обхода начаты, маркеры достигнут процесса, который уже был посещен другим маркером. Если эта ситуация возникает, обход прибывшего маркера будет прерван. Цель алгоритма теперь становится, чтобы свести вместе два маркера в одном процессе, где они будут убиты и новый обход будет начат. Сравните это с Peterson/Dolev и другими алгоритмами, где по крайней мере один из каждых двух идентификаторов проходит круг и продолжает проходить следующий. Понятие раундов заменено в Korach-Kutten-Moran алгоритме понятием уровней; два маркера вызовут новый обход только если они имеют один и тот же уровень, и вновь произведенный маркер имеет уровень на единицу больше. Если маркер встречается с маркером более высокого уровня, или достигает узла, уже посещенного маркером более высокого уровня, прибывающий маркер просто убит без того, чтобы влиять на маркер на более высоком уровне.
Алгоритм дается как Алгоритм 7.13. Чтобы свести вместе маркеры одного и того же уровня в одном процессе, каждый маркер может быть в одном из трех режимов: *annexing, chasing,* или *waiting*. Маркер представляется (q, l), где q - инициатор маркера и l - уровень. Переменная levp дает уровень процесса p, и переменная catp дает инициатора последнего маркера *annexing* , отправленного p (в настоящее время активный обход p). Переменная waitp - udef, если никакой маркер не ожидает в p, и его значение q, если маркер (q, levp) ожидает в p. Переменная lastp используется для маркеров в режиме *chasing*: она дает соседа, которому p отправил маркер annexing уровня levp, если маркер *chasing* не был послан сразу после этого; в этом случае lastp = udef. Алгоритм взаимодействует с алгоритмом обхода запросом к функции trav: эта функция возвращает соседа, которому маркер должен быть отправлен или decide, если обход заканчивается.

Маркер (q, *l*) вводится в режиме *annexing* и в этом режиме он начинает исполнять алгоритм обхода (в случае IV Алгоритма 7.13) пока не произойдет одна из следующих ситуаций.
(1) Алгоритм обхода заканчивается: q становится лидером в этом случае (см. Случай IV в Алгоритме 7.13).
(2) Маркер достигает узла p уровня levp > *l*: маркер убит в этом случае, (Этот случай неявен в Алгоритме 7.13; все условия в том алгоритме требуют *l*> levp или *l* = levp.)
(3) Маркер прибывает в узел, где ожидает маркер уровня *l*: два маркера убиты в этом случае, и новый обход начинается с того узла (см. Случай II в Алгоритме 7.13).
(4) Маркер достигает узла с уровнем l, который был наиболее недавно посещен маркером с идентификатором catp > q (см. Случай VI) или маркером *chasing* (см. Случай III): маркер ожидает в том узле.
(5) Маркер достигает узла уровня l, который был наиболее недавно посещен маркером *annexing* с идентификатором catp < q: маркер становится маркером *chasing* в этом случае и посылается через тот же самый канал что и предыдущий маркер (см. Случай V).
Маркер *chasing* (g, l) отправляется в каждом узле через канал, через который наиболее недавно переданный маркер был послан, пока одна из следующих ситуаций не происходит.
(1) Маркер прибывает в процесс уровня levp > l: маркер убит в этом случае.
(2) Маркер прибывает в процесс с маркером *waiting* уровня l: два маркера удалены, и новый обход начат этим процессом (см. Случай II).
(3) Маркер достигает процесса уровня l, где наиболее недавно передан маркер *chasing*: маркер становится *waiting* (см. Случай III).
Маркер *waiting* находится в процессе, пока одна из следующих ситуаций не происходит.
(1) Маркер более высокого уровня достигает того же самого процесса: маркер waiting убит (см. Случай 1).
(2) Маркер равного уровня прибывает: два маркера удалены, и обход более высокого уровня начат (см. Случай II).
В Алгоритме 7.13 переменные и информация маркеров, используемая алгоритмом обхода игнорируются. Заметьте, что если p получает маркер уровня выше чем levp, это маркер annexing, инициатор которого не p . Если обход заканчивается в p, p становится лидером и отправляет сообщение всем процессам, заставляя их закончиться.
Правильность и сложность. Для того чтобы продемонстрировать правильность Korach-Kutten-Moran алгоритма, покажем, что число маркеров, произведенных на каждом уровне уменьшается до одного, на некотором уровне чей инициатор будет избран.
Lemma 7.22 Если произведены k> ***l*** маркеров на уровне ***l***, по крайней мере один и не более k/2 маркеров произведены на уровне *l + 1*.

var lev_p : integer **init** – 1;
cat_p , $wait_p$: *P* **init** *udef',*
$last_p$: $Neigh_p$ **init** *udef:*

begin if *p is* initiator **then**
begin $lev_p := lev_p + 1$; $last_p := trav(p.\ lev_p)$;
$cat_p := p$; send (**annex**, p, lev_p) to $last_p$

```
        end ;
      while . . . (* Условие завершения, смотри текст *) do
        begin receive token (q,l) ;
              if token is annexing then t := A else t := C ;
              if l > lev_p then (* Case I *)
                 begin lev_p := l ; cat_p := q ;
                       wait_p := udef ; last_p := trav(q, l) ;
                       send ( annex, q, l ) to last_p
                 end
              else if l == lev_p and wait_p ≠udef then (* Случай II *)
                 begin wait_p := udef ; lev_p := lev_p + 1 ;
                       last_p := trav(p, lev_p) ; cat_p := p ;
                       send ( annex, p, lev_p ) to last_p
                 end
              else if l = lev_p and last_p = udef then (* Случай III *)
                       waitp := q
              else if l = lev_p and t = A and q = cat_p then (* Случай IV *)
                 begin last_p := trav(q, t);
                       if last_p = decide then p объявляет себя лидером
                                          else send ( annex, q,l) to last_p
                 end
              else if l = lev_p and ((t = A and q > cat_p) or t = C) then (* Случай V *)
                 begin send ( chase, q, t ) to last_p ; last_p := udef end
              else if l = lev_p then (* Случай VI *)
                       wait_p := q
        end
end
```

Алгоритм 7.13 korach-kutten-moran алгоритм.

Доказательство. Не более k/2 маркеров произведены на уровне *l + 1*, потому что, когда такой маркер произведен, два маркера уровня l убиты в то же самое время.

Предположим, что имеется уровень l такой, что k> l маркеров произведены на уровне l, но никаком маркер не произведен на уровне l + 1. Пусть q процесс с максимальным идентификатором, который производит маркер на уровне l.Маркер (q, l) не заканчивает обход, потому что он будет получен процессом p, который уже отправил другой маркер уровня l.Когда это случается впервые (q, l) становится chasing или, если p уже отправил маркер chasing, (q, l) становится waiting. В любом случае, уровне l есть маркеры chasing .

Пусть (r, l) маркер с минимальным идентификатором после, которого посылается маркер chasing . Маркер (r, l) сам не может быть chasing, потому что маркер chasing преследует только маркеры с меньшим идентификатором. Мы можем поэтому предполагать, что (r, 1) стал waiting, когда он впервые достиг процесса p′, который уже отправил другой маркер уровня l.Пусть p″ последний процесс на пути следования (r, l), который получил маркер annexing, после того как он отправил

(r, l), и маркер сменил режим на chasing r. Этот маркер chasing встретит (r, 1) в p

', или откажется от преследования, если маркер waiting был найден прежде, чем маркер достиг p '. В обоих случаях маркер на уровне l + 1 произведен, т.е. мы пришли к противоречию.

Теорема 7.23 Korach-Kutten-Moran алгоритм (Алгоритм 7.13) - алгоритм выбора.

Доказательство Предположим, что по крайней мере один процесс начинает алгоритм. Согдасно предыдущей лемме, число маркеров, произведенных на каждом уровня уменьшается, и будет иметься уровень, на котором точно один маркер, скажем (q, 1) произведен. Никакой маркер уровня l ' < l не заканчивает обход, следовательно ни один из этих маркеров не заставляет процесс стать избранным. Уникальный маркер на уровне l только сталкивается с процессами на уровнях меньше чем l, или с cat = q (если он достигает процесса, который уже посещал), и отправляется в обоих случаях. Следовательно обход этого маркера заканчивается, и q избран.

Функция f, как считается выпуклой если f (a) + f (b) ≤ f (+ b). Сложность алгоритма проанализирована здесь согласно предположению что f (x) - алгоритм обхода(см. Раздел 6.3) , где f - выпуклая функция.

Теорема 7.24 если f (x) - используется алгоритмом обхода, где f выпуклая , KKM алгоритм выбора не более *(1+log k)[f(N)+N]* сообщений если он начинается k процессами.

Доказательство. Если алгоритм начат k процессами, не более 2^{-l} маркеров произведены на уровне l, что означает, что самый высокий уровень - не более $\lfloor \log k \rfloor$.

Каждый процесс посылает маркеры annexing не более одного идентификатора на каждом уровне. Если на некотором уровне l имеются маркеры с идентификаторами p1, …, pj и имеются процессы Ni, которые отправили маркер annexing (pi, l), то из этого следует что $\Sigma_l^j N_i \le N$. Т.к. алгоритм обхода является f (x) - алгоритмом обхода, маркер annexing (pj, l) был послан не более f (Nj) раз, что дает общее количество сообщений передающих маркеры annexing уровня l не более $\Sigma_l^j f(N_i)$, что не превышает f (N), потому что f выпуклая. Каждый процесс посылает не более одного маркера chasing на каждом уровне, что дает не более N маркеров chasing на уровень.

Таким образом имеется не более (1 + log k) различных уровней, и не более f (N) + N сообщений посылаются на каждом уровене, что доказывает результат. □

Построение Attiya. Другое постоения алгоритма выбора из алгоритмов обхода давалось Attiya [Att87]. В этом построении обход одного маркера, скажем с идентификатором q , не прерывается до тех пор, пока маркер не достигнет процесса r уже посещенного другим маркером, скажем процесса p.Маркер annexing ждет в r, но посылает маркер "охотник", чтобы преследовать маркер p и затем ветнуться в r, если p может быть успешно побежден. Если охотник возвращается, не нужно начинать новый обход, а текущий обход маркера *q* продолжается, таким образом потенциально сокращается сложность по сообщениям. Чтобы позволять охотнику возвращаться, чтобы обработать r, сеть должна быть двунаправленая. Если используется f (x) - алгоритм обхода, результирующий алгоритм выбора имеет сложность по сообщениям приблизительно 3. $\Sigma_l^j f(N/i)$

7.4.2 Применения Алгоритма KKM

Если f (x) - алгоритм обхода для класса сетей существует, этот класс как считают - f (x) -обходимый.

Выбор в кольцах. Кольца - x-обходимо, следовательно KKM алгоритм выбирает лидера в кольце используя 2N log N сообщений.

Выбор в кликах. Клики - 2x-обходимы, следовательно KKM алгоритм выбирает лидера в клике, использующей 3N log N сообщений согласно Теореме 7.24. Более осторожный анализ алгоритма показывает, что сложность - фактически 2N log N + 0 (N) в этом случае. Каждый маркер преследуется, используя не более трех сообщений chasing, так что общее количество сообщения chasing в вычислении, ограничено $3.\Sigma_0^{\log N+1} 2^{-i} N = 0(N)$. Никакой алгоритм выбора для клик со сложностью лучше чем 2NlogN + 0 (N) не известен до настоящего времени. Нижняя граница $\Omega(N \log N)$ была доказана Korach, Moran, и Zaks [KMZ84].

Нижняя граница не соблюдается, если сеть имеет направление глобального смысла [LMW86]. В сети, которая имеет направление глобального смысла, каналы процесса помечены целыми числами от 1 до N-1, и там существует направленный гамильтонов цикл такой, что канал pq помечен в процессе p расстоянием от p до q в цикле: см. Раздел B. 3. Loui, Matsushita. И West [LMW86] дал 0 (N) алгоритм выбора для клик с направление глобального смысла, но на вычисление направление глобального смысла затрачивается $\Omega(N^2)$ сообщений, даже если лидер доступен [Tel94].

Выбор в торе. Торические сети - x-обходимы, следовательно KKM алгоритм выбирает лидера в торе используя 2N log N сообщений.

Тор должен быть помечен (то есть, каждый край помечен как *Up, Left* и т.д.) чтобы применить x-алгоритм обхода (Алгоритм 6.11). Peterson [Pet85] дал алгоритм выбора для решетки и тора, который использует 0 (N) сообщения и не требует, чтобы грани были помечены.

Выбор в гипекубах. Гиперкубы со смыслом направлений - x-обходимы (см. Алгоритм 6.12), следовательно KKM алгоритм выбирает лидера в гиперкубе, использование 2N log N сообщений. Tel [Tel93] предложил алгоритм выбора для гиперкубов со смыслом направлений, который использует только 0 (N) сообщений. Вычисление смысла направлений стоит 0 (1.51) сообщений [Tel94], это также и сложность GHS алгоритма когда он применяется к гиперкубу.

Tel [Tel93] дал алгоритм выбора для гиперкубов со смыслом направлений, который использует 0 (N) сообщений.

Упражнения к Главе 7

Раздел 7.1

Упражнение 7.1 Докажите, что сравнительный алгоритм выбора для произвольных сетей - алгоритм волны, если случай, в котором процесс становится лидером, расценен как случай решения.

Упражнение 7.2 Покажите, что сложность по времени Алгоритма 7.1 *2D*.

Раздел 7.2

Упражнение 7.3 Докажите, что идентификатор $\Sigma_1^m \mathbf{H}_i = (m + 1)\mathbf{H}_i$ - *m* используемый в Подразделе 7.2.1.

Упражнение 7.4 Покажите, что ln (N + 1) < HN < ln (N) + 1. (*ln* означает нату-

ральный логарифм.)

Упражнение 7.5 Рассмотрите алгоритм Chang-Роберта согласно предположению, что каждый процесс является инициатором. Для какого распределения идентификаторов по кольцу сложность по сообщениям минимальна и сколько сообщений обменены в этом случае?

Упражнение 7.6 Какова средняя сложностью алгоритма Chang-Роберта, если имеется точно S инициаторов, где каждый выбор процессов S, одинаково, вероятно будет набором инициаторов?

Упражнение 7.7 Дайте начальную конфигурацию для Алгоритма 7.7, для которого алгоритм фактически требует ⌊Jog N + 1⌋ раундов. Также дайте начальную конфигурацию, для которой алгоритм требует только двух раундов, независимо от числа инициаторов. Является ли это возможным для алгоритма, чтобы закончиться в одном круге?

Упражнение 7.8 Определите набор ecr (как определяет Lemma 7.10) для алгоритма Chang-Роберта.

Раздел 7.3

Упражнение 7.9 Примените вырождение к кольцевому алгоритму и сравните алгоритм с алгоритмом Chang-Роберта. Каково различие и каково влияние этого различия?

Упражнение 7.10Определите для каждого из семи типов сообщения, используемых в Gallager/Humblet/Spira алгоритме, может ли сообщение этого типа быть послано узлу в состоянии *sleep*.

Упражнение 7.11 Предположим, что GHS алгоритм использует дополнительную процедуру пробуждения, которая гарантирует, что каждый узел начинает алгоритм в пределах единиц N время.

Докажите по индукции, что после не более чем 5N l — 3N единиц времени каждый узел - на 1 уровне. Докажите, что алгоритм заканчивает в пределах 5NlogN единиц время.

Раздел 7.4

Упражнение 7.12 Покажите, что O (NlogN) алгоритм для выбора в плоских сетях существует.

Упражнение 7.13 Покажите, что существует 0 (NlogN) алгоритм выбора для тора без смысла направлений.

Упражнение 7.14 Покажите, что существует 0 (NlogN) алгоритм выбора для гиперкубов без смысла направлений.

Упражнение 7.15 Покажите, что существует O (N (logN + k)) алгоритм выбора для сетей с ограниченной степенью k (то есть, сети, где каждый узел имеет не более k соседей).

8 Обнаружение завершения

Вычисление распределенного алгоритма заканчивается, когда алгоритм достигает *конечной конфигурации*; то есть конфигурация, в которой никакие дальнейшие шаги алгоритма невозможны. Не всегда в предельной конфигурации

каждый процесс находится в *конечном состоянии*; то есть в таком состоянии,что нет ни одного события применимого к процессу. Рассмотрим конфигурацию, в которой каждый процесс находится в состоянии, которое позволяет получать сообщения, и все каналы пусты. Такая конфигурация является конечной, но состояния процессов могут быть промежуточными состояния в вычислении. В этом случае, процессы не знают, что вычисление закончилось; такое завершение вычисления называется *неявным*. Завершение называется *явным* в процессе, если состояние этого процесса в конечной конфигурации - конечное состояние. Неявное завершение вычисления также называется *завершением сообщений*, потому что после достижения конечной конфигурации сообщения больше не посылаются. Явное завершение также называется *завершением процессов*, потому что при явном завершении алгоритма процессы заканчивают свое выполнение.

Обычно более легкое разрабатывать алгоритм, который заканчивается неявно чем алгоритм , который заканчивается явно. Действительно, в время разработки алгоритма все аспекты надлежащего завершения процессов могут игнорироваться; пр разработке концентрируются на ограничении полного числа событий, которые могут произойти. С другой стороны, применение алгоритма может потребовать, чтобы процессы закончились явно. Только после явного завершения результаты вычисления могут быть расцененным как заключительные, и переменные используемые в вычислении освобождены. Тупик распределенного алгоритма также приводит конечной конфигурации; в этом случае после достижения конечной конфигурации вычисление должно быть начато снова.

В основных методах этой главы будет исследовано для преобразование заканчивающихся неявно в явно заканчивающиеся. Зная такой метод, алгоритм можно разрабатывать беря во внимание только завершение сообщений (то есть, гарантируя только то, что алгоритм закончит вычисления), после чего алгоритм преобразуется в алгоритм завершения процессов с помощью использования одного из предложенных методов. Методы состоят из двух дополнительных алгоритмов, которые взаимодействуют с данным алгоритмом завершения сообщений и друг с другом. Один из этих алгоритмов наблюдает за вычислением и некоторым образом обнаруживает , что вычисление основного алгоритма достигло конечной конфигурации. После обнаружения такого завершения он вызывает второй алгоритм, который отправляет сообщения завершения всем процессам, заставляя их войти в конечное состояние.

Наиболее трудной частью преобразования оказывается алгоритм, обнаруживающий завершение. Процедура отправки сообщений завершения довольно тривиальна и будет обсуждена кратко в Подразделе 8.1.3. В этой главе будет показано, что обнаружение завершения возможно для всех классов сетей, для которых можно получить волновой алгоритм. Эти классы включают сети, где лидер доступен, сети, где процессы имеют идкентификаторы, древовидные сети, и сети, в которых доступна топологическая информация, типа диаметра или количества процессов. С другой стороны, в Главе 9 будет показано, что для анонимных сетей неизвестного размера существует неявно заканчивающийся алгоритм, но нет явно заканчивающегося алгоритма для вычисления максимума по входам процессов. Следовательно, обнаружение завершения невозможно в анонимных сетях где неизвестен размер.

Для тех случаев, в которых возможно обнаружение завершения, мы установим нижении границы числа сообщений, используемых алгоритмом обнаружения

завершения. Будет показано, что существуют алгоритмы, которые удовлетворяют этим границам. Раздел 8.1 представляет проблему формально, предлагая модель поведения распределенного вычисления и представляя низшую границу и процедуру посылки сообщений завершения. Раздел 8.2 представляет несколько решений, основанные на использовании дерева (или леса) процессов, включающего по крайней мере все процессы, которые все еще производят вычисление. Решения в этом разделе не слишком сложны и удовлетворяют низшей границе Разделе 8.1. Эти первые два раздела содержат все фундаментальные результаты касающиеся существования и сложности алгоритмов обнаружения завершения. По различным причинам один алгоритм обнаружения завершения может быть более подходящий для конкретного применения чем другой алгоритм. Поэтому, Разделы 8.3 и 8.4 представляют некоторые другие решения.

8.1 Предварительные замечания

8.1.1 Определения

В этом подразделе будет определена модель распределенных вычислений для изучения проблемы завершения распределенных вычислений. Модель получена из модели в Главе 2, но все аспекты не имеющие отношения к проблеме завершения отброшены.

Множество состояний процесса p - Z_p, разделено в два подмножества - активных и пассивных состояний. Состояние процесса p - C_p активно если в нем к процессу p применимо внутреннее событи или событие посылки, и пассивено в остальных случаях. В пассивном состоянии C_p применимо только событие приема или вообще нет применимого события, т.е. C_p - конечное состояние процессо p. Процесс p будем называть активным, если он находится в активном состоянии и пассивным в противном случае. Очевидно, что сообщение может быть послано только активным процессом, и пассивный процесс может стать активным только, после получения сообщения. Активный процесс может стать пассивным, если он войдет в пассивное состояние. Сделаем некоторые из предположения для упрощения описания алгоритмов в этой главе.

(1) *Активный процесс становится пассивным только после внутреннего события*. Любой процесс можно достаточно просто модифицировать, для того чтобы он удовлетворял этому предположению. Пусть *(c, m, d)* событие посылки (или получения) процесса p, где d - пассивное состояние. Заменим это событие процесса p на *(c, m, d′)*, где d' является новым состоянием, и единственным событием, применимым в d' является внутреннее ссобытие *(d′, d)*. Так как d' является активным состоянием, p становится пассивным после внутреннего события *(d′, d)*.

(2) *Процесс всегда становится активным после получения сообщения*. Любой процесс можно достаточно просто модифицировать, для того чтобы он удовлетворял этому предположению. Пусть *(c, m, d)* событие получения процесса p, где d - пассивное состоянием. Заменим это событие на *(c, m, d′)*, где d' является новым состоянием, и единственным событием, применимым в d' является внутреннее событие *(d′, d)*. Так как d' является активным состоянием, p становится активным после события получения и пассивным в его следующем событии *(d′, d)*.

(3*) Внутреннее событие, после которого p становится пассивным является единственным внутренним событием процесса p.* Внутренние события, после

которых *p* переходит из одного активного состояния в другое активное состояние игнорируются, потому что алгоритм обнаружения завершения должен быть *забывающий*; использовать информацию о том, как будет происходить локальное вычисление процесса не представляется возможным. Все активные состояния поэтому одинаковы для алгоритма обнаружения завершения.
Из состояния процесса *p* нас интересует только активное оно или пассивное; эта информация будет представлена переменной $state_p$. Все переходы вычисления представлены в алгоритме 8.1

---

var $state_p$: (active, passim) ;

$\mathbf{S}_p$: { $state_p$ = active }
begin send (**mes**) **end**
$\mathbf{R}_p$: { A message (**mes**) has arrived at p }
begin receive (**mes**) ; $state_p$:= active **end**
$\mathbf{I}_p$: { $state_p$ = active }
begin $state_p$:= passive **end**

Алгоритм 8.1 ОСНОВНОЙ распределенный алгоритм.

Как обычно, предполагается, что в начальной конфигурации нет сообщений, которые находились бы в процессе передачи. Первоначально процессы могут быть активны или пассивны.
Чтобы отличить этот алгоритм от алгоритма обнаружения завершения, его часто называют основным алгоритмом, и вычисление тогда называется основным вычислением. Алгоритм обнаружения завершения также называют управляющим или добавочным алгоритмом, и его вычисление называют управляющими или добавочными вычислениями. Аналогично, сообщения называются основные или управляющими.
Предикат **term** определен так, что принимает значение истина в каждой конфигурации, в которой не применимо никакое событие основного вычисления; согласно следующей теореме, в этом случае все процессы пассивны и ни одно основное сообщение не находится в процессе передачи.

Теорема 8.1
term <==> $(\forall p \in P : state_p = passive)$
$\wedge(\forall pq \in E : M_{pq}$ *не содержит сообщений* (**mes**)*).*
Доказательство. Если все процессы пассивны, внутренние события и события посылки не применимы. Если, кроме того, ни один канал не содержит сообщение (**mes**), то ни одно событие получения не применимо, следовательно ни один основной случай не применим вообще.
Если некоторый процесс активен, событие посылки или внутреннее событие возможно в том процессе, и если некоторый канал содержит сообщение (**mes**), событие получения этого сообщения применимо. □
В конечной конфигурации основного алгоритма каждый процесс ожидает полу-

чения сообщения и остается ожидающим навсегда. Проблема, обсуждаемая в этой главе - какой управляющий алгоритм нужно добавить в систему, чтобы перевести процессы в конечное состояние после того, как основное вычисление достигло конечной конфигурации. Для объединенного алгоритма (основного алглоритма вместе с управляющим алгоритмом) конфигурация, удовлетворяющая предикату **term** не обязательно является конечной: в общем случае могут иметься применимые события для управляющегоалгоритма. В управляющем алгоритме происходит обмен сообщениями, они могут быть посланы пассивными процессами и не переводить пассивный процесс в активное состояние после получения сообщения.

Управляющий алгоритм состоит из алгоритма обнаружения завершения и алгоритма объявления завершения. Алгоритм обнаружения завершения вызывает алгоритм объявления, и этот алгоритм переводит процессы в конечное состояние. Алгоритм обнаружения должен удовлетворять следующим трем требованиям.

(1) **Невмешательство**.Он не должен влиять на вычисление основного алгоритма.

(2) **Живучесть**. Если предикат **term** истинен , алгоритм объявления должен быть вызван за конечное число шагов.

(3) **Безопасность**. Если алгоритм объявления вызыван, конфигурация должна удовлетворить предикату **term**.

Проблема обнаружения завершения была впервые сформулирована Francez [Fra80]. Francez предложил решение, которое не удовлетворяет приципу невмешательства; сначала вычисление основного алгоритма "замораживалось" блокировкой всех событий, затем проверялось является ли конфигурация конечной. При положительном ответе вызывался алгоритм объявления; в противном случае, основное вычисление "размораживалось", и процедура повторялась спустя некоторое время. Вышеупомянутые требования не выполняются при таком решении проблемы.Они требуют, чтобы алгоритм обнаружения работал "непрерывно", то есть, во время работы основного вычисления. В доказательствах правильности обнаружения завершения в этой главе не дается объяснений по поводу выполнения требования невмешательства, потому что из самого описания алгоритма обычно вполне ясно, что это требование удовлетворено.

Основное вычисление называется централизованным если в каждом начальном состоянии имеется точно один активный процесс и децентрализованным, если число активных процессов в начальной конфигурации произволено. Централизованные основные вычисления часто называют в литературе по обнаружения завершения диффузийными вычислениями. Управляющие вычисление называют централизованными, если имеется один специальный процесс для управления вычислением, и децентрализованными, если процессы все выполняют один и тот же управляющий алгоритм .

8.1.2 Две нижних границы

Сложность алгоритма обнаружения завершения выражается как и ранее параметрами N и $|E|$ и числом сообщений M, используемых основным вычислением. Сложность обнаружения завершения также связана со сложностью алгоритма волны; обозначим сложность по сообщениям лучшего алгоритма волны W. W конечно зависит от характеристик класса сетей, которые мы рассматриваем ,

например, характеристики типа, является ли лидер доступным, топологии, и начального знания процессов.
Будет показана сложность обнаружения завершения в наихудшем случае. Как для централизованных, так и для децентрализованных вычислений, она ограничена снизувеличиной *M*. Затем будет показано, что сложность обнаружения завершения для децентрализованных основных вычислений ограничена снизу величиной *W*. В конце этого подраздела будет обсуждена ниняя граница по сообщениям $|E|$, выведенная Chandrasekaran и Venkatesan [CV90],.
Теорема 8.2 *Для каждого алгоритма обнаружения завершения существует основное вычисление, которое использует M основных сообщений и для которого алгоритм обнаружения использует по крайне мере M управляющих сообщений.*
Доказательство. Если система может достигнуть конфигурации, в которой управляющий алгоритм может обмениваться бесконечным числом управляющих сообщений без наступления основного события, результат следует тривиально. Поэтому предположим, что далее при доказательстве управляющий алгоритм реагирует на каждое основное событие конечным числом сообщений.
Пусть γ конфигурация в который два процесса, *p* и *q*, активные и нет сообщений, находящихся в процессе передачи. Если основной алгоритм централизован, такая конфигурация может быть достигнута из начальной конфигурации, обменом одного основного сообщения; в остальных случаях, такие конфигурации включены как начальные.
Сначала рассмотрите вычисление где, начиная с конфигурации γ, оба процесса станут одновременно пассивными, то есть, система достигнет $\delta = I_p(I_q(\gamma))$. Завершение должно быть обнаружено за конечное число шагов; но ни *p* ни *q* не могут вызвать алгоритм объявления не получа сначала сообщение от другого процесса. Иначе, завершение могло бы быть обнаружено ошибочно в конфигурации, где некоторый другой процесс все еще активен. (Если завершение обнаружено третьим процессом, необходимы по крайней мере два сообщения.) Следовательно, по крайней мере одно управляющее сообщение должно быть использовано в конфигурации δ прежде, чем завершение может быть обнаружено. Без потери общности, предположите, что *p* пошлет управляющее сообщение в конфигурации δ. Рассмотрим вычисление, в котором, начинающийся с конфигурации γ, только *p* становится пассивным, то есть, система достигает конфигурации $\gamma_p = I_p(\gamma)$. Состояние *p* одинаково конфигурациях γ_p и δ, и следовательно, *p* также посылает управляющее сообщение в конфигурации γ_p. Управляющий алгоритм может продолжать свою работу, но это не приведет к обнаружению завершения, т.к. *q* все еще активен. После того, как управляющий алгоритм прекратит обмен сообщениями, *q* посылает основное сообщение *p*, чтобы возвратиться конфигурацию, где *p* и *q* активены. Управляющий алгоритм может продолжать свою работу, но после конечного числа шагов будет снова достигнута конфигурация, в котором *p* и *q* являются активными и нет сообщений, которые находятся в процессе передачи. Подведем итог,
(1) Конфигурация, в которой *p* и *q* являются активными и нет сообщений, которые находятся в процессе передачи, может быть достигнута из начальной конфигурации передачей по крайней мере одного основного сообщения;
(2) Основной алгоритм может переходить из одной такой конфигурации в другую, передачей одного сообщения и вынуждая управляющий алгоритм передать по крайней мере одно управляющее сообщение ;

(3) Если основное вычисление заканчивается после такой конфигурации, по крайней мере одно управляющее сообщение должно быть передано для обнаружения завершения.
Теорема таким образом доказана. □

Теорема 8.3 *Обнаружение завершение децентрализованного основного вычисления требует в худшем случае передачи по крайней мере W управляющих сообщений.*

Доказательство. Рассмотрим основное вычисление, в котором не происходи обмен сообщениями и где каждый активный процесс становится пассивным после его первого события. Это основное вычисление требует, чтобы алгоритм обнаружения был волновым алгоритмом, если обнаружение (вызов алгоритма объявления) расценить как принятие решения. Действительно, вызов алгоритма объявления должен произойти за конечное число шагов, что доказывает, что алгоритм обнаружения сам заканчивается и принимает решение. Если решению не предшествует событие в некотором процессе q, рассматривается другое основное вычисление, где q не станет пассивным. Решение каузально не зависит не от какого события в q, так что алгоритм обнаружения может ошибочно вызвать алгоритм объявления, в то время как q все еще активен. Поскольку алгоритм обнаружения является волновым алгоритмом, он использует по крайней мере W сообщений. □

Начало алгоритма обнаружения. Chandrasekaran и Venkate-Сан [CV90] получили нижнюю границу управляющих сообщений $\lceil E \rceil$ полагаясь два следующих дополнительных предположения.

Cl. Каналы - fifo.

C2. Алгоритм обнаружения завершения может начинать выполнение в любое время после того, как началось основное вычисление, то есть, в произвольной конфигурации основного вычисления.

Согласно этим предположениям неправильное обнаружение может произойти, если алгоритм обнаружения не пошлет управляющее сообщение через одно специальное ребро, скажем pq. Только перед началом алгоритма обнаружения, основное вычисление посылает одно дополнительное сообщение через канал pq.

---

var $SentStop_p$: boolean **init** false ;
$RecStop_p$: integer **init** 0;

Procedure *Announce;*
begin if not $SentStop_p$ **then**
begin $SentStop_p := true$;
forall $q \in Out_p$ **do** send (**stop**) to q
end
end

{ Сообщение (**stop**) пришло в p }
begin receive (**stop**) ; $RecStop_p := RecStop_p + 1$;
Announce ;
if $RecStop_p = \#In_p$ **then halt**
end

Алгоритм 8.2 алгоритм объявления.

Это сообщение не замечается управляющим алгоритмом, из которого выводится неверное обнаружение. Алгоритм Chandrasekaran и Venkatesan посылает управляющее сообщение через каждый канал, таким образом отправка всех сообщений происходит до начала работы управляющего алгоритма и получение сообщений происходит до обнаружения.
Можно показать,используя аргументы подобные тем, что использовались в [CV90], что проблема не имеет решение вообще, если предположение *C2* действует, а предположение *C1* - нет. В этой главе мы предполагаем, что управляющий алгоритм начинает работу в начальной конфигурации основного вычисления, то есть основное вычисление не исполняет никакое незамеченное событие до начала работы управляющего алгоритма. Если это предположение заменить на предположением *C2*, проблема имеет решение, тогда и только тогда, когда каналы - fifo, и решение найдено в [CV90] для этого случая.

8.1.3 Завершение Процессов

Чтобы объявить о завершение всем процессам, им посылается сообщение (**stop**). Каждый процесс посылает такое сообщение всем соседям, но делает это не более одного раза при локальном вызове алгоритма объявления или при получении сообщения (**stop**).При получении сообщения (**stop**) от каждого соседа, процесс выполняет оператор **stop** , переводя процесс конечное состояние.
Процедура объявления представлена Алгоритмом 8.2.
Алгоритм 8.2 может использоваться для произвольной связной топологии, включая направленные сети, и не требует ни лидера, ни идентификаторов, ни знания топологии вообще.

8.2 Деревья Вычислений и Леса

Решения, описанные в этом разделе основаны на динамическом поддержании направленных графов, называемых *графом вычисления*, узлы которого включают все активные процессы и все основные сообщения, находящиеся впроцессе пердачи. Завершение считается обнаруженным, если граф вычисления становится пустым. Решения этого раздела требуют, чтобы сеть была ненаправлена, то есть, сообщения могут передаваться в двух направлениях через каждый канал. Подраздел 8.2.1 описывает решение для централизованного основным вычисления, для которого граф вычисления является деревом с инициатором в качестве корня. Подраздел 8.2.2 обобщает это решение для децентрализованных основных вычислений и использует лес, в котором каждый инициатор основного вычисления является корнем дерева.

8.2.1 Dijkstra-Scholten Алгоритм

Алгоритм Dijkstra и Scholten [DS80] обнаруживает завершение централизованного основного вычисления (называемого диффузийным вычислением в [DS80]). Инициатор основного вычисления (называемого окружением в [DS80]), также играет специальную роль в алгоритме обнаружения и обозначается p_0.
Алгоритм обнаружения динамически поддерживает дерево вычислений $T = (V_T, E_T)$ со следующими двумя свойствами.
(1) T пусто или T - направленное дерево с корнем p_0.

(2) Множество V_T включает все активные процессы и все основные сообщения, находящиеся в процессе передачи.
Инициатор p_0 вызывает алгоритм объявления когда $p_0 \notin V_T$ согласно первому свойству, T пусто в этом случае, и согласно второму свойству, предикат **term** принимает значение истина .
Для сохранения свойств дерева вычислений во время выполнения основного вычисления, T должно расширятся при отправке основного сообщения или при переходе процесса, не принадлежащего дереву, в активное состояние. Когда p посылает основное сообщение (**mes**), (**mes**) вставляется в дерево, и отцом (**mes**) является p. Когда процесс p, не принадлежащий дереву, становится активным получая сообщения от q, q становится отцом p. Для того, чтобы представить отправителя сообщения явно, основное сообщение (**mes**) послаемой q будем обозначать (**mes**, q).
Удаление узлов из T также необходимо по двум причинам. Во первых, основное сообщение удаляется после его получения. Во вторых, для того чтобы гарантировать продвижение алгоритма обнаружения дерево должно быть опустошено за конечное число шагов после завершения.

```
var state_p   : (active, passive) init if p = p0 then active else passive;
    sc_p      : integer           init 0;
    father_p : P                  init if p == p0 then p else udef;

S_p: { state_p = active }
     begin send (mes, p) ; sc_p := sc_p + 1 end

R_p: { сообщение (mes, q) прибывает в p }
     begin receive (mes, q);, state_p := active;,
           if father_p = udef then father_p := q else send ( sig, q ) to q
     end
I_p: { state_p = active }
    begin state_p := passive ;
          if sc_p = 0 then (* Удаляем p из T *)
             begin if father_p = p then Announce else send (sig , father_p) to father_p ;
                   father_p := udef
             end
     end
A_p: { Сигнал (sig ,p) прибывает в p }
    begin receive (sig,p) ; sc_p := sc_p -1 ;
          if sc_p = 0 and state_p = passive then
             begin if father_p = p then Announce else send ( sig, father_p ) to father_p ;
                   father_p := udef
             end
    end
```

Алгоритм 8.3 dijkstra-scholten алгоритм.

Сообщения - листья T; процессы поддерживают переменную, которая считает

число их сыновей в *T*. Удаление сына процесса *p* происходит в другом процессе q; это происходит при получении сообщения сына или удалении сына процесса *q*. Для того, чтобы предотвратить искажение данных счетчика сына *p*, процессу *p* посылается сигнальное сообщение (**sig**, *p*) об удалении его сына. Это сообщение заменяет удаленного сына *p*, и его удаление, т.е. получение, происходит в процессе *p* и *p* при получении сигнала уменьшает на единицу счетчик сыновей. Алгоритм описан как Алгоритм 8.3. Каждый процесс *p* имеет переменную $father_p$, которая не определена если $p \notin V_T$, равна *p* если *p* является корнем, и является отцом *p*, если *p* - не корень в *T*. Переменная sc_p содержит число сыновей *p* в *T*.

Доказательство правильности строго устанавливает, что граф T, как определено, является деревом и что он становится пустым только после завершения основного вычисления. Для любой конфигурации γ Алгоритма 8.3, определено

$V_T = \{p : father_p \neq udef\} \cup$ {передается (**mes**,*p*) } ∪ {передается (**sig**,*p*) }

И

$E_T =$ $\{(p, father_p) : father_p \neq udef \wedge father_p \neq p\}$ ∪ { ((**mes**, *p*), *p*) *:* (**mes**, *p*) передается }∪{((**sig**,p), p) : (**sig**, p) передается }.

Безопасность алгоритма следует из утверждения P, определенного следующим образом

$P \equiv state_p = active \Rightarrow p \in V_p$ (1)

$\wedge (u, v) \in E_T \Rightarrow u \in V_T \wedge v \in V_T \cap P$ (2)

$\wedge sc_p = \#\{v : (v, p) \in E_T\}$ (3)

$\wedge V_T \neq \varnothing \Rightarrow$ *T* дерево с корнем p_0 (4)

$\wedge (state_p = passive \wedge sc_p = 0) \Rightarrow p \notin V_T$ (5)

Значение этого инварианта следующие. По определению, множество узлов *T* включает все сообщения (основные и управляющие), и согласно пункту (1) оно также включает все активные процессы. Пункт (2) скорее технический; он заявляет, что *T* - действительно граф, и все ребра направлены к процессам. Пункт (3) выражает правильность счетчика сыновей каждого процесса, и пункт (4) заявляет, что *T* - дерево, и p_0 - корень. Пункт (5) используется, чтобы показать, что дерево действительно разрушается, если основное вычисление заканчивается. Для доказательства правильности, обратите внимание, что из *P* следует, что $father_p = p$ только для $p = p_0$.

Lemma 8.4 *P* - инвариант Dijkstra-Scholten алгоритма.

Доказательство. Первоначально $state_p = passive$ для всех $p \neq p_0$ и $father_{p0} \neq udef$, который доказывает пункт (1). Также, $E_T = \varnothing$, что доказывает (2). Так как $sc_p = 0$ для каждого *p*, удовлетворяется (3). $V_T = \{p_0\}$ и $E_T = \varnothing$, таким образом *T* - дерево с корнем p_0, что доказывает (4). Единственный процесс в V_T - p_0, и p_0 активен.

***S**$_p$*: Никакой процесс не становится активным в S_p, и никакой процесс не удаляется из V_T, так что (1) сохраняется.

Применимость действия означает, что p, отец нового узла (**mes**, *p*), находится в V_T, что доказывает, что (2) сохраняется. В результате действия, V_T дополняется узлом (**mes**, *p*) и E_T дугой ((**mes**, *p*), *p*), что означает, что *T* остается деревом, и sc_p правильно увеличивается, чтобы представить нового сына *p*, следовательно

(3) и (4) сохраняются.
Никакой процесс не становится пассивным листом, и никакой процесс не вставлен в V_T , таким образом (5) сохраняется.

R$_p$: Или p уже был в V_T ($father_p \neq udef$) или p вставлен в V_T после действия, таким образом (1) сохраняется.
Если значение $father_p$ определено, его новое значение - q, и если сигнал послан процессом p, его отец - также q, и q находится в V_T, таким образом (2) сохраняется. Число сыновей q не изменяется, потому что сын (*mes*, q) процесса q заменяется сыном p или сыном (**sig**, q), так что sc_q остается правильным, который сохраняет (3).
Структура графа не изменяется, таким образом (4) сохраняется. Процесс p находится в V_T после действия в любом случае, таким образом (5) сохраняется.
I$_p$: Переход p в пассивное состояние сохраняют (1), (2), (3) и (4). Из того, что p был прежде активен следует, что p был в V_T. Если $sc_p = 0$, p удаляется из V_T, таким образом (5) сохраняется.
Затем мы рассматриваем что случится при удалении p из T, то есть, если p окажется пассивным листом T.
Если сигнал посылается процессом p, отец сигнала - последний отец p, который находится в V_T, следовательно (2) сохраняется. В этом случае, сигнал заменяет p как сын процесса $father_p$, следовательно $father_{father\,p}$ остается правильным, и (3) сохраняется. Структура графа не изменилась, следовательно (4) сохраняется.
Иначе, то есть, если $father_p = p$, $p = p_0$ и p, являющийся листом, означает, что p был единственным узелом T, так что удаление опустошает T , что сохраняет (4).
A$_p$: Получение сигнала уменьшает число сыновей p на единицу, и присваивание значения на sc_p гарантируетсохранение (3). То, что p был отецом сигнала, означает, что p был в V_T. Если $state_p = passive$ и после получения sc_p присваивается 0, p удаляется, таким образом сохраняется (5). Если p удаляется из V_T, инвариант сохраняется по тем же причинам, что и для действия ***I***$_p$. □

Теорема 8.5 *Dijkstra-Scholten алгоритм (Алгоритм 8.3) - правильный алгоритм обнаружения завершения и использует M управляющих сообщений.*

Доказательство. Пусть S сумма всех счетчиков сыновей, то есть, $S = \Sigma_{p\in P} sc_p$.
Первоначально S = 0, S увеличивается на единицу при посылке основного сообщения (в S_p), S - уменьшается на единицу, когда получается управляющее сообщение (в A_p), и S никогда не становится отрицательным (из (3)). Это означает, что число управляющих сообщений никогда не превышает число основных сообщений в любом вычислении.
Чтобы доказать живость алгоритма, предположим, что основное вычисление закончилось. После завершения только действия A_p может иметь место и т.к. S уменьшается на единицу при каждом таком переходе, алгоритм достигает конечной конфигурации. Заметьте, что в этой конфигурации, V_T не содержит никакие сообщения. Также, из (5), V_T не содержит никаких пассивных листьев. Следовательно, T не имеет никаких листьев, что означает, что T пусто. Дерево стало пустым, когда p_0 удалил себя, и программа такова, что p_0 на этом шаге вызывает алгоритм объявления.
Чтобы доказать безопасность, обратите внимание, что только p_0 вызывает алгоритм объявления, и делает это после того, как удаляет себя из V_T. Из (4), T пус-

то, когда это случается,что заключает в себе **term.** □
Dijkstra-Scholten алгоритм достигает привлекательного баланса между передачей основных и управляющих сообщений; для каждого основного сообщения, посланного от p в q алгоритм посылает точно одно управляющее сообщение от q в p. Передача управляющих сообщений имеет нижнюю границу представленную в Теореме 8.2, так что алгоритм является оптимальным алгоритмом обнаружения завершения централизованных вычислений для худшего случая.
В описании этого подраздела, все сообщения содержат явное указание на их отца, но обычно это не является необходимым, потому что отец основных (управляющих) сообщений всегда их отправитель (адресат).

8.2.2 Алгоритм Shavit-Francez

Dijkstra-Scholten алгоритм был обобщен для децентрализованных основных вычислений Shavit и Francez [SF86]. В их алгоритме, граф вычисления - лес, каждое дерево которого имеет в качестве корня инициатора основного вычисления. Дерево с корнем p обозначается T_p.
Алгоритм поддерживает граф $F = (V_p, E_p)$, такой что (1) F является пустым или F - лес, каждое дерево которого имеет в качестве корня инициатора; (2) V_p включает все активные процессы и все основные сообщения. Как в алгоритме Dijkstra-Scholten завершении обнаруживается, когда граф становится пустым. К сожалению, в случае леса не так просто выяснить,является ли граф пустым. Действительно, свойство дерева вычислений иметь в качестве корня инициатора означает, что пустота дерева замечается корнем, который и вызывает алгоритм объявления. В случае леса, каждый инициатор замечает пустоту только собственного дерева, но это не означает, что весь леса пуст.
Проверка пустоты всех деревьев выполняется отдельной волной. Лес поддерживает дополнительное свойство, что если дерево T_p стало пустым, оно остается пустым и после этого. Обратите внимание, что это не мешает p стать активным; если p становится активным после разрушения дерева, p вставляется в дерево другого инициатора. Каждый процесс участвует в волне только, если его дерево разрушилось; когда волна принимает решение, вызывается алгоритм объявления. (вызов объявления не нужен, если выбранный волновой алгоритм генерирует решение в каждом процессе; в этом случае, процесс просто останавливается после принятия решения и завершения волнового алгоритма.)
Алгоритм дается как Алгоритм 8.4, в котором волновой алгоритм не показан явно. Каждый процесс p имеет переменную $father_p$, которая не определена, если $P \notin V_T$, и равна p если p является корнем или равна отцу p, если p не-корень в F. Переменная sc_p содержит число сыновей p в F. Логическая переменная *empty* истинна, тогда и только тогда, когда дерево p пусто.
Доказательство правильности алгоритма подобно доказательству Dijkstra-Scholten алгоритма. Для любой конфигурации γ Алгоритма 8.4, определим

$V_F = \{p : father_p \neq udef\}$ U {передается (**mes**, p) } U {передается (**sig**,p) }
И
$Ep =$ {(P, $father_p$): $father_p \neq udef \wedge father_p \neq$ p}
U {((**mes**, p), p) : (**mes**, p) передается } U {((**sig**,p}, p) : (**sig**,p} передается }.
Безопасность алгоритма будет следовать от утверждения Q, определенный ни-

же. Это инвариант алгоритма, и доказательство инвариантности подобно доказательству Lemma 8.4. Значение пунктов (1)-(5) из Q такие же как для инварианта алгоритма Dijkstra-Scholten и пункт (6) выражает тот факт, что каждый процесс правильно отслеживает,является ли он все еще корнем дерева в лесе. Конечно, лес пуст, если никакой процесс не является корнем.

$$
\begin{aligned}
Q \Leftrightarrow\ & state_p = active \Rightarrow p \in V_F && (1)\\
& \wedge\ (u, v) \in E_F \Rightarrow u \in V_F \wedge v \in V_F \cap P && (2)\\
& \wedge\ sc_p = \#\{\, v : (v, p) \in Ep \,\} && (3)\\
& \wedge\ V_F \neq \emptyset \Rightarrow F \text{ is a forest} && (4)\\
& \wedge\ (state_p = passive \ \mathrm{A}\ sc_p = 0) \Rightarrow p \notin Vp && (5)\\
& \wedge\ empty_p \Leftrightarrow \mathrm{Tp\ is\ empty} && (6)
\end{aligned}
$$

Lemma 8.6 Q - инвариант Shavit-Francez алгоритма.
Доказательство. Первоначально $state_p = passive$ для каждого не-инициатора p, и $father_p = p$ для каждого инициатора p, что доказывает (1). Также, $Ep = \emptyset$, что доказывает (2). Так как $sc_p = 0$ для каждого p, (3) удовлетворяется. $V_F = \{p : p$ - инициатор$\}$ и $E_F = \emptyset$, так что F - лес, состоящий из деревьев, содержащих один узел для каждого инициатора, что доказывает (4). Процессы в V_F - инициаторы, которые являются активными; это доказывает (5). Первоначально $empty_p$ равны (p - не-инициатор) и T_p действительно пуст, тогда и только тогда, когда p - не инициатор, что доказывает (6).
$\mathbf{S}_p$: Никакой процесс не становится активным в $\mathbf{S}_p$, и никакой процесс не удаляется из V_F, таким образом (1) сохраняется.
Применимость действия означает, что p, отец нового узла, находится в V_F, таким образом (2) сохраняется.
В результате действия, V_F пополняется вершиной (**mes**, p) и E_F ребром ((**mes**, p), p), что означает, что F остается лесом, и sc_p правильно увеличивается на единицу, чтобы представить наличие нового сына процесса p, таким образом (3) и (4) сохраняются.
Никакой процесс не становится пассивным листом, и никакой процесс не вставляется в V_F, таким образом (5) сохраняется.
Поскольку новый лист добавлен в не-пустое дерево, никакое дерево не становится непустым, и поскольку ни одна переменная *empty* не изменяется, (6) сохраняется.
$\mathbf{R}_p$: p уже был в V_F ($father_p \neq udef$) или p вставлен после действия, таким образом (1) сохраняется.
Если значение $father_p$ определено, его новое значение - q, и если послан сигнал, его отец также q, и q находится в V_T, таким образом (2) сохраняется.
Число сыновей процесса q не изменяется, потому что сын (**mes**, q) процесса q заменяется сыном p или сыном (**sig**, q), таким образом sc_q оставляет правильным (3).
Структура графа не изменяется, таким образом (4) сохраняется. Никакой процесс не становится пассивным листом, и никакой процесс не вставляется в V_F, таким образом (5) сохраняется.
Никакое дерево не становится пустым, следовательно (6) сохраняется.
$\mathbf{I}_p$: Перевод p в пассивное состояние сохраняет (1), (2), (3), и (4). То, что p преж-

де был активен означает, что p был в V_F. Если $sc_p = 0$, p удаляется из V_F, таким образом (5) сохраняется.
Затем мы рассмотрим что случится, если p удалить из F, то есть если p окажется, пассивным листом F. Если послан сигнал, отец сигнала - последний отец процесса p, который находится в V_F, следовательно (2) сохраняется. В этом случае, сигнал заменяет p на сына процесса $father_p$, следовательно $sc_{father\ p}$ остается правильным, и (3) сохраняется. Структура графа не изменилась, следовательно (4) сохраняется. Никакое дерево не становится пустым, таким образом (6) сохраняется. Иначе, то есть, если $father_p = p$, p был корнем и то, что p является листом означает, что p единственная иуршина в T_p ,таким образом ее удаление делает T_p пустым и присваивание $empty_p$ сохраняет (6).

---

var $state_p$: *(active, passive)* **init if** p is initiator **then** *active* **else** *passive* ;
sc_p : integer **init** 0 ;
$father_p$: P **init if** *p is* initiator **then** *p* **else** *udef* ;
$empty_p$: boolean **init if** *p is* initiator **then** false **else** true ;

S_p: { $state_p = active$ }
begin send (**mes**, p) ; $sc_p := sc_p + 1$ **end**

R_p: { Сообщение (**mes**, *q)* пришло в *p* }
begin receive (**mes**, *q)* ; $state_p := active$;
if $father_p = udef$ **then** $father_p := q$ **else** send (**sig**, *q)* to *q*
end

I_p: { $state_p = active$ }
begin $state_p := passive$;
if $sc_p = 0$ **then** (* Delete p from *F* *)
begin if $father_p$ = p **then** $empty_p := true$ **else** send (**sig**, $father_p$ } to $father_p$;
$father_p := udef$
end
end

A_p: { Сигнал (**sig**,p) пришел в *p* }
begin receive (**sig**,p) ; $sc_p := sc_p - 1$;
if $sc_p = 0$ **and** $state_p = passive$ **then**
begin if $father_p = p$ **then** $empty_p := true$ **else** send (**sig**, $father_p$) to $father_p$;
$father_p := udef$
end
end

Процессы одновременно выполняют волновой алгоритм, в котором посылка или принятие решения процессом p разрешается только, если $empty_p$ истина и

тогда *decide* вызывает алгоритм объявления .

Алгоритм 8.4 shavit-francez алгоритм.

$\mathbf{A}_p$: получение сигнала уменьшает число сыновей процесса p на единицу, и присваивание sc_p гарантирует, что (3) сохраняется. То, что p был отцом сигнала означает, что p был в V_F. Если $state_p$=*passive* и после получение сигнала $sc_p = 0$, p удаляется, таким образом (5) сохраняется.
Если p удаляется из V_F, инвариант сохраняется по тем же причинам, что и при действии $\mathbf{I}_p$.

Теорема 8.7 Алгоритм Shavit-Francez (Алгоритм 8.4) - правильный алгоритм обнаружения завершения и использует $M + W$ управляющих сообщени.
Доказательство. Также как в Теореме 8.5 можно показано, что число сигналов не превышает число основных сообщений. Помимо сигналов, управляющий алгоритм только посылает сообщения для одной волны. Из этого следует, что послано не более $M + W$ управляющих сообщений.
Чтобы доказать живость алгоритма предположим, что основное вычисление закончилось. За конечное число шагов алгоритм обнаружения завершения достигает конечной конфигурации, и можно показать, как это было сделано в Теореме 8.5, что в этой конфигурации F пуст. Следовательно, все события волны возможны в каждом процессе, и то, что конфигурация является конечной теперь означает, что все события волны были выполнены, включая по крайней мере одно принятие решения, при котором был вызван алгоритм объявления.
Чтобы доказывать безопасность, обратим внимание на то, что алгоритм объявления вызывается после принятия решения в волновом алгоритме. Это означает, что каждый процесс p послал сообщение волны или принял решение, и из алгоритма следует, что $empty_p$ истина, когда p сделает это. Никакое действие не присваивает переменной *empty* значение ложь повторно, так что (для каждого p) $empty_p$ истина, когда вызывается алгоритм объявления. Из (6), V_F пусто, что имеет следствием **term.** □

Число управляющих сообщений, используемых алгоритмом Shavit-Francez имеет тот же порядок, что и нижнии границы, выведенные в Теоремах 8.2 и 8.3.
Алгоритм является оптимальным алгоритмом для обнаружения завершения децентрализованных вычислений для худшего случая (если используется оптимальный волновой алгоритм).
Применение алгоритмов, рассматриваемых в этом разделе требует, чтобы каналы связи были двунаправленными; для каждого основного сообщения, посланного от p в q сигнал должен быть послан от q в p. Сложность с средним случаем равняется сложности в худшем случае; каждое выполнение требует одного сигнального сообщения на одно основное сообщение и, в случае Shavit-Francez алгоритма, точно одного выполнение волны.

8.3 Решения, основанные на волнах

По двум причинам полезно рассмотреть некоторые другие алгоритмы для обнаружения завершения кроме двух алгоритмов, данных в Разделе 8.2. Во первых,

Описанные алгоритмы могут использоваться только когда каналы двунаправленные. Во вторых, хотя сложность по сообщениям этих алгоритмов оптимальна в худшем случае, существуют алгоритмы, которые имеют лучшую сложность в среднем случае. Алгоритмы предыдущего раздела используют их число сообщений в наихудшем случае в каждом выполнении.
В этом разделе будут изучаться некоторые алгоритмы, основанные на повторном выполнении алгоритма волны; в конце каждой волны, либо обнаружевается завершение, либо начинается новая волна. Завершение обнаружено, если локальное условие окажется удовлетворенным в каждом процессе.
Сначала мы рассмотрим конкретные примеры алгоритмов, в которых во всех случаях волновой алгоритм является кольцевым алгоритмом. Для этого предположим, что кольцо вложено как подтопология сети ; но обмен основными сообщениями не ограничен каналам, принадлежащими к кольцу. Процессы перенумерованы с p_o до P_{N-1} и кольцевой алгоритм начинается процессом p_o , посылая маркер процессу P_{N-1}. Процесс p_{i+1} (для $i < N-1$) передает маркер процессу p_i. Передвижение маркера заканчивается, когда маркер получает процесс p_0.
Первое решение, обсуждаемое в этом классе - алгоритм Dijkstra, Feijen, и Van Gasteren (Подраздел 8.3.1); этот алгоритм обнаруживает завершение вычислений с синхронным прохождением сообщений. Несколько авторов обобщили алгоритм для вычислений с асинхронным прохождением сообщений; главная проблема здесь состоит в том, чтобы проверить, что каналы связи пусты. Мы обсуждаем решение Safra (Подраздел 8.3.2), в котором в каждом процессе подсчитывается число сообщений, которые посланы и получены; сравнивая счетчики можно определить действительно ли каналы являются пустыми. Также возможно использовать подтверждения для этой цели (Подраздел 8.3.3); но это решение снова требует, чтобы каналы были двунаправленными и чтобы число управляющих сообщений равнялось по крайней мере числу, используемому алгоритмом Shavit-Francez.
В Подразделе 8.3.4 принцип обнаружения будет обобщен для использования произвольного волнового алгоритма.

8.3.1 Алгоритм Dijkstra-Feijen-Van Gasteren

Алгоритм Dijkstra, Feijen, и Van Gasteren [DFG83] обнаруживает завершение основного вычисления, используя синхронное прохождение сообщений; действия такого вычисления даются как Алгоритм 8.5. В этих вычислениях, завершение описано с помощью предиката

$$\textbf{term} \Leftrightarrow \forall p : state_p = passive$$

term $\Leftrightarrow \forall p : state_p = passive$

---

var $state_p$ *: (active, passive);,*

C $_{pq}$: { $state_p = active$ }
begin (* *p* посылает основное сообщение, которое получает *q* *)
$state_p := active$
end

I_p: { $state_p = active$ }
begin $state_p := passive$ **end**

Алгоритм 8.5 основной алгоритм с синхронными сообщениями.

Сравните алгоритм и **term** с Алгоритмом 8.1 и Теоремой 8.1.
Алгоритм разработан как последовательность небольших шагов, каждый шаг прост для понимания, и правильность следует из инварианта, который разработан вместе с алгоритмом. Обработка взята из [DFG83]. Обзначим номер процесса, содержащего маркер t ,или если маркер находится в процессе передачи, номер процесса, к которому направляется является маркер. Отправление маркера может быть выполнено только процессом p_t, при этом t уменьшается на 1. Волна заканчивает когда $t = 0$; следовательно инвариант P должен быть выбран такой, что можно было сделать заключение о наличии завершения из P, $t = 0$, и другой информации в p_0. Инвариант должен сохранятся, когда p_0 начинает волну, то есть, когда $t = N - 1$.
Сначала положим $P = P_0$, где

$$P_0 \equiv \forall i\ (N > i > t) : state_p = passive.$$

Действительно, P_0 истинен когда $t = N - 1$, и если $t = 0$ и $statep_{p0} = passive,$ из этого утверждения можно сделать заключение о завершение. Отправление маркера сохраняет P_0, если только маркер отправляют пассивные процессы, поэтому мы принимаем следующее правило.
Правило 1. Процесс только тогда управляет маркером, когда он пассивен.
В этом режиме, P сохраняется с помощью отправления маркера и также с помощью внутренних действи1; к сожалению, P не сохраняется действиями связи. Предикат P_0 может принимать значение ложь, когда процесс p_j активизируется процессом p_i, где $j > t$ и $i \leq t$; см. Упражнение 8.4. Так как P_0 может принять значение ложь, P заменяется более слабым утверждением $(P_0 \vee P_1)$, где P_1 выбирается так, что каждый раз когда P_0 принимает значение ложь, P_1 является истинным. Каждому процессу присваивается цвет, белый или черный, и пусть $P = (P_0 \vee P_1)$ где

$$P_1 \equiv \exists j\ (t \geq j \geq 0) : color_{pj} = black.$$

Каждый раз, когда P_0 принимает значение ложь , P_1 является или становится истинным, если цвет посылающего процесса черный.

Правило 2. Посылающие процессы становятся черными.

Так как $(P \wedge color_{p0} = white \wedge t = 0) \Rightarrow \neg P_1$, все еще возможно обнаружить завершение с новым инвариантом, а именно, смотря является ли p_0 белым (и пассивным) когда он обрабатывает маркер.
Ослабление P предотвращает обращение предиката в ложь при совершении собыий получения и передачи сообщений; но более слабое утверждение может принять значение ложь при отправлении маркера, а именно, если процесс t - единственный черный процесс и он передает маркер. Ситуация исправляется дальнейшим ослаблением P. Пусть маркер тоже имеет цвет (белый или черный), и P ослабим до $(P_0 \vee P_1 \vee P_2)$, где

$P_2 \equiv$ маркер черный.

Отправление маркера сохраняет P_2, если черные процессы отправляют черный маркер.

Правило 3. Когда черный процесс отличный от p_0 посылает маркер, маркер становится черным.

Так как (маркер белый) $\Rightarrow \neg P_2$, завершение все еще может обнаружиться процессом p_0, а именно, по тому получает ли он белый маркер (и белый ли он сам и пассивный).

Действительно, теперь можно проверить, что внутренние действия, основная соммуникации, и отправление маркера сохраняют P. Присвоение маркеру черного цвета представляет явление неудачных волн; завершение не может быть определено процессом p_0, если возвращающийся маркер черный. Если волна заканчивается неудачно, должна быть начата новая.

Правило 4. Когда волна заканчивается неудачно, p_0 начинает новую волну.

Следующая волна будет конечно столь же неудачна как предшествующая, если нет никакого способа черным процессам стать белыми снова; действительно, черные процессы были окрашивают маркер в черный цвет при его отправлении, поэтому следующая волна также заканчивается неудачно.

Заметьте, что процесс p, окрашивающий маркер в белый цвет, не изменяет значение P на ложь если $i > t$, и P всегда принимет значение истина, когда p_0 начинает волну, посылая маркер к P_{N-1}. Из этого следует, что окрашивание в белый цвет может благополучно иметь место при отправления маркера.

Правило 5. Каждый процесса становиться белым сразу после посылки маркера.

Это гарантирует конечный успех волны после завершения основного вычисления. Алгоритм дается как Алгоритм 8.6.

---

var $state_p$: *(active, passive) ;*
$color_p$: *(white, black)* ;

C$_{pq}$: { *statep = active* }
begin (* p посылает основное сообщение, которое получает *q* *)
color$_p$:= black; (* Правило 2 *)
state $_q$ *:= active*
end

I$_p$: { $state_p$ *= active* }
begin $state_p$ *:= passive* **end**

Начало обнаружения, исполняется один раз процессом p_0:
begin send (**tok,** *white) to* p_{N-1} **End**

T$_p$: (* Процесс p обрабатывает маркер (**tok** ,c) *)
{ $state_p$ == *passive* } (* Правило I *)
begin if p = p_0
then if (c = *white* $\wedge$ $color_p$ = *white)*
then *Announce*

```
            else send ( tok, white} to p_{N-1}  (* Правило 4 *)
        else if (color_p = white)  (*Правило 3 *)
            then send ( tok, c ) to Next_p
            else send ( tok, black ) to Next_p ;
        color_p := white  (*Правило 5 *)
end
```

Алгоритм 8.6 dukstra-feuen-van gasteren алгоритм.

Теорема 8.8 *Dijkstra-Feijen- Фургон Gasteren алгоритм (Алгоритм 8.6) - правильный алгоритм обнаружения завершения для основных вычислений, использующих синхронное прохождение сообщений.*

Доказательство. Предикат $P \equiv (P_0 \vee P_1 \vee P_2)$ и алгоритм были разработаны таким образом, что P является инвариантом алгоритма. Завершение считается обнаруженным когда пассивный, белый p_0 обрабатывает белый маркер. Действительно, при этом из цвета маркера следует, что $\neg P_2$, из цвета процесса p_0 и из t = 0 следует $\neg P_1$, а из P_0 и состояния p_0 следует **term**. Следовательно алгоритм безопасен.

Чтобы доказать живость, предположим, что основное вычисление закончилось. После этого, все процессы отправляют маркеры без задержки, сразу после того, как их получают. Когда маркер заканчивает первый полный обход, начатый после завершения, все процессы окрашены в белый цвет и после того, как маркер заканчивает следующий обход, обнаруживается завершение. □

Теперь мы попытаемся оценить число управляющих сообщений, используемых алгоритмом. Основное вычисление, используемое в доказательстве Теоремы 8.2 заставляет алгоритм использовать по крайней мере один обход маркера для каждых двух основных сообщений; следовательно сложность алгоритма в худшем случае - ½ $N.M$ управляющих сообщений; см. Упражнение 8.5.

Алгоритм может использовать значительно меньшее количество сообщений в "среднем" основном вычислении. Предположим, что основное вычисление имеет сложность по времени T. Т.к. маркер всегда отправляется последовательно, не неблагоразумно предположить, что маркер отправляется приблизительно T раз прежде, чем заканчивается основное вычисление. (Даже эта оценка может быть слишком пессимистичной, т.к. отправление маркеров приостановлено в активных процессах.) Т.к. маркер отправляется менее чем $3N$ раза после завершения, алгоритм в этом случае использует $T + 3N$ управляющих сообщений. Сложность основного вычисления - по крайней мере T (а именно, сложность по времени), но если вычисление содержит достаточный параллелизм, сложность сообщения может достигать $\Omega(N.T)$.Если параллелизм позволяет каждому процессу посылать постоянное число сообщений в единицу времени, сложность по сообщениям основного вычисления - $N.T.\alpha$, то есть $\Omega(N.T)$. Число управляющих сообщений, который является 0 (N + T), тогда намного лучше чем можно ожидать от сложности обнаружения завершения в худшем случае.

8.3.2 Подсчет Основных Сообщений: Алгоритм Сафра

Синхронность прохождения сообщений, принятая для основного вычисления в алгоритме Dijkstra-Feijen-Van Gasteren - серьезное ограничение для его общего применения. Несколько авторов обобщили этот алгоритм для вычислений с

асинхронным прохождением сообщений (cf. Алгоритм 8.1). В данном подразделе будет обсуждено решение Сафра [Dij87]; в нем сложность в среднем случае сопоставима с сложностью алгоритма Dijkstra-Feijen-Van Gasteren.
Определим для каждой конфигурации число сообщений находящихсы в процессе передачи как *B*. Тогда **term** эквивалентен

$$(\forall p : state_p = passive) \wedge B = 0.$$

Снова инвариант *P* будет составлен так, что завершение можно будет определить из *P*, *t = 0*, и другой информации из p_0. Инвариант должен сохраняться, когда p_0 начинает волну, то есть, когда *t = N - 1*.
Чтобы информация о *B* была доступна в процессах (распределенным способом), процесс *p* снабжается счетчеком сообщений mc_p , и процессы поддерживают P_m как инвариант, где

$$P_m \equiv B = \sum_{p \in P} mc_p .$$

Инвариант P_m получен, когда первоначально $mc_p = 0$ для каждого *p*, и процессы подчиняются следующему правилу.
Правило M. Когда процесс *p* посылает сообщение, счетчик сообщений увеличивается на 1; когда процесс *p* получает сообщение, счетчик сообщений уменьшается на 1.
Инвариант должен позволять p_0 решать,что содержит **term**, когда он получает маркер (t = 0). Т.к. **term** теперь также включает ограничение на значение *B*, маркер будет использоваться для передачи целого числа *q* для вычисления суммы счетчиков сообщений процессов, которые его отправили. Попробуем установить $P = P_m \wedge P_0$, где

$$P_0 \equiv (\forall i\ (N > i > t) : state_{Pi} = passive) \wedge (q = \sum_{N>i>t} mc_{Pi}) .$$

Действительно, P_0 истинен, когда t = N -1 и q = 0, и если t = 0 тогда из *P* следует, что

$$(\forall i > 0 : state_{Pi} = passive) \wedge (mc_{P0} + q = B),$$

так что p_0 может определить завершение если $state_{P0} = passive$ и $mc_{P0} + q = 0$.
Утверждение P_0 устанавливается, когда p_0 начинает волну, посылая маркер в P_{N-1} с q = 0. Отправление маркера сохраняет P_0, только если отправляют маркер пассивные процессы и добавляют значение их счетчека сообщений; поэтому мы принимаем следующее правило.
Правило 1. Процесс обрабатывает маркер только когда он пассивен, и когда процесс посылает маркер он прибавляет значение своего счетчика сообщений к q.
При этом, P сохраняется при отправлении маркера и также при внутренних действиях; к сожалению, P не сохраняется при получении сообщения процессом p, с $i > t$. При получении сообщения P_0 принимает значение ложь, то есть, это применимо только когда B> 0. Т.к. Почтовый сохраняется перед тем как принимает значение ложь,сохраняется P_1, где

$$P_1 \equiv (\sum_{i \le t} mc_{Pi} + q) > 0.$$

Утверждение P_1 остается истинным при получении сообщения процессом p_i с $i > t$; следовательно, более слабое утверждение *P*, определенное как $P_m \wedge (P_0 \vee P_1)$ сохраняется при полученииисообщения процессом p_i с $i > t$.

---

var $state_p$: *{active, passive) ;*

$color_p$: *(white, black)* ;
mc_p : integer **init** 0 ;

S$_p$: { $state_p = active$ }
begin send (**mes**) ;
$mc_p := mc_p + 1$ (* Правило M *)
end

R$_p$: { Сообщение (**mes**) прибывает в p }
begin receive (**mes**) ; $state_p := active$;
$mc_p :=$ mc$_p$ - 1; (*Правило M *)
$color_p := black$ (*Правило 2 *)
end

I$_p$: { $state_p = active$ }
begin $state_p := passive$ **end**

Начало определения, выполняется один раз процессом p$_0$:
begin send (**tok**, *white, 0)* to p$_{N-1}$ **end**

T$_p$: (* Процесс *p* обрабатывает маркер (**tok**,c,q) *)
{ $state_p = passive$ } (*Правило I *)
begin if p = p$_0$
then if (c = *white)* $\wedge$ $(color_p$ = *white)* $\wedge(mc_p + q = 0)$
then *Announce*
else send (**tok**, *white,0)* to p$_{N-1}$ (*Правило 4 *)
else if $(color_p = white)$ (*Правила I and 3 *)
then send (**tok**, c, $q + mc_p$) to $Next_p$
else send (**tok**, *black,* $q + mc_p$) to $Next_p$;
$color_p := white$ (*Правило 5 *)
end

Алгоритм алгоритма 8.7 safra's.

Это измененное утверждение все еще позволяет обнаружить завершение процессу p_0 при тех же самых условиях, потому что, если t = 0, P$_1$ читает $mc_{P0} + q > 0$, таким образом если mc_{P0} + q = 0 (это уже требовалось для обнаружения), ¬P$_1$ сохраняется. Отправление маркера сохраняет P$_1$, и тоже самое происходит при посылке основного сообщения. К сожалению, P$_1$ может принималь значение ложь при получении сообщения процессом p$_i$ с $i \le t$. Как и в Подразделе 8.3.1, эта ситуация исправляется назначение цвета каждому процессу по следующему правилу:
Правило 2. Процесс получающий сообщение становится черным и P заменяется на $P_m \wedge (P_0 \vee P_1 \vee P_2)$, где

$$P_2 \equiv \exists j\ (t \ge j \ge 0) : color_p = black.$$

При каждом получении соощения, при котором P_1 принимает значение ложь, P_2 принимает значение истинна, так что P не принимает значение ложь при любом

основном действии. Так как $(P \wedge color_{P0} = \text{white} \wedge t = 0) \Rightarrow \neg P_2$, все еще возможно обнаружить завершение с новым утверждением, а именно, проверяя является ли p_0 белым (и пассивным) когда он обрабатывает маркер.
Ослабление P было успешно предотвращает изменение значения P после основных событий; но более слабое утверждение может принять значение ложь при отправлении маркера, а именно, если процесс t - единственный черный процесс и он посылает маркер. Ситуация исправляется дальнейшим ослаблением P.
Маркеру также назначается цвет (белый или черный) , и P ослабляется до $P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$, где

$$P_3 \equiv \text{маркер черный.}$$

Отправление маркера сохраняет P_3, если черные процессы отправляют черный маркер.
Правило 3. Когда черный процесс посылает маркер, маркер становится черным.
Т.к. (маркер белый) $\Rightarrow \neg P_3$ завершение может все еще обнаруживаться процессом p_0, а именно, проверкой получает ли он белый маркер (и белый ли он сам и пассивный).
Действительно, теперь можно быть уверенным, что внутренние действия, основные коммуникации и отправление маркеров сохраняют P. Волна заканчивается неудачно, когда маркер возвращается в p_0, если он черный, p_0 черный, или $mc_{P0} + q \neq 0$. Если волна заканчивается неудачно, должна быть начата новая волна.
Правило 4. Когда волна заканчивается неудачно, p_0 начинает новую волну.
Следующая волна закончилась бы так же неудачно как предыдущая, если бы не было способа для черных процессов стать снова белыми; действительно, черные процессы окрашивали бы маркер при его отправлении в черный цвет, из-за чего следующая волна закончилась бы также неудачно.
Заметим, что процесс p окрашивающий в белый цвет не изменяет заначение P на ложь если $i > t$, и что P всегда становится истинным когда p_0 начинает волну, посылая маркер в p_{N-1}.Из этого следует, что окрашивание в белый цвет может благополучно иметь место при отправлении маркера.
Правило 5. Каждые процесс становится белым сразу после посылки маркера.
При этом гарантируется конечный успех волны после завершения основного вычисления. Алгоритм дается как Алгоритм 8.7.
Теорема 8.9 Алгоритм Сафра (Алгоритм 8.7) - правильный алгоритм обнаружения завершения для вычислений с асинхронным прохождением сообщений.
Доказательство. Алгоритм был разработан так, что предикат P, определенный как $P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$ - инвариант алгоритма.
Чтобы показать безопасность, заметим, что завершение обнаружевается при t = 0, $state_{P0} = passive$, $color_{P0} = white$, и $mc_{P0} + q = 0$.Из этих условий $\neg P_3$, $\neg P_2$, и $\neg P_1$, следовательно $P_m \wedge P_0$, из чего вместе с $state_{P0} = passive$ и $mc_{P0} + q = 0$ следует **term**.
Чтобы показать живость, заметим, что после завершения основного вычисления счетчики сообщений не изменяют свои значения и их сумма, равняется 0. Волна, начатая в такой конфигурациизаканчивается с $mc_{P0} + q = 0$ и все процессы окрашены в белый цвет, что гарантирует, что следующая волна будет успешной. □
В отличие от алгоритма Dijkstra-Feijen-Van Gasteren, алгоритм Сафра не имеет

ограниченной сложности для худшего случая; маркер может быть пропущен неограниченное число раз, в то время как все процессы - пассивные, но некоторые основные сообщения находятся в процессе передачи в течение длительный периода времени. Что касается алгоритма Dijkstra-Feijen-Van Gasteren, для основного вычисления со сложностью времени T можно ожидать выполнения $\Theta(T + N)$ сообщений.

Векторно-расчетный алгоритм. Mattern [Mat87] предложил алгоритм, сопоставимый с алгоритмом Сафра, но который поддерживает отдельный счетчик сообщений для каждого адресата. Для процесса p счетчик сообщений - массив $mc_p[P]$. Когда *p* посылает сообщение *q*, *p* изменяет счетчик $mc_p[q] := mc_p[q]+1$ и когда *p* получает сообщение, он изменяет счетчик $mc_p[p] := mc_p[p] - 1$. Маркер также содержит массив счетчиков, и когда p обрабатывает маркер, к mc_p добавляется в маркер и обнуляется (массив в начале содержит нули). Завершение обнаружено, когда маркер равняется 0.

Алгоритм векторного-подсчета имеет некоторые преимущества по сравнению с алгоритмом Сафра, но также и страдает от некоторых серьезных неудобств. Одно из преимуществ алгоритма состоит в том, что завершение обнаруживается быстрее, а именно за один обход маркера после возникновения завершения. Второе преимущество состоит в том, что пока вычисление не закончилось, маркер приостанавливается чаще, что может уменьшить число управляющих сообщений, используемых алгоритмом. В алгоритме Сафра, каждый пассивный процесс передает маркер; в алгоритме векторного подсчета такой процесс *p* не будет отправлять маркер если из информации, содержащейся в маркере следует, что сообщение находиться на пути к *p*.

Главное неудобство алгоритма векторного подсчета состоит в том, что маркер содержит большое количество информации (а именно, целое число для каждого процесса), которую нужно передавать в каждом управляющем сообщении. Это неудобство не обременительно, если число процессов не велико. Другое неудобство состоит в том, что требуется знание идентификаторов других процессов. Если вектор представлен как массив, каждый процесс должен заранее знать идентификаторы всех процессов, но это требование может быть смягчено, если вектор представлен как множество пар целого числа и идентификатора. Первоначально каждый процесс должен знать по крайней мере идентификаторы соседей (чтобы правильно увеличивать счетчик), а другие идентификаторы будут изучены в течение вычисления.

8.3.3 Использование Подтверждений

Алгоритм Сафра подсчитывает основные сообщения, которые посланы и получены, чтобы знать есть ли основные сообщения, находящиесы в процессе передачи. Также возможно гарантировать это используя подтверждения; несколько авторов предложили такое усовершенствование алгоритма Dijkstra-Feijen-Van Gasteren, см., например, Naimi [Nai88]. Это разновидность принципа обсуждается лишь кратко, потому что результирующий алгоритм не во всех смыслах является усовершенствованием алгоритма Shavit-Francez, и поэтому устарел.

Сначала, заметим, что *никакое сообщение не находится в процессе передачи* эквивалентено *для всех p никакое сообщение, посланное p не находится в процессе передачи*. Каждый процесс ответственен за сообщения, которые он послало, то есть, он должен позабититься о том, чтобы алгоритм объявления не был

вызван, пока не будет уверенности в том, что все основные сообщения, посланные им получены. Метод обнаружения определяет для каждого процесса *p* локальное условие *quiet(p)* таким образом, что

quiet(p) $\Rightarrow$ *(state$_p$ = passive* $\wedge$ никакие основный сообщения, посланные процессом *p* не находятся в процессе передачи)

удовлетворен. Тогда ($\forall$p : *quiet(p))* $\Rightarrow$ **term.**

Чтобы устанавить, что никакое сообщение, посланное *p* не находится в процессе передачи, каждое сообщение подтверждается, и каждый процесс поддерживает счетчик числа подтверждений, которые он должен еще получить. Формально, утверждение P_α определяется как

P_α = $\forall$ p : *(unack$_p$ =* # (передается сообщение,посланное *p)* + # (передается подтверждение для *p*))

и поддерживается инвариант в соответствии с следующим правилом.

---

var *state$_p$: (active, passive) ;*
color$_p$: (white, black) ;
unack$_p$: integer **init** 0 ;

S$_p$: { *state$_p$ = active* }
begin send (**mes**) ; *unack$_{p:}$= unack$_p$* + 1 : (* Правило A *)
color$_p$:= *black (*Правило B *)
end

R$_p$: { сообщение (**mes**) из *q* прибыло в p }
begin receive (**mes**) ; *state$_p$:= active ;*
send (**ack**) to q (*Правило A*)
end

I$_p$: { *state$_p$ = active* }
begin *state$_p$* := *passive* **end**

A$_p$: { подтверждение (**ack**) прибыло в p }
begin receive (**ack**) ; *unack$_p$:= unack$_p$* – 1 **end** (* Rule A *)

Начало определения, исполняется один раз процессом p$_0$:
begin send *(* **tok,** *white)* to p$_{N-1}$ **end**

T$_p$: (* Процесс *p* обрабатывает маркер (**tok**,c) *)
{ *state$_p$ = passive* $\wedge$ *unack$_p$ = 0* }
begin if p = p$_0$
then if *c = white* $\wedge$ *color$_p$ = white*
then *Announce*
else send (**tok**, *white)* to *p$_{N-1}$*
else if *(color$_p$ = white)*
then send (**tok,** *c)* to *Next$_p$*
else send *(* **tok,** *black)* to *Next$_p$* ;

$color_p$:= *white* (*Правило B *)
end

Алгоритм 8.8 обнаружения завершения, использующие подтверждения.

Правило A. При посылке сообщения, *p* увеличивает $unack_p$, при получение сообщения от *q*, *p* посылает подтверждение *q* ; при получении подтверждения, p уменьшает на 1 $unack_p$.
Требования для *quiet* (а именно, что из *quiet(p)* следует, что *p* пассивен и никакое основное сообщение, посланное p не находится в процессе передачи) будут удовлетворены, если *quiet* определить как

$$quiet(p) \equiv (state_p = passive \wedge unack_p = 0).$$

Начало алгоритма обнаружения похоже на начало алгоритма Dijkstra-Feijea-Van Gasteren. Начинаем с рассмотрения утверждение P_0 ,определенного как

$$P_0 \equiv \forall\, i\ (N > i > t) : quiet(p).$$

Представление P_1 нужно выбирать осторожно, потому что активация процесса p_j с $j > t$ процессом p_i с $i \le t$ не имеет место в том же самом событии,что и посылка сообщения процессом p_i. Это имеет место, однако, что, когда p_j активизирован (таким образом, что P_0 ложь), $unack_{Pi} > 0$. Следовательно, если утверждение P_b определенное как

$$P_b \equiv \forall\, p : (unack_p > 0 \Rightarrow color_p = black),$$

Поддерживается наблюдением
Правило B. Когда процесс посылает сообщение, он становится черным; процесс становится белым только, когда он *quiet*.
Заключение снова подтветждает, что когда P_0 обращается в ложь, P_1 сохраняется, следовательно ($P_0 \vee P_1$) не обращается в ложь.

Результирующий алгоритм дается как Алгоритм 8.8, и инварианта - $P_a \vee P_b \vee (P_0 \wedge P_1 \wedge P_2)$, где

$P_a \equiv \forall$ p : *(*$unack_p$ =: #(передается сообщение посланное *p)*
+ #(передается подтверждение для p))

$P_b \equiv \forall$ p : $(unack_p > 0 \Rightarrow color_p = black)$

$P_0 \equiv \forall\, i\ (N > i > t) : quiet(p)$

$P_1 \equiv \exists\ i\ (t \ge i \ge O)$: $color_{Pi}$, = *black*

$P_2 \equiv$ маркер черный.

Теорема 8.10 *Алгоритма 8.8 - правильный алгоритм обнаружения завершения для вычислений с асинхронным прохождением сообщений.*
Доказательство. Завершение объявляется, когда p_0 *quiet* и обрабатывает белый маркер. Из этих условий следует, что $\neg P_2$ и $\neg P_1$, а следовательно $P_a \wedge P_b \wedge P_0$ сохраняются. Вместе с *quiet*(p_0) (p_0) это означает, что все процессы *quiet*, следовательно сохраняется **term.**
Когда основное вычисление заканчивается, через некоторое время получены все подтверждения, и все процессы становятся *quiet*. Когда заканчивается первая волна, которая начинается, когда все процессы *quiet*, все процессыокрашены в белый цвет, и завершение объявляется в конце следующей волны. □
Решение, основанное на ограниченной задержке сообщений. В [Tel91b, Раздел 4.1,3] классе решений обнаружения завершения (и других проблем) описывается решение основанное на предположении, что задержка сообщений огра-

ничена
постоянной μ. (См. также Раздел 3.2). В этих решениях, процесс не является *quiet* промежуток времяни μ после отправления последнего сообщения, также процесс остается черным, пока он не *quiet*, как описано в решении основанном на использовании подтверждений. Процесс p становится *quiet* если (1) прошло по крайней мере μ времяни после того как прцесс *p* посылал последний раз сообщения и *p* пассивен. Полный формальный вывод алгоритма предоствлен читателю.

8.3.4 Обнаружение завершения с помощью волн

Все алгоритмы обнаружения завершения, обсужденные пока в этом разделе используют кольцевую подтопологию для управляющих коммуникаций; все алгоритмы основаны на алгоритме волны для колец. Подобные решения были предложены для другой топологии; например, Francez и Rodeh [FR82] и Topor [Top84] предложили алгоритм, использующий корневое дерево охватов управляющих соммуникаций. Tan и Van Leeuwen [TL86] предложили децентрализованные решения к кольцевым сетям, для сетей деревьев, и для произвольных сетей. Изучение этих решений показывает, что они очень похожи друг на друга, за исключением алгоритма волны, на который они опираются.
В этом подразделе делается набросок для вывода алгоритма обнаружения завершения (и инварианта), основанного на произвольном алгоритме волны, а не на специально определенном алгоритме (кольцевом алгоритме). Для каждой волны, первое событие, в котором процесс посылает сообщение для волны или принимает решение, называется посещением того процесса. Предполагается, что, если необходимо, процесс может отложить посещение, пока не удовлетворено локальное условие процесса. Последующие события той же самой волны больше не приостанавливаются.
Этот подраздел представляет вывод только для случая синхронного прохождения сообщений основного вычисления (как для вывода в Подразделе 8.3.1). Этот вывод можно обобщить для асинхронного случая, подобно тому как это сделано в Подразделе 8.3.2 и 8.3.3.
Инвариант алгоритма должен позволить обнаружить завершение, когда волна принимает решение; поэтому, сначала мы устанавливаем $P = P_0$, где

$P_0 \equiv$ все посещенные процессы пассивны.

Действительно, поскольку все процессы были посещены, когда произошло принятие решения, это утверждение позволяет обнаружение завершения, когда волна принимает решение. Кроме того, P_0 устанавливается когда волна начинается (нет еще посещенных процессов). При работе алгоритма волны P_0 сохраняется по правилу 1, представленному ниже.
Правило 1. Только пассивные процессы посещаются волной. К сожалению, P_0 принимает значение ложь, когда посещенный процесс активизируется непосещенным процессом. Поэтому, каждый процесс обеспечивается цветом, и P ослаблляется до $(P_0 \vee P_1)$, где

$P_1 \equiv$ имеется непосещенный черный процесс.

Более слабый инвариант сохраняется согласно правилу 2.
Правило 2. Процесс посылающий сообщение становится черным.
Волна может изменить значение более слабого утверждение, если посещен единственный непосещенной черный процесс. Ситуация исправляется даль-

нейшим ослаблением P. Каждый процесс представляет цвет, белый или черный, как входное данное для волну. Волна измененяется так, чтобы вычислить самый *темный* из представленных цветов; вспомним, что волны могут вычислять infirna, и " самый темный " является infirnum. Когда волна принимает решение, будет вычислен самый темный из всех представленных цветов; это будет белый цвет, если все процессы представляют белый и черный, если по крайней мере один процесс представляет черный. В время волны, волна будет называться белой, если ни один процесс еще не представляет черный цвет; и черной, если по крайней мере один процесс уже представляет черный цвет. Таким образом процесс, когда он посещается, либо представляет белый цвет, что не изменяет цвет волны, либо представляет черный цвет,что окрашивает волну в черный цвет. P ослабляется до $(P_0 \vee P_1 \vee P_2)$, где

$$P_2 \equiv \text{ волна черная.}$$

Это утверждение сохраняется по следующему правилу.

Правило 3. Посещенный процесс представляет волне свой текущий цвет.

Действительно, все основные коммуникации также как деятельность волны сохраняют это утверждение, которое является поэтому инвариантом. Волна заканчивается неудачно, если процессы принимают решение для черной волны, но в этом случае просто начинается новая волна. Новая волна может быть успешной, только если процессы могут стать белыми, и это случается немедленно после посещения волны.

Правило 4. Решающий узел в черной волне начинает новую волну.

Правило 5. Процессы немедленно становятся белыми после каждого посещения волны.

Эти правила гарантируют возможный успех волны после завершения основного вычисления. Действительно, если основное вычисление закончилось, первая волна, начатая после завершение, окрашивает все процессы в белый цвет, и следующая волна заканчивается успешно.

В этом алгоритме только одна волна может бежать в любой время. Если две волны, скажем A и B, бегут одновременно, окрашивание процесса в белый цвет после посещения волной B может нарушить инвариант для волны A. Поэтому, если алгоритм обнаружения должен быть децентрализован, должен также использоваться децентрализованный алгоритм волны, чтобы все инициаторы алгоритма обнаружения сотрудничали в той же самой волне. Также возможно использовать другой принцип обнаружения, в котором различные волны могут вычислять одновременно без того, чтобы нарушить правильное действие алгоритма обнаружения; см. Подраздел 8.4.2.

8.4 Другие Решения

Еще два решения проблемы обнаружения завершения будут обсуждены в этом разделе: алгоритм восстановления кредита и алгоритм временных пометок.

8.4.1 Алгоритм восстановления кредита

Mattern [Mat89a] предложил алгоритм, который обнаруживает завершение очень быстро, а именно, за одну единицу времени после возникновения (при принятии предположений идеализации времени из Определения 6.31). Алгоритм обнаруживает завершение централизованного вычисления и предполагает, что каждый процесс может послать сообщение инициатору вычисления непосредственно (то

есть, сеть содержит звезду с инициатором в центре).
В алгоритме каждому сообщению и каждому процессу назначается значение кредита, которое всегда находится между 0 и 1 (включая границы), и алгоритм поддерживает следующие утверждения как инварианты.
S1. Сумма всех кредитов (в сообщениях и процессах) равняется 1.
S2. Основное сообщение имеет положительный кредит.
S3. Активный процесс имеет положительный кредит.
Процессы имеют положительный кредит, когда правилами не предписано (то есть, пассивным процессам) посылать их кредиты инициатору. Инициатор действует как банк, собирая все кредиты, посланные ему, в переменной ret .
Когда инициатор имеет все кредиты, требование для активных процессов и основных сообщений иметь положительный кредит означает, что не имеется никаких таких процессов и никаких таких сообщений; следовательно **term** сохраняется.
Правило 1. Когда ret = 1, инициатор вызывает алгоритм объявления.
Для выполнения требования живости, все кредиты в конечном счете должны быть переданы инициатору при возникновении завершения. Если основное вычисление закончилось, больше нет основных сообщений, и нас интересуют только кредиты, поддерживаемые процессами.

---

var $state_p$ *: (active, passive)* **init if** $p = p_0$ **then** *active* **else** *passive* ;
$cred_p$ *:* fraction **init if** $p = p_0$ **then** I **else** 0 ;
ret *:* fraction **init** 0 ; for p_0 only

$\mathbf{S}_p$: { $state_p = active$ } (* Праволо 3 *)
begin send (**mes**,$cred_p$ / 2) : $cred_p := cred_p / 2$ **end**

$\mathbf{R}_p$: { Сообщение (**mes**,c) прибыло в p }
begin receive (**mes**,c) ; $state_p := active;$
$cred_p := cred_p + c$ (* Правила 4 and 5b *)
end

$\mathbf{I}_p$: { $state_p = active$ }
begin $state_p := passive$;
send (**ret**, $cred_p$) to p_0 ; $cred_p$:== 0 (* Правило 2 *)
end

$\mathbf{A}_{P0}$: { Сообщение (**ret,** c) прибыло в p_0 }
begin receive (**ret**, c) ; $ret := ret + c$;
if *ret* = 1 then *Announce (** Правило I *)
end

Алгоритм 8.9 Алгоритм восстановления кредита.

Правило 2. Когда процесс становится пассивным, он посылает свой кредит инициатору.
В начальной конфигурации только инициатор активен и имеет положительный кредит, а именно 1, и *ret* = 0, что означает, что S1- S3 удовлетворz.ncz. Инвари-

ант должен поддержаться в течение вычисления; об этом заботятся следующие правила. Сначала, каждому основному сообщению при посылке нужно дать положительный кредит; к счастью, отправитель активен, и следовательно имеет положительный кредит.

Правило 3. Когда активный процесс *p* посылает сообщение, кредит разделяется между *p* и сообщением.

Процессу при его активизации нужно дать положительный кредит; к счастью, сообщение, которое он получает при этом, содержит положительный кредит.

Правило 4. При активизации процесса ему дается кредит активизирующего сообщения.

Единственная ситуация, не охваченная этими правилами - получение основного сообщения уже активным процессом. Процесс уже имеет положительный кредит, следовательно не нуждается в кредите сообщения, чтобы удовлетворить S3; однако, кредит не может быть разрушен, поскольку это привело бы вело бы к нарушению S1 . Процесс получающий сообщение может обращаться с кредитом двумя различными способами, оба порождают правильные алгоритмы.

Правило 5a. Когда активный процесс получает основное сообщение, кредит этого
сообщения посылается инициатору.

Правило 5b. Когда активный процесс получает основное сообщение, кредит того сообщения добавляется к кредиту процесса.

Алгоритм дается как Алгоритм 8.9. В этом алгоритме, принимается, что каждый процесс знает имя инициатора (по крайней мере, когда он сначала становится пассивным) и алгоритм использует правило 5b. Когда инициатор становится пассивным, он посылает сообщение самому себе.

Теорема 8.11 Алгоритм восстановления кредита (Алгоритм 8.9) - правильный алгоритм обнаружения завершения.

Доказательство. Алгоритм осуществляет правила 1-5, из чего следует, что $S_1 \wedge S_2 \wedge S_3$ инвариант, где

$S_1 \equiv 1 = (\Sigma_{(\mathbf{mes},\, c)}\, c)+ (\Sigma_{p \in P}\, cred_p)+ (\Sigma_{(\mathbf{ret},\, c)}\, c)+ret$

$S_2 \equiv \forall(\ \mathbf{mes},\ c\ )$ в процессе передачи : $c > 0$

$S_3 \equiv \forall p \in P$: $(state_p = passive \Rightarrow cred_p = 0) \wedge (state_p = active \Rightarrow cred_p > 0)$.

Завершение обнаружено, когда *ret* = 1, который вместе с инвариантом означает, что **term** выполняется.

Чтобы показать живучесть, заметим что после завершения не происходят никакие основные действия, следовательно происходят только получения сообщений (**ret**, c), и каждое получение уменьшает на 1 число сообщений находящихся в процессе передачи. Следовательно, алгоритм достигает конечной конфигурации. В такой конфигурации не имеется никаких основных сообщений (соглачно **term**), $cred_p = 0$ для всех *p* (согласно **term** и S3), и не имеется никакого сообщения (**ret**, c) (конфигурация конечная). Следовательно, *ret* = 1(из S1), и завершение обнаружено. □

Если осуществляется правило 5a, число управляющих сообщений равняется числу основных сообщений плюс один. (Здесь мы также считаем сообщение, посланное p_0 самому себе после того, как он стал пассивным.) Если осуществляется правило 5b, число управляющих сообщений равняется числу внутренних событий в основном вычислении плюс один, не больше числа основных сооб-

щений плюс один. Казалось бы, что правило 5b более предпочтительно с точки зрения сложности по сообщениям управляющего алгоритма. Иная ситуация возникает при рассмотрении битовой сложности. Согласно правилу 5a, каждое значение кредита в системе кроме *ret* - отрицательная степень 2 (i.e .., 2^{-i} для некоторого целого числа *i*). Представление кредита отрицательным логарифмом уменьшает число передаваемых бит.
Алгоритм восстановления кредита - единственный алгоритм в этой главе, который требует включения дополнительной информации (а именно, кредита) в основные сообщения. Добавление информации к основным сообщениям называется piggybacking. Если piggybacking не желателен, кредит сообщения может быть передан в управляющем сообщении, посланном сразу после основного сообщения. (Алгоритм следующего подраздела также требует piggybacking, если это осуществлено, используя логические часы Лампорта.)
Проблема может возникнуть, если кредиты (сообщений и процессов) хранятся в установленном числе бит. В этом случае существует самый маленький положительный кредит, и не возможно разделить это количество кредита на два. Когда кредит с наименьшим возможным значением нужно разделить, основное вычисление приостанавливается на время пока процесс не приобретет дополнительный кредит от инициатора. Инициатор вычитает этот кредит из *ret* (*ret*, может получиься в результате отрицательным) и передает его процессу, который возобновляет основное вычисление после получения. Это увеличение кредита вызывает блокирование основного вычисления, что противоречит требованию невмешательства алгоритма обнаружения завершения в основное вычисление. К счастью, эти действия редки.

8.4.2 Решения, использующие временные пометки

Этот подраздел обсуждает решения проблемы обнаружения завершения, основанной на использовании временных пометок. Предполагается, чтопроцессы оборудованы для этой цели часами (Подраздел 2.3.3); могут использоваться часы аппаратных средств ЭВМ также как логические часы Лампорта (Подраздел 2.3.3). Принцип обнаружения был предложен Rana [Ran83].
Подобно решениям Подраздела 8.3.3, решение Рана основано на локальном предикате *quiet(p)* для каждого процесса *p*, где
$quiet(p) \Rightarrow state_p = passive \wedge$ в не передаются соощения посланные процессом *p*, что означаетс, что$(\forall$p *quiet(p))* $\Rightarrow$ ***term.*** Как и прежде, *quiet* определяется как $quiet(p) \equiv (state_p = passive \wedge unack_p = 0)$.
Алгоритм стремится проверить для некоторого момента времени t, все ли процессы *quiet* ; при положительном ответе следует заключение о завершении. Реализуется это волной, которая опрашивает каждый процесс был ли он *quiet* в тот момент или позже; процесс, который не был *quiet*, не отвечает на сообщения волны, эффективно гася волну.

var $state_p$ *: (active, passive)* ;
θ_p *:* integer **init 0** ; (* Логические часы *)
$unack_p$ *:* integer **init** 0 ; (* Число сообщений оставшихся без ответа*)
qt_p *:* integer **init** 0 ; (* Время последнего перехода на *quiet **)

S$_p$*: { state*$_p$ *= active }*
begin θ_p := θ_p + 1 ; send (**mes,** θ_p*) ', unack* $_p$:= *unack* $_p$ + 1 **end**

R$_p$: { Сообщение (**mes**, θ*)* из *q* прибыло в *p }*
begin receive (**mes,** θ*) ;* θ_p := max(θ_p, θ*)* + 1 ;
send (**ack,** θ_p *)* to *q* ; *state*$_p$ *:= active*
end

I$_p$: { *state*$_p$ *= active }*
begin θ_p := θ_p + 1 ; *state*$_p$ *:= passive* ;
if *unack*$_p$ = 0 **then** (* p становится *quiet* *)
begin *qt*$_p$:= θ_p ; send (**tok**, θ_p , *qt*$_p$ *, p)* to *Next*$_p$ **end**
end

A$_p$: { Подтверждение (**ack,** θ) прибыло в *p* }
begin receive (**ack,** θ *) ;* θ_p :== max(θ_p, θ*)* + 1 ;
unack$_p$ *:= unack*$_p$ *- 1* ;
if *unack*$_p$ *= 0* **and** *state*$_p$ *= passive* **then** (* p сиановится *quiet* *)
begin *qt*$_p$:= θ_p ; send (**tok**, θ_p, *qt*$_p$ *,p)* to *Next*$_p$ **end**
end

T$_p$: { Маркер *(* **tok,** θ*, qt, q)* прибывает в p }
begin receive (**tok,** θ*, qt, q}* ; θ_p := *max(*θ_p*,* θ*)* + 1 ;
if *quiet(p)* **then**
if p = *q* **then** *Announce*
else if *qt* ≥ *qt*$_p$ **then** send *(***tok** , θ_p *, qt, q) to Next*$_p$
end

Алгоритм 8.10 алгоритм RANA.

В отличие от решений в Разделе 8.3 посещение волной процесса *p* не затрагивает переменные процесса *p,* используемые для обнаружения завершения. (Посещение волны может затрагивать переменные алгоритма волны и, если используются логические часы Лампорта, часы процесса.) В следствии этого правильное действие алгоритма не нарушается параллельным выполнением нескольких волн.
Алгоритм Рана децентрализован; все процессы выполняют один и тот же алгоритм обнаружения. Децентрализованный алгоритм также можно получить обеспечив алгоритм Подраздела 8.3.4 децентрализованным алгоритмом волны. В решении Рана процессы могут начинать частные волны, которые бегут одновременно.
Процесс *p*, когда становится *quiet*, сохраняет время *qt*$_p$, в которое это случается, и начинает волну, чтобы проверить, все ли процессы *quiet* со времяни *qt*$_p$. Если дело обстоит так, завершение обнаружено. Иначе, будет иметься процесс, который становится *quiet* позже, и новая волна будет начата. Алгоритм 8.10 испольлзует этот принцип, используя часы Лампорта и используя кольцевой алгоритм как волновой алгоритм.
Теорема 8.12 *Алгоритм Рана (Алгоритм 8.10) - правильный алгоритм обнару-*

жения завершения.

Доказательство. Чтобы доказывать живучесть алгоритма, предположим что **term** сохраняется в конфигурации γ, в которой все еще передаеются подтверждения. Тогда происходят только действия $\mathbf{A}_p$ and $\mathbf{T}_p$. Поскольку каждое действие $\mathbf{A}_p$ уменьшает на 1 число сообщений (**ack,** θ *)* находящихся в процессе передачи, происходит только конечное число этих шагов. Каждый процесс становится *quiet* не более одного раза; следовательно маркер генерируется не более N раз, и каждый маркер передается не более N раз. Следовательно за $a + N^2$ шагов алгоритм обнаружения завершения достигает когнечной конфигурации δ, в которой **term** все еще сохраняется.

Пусть p_0 процесс с максимальным значением qt в δ, то есть, в конечной конфигурации $qt_{P0} \geq qt_P$ для каждого процесса p. Когда p_0 стал *quiet* в последний раз (то есть, во время qt_{P0}), он передает маркер (**tok**,qt_{P0} ,qt_{P0} ,p_0 *).*Этот маркер проходит полный круг по кольцу и возвращается к p_0. Действительно, каждый процесс p должен быть *quiet* и удовлетворять $qt_P \leq qt_{P0}$, когда он получает этот маркер. Если нет, p установил бы часы на значение большее чем qt_{P0} после получения маркера и стал бы *quiet* позже чем p_0, противореча выбору p_0. Тогда маркер возвратился к p_0, p_0 был еще *quiet*, и следовательно вызвал алгоритм объявления.

Чтобы доказавать безопасность алгоритма, предположим что p_0 вызвал алгоритм объявления; это произойдет, когда p_0 *quiet* и получает назад макер (**tok**,qt_{P0} ,qt_{P0} ,p_0 *),* который был отправлен всеми процессами. Доказательство приводит к противоречию. Предположим, что **term** не сохраняется, когда p_0 обнаруживает завершение; это означает, что имеется процесс p такой, что p не *quiet*. В этом случае p стал не *quiet* после отправления маркера p_0; действительно, p был *quiet*, когда он отправил этот маркер. Пусть q первый процесс, который стал не *quiet* после отправления маркера *(***tok***, θ, qt,* p_0*).* Это означает, что q был активизирован при получении сообщения от процесса, скажем r, который еще не отправил маркер процесса p_0.

(Иначе r стал бы не *quiet* после отправления маркера, но прежде, чем q стал не *quiet*, что противоречит выбору q.)

Теперь после отправления маркера $\theta_q > qt_{P0}$ продолжает сохраняться. Это означает, что подтверждение для сообщения, которое сделало q не *quiet*, послается r с временной пометкой $\theta_0 > qt_{P0}$. Таким образом, когда r стал *quiet*, после получения этого подтверждения, $\theta_r > qt_{P0}$ сохраняется, и следовательно $qt_r > qt_{P0}$ сохраняется, когда r получает маркер. Согласно алгоритму r не отправляет маркер; т.о. мы пришли к противоречию. □

Описание этого алгоритма, который не полагался на кольцевую топологию, было представлено Huang [Hua88].

Упражнения к Главе 8

Раздел 8.1

Упражнение 8.1 Оаарактеризуйте активные и пассивные состояния Алгоритма А.2. Где эти состояния находятся в Алгоритме А.1?

Раздел 8.2

Сложность по времени алгоритма обнаружения завершения определена как число единиц времени в худшем случае (согласно идиализационным предположениям Определения 6.31) между завершением основного вычисления и вызова алгоритмя объявления.

Упражнение 8.2. Что является сложностью по времени Dijkstra-Scholten алгоритма?

Упражнение 8.3. Shavit-Francez алгоритм применяется в произвольной сети с уникальными идентификаторами, и для того, чтобы минимизировать накладные расходы на управляющие сообщения Gallager-Humblet-Spira алгоритм используется как алгоритм волны. Сложность времени обнаружения - $\Omega(NlogN)$.
Можите ли вы улучшить сложность по времени до *0 (N)* за счет обмена *0 (N)* дополнительных управляющих сообщений?

Раздел 8.3

Упражнение 8.4. Почему предикат P_0 в выводе алгоритма Dijkstra-Feijen-Van Gasteren, не принимает значение ложь, если p_j активизирован p_i, где $j \leq t$ или $i > t$?

Упражнение 8.5 Покажите, что для каждого *m* существует основное вычисление, которое использует *m* сообщений и заставляет алгоритм Dijkstra-Feijen-Van Gasteren использовать $m(N - 1)$ управляющих сообщений.

Раздел 8.4

Упражнение 8.6. Какие модификации должны быть сделаны в Алгоритме 8.9, чтобы осуществить правило 5a алгоритма восстановления кредита, вместо правила 5b?

Упражнение 8.7 В алгоритме Рана принято, что процессы имеют идентификаторы. Теперь примите вместо этого, что процессы анонимны, но имеют средства посылки сообщений их преемникам в кольце, и что число процессов известен. Измените Алгоритм 8.10, чтобы работать согласно этому предположению.

Упражнение 8.8 Покажите правильность алгоритма Рана (Алгоритм 8.10) из инварианта алгоритма.

13 Отказоустойчивость в Асинхронных Системах

Эта глава рассматривает разрешимость проблем решения в асинхронных распределенных системах. Результаты организованы вокруг фундаментального результата Фишера, Линча и Патерсона [FLP85], представленного в Разделе 13.1. Сформулированный как доказательство невозможности для класса алгоритмов решения, результат можно также трактовать как список предположений, которые совместно исключают разрешение проблем решения. Смягчение этих предположений позволяет получить практические решения различных проблем, как показано в последующих разделах. Дальнейшее обсуждение см. в Подразделе 13.1.3.

13.1 Невозможность согласия

В этом разделе доказывается фундаментальная теорема Фишера, Линча и Патерсона [FLP85] об отсутствии асинхронных, детерминированных 1-аварийно устойчивых протоколов согласия. Результат показан рассуждением, включающим в себя законные последовательности выполнения алгоритмов. Сначала введем обозначения (вдобавок к введенным в Разделе 2.1) и укажем элементарные результаты, которые окажутся полезными далее.

13.1.1 Обозначения, Определения, Элементарные Результаты

Последовательность событий $\sigma = (e_1,...,e_k)$ *применима* в конфигурации γ, если e_1 применима в γ, e_2 - в $e_1(\gamma)$, и т.д. Если δ - результирующая конфигурация, то, чтобы явно указать события, ведущие от γ к δ, мы пишем $\gamma \to^{\sigma} \delta$ или $\sigma(\gamma) = \delta$. Если $S \subseteq \mathrm{P}$ и σ содержит только события в процессах из S, мы также пишем $\gamma \to_S \delta$.

Утверждение 13.1 *Пусть последовательности* σ_1 *и* σ_2 *применимы в конфигурации* γ*, и пусть ни один процесс не участвует одновременно в* σ_1 *и* σ_2*, тогда* σ_2 *применима в* $\sigma_1(\gamma)$*,* σ_1 *применима в* $\sigma_2(\gamma)$*, и* $\sigma_2(\sigma_1(\gamma)) = \sigma_1(\sigma_2(\gamma))$.

Доказательство. Следует из повторного применения Теоремы 2.19. □

Процесс p имеет входную переменную x_p, доступную только для чтения, и выходной регистр однократной записи y_p с начальным значением b. Входная конфигурация полностью определяется значением x_p для каждого процесса p. Процесс p может *принять решение* о значении (обычно 1 или 0) записью его в y_p; начальное значение b *не является* значением решения. Предполагается, что корректный процесс исполняет бесконечно много событий при законном выполнении; в крайнем случае, процесс всегда может выполнять (возможно пустое) внутреннее событие.

Определение 13.2 *t-аварийное законное выполнение - выполнение, в котором по меньшей мере N-t процессов исполняют бесконечно много событий, и каждое сообщение, посылаемое корректному процессу, получается. (Процесс корректен, если исполняет бесконечно много событий.)*

Максимальное число сбойных процессов, с которым может справиться алгоритм, называется *способностью восстановления* алгоритма, и всегда обозначается t. В этом разделе демонстрируется невозможность существования асинхронного, детерминированного алгоритма со способностью восстановления 1.

Определение 13.3 *1-аварийно-устойчивый алгоритм согласия - алгоритм, удовлетворяющий следующим трем требованиям.*

(1) **Завершение.** *В каждом 1-аварийном законном исполнении, все корректные процессы принимают решение.*

(2) **Согласованность**. *Если в достижимой конфигурации* $y_p \neq b$ *и* $y_q \neq b$ *для корректных процессов* p *и* q, *то* $y_p = y_q$.

(3) **Нетривиальность**. *Для* $v = 0$ *и для* $v = 1$ *существуют достижимые конфигурации, в которых для некоторого* p $y_p = v$.

Для $v = 0,1$ конфигурация называется *v-решенной*, если для некоторого p $y_p = v$; конфигурация называется *решенной*, если она 0-решенная или 1-решенная. В v-решенной конфигурации какой-нибудь процесс принял решение v. Конфигурация называется *v-валентной*, если все решенные конфигурации, достижимые из нее, v-решенны. Конфигурация называется *бивалентной*, если из нее достижимы как 0-валентные, так и 1-валентные конфигурации, и *унивалентной*, если она либо 1-валентная, либо 0-валентная. В унивалентной конфигурации, хотя никакое решение не было обязательно принято никаким процессом, окончательное решение уже неявно определено.

Конфигурация γ t-устойчивого протокола называется *развилкой*, если существует множество T (самое большее) из t процессов и конфигурации γ_0 и γ_1 такие, что $\gamma \to_T \gamma_0$, $\gamma \to_T \gamma_1$, и γ_v v-валентна. Неформально, γ - развилка, если подмножество из t процессов может добиться 0-решенности так же, как и 1-решенности. Следующее утверждение формально фиксирует, что в любой момент оставшиеся процессы должна вынести аварию самое большее t процессов.

Утверждение 13.4 *Для каждой достижимой конфигурации t-устойчивого алгоритма и каждого подмножества S по меньшей мере из N-t процессов существует решенная конфигурация* σ *такая, что* $\gamma \to_S \delta$.

Доказательство. Пусть γ и S удовлетворяют условию и рассмотрим выполнение, которое достигает конфигурации γ и содержит бесконечно много событий в каждом процессе из S впоследствии (и никаких шагов процессов не из S). Это выполнение - t-аварийное законное, и процессы в S корректны; следовательно они достигают решения □

Лемма 13.5 *Достижимой развилки не существует.*

Доказательство. Пусть γ - достижимая конфигурация и T - подмножество самое большее из t процессов.

Пусть S будет дополнением T, т.е., $S = \mathrm{P} \setminus T$. В S по меньшей мере N-t процессов, следовательно существует решенная конфигурация σ такая, что $\gamma \to_S \delta$ (Утверждение 13.4). Конфигурация δ либо 0-, либо 1-решенная; положим, что она 0-решенная.

Сейчас будет показано, что $\gamma \to_T \gamma'$ ни для какой 1-валентной γ'; пусть γ' - любая такая конфигурация, что $\gamma \to_T \gamma'$. Так как шаги в T и S заменяются

(Утверждение 13.1), есть конфигурация δ', которая достижима и из δ, и из γ'. Так как δ - 0-решенна, то и δ' - тоже, что показывает не 1-валентность γ'. □

13.1.2 Доказательство невозможности

Сначала, используя нетривиальность проблемы, покажем что существует бивалентная начальная конфигурация (Лемма 13.6). Вподедствии будет показано, что начиная с бивалентной конфигурации, каждый доступный шаг можно исполнять без перехода в унивалентную конфигурацию (Лемма 13.7). Этого достаточно, чтобы показать невозможность алгоритмов согласия (Теорема 13.8). В дальнейшем, пусть A - 1-аварийно-устойчивый алгоритм согласия.

Лемма 13.6 *Для A существует бивалентная начальная конфигурация.*

Доказательство. Так как A нетривиален (Определение 13.3), то есть достижимые 0- и 1-решенные конфигурации; пусть δ_0 и δ_1 - начальные конфигурации такие, что v-решенная конфигурация достижима из δ_v.

Если $\delta_0 = \delta_1$, эта начальная конфигурация бивалентна и результат имеет силу. Иначе, есть начальные конфигурации γ_0 и γ_1 такие, что v-решенная конфигурация достижима из γ_v, и γ_0 и γ_1 различаются входом одного процесса. Действительно, рассмотрим последовательность начальных конфигураций, начинающуюся с δ_0 и заканчивающуюся δ_1, в которой каждая следующая начальная конфигурация отличается от предыдущей в одном процессе. (Эта последовательность получается инвертированием входных битов одного за другим.) Из первой конфигурации в последовательности, δ_0, достижима 0-решенная конфигурация, и из последней, δ_1, достижима 1-решенная конфигурация. Так как решенная конфигурация достижима из каждой начальной конфигурации, описанные γ_0 и γ_1 можно найти в последовательности. Пусть p - процесс, в котором γ_0 и γ_1 различны.

Рассмотрим законное выполнение, начинающееся с γ_0, в которой p не делает шагов; это выполнение 1-аварийно законное и следовательно достигает решенной конфигурации β. Если β 1-решенная, γ_0 бивалентна. Если β 0-решенная, заметьте, что γ_1 отличается от γ_0 только в p, а p не делает шагов в выполнении; следовательно β достижима из γ_1, что показывает бивалентность γ_1. (Более точно, конфигурация β' достижима из γ_1, где β' отличается от β только в состоянии p; следовательно β' 0-решенная.) □

Чтобы поñòðîèòь законное выполнение без принятия решения мы должны показать, что каждый процесс может сделать шаг, и что каждое сообщение может быть получено не обуславливая принятие решения. Пусть *шаг s* обозначает получение и обработку отдельного сообщения или спонтанное действие (внутреннее или посылки) отдельного процесса. Состояние процесса, делающего шаг, может привести к различным событиям. Прием сообщения применим, если оно в пути, и спонтанный шаг всегда применим.

Лемма 13.7 *Пусть γ - достижимая бивалентная конфигурация и s - применимый шаг для процесса p в γ. Существует последовательность событий σ такая, что s применим в $\sigma(\gamma)$, и $s(\sigma(\gamma))$ бивалентна.*

Доказательство. Пусть C - множество конфигураций, достижимых из γ без применения s, т.е., C = $\{\sigma(\gamma)$: s не происходит в $\sigma\}$; s применим в каждой конфигурации C (напомним, что s - шаг, а не отдельное событие).

В C есть конфигурации α_0 и α_1 такие, что из $s(\alpha_v)$ достижима v-решенная конфигурация. Чтобы убедится в этом, заметим, что, т.к. γ бивалентна, из нее достижимы v-решенные конфигурации β_v для v =0,1. Если $\beta_v \in C$ (т.е. для достижения решенной конфигурации s не применялся), заметим, что $s(\beta_v)$, тем не менее, v-решенная, поэтому выберем $\alpha_v = \beta_v$. Если $\beta_v \notin C$ (т.е. для достижения решенной конфигурации s применялся), выберем α_v как конфигурацию, из которой применялся s.

Если $\alpha_0 = \alpha_1$, $s(\alpha_0)$ - искомая бивалентная конфигурация. Предположим, что $\alpha_0 \neq \alpha_1$, и рассмотрим конфигурации на путях от γ до α_0 и α_1. Две конфигурации на этих путях называются соседними, если одна получается из другой за один шаг. Так как 0-решенная конфигурация достижимаа из $s(\alpha_0)$ и 1-решенная конфигурация достижима из $s(\alpha_1)$, то

(1) на путях есть конфигурация γ' такая, что $s(\gamma')$ бивалентна; или

(2) есть соседи γ_0 и γ_1 такие, что $s(\gamma_0)$ 0-валентна и $s(\gamma_1)$ - 1-валентна.

В первом случае $s(\gamma')$ - искомая бивалентная конфигурация и лемма доказана. Во втором случае, одна конфигурация из γ_0 и γ_1 - развилкой, что является противоречием. Действительно, предположим, что γ_1 получена за один шаг из γ_0, т.е., $\gamma_1 = e(\gamma_0)$ для события e в процессе q. Теперь $s(e(\gamma_0))$ - это $s(\gamma_1)$ и, следовательно, 1-валентна, но $e(s(\gamma_0))$ не 1-валентна, т.к. $s(\gamma_0)$ уже 0-валентна. Итак, e и s не заменяются, что подразумевает (Теорема 2.19) , что p = q, но тогда достижимая конфигурация γ_0 удовлетворяет $\gamma_0 \rightarrow_p s(\gamma_0)$ и $\gamma_0 \rightarrow_p s(e(\gamma_0))$. Так как первая 0-валентна, а последняя 1-валенттна, γ_0 - развилка, что является противоречием. □

Теорема 13.8 *Асинхронного, детерминированного, 1-аварийно-устойчивого алгоритма согласия не существует.*

Доказательство. Если предположить, что такой алгоритм существует, можно построить законное выполнение без принятия решения, начиная с бивалентной начальной конфигурации γ_0.

Когда построение дойдет до конфигурации γ_i, выберем в качестве s_i применимый шаг, который был применим самое большое число раз. По предыдущей лемме, выполнение можно расширить так, что исполняется s_i и достигается бивалентная конфигурация γ_{i+1}.

Такое построение дает бесконечное законное выполнение, в котором все процессы корректны, но решение никогда не будет принято. □

13.1.3 Обсуждение

Вывод утверждает, что не существует асинхронных, детерминированных, 1-аварийно-устойчивых алгоритмов решения для проблемы согласия; это исключает алгоритмы для класса нетривиальных проблем. (см. Подраздел 12.2.2).

К счастью, некоторые предположения, лежащие в основе результата Фишера, Линча и Патерсона, можно выразить явно, и результат, как оказывается, очень чувствителен к ослаблению любого из них. Несмотря на вывод о невозможности, многие нетривиальные проблемы имеют решения, даже в асинхронных системах и где процессы могут отказывать.

(1) *Ослабленная модель отказов*. Раздел 13.2 рассматривает модель отказов изначально-мертвых процессов, которая слабее, чем модель аварий, и в этой модели согласие и выборы детерминированно достижимы.

(2) *Ослабленная координация*. Раздел 13.3 рассматривает проблемы, которые требуют менее тесной координации между процессами, чем согласие, и показывает, что некоторые из этих проблем, включая переименование, разрешимы в модели аварий.

(3) *Рандомизация*. Раздел 13.4 рассматривает протоколы с уравненными вероятностями, где требование завершения достаточно ослаблено, чтобы обеспечить решения даже при присутствии Византийских отказов.

(4) *Слабое требование завершения*. Раздел 13.5 рассматривает другое ослабление требования завершения, а именно где разрешение требуется только когда данный процесс корректен; здесь также возможны Византийско-устойчивые решения.

(5) *Синхронность*. Влияние синхронности изучается далее в Главе 14.

Возможны довольно тривиальные решения, если одно из трех требований Определения 13.3 просто опущено; см. Упражнение 13.1. Исключение предположения (неявно использованного в доказательстве Леммы 13.6) о том, что возможны все комбинации входов, изучается в Упражнении 13.2.

13.2 Изначально-мертвые Процессы

В модели изначально-мертвых процессов, ни один процесс не может отказать после исполнения события, следовательно, при законном выполнении каждый процесс исполняет либо 0, либо бесконечно много событий.

Определение 13.9 *t-изначально-мертвых законное выполнение - выполнение, в котором по крайней мере N-t процессов активны, каждый активный процесс исполняет бесконечно много событий, и каждое сообщение, посылаемое корректному процессу, принимается.*

В t-изначально-мертвых-устойчивом алгоритме согласия, каждый корректный процесс принимает решение в каждом t-изначально-мертвых законном выполнении. Согласованность и нетривиальность определяются так же, как в модели аварий.

---

var $Succ_p$, $Alive_p$, $Rcvd_p$: sets of processes **init** 0;

begin shour <**name**, p>;
(* т.е.: **forall** $q \in$ P **do** send<**name**, p> to q *)
while $\#Succ_p$ < L
do begin receive<**name**, q>; $Succ_p := Succ_p \cup \{q\}$ **end**;
shout<**pre**, p, $Succ_p$>;
$Alive_p := Succ_p$;
while $Alive_p \not\subset Rcvd_p$
do **begin** receive<**pre**, q, $Succ_p$>;
$Alive_p := Alive_p \cup Succ \cup \{q\}$;
$Rcvd_p := Rcvd_p \cup \{q\}$;
end;
Вычислить узел в G
end

АЛГОРИТМ 13.1 ВЫЧИСЛЕНИЕ УЗЛА.

Так как процессы не отказывают после посылки сообщения, то для процесса безопасно ждать приема сообщения от p, зная, что p уже послал по меньшей мере одно сообщение. Будет показано, что проблемы согласия и выборов разрешимы в модели изначально-мертвых, пока отказывает меньшинство процессов (t < N/2). Большее число изначально-мертвых процессов не допускается (см. Упражнение 13.3).

Соглашение о подмножестве корректных процессов. Сначала представляется алгоритм Фишера, Линча и Патерсона [FLP], с помощью которого каждый из корректных процессов вычисляет *одну и ту же* совокупность корректных процессов. Способность восстановления этого алгоритма $\lfloor (N-1)/2 \rfloor$; пусть L равно $\lceil N+1/2 \rceil$, и заметим, что корректных процессов по меньшей мере L. Алгоритм работает в два этапа; см. Алгоритм 13.1.

Заметим, что процессы посылают сообщения сами себе; это делается во многих устойчивых алгоритмах и облегчает анализ. Здесь и в дальнейшем, операция “shout<**mes**>” означает

forall $q \in$ P **do** send<**mes**> to q.

Эти процессы строят ориентированный граф G, “выкрикивая” свой идентификатор (в сообщении <**name**, p>) и ожидая приема L сообщений. Так как кор-

ректных процессов по меньшей мере L, каждый корректный процесс получает достаточно много сообщений для завершения этой части. Преемники p в графе G - вершины q, из которых p получил сообщение <**name**, q>.

Изначально-мертвый процесс не получал и не посылал никаких сообщений, следовательно он формирует изолированную вершину в G; у корректного процесса есть L преемников, следовательно, он не изолирован. Узел - это сильносвязный компонент без исходящих дуг, содержащий по меньшей мере две вершины. В G есть узел, содержащий корректные процессы, и, так как каждый корректный процесс имеет степень выхода L, этот узел имеет размер по меньшей мере L. В результате, так как $2L > N$, существует ровно один узел; назовем его K. В конечном счете, так как корректный процесс p имеет L преемников, по меньшей мере один из них принадлежит K, что означает, что все процессы в K - потомки p.

Следовательно, на втором этапе алгоритма, процессы образуют индуцированный подграф графа G, содержащий по меньшей мере их потомков, получая множество преемников от каждого процесса, который, как они знают, корректен. Так как процессы не отказвыают после посылки сообщения, на этом этапе не возникает тупика. Действительно, p ждет сообщения от q только если на первом этапе некоторый процесс получил сообщение <**name**, q>, показывающее на корректность q.

После завершения Алгоритма 13.1 каждый корректный процесс получил набор преемников каждого из своих потомков, позволяя таким образом вычислить уникальный узел в G.

Согласие и выбор. Поскольку все корректные процессы договариваются об узле корректных процессов, избрать процесс теперь тривиально; избирается процесс с самым большим идентификатором в K. Теперь так же просто достигнуть согласия. Каждый процесс вещает, вместе со своими преемниками, свой вход (x). После вычисления K, процессы принимают решение о значении, которое является функцией совокупности входов в K (например, значение, которое встречается наиболее часто, ноль в случае ничьей).

Алгоритмы узел-соглашения, согласия, и выбора обменивают $O(N^2)$ сообщениями, где сообщение может содержать список из L имен процессов. Были предложены более эффективные алгоритмы выбора. Итаи и другие [IKWZ90] привели алгоритм, использующий $O(N(t + \log N))$ сообщения и показали, что это является нижней границей. Масузава и другие [MNHT89] рассмотрели проблему для клик с чувством направления и предложили алгоритм $O(Nt)$ сообщений, который также является оптимальным.

Любой алгоритм выбора, выбирая корректный процесс в качестве лидера также решает проблему согласия; лидер вещает свой вход и все корректные процессы принимают решения по нему. Следовательно, вышеупомянутые верхние границы остаются в силе также для проблемы согласия для изначально-мертвых процессов. В модели аварий, однако, наличие лидера не помогает в решении проблемы согласия; сам лидер может отказать до вещания своего входа. Кроме того, проблема выбора не разрешима в модели аварийного отказа, что будет показано в следующем разделе.

13.3 Детерминированно Достижимые Случаи

Проблема согласия, изучаемая до сих пор, требует, чтобы каждый процесс принял решение об одном и том же значении; этот раздел изучает разрешимость задач, которые требуют менее близкой координации между процессами. В Подразделе 13.3.1 представлено решение практической проблемы, а именно, переименование совокупности процессов в малом пространстве имен. В Подразделе 13.3.2 выведенные ранее результаты о невозможности расширяются, чтобы охватить больший класс проблем решения.

Распределенная задача описывается множествами возможных входных и выходных значений X и D, и (возможно частичным) отображением

$$T: X^N \to \mathrm{P}(D^N).$$

Интерпретация отображения T: если вектор $\bar{x} = (x_1, \ldots, x_N)$ описывает вход процессов, то $T(\bar{x})$ - набор допустимых выходов алгоритма, описанный как вектор решения $\bar{d} = (d_1, \ldots, d_N)$. Если T - частичная функция, допустима не каждая комбинация входных значений.

Определение 13.10 *Алгоритм является t-аварийно устойчивым решением для задачи T если он удовлетворяет следующим утверждениям.*

(1) **Завершение.** *В каждом t-аварийно законном выполнении, все корректные процессы принимают решение.*

(2) **Непротиворечивость**. *Если все процессы корректны, вектор решения* $\bar{d}$ *находится в* $T(\bar{x})$.

Условие непротиворечивости подразумевает, что в выполнении, где подмножество процессов принимает решение, частичный вектор решений всегда можно расширить до вектора в $T(\bar{x})$. Множество D_T обозначает совокупность всех векторов выхода, то есть, диапазон T.

(1) *Пример: согласие*. Проблема согласия требует, чтобы все решения были равны, т.е.,

$$D_{согл} = \{(0,0,\ldots,0),(1,1,\ldots,1)\}.$$

(2) *Пример: выбор*. Проблема выбора требует, чтобы один процесс принял решение 1, а другие 0, т.е.,

$$D_{выб} = \{(1,0,\ldots,0),(0,1,\ldots,0),(0,0,\ldots,1)\}.$$

(3) *Пример: приблизительное соглашение*. В проблеме ε *-приблизительного соглашения* каждый процесс имеет действительное входное значение и принимает решение о действительном выходном значении. Максимальное различие между двумя значениями выхода самое большее e, и выходы должны быть заключены между двумя входами.

$$D_{прибл} = \{(d_1, \ldots, d_N): \max(d_i) - \min(d_i) \le \varepsilon\}.$$

(4) *Пример: переименование*. В *проблеме переименования* каждый процесс имеет отдельный идентификатор, который может браться из произвольно большой области. Каждый процесс должен принять решение о новом имени, из меньшей области 1, ..., K, так, чтобы все новые имена различались.

$$D_{переименования} = \{(d_1,\ldots,d_N): i \neq j \Rightarrow d_i \neq d_j\}.$$

В *сохраняющей порядок* версии проблемы переименования, новые имена должны сохранять порядок старых имен, то есть, $x_i < x_j \Rightarrow d_i < d_j$.

13.3.1 Разрешимая Проблема: Переименование

В этом подразделе будет представлен алгоритм для переименования Аттийи и других [ABND+90]. Алгоритм допускает до $t < N/2$ аварий (t - параметр алгоритма) и осуществляет переименование в пространстве имен размера $K = (N - t/2)(t+1)$.

Верхняя граница t. Мы сначала покажем, что никакой алгоритм переименования не сможет выдержать N/2 или большее количество сбоев; фактически, почти все аварийно-устойчивые алгоритмы имеют ограничение t<N/2 на число неисправностей, и приведенное ниже доказательство можно адаптировать к другим проблемам.

Теорема 13.11 *При* $t \geq N/2$ *алгоритмов для переименования не существует.*

Доказательство. Если $t \geq N/2$, можно сформировать две непересекающихся группы процессов S и T размера N-t. Вследствие возможности сбоя t процессов, группа должна уметь принимать решение "самостоятельно", то есть, без взаимодействия с процессами вне группы (см. Утверждение 13.4). Но тогда группы могут *независимо* достичь решения в одиночном выполнении; сложный момент доказательства - показать, что эти решения могут быть взаимно противоречивы. Мы продолжим формальную аргументацию для случая переименования.

По утверждению 13.4, для каждой начальной конфигурации γ существует конфигурация δ_S такая, что все процессы в S приняли решение и $\gamma \rightarrow_S \delta_S$; то же справедливо и для T. Действие алгоритма внутри группы из N-t процессов определяет отношение векторов из N-t начальных идентификаторов к векторам из N-t новых имен. Так как начальное пространство имен неограниченно, и новые имена получаются из ограниченного диапазона, то имеются непересекающиеся входы, которые отображаются на перекрывающиеся выходы. То есть, имеются входные векторы (длины N-t) $\bar{u}$ и $\bar{v}$ такие, что $u_i \neq v_j$ для всех i, j, но им соответствуют векторы выхода $\bar{d}$ и $\bar{e}$ такие, что $d_i = e_j$, для некоторых i, j.

Некорректное выполнение теперь создается следующим образом. Начальная конфигурация γ имеет входы $\bar{u}$ в группе S и $\bar{v}$ в группе T; заметьте, что все начальные имена различны (начальные имена вне обеих групп могут быть выбраны произвольно). Пусть σ_T - последовательность шагов, которыми группа T

достигает, из γ, конфигурации δ_T, в которой процессы в T остановились (приняли решение) на именах $\bar{e}$. По утверждению 13.1, эта последовательность все еще применима в конфигурации γ_S, в которой процессы в S остановились на именах $\bar{d}$. В $\sigma_T(\gamma_S)$, два процесса остановились на одном и том же имени (потому что $d_i = e_j$), что указывает на противоречивость алгоритма. □

Далее предполагается, что t < N/2.

```
var     V_p: set of identities;
        c_p: integer;
begin   V_p:= {x_p} ;  c_p: = 0; shout <set, V_p>;
        while true
        do      begin  receive<set, V>
                       if V = V_p then
                                begin  c_p:= c_p + 1;
                                       if c_p = N − t and y_p = b then
                                       (* V_p впервые не изменялось: принять решение*)
                                       y_p:= (#V_p, rank(V_p, x_p))
                                end
                       else if V ⊆ V_p then
                                skip   (*Игнорировать "старую" информацию*)
                       else     (*новый вход; обновить Vp и начать счет заново*)
                                begin  if V_p ⊂ V then c_p:= 1 else c_p:= 0;
                                       V_p:= V_p ∪ V ; shout<set, V_p>
                                end
                end
end
```

АЛГОРИТМ 13.2 ПРОСТОЙ АЛГОРИТМ ПЕРЕИМЕНОВАНИЯ.

Алгоритм переименования. В алгоритме переименования (Алгоритм 13.2), процесс p сохраняет множество V_p входов процесса, которые p видел; первоначально, V_p содержит только x_p. Каждый раз, когда p получает множество входов, включая те, которые являются новыми для p, V_p расширяется новыми входами. При старте и каждый раз, когда V_p расширяется, p "выкрикивает" свое множество. Как видно, множество V_p растет только в течение выполнения, т.е., последующие значения V_p полностью упорядочиваются при включении, и, кроме того, V_p содержит самое большее N имен. Следовательно, процесс p "выкрикивает" свое множество самое большее N раз, что показывает, что алгоритм завершается и что сложность по сообщениям ограничена $O(N^3)$.

Далее, p считает (в переменной c_p) сколько раз он получил копии своего текущего множества V_p. Первоначально c_p равна 0, и увеличивается каждый раз, когда получается сообщение, содержащее V_p. Получение сообщения <**set**, V> может вызвать рост V_p, что требует сброса c_p. Если новое значение V_p равняется V (то есть, если V - строгое надмножество старого V_p), c_p устанавливается в 1, иначе в 0.

Говорят, что процесс p, достигает *устойчивого множества* V если c_p становится равным N-t, когда значение V_p - V. Другими словами, p получил в (N-t)-й раз текущее значение V Vp.

Лемма 13.12 *Устойчивые множества полностью упорядочены, то есть, если q достигает устойчивого множества V_1 и r достигает устойчивого множества V_2, то $V_1 \subseteq V_2$ или $V_2 \subseteq V_1$.*

Доказательство. Предположим, что q достигает устойчивого множества V_1 и r достигает устойчивого множества V_2. Это подразумевает, что q получил <set, V_1> от N-t процессов и r получил <**set**, V_2> от N-t процессов. Так как 2(N-t) > N, то есть по крайней мере один процесс, допустим p, от которого q получил <set, V_1> и r получил <set, V_2>. Следовательно, как V_1 так и V_2 - значения V_p, что означает, что одно включено в другое. □

Лемма 13.13 *Каждый корректный процесс по крайней мере однажды достигает устойчивого множества в каждом законном t-аварийном выполнении.*

Доказательство. Пусть p - корректный процесс; множество V_p может только расширяться, и содержит самое большее N входных имен. Следовательно, для V_p достигается максимальное значение V_0. Процесс p “выкрикивает” это значение, и сообщение <**set**, V_0> получается каждым корректным процессом, что показывает, что каждый корректный процесс в конечном счете имеет *надмножество* V_0.

Однако, это надмножество *не* строгое; иначе корректный процесс послал бы строгое надмножество V_0 к p, что противоречит выбору V_0 (как самого большого множества когда-либо побывавшего в p). Следовательно, каждый корректный процесс q имеет значение $V_q = V_0$ по крайней мере один раз при выполнении, и следовательно каждый корректный процесс посылает p сообщение <**set**, V_0> в течение выполнения. Все эти сообщения получаются при выполнении, и, поскольку V_p никогда не увеличивается за пределы V_0, они все подсчитываются и заставляют V_0 стать устойчивым в p. □

После достижения устойчивого множества V впервые, процесс p останавливается на паре (s, r), где s - размер V, и r - положение x_p в V. Устойчивый множест-

во было получено от N-t процессов, и следовательно содержит по крайней мере N-t входных имен, что показывает $N - t \le s \le N$. Положение в множестве размера s удовлетворяет $1 \le r \le s$. Число возможных решений, следовательно, $K = \sum_{s=N-t}^{N} s$, что равняется $(N - t/2)(t+1)$; если нужно, можно использовать фиксированное отображение пар на целые числа в диапазоне 1,..., K (Упражнение 13.5).

Теорема 13.14 *Алгоритм 13.2 решает проблему переименования с выходным пространством имен размера* $K = (N - t/2)(t+1)$.

Доказательство. Так как, в любом законном t-аварийном выполнении каждый корректный процесс достигает устойчивого множества, каждый корректный процесс останавливается на новом имени. Чтобы показать, что все новые имена различны, рассмотрим устойчивые множества V_1 и V_2, достигаемые процессами q и r соответственно. Если эти множества имеют различные размеры, решения q и r различны, потому что размер включается в решение. Если множества имеют один и тот же размер, то по Лемме 13.12, они равны; тогда q и r имеют различный ранг в множестве, что снова показывает, что их решения различны. □

Обсуждение. Заметьте, что процесс не завершает Алгоритм 13.2 после принятия решения о своем имени; он продолжает алгоритм, чтобы "помочь" другим процессам тоже принять решение. Аттийя и другие [ABND+90] показывают, что это необходимо, потому что алгоритм должен справиться с ситуацией, когда некоторые процессы настолько медленны, что выполняют первый шаг после того, как некоторые другие процессы уже приняли решение.

Простой алгоритм, представленный здесь не самый лучший в отношении размера пространства имен, используемого для переименования. Аттийя и другие [ABND+90] привели более сложный алгоритм, который назначает имена в диапазоне от 1 до N + t. Результаты следующего подраздела предполагают нижнюю границу размера нового пространства имен для аварийно-устойчивого переименования N + 1.

Аттийя и другие предложили также алгоритм для переименования, сохраняющего порядок. Он осуществляет переименование на целые числа в диапазоне от 1 до $K = 2^t \cdot (N - t + 1) - 1$, что, как было показано, является самым маленьким размером пространства имен, позволяющего t-аварийно-устойчивое переименование, сохраняющее порядок.

13.3.2 Расширение Результатов Невозможности

Результат о невозможности согласия (Теорема 13.8) был обобщен Мораном и Вольфшталом [MW87] для более общих проблем решения. *Граф решения* задачи T - граф $G_T = (V, E)$, где $V = D_T$ и

$$E = \{(\overline{d_1}, \overline{d_2}): \overline{d_1} \text{ и } \overline{d_2} \text{ отличаются точно в одном компоненте}\}.$$

Задача T называется *связной*, если G_T - связный граф, и *несвязной* иначе. Моран и Вольфштал предположили, что входной граф задачи T (определенный аналогично графу решения) связный, то есть, как в доказательстве Леммы 13.6 мы можем двигаться между любыми двумя входными конфигурациями, изменяя по порядку входы процесса. Кроме того, результат невозможности был доказан для не-тривиальных алгоритмов, то есть, алгоритмов, которые удовлетворяют, в дополнение к (1) завершению и (2) непротиворечивости,

(1) **Нетривиальность**. Для каждого $\overline{d} \in D_T$ имеется достижимая конфигурация, в которой процессы остановились на (приняли решение) $\overline{d}$.

Теорема 13.15 *Нетривиального 1-аварийно-устойчивого алгоритма решения для несвязной задачи T не существует.*

Доказательство. Предположим, напротив, что такой алгоритм, A, существует; из него можно получить алгоритм согласия A', что противоречит Теореме 13.8. Чтобы упростить аргументацию, мы полагаем, что G_T содержит два связных компонента, "0" и "1".

Алгоритм A’ сначала моделирует A, но вместо того, чтобы остановиться на значении d, процесс “выкрикивает” <**vote**, d> и ждет получения N-1 сообщений голосования. Тупика не возникает, потому что все корректные процессы принимают решение в A; следовательно по крайней мере N-1 процессов “выкрикивают” сообщение голосования.

После получения сообщений, у процесса p есть N-l компонентов вектора в D^N . Этот вектор можно расширить значением процесса, от которого голос не был получен так, чтобы весь вектор находился в D_T . (Действительно, непротиворечивое решение принято этим процессом, или все еще возможно.)

Теперь заметим, что различные процессы могут вычислять различные расширения, но эти расширения принадлежат одному и тому же связному компоненту графа G_T . Каждый процесс, который получил N-1 голосов, останавливается на (принимает решение) имени связанного компонента, которому принадлежит расширенный вектор. Остается показать, что A' является алгоритмом согласия.

Завершение. Выше уже обсуждалось, что каждый корректный процесс получает по крайней мере N-1 голосов.

Соглашение. Мы сначала докажем, что существует вектор $\overline{d_0} \in D_T$ такой, что каждый корректный процесс получает N-1 компонентов $\overline{d_0}$.

Случай 1: *Все процессы нашли решение в A*. Пусть $\overline{d_0}$ будет вектором достигнутых решений; каждый процесс получает N-1 компонентов $\overline{d_0}$, хотя "недостающий" компонент может быть различным для каждого процесса.

Случай 2: *Все процессы за исключением одного, допустим r, нашли решение в A*. Все корректные процессы получают одни и те же N-1 решений,

а именно решения всех процессов за исключением r. Возможно, что r потерпел аварию, но, так как возможно , что r просто очень медленный, он все же сможет достичь решения, то есть, существует вектор $\overline{d_0} \in D_T$, который расширяет решения, принятые на настоящий момент.

Из существования $\overline{d_0}$ следует, что каждый процесс принимает решение о связном компоненте этого вектора.

Нетривиальность. Из нетривиальности A, можно достичь векторы решения как в компоненте 0, так и в компоненте 1; по построению A' оба решения возможны.

Таким образом, A' является асинхронным, детерминированным, 1-аварийно-устойчивым алгоритмом согласия. Алгоритма A не существует по Теореме 13.8. □

Обсуждение. Требование нетривиальности, утверждающее, что каждый вектор решения в D_T достижим, является довольно сильным. Можно спросить, могут ли некоторые алгоритмы, которые являются тривиальными в этом смысле тем не менее быть интересными. В качестве примера, рассмотрим Алгоритм 13.2 для переименования; с ходу не видно, что он нетривиален, то есть, *каждый* вектор с отдельным именем достижим (да, достижим); еще менее понятно то, почему нетривиальность может представлять интерес в этом случае.

Исследование доказательства Теоремы 13.15 показывает, что в доказательстве можно использовать более слабое требование нетривиальности, а именно, что векторы решения достижимы по крайней мере в двух различных связных компонентах G_T. Такую ослабленную нетривиальность можно иногда вывести из формулировки проблемы.

Фундаментальная работа о задачах решения, которые являются разрешимыми и неразрешимыми при наличии одного сбойного процессора, была выполнена Бираном, Мораном и Заксом [BMZ90]. Они дали полную комбинаторную характеристику разрешимых задач решения.

13.4 Вероятностные Алгоритмы Согласия

В доказательстве Теоремы 13.8 показано, что каждый асинхронный алгоритм согласия имеет бесконечные выполнения, в которых никакое решение не принимается. К счастью, для хорошо подобранных алгоритмов такие выполнения могут быть достаточно редки и иметь вероятность 0, что делает алгоритмы очень полезными в вероятностном смысле; см. Главу 9. В этом разделе мы представляем два вероятностных алгоритма согласия, один для модели аварий, другой для Византийской модели; алгоритмы были предложены Брахой и Туэгом [BT85]. В обоих случаях сначала доказывается верхний предел для способности восстановления ($t < N/2$ и $t < N/3$, соответственно) и что и оба алгоритма удовлетворяют соответствующей границе.

В требованиях правильности для этих вероятностных алгоритмов согласия, требование завершения сделано вероятностным, то есть, заменено более слабым требованием сходимости.

(1) **Сходимость**. Для каждой начальной конфигурации,

$$\lim_{k\to\infty} \Pr[\text{корректный процесс не принял решение после k шагов}] = 0.$$

Частичная правильность (Соглашение) должна удовлетворяться при каждом выполнении; возникающие в результате вероятностные алгоритмы имеют класс Las Vegas (Подраздел 9.1.2).

Вероятность принимается всеми выполнениями, начинающимися в данной начальной конфигурации. Чтобы вероятности были значимыми, должно быть задано распределение вероятности над этими выполнениями. Это можно сделать использованием рандомизации в процессах (как в Главе 9), но здесь вместо этого определяется распределение вероятности на прибытиях сообщений.

Распределение вероятности на выполнениях, начинающихся в данной начальной конфигурации, определяется предположением о *законном планировании*. Оба алгоритма функционируют в раундах; в раунде процесс "выкрикивает" сообщение и ждет получения N-t сообщений. Определим R(q, p, k) как событие, когда в раунде k процесс p получает (раунд-k) сообщение q среди первых N-t сообщений. Законное планирование означает, что

(1) $\exists \varepsilon > 0 \;\; \forall p,q,k: \Pr[R(p,q,k)] \geq \varepsilon$.

(2) Для всех k и различных процессов p, q, r, события R(q, p, k) и R(q, r, k) независимы.

Заметьте, что Утверждение 13.4 также выполняется для вероятностных алгоритмов, когда требуется сходимость (завершение с вероятностью один). Действительно, так как достижимая конфигурация достигается с положительной вероятностью, решенная конфигурация должна быть достижима из каждой достижимой конфигурации (хотя не обязательно достигаемой в каждом выполнении).

13.4.1 Аварийно-устойчивые Протоколы Согласия

В этом подразделе изучается проблема согласия в модели аварийного отказа. Сначала доказывается верхняя граница $t < N/2$ способности восстановления, потом приводится алгоритм со способностью восстановления $t < N/2$.

Теорема 13.16 *t-аварийно-устойчивого протокола согласия для* $t \geq N/2$ *не существует.*

Доказательство. Существование такого протокола, допустим P, подразумевает следующий три требования.

Требование 13.17 *P имеет бивалентную начальную конфигурацию.*

Доказательство. Аналогично доказательству Леммы 13.6; детали оставлены читателю. □

Для подмножества процессов S, конфигурация γ называется S-валентной, если и 0- и 1-решенные конфигурации достижимы из γ с помощью только шагов в S. γ называется S-0-валентной если, делая шаги только в S, 0-решенная конфи-

гурация, и никакая 1-решенная конфигурации, может быть достигнута, S-1-валентная конфигурация определяется аналогично.

Разделим процессы на две группы, S и T, размера $\lfloor N/2 \rfloor$ и $\lceil N/2 \rceil$.

Требование 13.18 *Достижимая конфигурация γ является или S-0-валентной и T-0-валентной, или S-1-валентной и T-1-валентной.*

Доказательство. Действительно, высокая способность восстановления протокола подразумевает, что и S и T могут достигать решения независимо; если возможны *различные* решения, можно достичь противоречивой конфигурации, объединяя планы. □

Требование 13.19 *P не имеет достижимой бивалентной конфигурации.*

Доказательство. Пусть дана достижимая бивалентная конфигурация γ и предположим, что это γ S-l-валентна и T-1-валентна (используем Требование 13.18). Однако, γ бивалентна, поэтому (ясно из связи между группами) 0-решенная конфигурация δ_0 также достижима из γ. В последовательности конфигураций от γ до δ_0 имеются две последующих конфигурации γ_1 и γ_0, где γ_v является и S-v-валентной и T-v-валентной. Пусть p - процесс, вызывающий переход из γ_1 в γ_0. Теперь невыполнимо $p \in S$, потому что γ_1 S-1-валентна и γ_0 S-0-валентна; аналогично невыполнимо $p \in T$. Мы пришли к противоречию. □

Противоречие существованию протокола P является результатом Требований 13.17 и 13.19; таким образом Теорема 13.16 доказана. □

Аварийно-устойчивый алгоритм согласия Брахи и Туэга. Аварийно-устойчивый алгоритм согласия, предложенный Брахой и Туэгом [BT85] функционирует в *раундах*: в раунде k процесс посылает сообщение всем процессам (включая себя) и ждет получения N-t сообщений раунда k. Ожидание такого числа сообщений не представляет возможность тупика (см. Упражнение 13.10).

В каждом раунде, процесс p "выкрикивает" голос за 0 или за 1 вместе с весом. Вес - число голосов, полученных для этого значения в предыдущем раунде (1 в первом раунде); голос с весом, превышающим N/2, называется *свидетелем*. Хотя различные процессы в раунде могут голосовать по-разному, в одном раунде никогда нет свидетелей различных значений, как будет показано ниже. Если процесс p получает свидетеля в раунде k, p голосует за свое значение в раунде k+1; иначе p голосует за большинство полученных голосов. Решение принимается, если в раунде получено больше, чем t свидетелей; решительный процесс выходит основной цикл и свидетели криков в течение следующих двух раундов, чтобы дать возможность другим процессам решить. Протокол дан как Алгоритм 13.3.

---

var	$value_p$	: (0, 1)	**init** x_p (*голос p*)
	$round_p$	: integer	**init** 0 (*номер раунда*)
	$weight_p$	: integer	**init** 1 (*Вес голоса p*)

```
        msgs_p[0..1]    : integer      init 0   (*Счетчик полученных голосов*)
        witness_p[0..1] : integer      init 0   (*Счетчик полученных свидетелей *)
begin
        while y_p = b do
        begin witness_p[0], witness_p[1], msgs_p[0], msgs_p[1] := 0,0,0,0; (*сброс счетчиков*)
                shout<vote, round_p, value_p, weight_p>;
                while msgs_p[0] + msgs_p[1] < N - t do
                begin receive<vote, r, v, w>;
                        if r > round_p then                    (*Будущий раунд…*)
                                send< vote, r, v, w> to p      (*…обработать позже*)
                        else if r = round_p then
                        begin msgs_p[v] := msgs_p[v] + 1;
                                if w > N/2 then                (*Свидетель*)
                                  witness_p[v] := witness_p[v] + 1
                        end
                        else (*r < round_p, ignore*) skip
                end;
                (*Выбрать новое значение: голос и вес в следующем раунде*)
                if witness_p[0] > 0 then value_p := 0
                else if witness_p[1] > 0 then value_p := 1
                else if msgs_p[0] > msgs_p[1] then value_p := 0
                else value_p := 1;
                weight_p := msgs_p[value_p];
                (*Принять решение, если более t свидетелей*)
                if witness_p[value_p] > t then y_p := value_p;
                round_p := round_p + 1
        end;
        (*Помочь другим процессам принять решение*)
        shout<vote, round_p, value_p, N-t>;
        shout<vote, round_p+1, value_p, N-t>
end
```

Алгоритм 13.3 Аварийно-устойчивый алгоритм согласия

Голоса, прибывающие для более поздних раундов должны быть обработаны в соответствующем раунде; это моделируется в алгоритме с помощью посылки сообщения самому процессу для обработки позже. Заметьте, что в любом раунде процесс получает самое большее один голос от каждого процесса, общим количеством до N-t голосов; так как более, чем N-t процессов могут "выкрикивать" голос, процессы могут принимать во внимание различные подмножества "выкрикиваемых" голосов. Мы впоследствии покажем несколько

свойств алгоритма, которые вместе означают, что это - вероятностный аварийно-устойчивый протокол согласия (Теорема 13.24).

Лемма 13.20 *В любом раунде никакие два процесса не свидетельствуют за различные значения.*

Доказательство. Предположим, что в раунде k, процесс p свидетельствует за v, и процесс q свидетельствует за w; k > 1, потому что в раунде 1 никакие процессы не свидетельствуют. Предположение подразумевает, что в раунде k-1, p получил больше чем N/2 голосов за v, и q получил больше чем N/2 голосов за w. Вместе задействовано более N голосов; следовательно, процессы от которых p и q получили голоса перекрываются, то есть, есть r, который послал v-голос процессу p и w-голос процессу q. Это означает, что v =w. □

Лемма 13.21 *Если процесс принимает решение, то все корректные процессы принимают решение об одном и том же значении, и самое большее два раунда спустя.*

Доказательство. Пусть k будет первым раундом, в котором принимается решение, p - процесс, принимающий решение в раунде k, и v - значение решения p. Решение подразумевает, что в раунде k имелись v-свидетели; следовательно, по Лемме 13.20 не имелось свидетелей других значений, так что никакое другое решение не принимается в раунде k.

В раунде k имелось более t свидетелей v (это следует из решения p), следовательно, все корректные процессы получают по крайней мере одного v-свидетеля в раунде k. В результате, все процессы, которые голосуют в раунде k + 1, голосуют за v (заметьте также, что p все еще “выкрикивает” голос в раунде k + 1). Это означает, что, если решение вообще принимается в раунде k + 1, это решение v.

В раунде k + 1 предлагаются только v-голоса, следовательно все процессы, которые голосуют в раунде k + 2 свидетельствуют за v в этом раунде (p тоже). В результате, в раунде k + 2 все корректные процессы, которые не приняли решения в более ранних раундах, получают N-t v-свидетелей и останавливаются на v. □

Лемма 13.22 $\lim_{k\to\infty} \Pr$ *[никакого решения не принято в раунде* $\le k$ *] = 0.*

Доказательство. Пусть S - множество N-t корректных процессов (такое множество существует) и предположим, что до раунда k_0 не принято никакого решения. Предположение законного планирования подразумевает, что, для некоторого $\rho > 0$, в любом раунде вероятность того, что каждый процесс в S получает точно N-t голосов процессов в S, по крайней мере ρ. Это происходит в трех последующих раундах k_0, $k_0 + 1$ и $k_0 + 2$ с вероятностью по крайней мере $\psi = \rho^3$.

Если это происходит, процессы в S получают одни и те же голоса в раунде k_0 и следовательно выбирают одно и то же значение, допустим v_0 в раунде k_0. Все

процессы в S голосуют за v_0 в раунде k_0+1, что означает, что каждый процесс в S получает N-t голосов за v_0 в раунде k_0+1. Это значит, что процессы в S за v_0 в раунде k_0+2; следовательно они все получают N-t > t свидетелей v_0 в раунде k_0+2, и все принимают решение v_0 в этом раунде. Отсюда

$$\textbf{Pr}\ [\text{Процессы в S не приняли решения в раунде k + 2}]$$
$$\leq \psi \times \Pr\ [\text{Процессы в S не не приняли решения до раунда k}],$$

что подтверждает результат. □

Лемма 13.23 *Если все процессы начинают алгоритм с входом v, то все они принимают решение v в раунде 2.*

Доказательство. Все процессы получают только голоса за v в раунде 1, так что все процессы свидетельствуют за v в раунде 2. Это означает, что все они они принимают решение в этом раунде. □

Теорема 13.24 *Алгоритм 13.3 - вероятностный, t-аварийно-устойчивый протокол согласия при t < N/2.*

Доказательство. Сходимость показана в Лемме 13.22, а соглашение - в Лемме 13.21; нетривиальность следует из Леммы 13.23. □

Зависимость решения от входных значениях анализируется далее в Упражнении 13.11.

13.4.2 Византийско-устойчивые Протоколы Согласия

Византийская модель сбоев более недоброжелательна, чем модель аварий, потому что Византийские процессы могут выполнять произвольные переводы состояний и могут посылать сообщения, которые расходятся с алгоритмом. В дальнейшем мы будем использовать запись $\gamma \to \delta$ (или $\gamma \to_S \delta$) для обозначения того, что имеется последовательность *корректных шагов*, то есть, переходов протокола (в процессах S), ведущих систему из γ в δ. Аналогично, γ достижима, если имеется последовательность корректных шагов, ведущих из начальной конфигурации в γ. Злонамеренность Византийской модели подразумевает более низкий максимум способности восстановления, чем для модели аварийного отказа.

Теорема 13.25 *t-Византийско-устойчивого протокола согласия при* $t \geq N/3$ *не существует.*

Доказательство. Предположим, напротив, что такой протокол существует. Читателю снова предоставляется показать существование бивалентной начальной конфигурации любого такого протокола (используйте, как обычно, нетривиальность).

Высокая способность восстановления протокола означает, что можно выбрать два множества S и T таких, что $|S| \geq N - t$, $|T| \geq N - t$, и $|S \cap T| \leq t$. Словом, и S и T достаточно большие, чтобы выжить независимо, но их пересечение может быть полностью злонамеренно. Это используется для демонстрации того, что никакие бивалентные конфигурации не являются достижимыми.

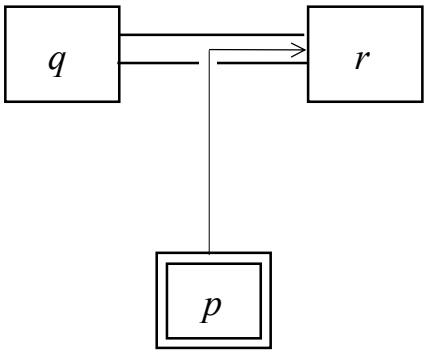
Заявление 13.26 *Достижимая конфигурация γ является или S-0-валентной и T-0-валентной, или S-1- валентной и T-1-валентной.*

Доказательство. Так как γ достигается последовательностью корректных шагов, все возможности для выбора множества t процессов, которые дают сбой, все еще открыты. Предположим, напротив, что S и T могут достичь *разных* решений, то есть, $\gamma \rightarrow_S \delta_v$ и $\gamma \rightarrow_T \delta_{\bar{v}}$, где $\delta_v(\delta_{\bar{v}})$ - конфигурация, где все процессы в S (в T) остановились на v ($\bar{v}$). Можно достичь противоречивого состояния, предполагая, что процессы в $S \cap T$ злонамеренные, и объединяя планы следующим образом. Начиная с конфигурации γ, процессы в $S \cap T$ сотрудничают с другими процессами в S в последовательности, ведущей к v-решению в S. Когда это решение было принято процессами в S, злонамеренные процессы восстанавливают свое состояние как в конфигурации γ и впоследствии сотрудничают с процессами в T в последовательности, ведущей к $\bar{v}$ решению в T. Из этого получается конфигурация, в которой корректные процессы приняли решение по-разному, что находится в противоречии с требованием соглашения. □

Заявление 13.27 *Достижимой бивалентной конфигурации не существует.*

Доказательство. Пусть дана достижимая бивалентная конфигурация γ и предположим, что γ является, и S-1-валентной и T-1-валентной (Заявление 13.26). Однако, γ бивалентна, поэтому из γ также достижима 0-решенная конфигурация δ_0 (очевидно, в сотрудничестве между S и T). В последовательности конфигураций из γ в δ_0 имеются две вытекающих конфигурации γ_1 и γ_0, причем γ_v и S-v-валентна и T-v-валентна. Пусть p - процесс, вызывающий переход из γ_1 в γ_0. Теперь не выполняется $p \in S$, потому что γ_1 S-1-валентна и γ_0 S-0-валентна; аналогично не выполняется $p \in T$. Пришли к противоречию. □

Последнее заявление противоречит существованию бивалентных начальных конфигураций. Таким образом Теорема 13.25 доказана. □



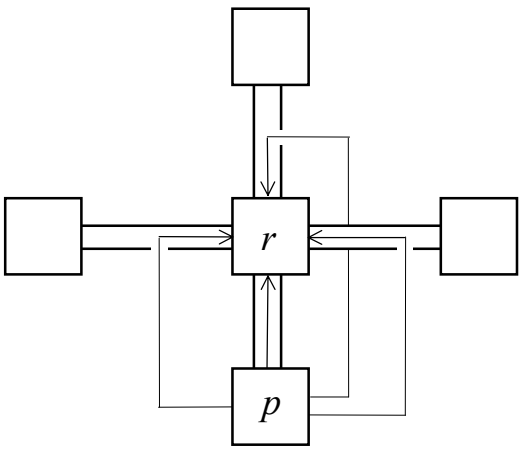


Рисунок 13.4 Византийский процесс, моделирующий другие процессы.

Византийско-устойчивый алгоритм согласия Брахи и Туэга. При t < N/3, t-Византийско-устойчивые протоколы согласия существуют. Необходимо, чтобы система связи позволяла процессу определять, каким процессом было послано полученное сообщение. Если Византийский процесс p может послать корректному процессу r сообщение и успешно симулировать получение процессом r сообщения *от q* (см Рисунок 13.4), проблема становится неразрешимой. Действительно, процесс p может моделировать достаточно много корректных процессов, чтобы навязать неправильное решение в процессе r.

Подобно аварийно-устойчивому протоколу, Византийско-устойчивый протокол (Алгоритм 13.5) функционирует в раундах. В каждый раунде каждый процесс может представлять на рассмотрение голоса, и решение принимается, когда достаточно много процессов голосуют за одно и то же значение. Более низкая способность восстановления (t < N/3) устраняет необходимость в различении свидетелей и не-свидетелей; процесс принимает решение после принятия более (N + t) /2 голосов за одно и то же значение.

Злонамеренность этой модели отказов требует, однако, введения механизма проверки голоса, что является затруднением протокола. Без такого механизма Византийский процесс может нарушать голосование среди корректных процессов, посылая *различные* голоса различным корректным процессам. Такое злонамеренное поведение не возможно в модели аварийного отказа. Механизм проверки гарантирует, что, хотя Византийский процесс r может посылать различные голоса корректным процессам p и q, он не может обмануть p и q, чтобы они приняли различные голоса за r (в некотором раунде).

Механизм проверки основан на отражении сообщений. Процесс “выкрикивает” свой голос (как *initial*, **in**), и каждый процесс, после получения первого голоса за некоторый процесс в некотором раунде, отражает эхом голос (как *echo*, **ec**). Процесс примет голос, если для него были приняты более (N+t)/2 отраженных сообщений. Механизм проверки сохраняет (частичную) правильность коммуникации между корректными процессами (Лемма 13.28), и корректные процессы никогда не принимают различные голоса за один и тот же процесс (Лемма 13.29). Тупиков нет (Лемма 13.30).

Мы говорим, что процесс p *принимает v-голос* за процесс r в раунде k, если p увеличивает $msgs_p[v]$ после получения сообщения голоса <**vote**, **ec**, r, v, k>. Алгоритм гарантирует, что p проходит раунд k только после принятия N-t голосов, и что p принимает также самое большее один голос за каждый процесс в каждом раунде.

---

var $value_p$: (0, 1) **init** x_p ;

```
        round_p          : integer      init 0;
        msgs_p[0..1]     : integer      init 0;
        echos_p[P,0..1]  : integer      init 0;
repeat forall v,q do begin msgs_p[v]:= 0 ; echos_p[q,v]:= 0 end;
        shout<vote, in, p, value_p , round_p >;
        (*Теперь принять N-t голосов для текущего раунда*)
        while msgs_p[0]+msgs_p[1] < N − t do
        begin receive<vote, t, r, v, rn> from q;
              if <vote, t, r, *, rn> уже был получен от q
                    then skip      (*q повторяет, должно быть, Византийский*)
              else if t=in and q ≠ r
                    then skip      (*q лжет, должно быть, Византийский *)
              else if rn > round_p
                    then (*обработать сообщение в более позднем раунде*)
                         send<vote, t, r, v, rn> to p
              else  (*Обработать или отразить сообщение голоса*)
               case t of
                    in:   shout<vote, ec, r, v, rn>
                    ec:   if rn = round_p then
                            begin echos_p[r,v]:= echos_p[r,v]+1;
                                  if echos_p[r,v] = ⌊(N + t) / 2⌋+1
                                    then msgs_p[v]:= msgs_p[v]+1
                            end
                          else skip      (*старое сообщение*)
               esac
        end;
        (*Выбрать значение для следующего раунда*)
        if msgs_p[0] > msgs_p[1] then value_p:= 0 else value_p:= 1;
        if msgs_p[value_p] > (N + t) / 2 then y_p:= value_p ;
        round_p:= round_p + 1
until false
```

Алгоритм 13.5 Византийско-устойчивый алгоритм согласия.

Лемма 13.28 *Если корректный процесс p принимает в раунде k голос v за корректный процесс r, то r голосовал за v в раунде k.*

Доказательство. Процесс p принимает голос после получения сообщения <**vote**, **ec**, r, v, k> от более (N+t)/2 (различных) процессов; по крайней мере один корректный процесс s послал такое сообщение p. Процесс s посылает эхо p после получения сообщения <**vote**, **in**, r, v, k> от r, что означает, так как r корректен, что r голосует за v в раунде k. □

Лемма 13.29 *Если корректные процессы p и q принимают голос за процесс r в раунде k, они принимают тот же самый голос.*

Доказательство. Предположим, что в раунде k процесс p принимает v-голос за r, а процесс q принимает w-голос. Таким образом, p получил <**vote**, **ec**, r, v, k> от более (N+t)/2 процессов, и q получил <**vote**, **ec**, r, w, k> от более (N+t)/2 процессов. Так как имеется только N процессов, то, должно быть, более t процессов послали <**vote**, **ec**, r, v, k> процессу p *и* <**vote**, **ec**, r, w, k> процессу q. Это значит, что по крайней мере один корректный процесс сделал так, и следовательно v = w. □

Лемма 13.30 *Если все корректные процессы начинают раунд k, то они принимают достаточно много голосов в этом раунде, чтобы закончить его.*

Доказательство. Корректный процесс r, начинающий раунд k с $value_r = v$, "выкрикивает" начальный голос для этого раунда, который отражается всеми корректными процессами. Таким образом, для корректных процессов p и r, <**vote**, **ec**, r, v, k> посылается p по крайней мере N-t процессами, позволяя p принять v-голос за r в раунде k, если не принято ранее N-t других голосов. Отсюда следует, что процесс p принимает N-t голосов в этом раунде. □

Теперь доказательство правильности протокола похоже на доказательство правильности аварийно-устойчивого протокола.

Лемма 13.31 *Если корректный процесс принимает решение (останавливается на) v в раунде k, то все корректные процессы выбирают v в раунде k и всех более поздних раундах.*

Доказательство. Пусть S - множество по крайней мере (N + t)/2 процессов, для которых p принимает v-голос в раунде k. Корректный процесс q принимает в раунде k N-t голосов, включая по крайней мере $|S| - t > (N - t)/2$ голосов за процессы в S. По Лемме 13.29, q принимает более (N-t)/2 v-голоса, что означает, что q выбирает v в раунде k.

Чтобы показать, что все корректные процессы выбирают v в более поздних раундах, предположим, что все корректные процессы выбирают v в некотором раунде l; следовательно, все корректные процессы голосуют за v в раунде l+1. В раунде l+1 каждый корректный процесс принимает N-t голосов, включая более (N-t)/2 голосов за корректные процессы. По Лемме 13.28, корректный процесс принимает по крайней мере (N-t)/2 v-голоса, и, следовательно, снова выбирает v в раунде l+1. □

Лемма 13.32 $\lim_{k \to \infty}$ ***Pr*** *[Корректный процесс p не принял решения до раунда k] =* ***0****.*

Доказательство. Пусть S - множество по крайней мере N-t корректных процессов и предположим, что p не принял решения до раунда k. С вероятностью $\psi > 0$ все процессы в S принимают в раунде k голоса за *одну и ту же* совокупность N-t процессов и, в раунде k + 1, только голоса за процессы в S. Если это проис-

ходит, процессы в S голосуют одинаково в раунде k + 1 и принимают решение в раунде k + 1. Отсюда

$$\textbf{Pr}\,[\text{Корректный процесс p не принял решения до раунда } k+2]$$

$$\leq \psi \times \Pr\,[\text{Корректный процесс p не принял решения до раунда k}],$$

что подтверждает результат. □

Лемма 13.33 *Если все корректные процессы начинают алгоритм с входом v, в конечном счете принимается решение v.*

Доказательство. Как в доказательстве Леммы 13.31 можно показать, что все корректные процессы выбирают v снова в каждом раунде. □

Теорема 13.34 *Алгоритм 13.5 - вероятностный, t-Византийско-устойчивый протокол согласия при t < N/3.*

Доказательство. Сходимость показана в Лемме 13.32 и соглашение - в Лемме 13.31; нетривиальность следует из Леммы 13.33. □

Зависимость решения от входных значений проанализирована далее в Упражнении 13.12. Алгоритм 13.5 описывается как бесконечный цикл для простоты представления; мы в заключение описываем, как можно модифицировать алгоритм, чтобы он завершался в каждом решающем процессе. После принятия решения v в раунде k процесс p выходит из цикла и “выкрикивает” "множественные" голоса <**vote**, **in**, p, k+, v> и отражает <**vote**, **ec**, *, k+, v>. Эти сообщения интерпретируются как начальный и отражаемый голоса для всех раундов после k. Действительно, p голосует за v во всех более поздних раундах, и все корректные процессы будут голосовать так же (Лемма 13.31). Следовательно, множественные сообщения - такие, которые были бы посланы процессом p при продолжении алгоритма, с возможным исключением для отражений злонамеренных начальных голосов.

13.5 Слабое Завершение

В этом разделе изучается проблема асинхронного Византийского вещания. Цель вещания состоит в том, чтобы сделать значение, которое присутствует в одном процессе g, *командующем*, известным всем процессам. Формально, требование нетривиальности для протокола согласия усилено заданием того, что значение решения является входом командующего, если он корректен:

(3) **Зависимость**. Если командующий корректен, все корректные процессы останавливаются на (принимают решение о) его входе.

При таком уточнении, однако, командующий становится единичной точкой отказа, что означает, что проблема не разрешима, как выражено в следующей теореме.

Теорема 13.35 *1-Византийско-устойчивого алгоритма, удовлетворяющего сходимости, соглашению, и зависимости, даже если сходимость требуется только, если командующий послал по крайней мере одно сообщение, не существует.*

Доказательство. Рассмотрим два сценария. В первом командующий считается Византийским; сценарий служит, чтобы определить достижимую конфигурацию γ. Затем получается противоречие во втором сценарии.

(1) Предположим, что командующий - Византийский и что он посылает сообщение, чтобы инициализировать вещание "0" процессу p_0 и сообщение, чтобы инициализировать вещание "1" процессу p_1. Затем командующий останавливается. Назовем возникающую в результате конфигурацию γ.

Из сходимости следует, что решенная конфигурация может быть достигнута даже если отказывает командующий; пусть S = P \ {g}, и предположим, что $\gamma \rightarrow_S \delta_0$, где δ_0 0-решенная.

(2) Для второго сценария, предположим, что командующий корректен и имеет вход 1, что он посылает сообщения, чтобы инициализировать вещание 1 процессам p_0 и p_1, после которого его сообщения задерживаются в течение очень длительного времени. Теперь предположим, что p_0 - Византийский, и, после получения сообщения, изменяет свое состояние на состояние в γ, то есть, притворяется, что получил 0-сообщение от командующего. Так как $\gamma \rightarrow_S \delta_0$, то теперь можно достичь 0-решения без взаимодействия с командующим, что не дозволяется, потому что командующий корректен и имеет вход 1.

□

Невозможность следует из возможности того, что командующий инициализирует вещание и останавливается (первый сценарий) без предоставления достаточной информации о своем входе (что используется во втором сценарии). Теперь покажем, что (детерминированное) решение возможно, если завершение требуется только в случае, когда командующий корректен.

Определение 13.36 *t-Византийско-устойчивый алгоритм вещания - алгоритм, удовлетворяющий следующим трем требованиям.*

(1) **Слабое завершение**. Все корректные процессы принимают решение, или никакой корректный процесс не принимают решения. Если командующий корректен, все корректные процессы принимают решение.

(2) **Соглашение**. Если корректные процессы принимают решение, они останавливаются на одном и том же значении.

(3) **Зависимость**. Если командующий корректен, все корректные процессы останавливаются на его входе.

Можно показать, пользуясь аргументами, подобными используемым в доказательстве Теоремы 13.25, что способность восстановления асинхронного Византийского алгоритма вещания ограничена t < N/3. Алгоритм вещания Брахи и Туэга [BT85], данный как Алгоритм 13.6, использует три типа сообщений голосов: *начальные* (initial) сообщения (тип **in**), *отраженные* (echo) сообщения (тип

ec), и *готовые* (ready) сообщения (тип **re**). Каждый процесс подсчитывает для каждого типа и значения, сколько сообщений были получены, считая самое большее одно сообщение, полученное от каждого процесса.

Командующий инициализирует вещание, “выкрикивая” начальный голос. После получения начального голоса от командующего, процесс “выкрикивает” отраженный голос, содержащий то же самое значение. Когда было получено более (N+t)/2 отраженных сообщения со значением v, “выкрикивается” готовое сообщение. Число отраженных сообщений достаточно велико, чтобы гарантировать, что никакие корректные процессы не посылают готовых сообщений для различные значения (Лемма 13.37). Получение более t готовых сообщений для одного и того же значения (что означает, что по крайней мере один корректный процесс послал такое сообщение) также вызывает “выкрикивание” готовых сообщений. Получение более 2t готовых сообщений для одного и того же значения (что означает, что более t корректных процессов послали такое сообщение) вызывает принятие решения для этого значения. В Алгоритме 13.6 не принято никаких мер, чтобы предотвратить “выкрикивание” готового сообщения корректным процессом дважды, т.к. такое сообщение все равно игнорируется корректными процессами.

---

var $msgs_p[(in,ec,re),0..1]$: integer **init** 0;

Только дёу командующего: shout<**vote**, **in**, x_p>

Äëÿ âñåõ ïðîöåññîâ:
while $y_p = b$ **do**

```
begin  receive<vote, t, v> from q;
       if от q уже было получено сообщение голоса <vote, t, v>
              then skip      (*q повторяется, игнорировать*)
       else if t = in and q ≠ g
              then skip      (*q подражает g, должно быть, Византийский*)
       else   begin  msgs_p[t,v] := msgs_p[t,v] + 1;
                     case t of
                       in:  if msgs_p[in,v] = 1 then shout<vote, ec, v>
                       ec:  if msgs_p[ec,v] = ⌈(N+t)/2⌉ + 1
                                    then shout<vote, re, v>
                       re:  if msgs_p[re,v] = t + 1 then shout<vote, re, v>;
                            if msgs_p[re,v] = 2t + 1 then y_p := v;
                     esac
              end
end
```

Алгоритм 13.6 Византийско-устойчивый алгоритм вещания.

Лемма 13.37 *Никакие два корректных процесса не посылают готовых сообщений для различных значений.*

Доказательство. Корректный процесс принимает самое большее одно начальное сообщений (от командующего), и следовательно посылает отраженные сообщения для самое большее одного значения.

Пусть p - первый корректный процесс, который шлет готовое сообщение для v, и q - первый корректный процесс, который шлет готовое сообщение для w. Хотя готовое сообщение может быть послано после получения достаточно большого числа готовых сообщений, дело обстоит не так для первого корректного процесса, который посылает готовое сообщение. Это происходит из-за того, что перед его посылкой должны быть получены t+1 готовых сообщения, что означает, что готовое сообщение от по крайней мере одного корректного процесса уже было получено. Таким образом, p получил v-отражения от более (N+t)/2 процессов и q получил w--отражения от более (N+t)/2 процессов.

Так как имеется только N процессов и $t < N/3$, есть более t процессов, включая по крайней мере один корректный процесс r, от которых p получил v-отражение, а q получил w-отражение. Так как r корректен, то $v = w$.

□

Лемма 13.38 *Если корректный процесс принимает решение, то все корректные процессы принимают решение относительно одного и того же значения.*

Доказательство. Чтобы остановиться на v, для v должно быть получено более 2t готовых сообщений, которые включают в себя более t готовых сообщений от корректных процессов; по Лемме 13.37 решения будут согласованными.

Предположим, что корректный процесс p останавливается на v; p получил более 2t готовых сообщений, включая более t сообщений от корректных процессов. Корректный процесс, посылающий готовое сообщение к p, посылает это сообщение всем процессам, что означает, что все корректные процессы получают более t готовых сообщений. Это, в свою очередь, значит, что все корректные процессы посылают готовое сообщение, так что каждый корректный процесс в конечном счете получает $N-t > 2t$ готовых сообщений и принимает решение.

□

Лемма 13.39 *Если командующий корректен, все корректные процессы останавливаются на его входе.*

Доказательство. Если командующий корректен, он не посылает начальных сообщений со значениями, отличными от своего входа. Следовательно, никакой корректный процесс не пошлет отраженных значений, отличных от входа командующего, что означает, что самое большее t процессов посылают неверные отражения. Такого количества неверных отражений недостаточно для того, чтобы корректные процессы посылали готовые сообщения для неверных значений, что означает, что самое большее t процессов посылают неверные готовые сообщения. Такого количества неверных готовых сообщений недостаточно для того, чтобы корректный процесс посылал готовые сообщения или принимал решения, что означает, что никакой корректный процесс не посылает неверного готового сообщения и не принимает неправильного решения.

Если командующий корректен, он посылает начальный голос со своим входом всем корректным процессам, и все корректные процессы "выкрикивают" отражение с этим значением. Следовательно, все корректные процессы получат по крайней мере N-t > (N+t)/2 корректных отраженных сообщений и "выкрикнут" готовое сообщение с корректным значением. Таким образом, все корректные процессы получат по крайней мере N-t > 2t верных готовых сообщений и примут верное решение. □

Теорема 13.40 *Алгоритм 13.6 - асинхронный t-Византийско-устойчивый алгоритм вещания при t < N/3.*

Доказательство. Слабое завершение следует из Лемм 13.39 и 13.38, соглашение - из Леммы 13.38, и зависимость - из Леммы 13.39. □

Упражнения к Главе 13

Раздел 13.1

Упражнение 13.1 *Удаление любого из трех требований Определения 13.3 (завершения, соглашения, нетривиальности) для проблемы согласия позволяет принять очень простое решение. Покажите это, представив три простых решения.*

Упражнение 13.2 *В доказательстве Леммы 13.6 предполагается, что каждое из 2^N назначений бит N процессам производит возможную входную конфигурацию.*

Приведите детерминированные, 1-аварийно устойчивые протоколы согласия для каждого из следующих ограничений на входные значения.

(1) Дано, что четность входа является четной (то есть, имеется четное число процессов со входом 1) в каждой начальной конфигурации.

(2) Имеются два (известных) процесса r_1 и r_2, и каждая начальная конфигурация удовлетворяет $x_{r_1} = x_{r_2}$.

(3) В каждой начальной конфигурации имеется, по крайней мере, $\lceil (N/2)+1 \rceil$ процессов с одним и тем же входом.

Раздел 13.2

Упражнение 13.3 *Покажите, что при $t \geq N/2$ t-изначально-мертвых-устойчивого алгоритма выбора нет.*

Раздел 13.3

Упражнение 13.4 *Покажите, что никакой алгоритм для ε-приблизительного соглашения не может вынести $t \geq N/2$ сбоев.*

Упражнение 13.5 *Дайте биекцию из множества*

$$\{ (S, r):\ N - t \le s \le N \ \text{and}\ 1 \le r \le s \}$$

на целые числа в диапазоне [1, ..., K].

Проект 13.6 *Алгоритм 13.2 нетривиален?*

Упражнение 13.7 *Адаптируйте доказательство Теоремы 13.15 для случая, когда G_T состоит из k связных компонент.*

Упражнение 13.8 *В этом упражнении мы рассматриваем проблему [k, l]-выбора, который обобщает обычную проблему выбора. Проблема требует, чтобы все корректные процессы остановились или на 0 ("побежденный") или на 1 ("избранный"), и что число процессов, которые принимают решение 1 находится между k и l (включительно).*

(1) Каковы использования [k, l]-выбора?

(2) Покажите, что не существует детерминированного 1-аварийно-устойчивого алгоритма для [k, k]-выбора (если $0 < k < N$).

(3) Приведите детерминированный t-аварийно-устойчивый алгоритм для [k, k+2t]-выбора.

Раздел 13.4

Упражнение 13.9 *Означает ли требование сходимости, что ожидаемое число шагов ограничено?*

Ограничено ли ожидаемое число шагов во всех алгоритмах этого раздела?

Упражнение 13.10 *Покажите, что, если все корректные процессы начинают раунд k аварийно-устойчивого алгоритма согласия (Алгоритм 13.3), то все корректные процессы также закончат раунд k.*

Упражнение 13.11

(1) Докажите, что если более (N+t)/2 процессов начинают аварийно-устойчивый алгоритм согласия (Алгоритм 13.3) с входом v, то решение для v принимается за три раунда.

(2) Докажите, что если более (N-t)/2 процессов начинают этот алгоритм с входом v, то решение для v возможно.

(3) Является ли решение для v возможным, если ровно (N-t)/2 процесса начинают алгоритм с входом v?

(4) Каковы бивалентные входные конфигурации алгоритма?

Упражнение 13.12

(1) Докажите, что, если более (N+t)/2 корректных процессов начинают Алгоритм 13.5 с входом v, то в конечном счете принимается v-решение.

(2) Докажите, что если более (N+t)/2 корректных процессов начинают Алгоритм 13.5 с входом v и $t < N/5$, то v-решение принимается в течение двух раундов.

Раздел 13.5

Упражнение 13.13 *Докажите, что при t>N/3 асинхронного t-Византийско-устойчивого алгоритма вещания не существует.*

Упражнение 13.14 *Докажите, что в течение выполнения Алгоритма 13.6 корректными процессами посылается самое большее N (3N + 1) сообщений.*

14 Отказоустойчивость в Синхронных Системах

Предыдущая глава изучала степень отказоустойчивости, достижимой в полностью асинхронных системах. Хотя достижима приемлемая устойчивость, надежные системы на практике всегда синхронные в том смысле, что они полагаются на использование таймеров и верхних пределов времени доставки сообщений. В этих системах достижима более высокая степень устойчивости, алгоритмы более простые, и алгоритмы в большинстве случаев гарантируют верхнюю границу времени ответа.

Синхронность системы делает невозможным для сбойных процессов приведение корректных процессов в замешательство, не посылая информацию; действительно, если процесс не получает сообщение когда ожидается, вместо него используется значение по умолчанию, и отправитель становится подозреваемым в отказе. Таким образом, потерпевшие крах процессы немедленно обнаруживаются и не представляют никакие проблем в синхронных системах; мы концентрируемся на Византийских сбоях в этой главе.

В Разделе 14.1 изучается проблема выполнения вещания в синхронных сетях; мы представим верхнюю границу способности восстановления ($t < N/3$), а также два алгоритма с оптимальной способностью восстановления. Алгоритмы детерминированы и достигают согласия; предполагается, что все процессы знают, когда начинается вещание. Так как согласие не детерминированно достижимо в асинхронных системах (Теорема 13.8), то в присутствии сбоев (даже одиночной аварии), синхронные системы проявляют определенно более сильную вычислительную мощность чем асинхронные.

Так как авария и отсутствие посылки информации обнаруживаются (и следовательно "безобидны") в синхронных системах, только Византийские процессы способны нарушить вычисление, посылая ошибочную информацию или о своем собственном состоянии или неправильно пересылая (forwarding) информацию. В Разделе 14.2 будет показано, что устойчивость синхронных систем может быть далее расширена с помощью методов для установления подлинности информации. Эти механизмы делают невозможной “ложь” злонамеренных процессов об информации, полученной от других процессов. Тем не менее, возможность посылки противоречивой информации о собственном состоянии процесса остается. Также показывается, что реализация установления подлинности на практике возможна при использовании криптографических методов.

Алгоритмы в Разделах 14.1 и 14.2 предполагают идеализированную модель синхронных систем, в которых вычисление идет в импульсах (*раундах*); см. Главу 11. Существенно более высокая способность восстановления синхронных

систем по сравнению с асинхронными системами означает невозможность любой 1-аварийно-устойчивой детерминированной реализации импульсной модели в асинхронной модели. (Такая реализация, *синхронизатор*, возможна в надежных сетях; см. Раздел 11.3).

Реализация импульсной модели возможна, однако, в асинхронных сетях ограниченной задержки (Подраздел 11.1.3), где процессы обладают часами, и известна верхняя граница задержки сообщений. Реализация возможна, даже если часы идут неверно и до одной трети процессов злонамеренно отказывают. Наиболее трудная часть реализации - надежно синхронизировать часы процессов, проблема, которая будет обсуждена в Разделе 14.3.

14.1 Синхронные Протоколы Решения

В этом разделе мы представим алгоритмы для Византийско-устойчивого вещания в синхронных (импульсных) сетях; мы начнем с краткого обзора модели импульсных сетей, которые определены в Разделе 11.1.1. В синхронной сети процессы функционируют в импульсах, пронумерованных 1, 2, 3, и так далее; каждый процесс может выполнять неограниченное число импульсов, пока его локальный алгоритм не завершается. Начальная конфигурация (γ_0) описывается начальными состояниями процессов, и конфигурации после i-го импульса (обозначается γ_1) также описывается состояниями процессов. В импульсе i, каждый процесс сначала посылает конечное множество сообщений, в зависимости от своего состояния в γ_{i-1}. Впоследствии каждый процесс получает все сообщения, посланные ему в этом импульсе, и вычисляет новое состояние на основе старого и совокупности сообщений, полученных в импульсе.

Модель импульса - идеализированная модель синхронных вычислений. Синхронность отражается в

(1) очевидно одновременном возникновении переходов состояний в процессах; и

(2) гарантии того, что сообщения импульса получаются до переходов состояний этого импульса.

Эти идеализированные предположения могут быть ослаблены до более реалистичных предположений, а именно (1) доступности аппаратных часов и (2) верхней границы времени доставки сообщений. Возникающая в результате модель *асинхронных сетей ограниченных задержек* позволяет очень эффективно реализовать модель импульса (см. Раздел 11.1.3). Как показано во Главе 11, одновременность переходов состояний - только видимость. В реализации модели переходы состояний могут происходить в разное время, если только гарантируется своевременное получение всех сообщений. Кроме того, реализация должна допускать неограниченное число импульсов процесса. Последнее требование исключает реализации Главы 11 из использования их в отказоустойчивых прикладных программах, потому что они все страдают тупиками, большинство из них даже в случае одиночной потери сообщения. Как уже было упомянуто, к устойчивой реализации модели импульса мы обратимся в Разделе 14.3.

Так как модель импульса гарантирует доставку сообщений в одном и том же импульсе, процесс способен определить, что сосед *не посылал ему сообщения*.

Это свойство, отсутствующее в асинхронных системах, предлагает решение для проблемы согласия, и даже для проблемы надежного вещания, в синхронных системах, что мы вскорости и увидим.

В проблеме Византийского-вещания отдельному процессу g, *командующему (general)*, дается вход x_g, который берется из множества V (обычно {0, 1}). Процессы, отличные от командующего, назвыаются *помощниками (lieutenants)*. Должны выполняться следующие три требования.

(1) **Завершение.** Каждый корректный процесс p остановится на значении $y_p \in V$.

(2) **Соглашение**. Все корректные процессы останавливаются на одном и том же значении.

(3) **Зависимость**. Если командующий корректен, все корректные процессы останавливаются на x_g.

Можно, кроме этого, требовать **одновременности**, то есть, что все корректные процессы принимают решение в одном и том же импульсе. Все алгоритмы, обсуждаемые в этом и следующем разделах, удовлетворяют требованию **одновременности**; см. также Подраздел 14.2.6.

14.1.1 Граница Способности восстановления

Способность восстановления синхронных сетей после Византийских сбоев, как в случае асинхронных сетей (Теорема 13.25), ограничена t < N/3. Эта граница была впервые продемонстрирована Пизом, Шостаком и Лампортом [PSL80] представлением нескольких сценариев для алгоритма в присутствии N/3 или более Византийских процессов. В отличие от сценариев, используемых в доказательстве Теоремы 13.25, здесь корректные процессы получают противоречивую информацию, позволяющую заключить, что некоторые процессы являются сбойными. Однако, как оказывается, невозможно определить, какие процессы являются ненадежными, и неверные процессы могут навязать неверное решение.

Теорема 14.1 *t-Византийско-устойчивого протокола вещания при t>N/3 не существует.*

Доказательство. Как в ранних доказательствах, способность восстановления N/3 или выше позволяет разделить процессы на три группы (S, T, и U), каждая из которых может быть полностью сбойной. Группа, содержащая командующего, называется S. Противоречие получается из рассмотрения трех сценариев, изображенных на Рисунке 14.1, где сбойная группа обозначена двойным блоком.

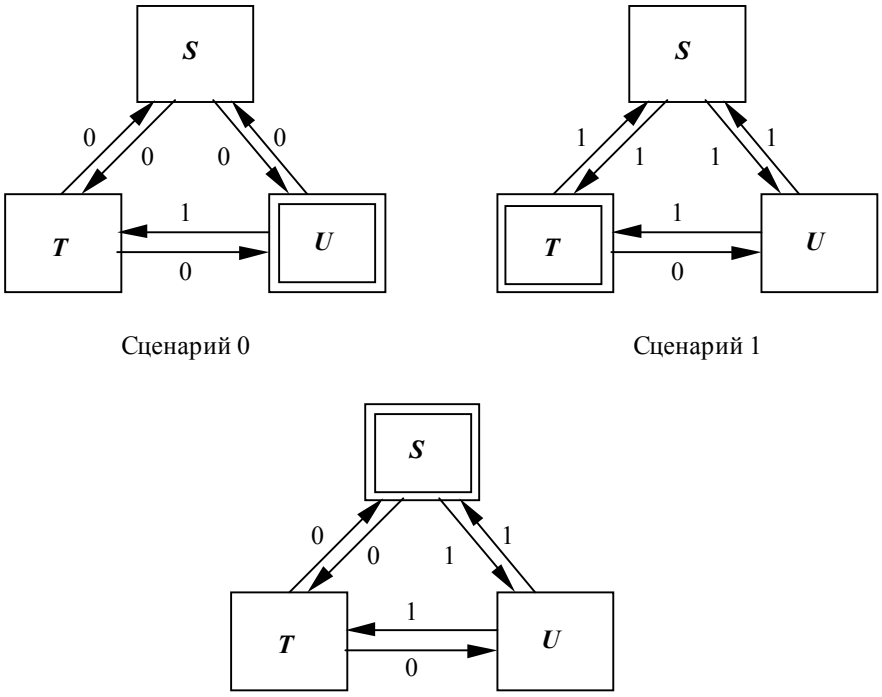


РИСУНОК 14.1 СЦЕНАРИИ ДЛЯ ДОКАЗАТЕЛЬСТВА ТЕОРЕМЫ 14.1.

В сценарии 0 командующий вещает значение 0, и процессы в группе U сбойные; в сценарии 1 командующий вещает 1 и процессы в T сбойные. В импульсе i сценария 0 процессы группы U посылают процессам группы T в точности те сообщения, которые они послали бы (согласно протоколу) в 1 сценарии. (То есть, сообщения, посланные в ответ на сообщения, полученные в импульсе i-1 сценария 1.) Процессам в S они посылают сообщения, направляемые протоколом. Процессы в S и T, конечно, посылают корректные сообщения во всех импульсах. Заметьте, что в этом сценарии только процессы группы U посылают неправильные сообщения, и спецификации протокола предписывают, что все корректные процессы, включая группу T, останавливаются на 0.

Сценарий 1 определен аналогично, но здесь процессы в T сбойные и они посылают сообщения, которые должны были послать в сценарии 0. В этом сценарии процессы в U останавливаются на 1.

В заключение рассмотрим сценарий 2, где процессы в S сбойные и ведут себя следующим образом. Процессам в T они посылают сообщения сценария 0 и процессам в U они посылают сообщения сценария 1. Теперь можно показать индукцией по номеру импульса, что сообщения, посланные T к U (или, от U к T) - точно те, что посланы в сценарии 0 (или 1, соответственно). Следовательно, для процессов в T сценарий 2 неотличим от сценария 0 и для процессов в U он неотличим от сценария 1. Из этого следует, что процессы в T останавливаются на 0, и процессы в U останавливаются на 1. Противоречие. □

В доказательстве используется то, что Византийские процессы могут посылать сообщения 1-сценария, даже если они получили только сообщения 0-сценария. То есть, процессы могут "лгать" не только о своем собственном состоянии, но

также и о сообщениях, которые они получили. Именно эту возможность можно устранить с помощью установления подлинности, как описано в Разделе 14.2; это ведет к способности восстановления N-1.

14.1.2 Алгоритм Византийского вещания

В этом подразделе будет показано, что верхняя граница способности восстановления, показанная в предыдущем подразделе, точна. Кроме того, противопоставляя ситуации в асинхронных сетях, максимальная способность восстановления достижима при использовании детерминированных алгоритмов. Мы представляем рекурсивный алгоритм, также Пиза и других [PSL80], который допускает t Византийских отказа при t < N/3. Способность восстановления - параметр алгоритма.

Алгоритм *Broadcast*(N, 0) дан как Алгоритм 14.2; он не допускает отказов (t = 0), и если отказов не происходит, все процессы останавливаются на входе командующего в 1 импульсе. Если отказ происходит, соглашение может быть нарушено, но завершение (и одновременность), тем не менее, гарантируется.

Импульс

1: Командующий посылает <**value**, x_g> всем процессам,
помощники не посылают.
Получить сообщения импульса 1.
Командующий принимает решение на x_g.
Помощники принимают решение следующим образом:
if от g в импульсе 1 было получено сообщение <**value**, x>
then принять решение *x*
else принять решение *udef*

АЛГОРИТМ 14.2 *BROADCAST (N, 0)*.

Протокол для способности восстановления t>0 (Алгоритм 14.3) использует рекурсивные вызовы процедуры для способность восстановления t-1. Командующий посылает свой вход всем помощникам в импульсе 1, и в следующем импульсе, каждый помощник начинает вещание полученного значения другим помощникам, но это вещание имеет способность восстановления t-1. Эта уменьшенная способность восстановления - трудноуловимый момент алгоритма, потому что (если командующий корректен) все t Византийские процессы могут находиться среди помощников, так что фактическое число отказов может превышать способность восстановления вложенного вызова *Broadcast*. Чтобы доказать правильность возникающего в результате алгоритма, необходимо рассуждать, используя способность восстановления t *и* фактическое число сбойных процессов f (см. Лемму 14.3). В импульсе t+1 вложенные вызовы производят решение, поэтому помощник p принимает решение в N-1 вложенных вещаниях. Эти N-1 решения хранятся в массиве W_p, из которого решение p получается большинством голосов (значение, полученное непосредственно от командующего, здесь игнорируется!). Для этого на массивах определяется детерминиро-

ванная функция *major*, с таким свойством, что, если v имеет большинство в W, (то есть, более половины элементов равны, то *major(W)=v*.

Импульс

1: Командующий посылает <**value**, x_g> всем процессам,

помощники не посылают.

Получить сообщения импульса 1.

Помощник p действует следующим образом.

if от g в импульсе 1 было получено сообщение <**value**, x>

then $x_p := x$ **else** $x_p := udef$

Объявить x_p другим помощникам, действуя как командующий

в $Broadcast_p(N-1, t-1)$ в следующем импульсе

(t+1)**: получить сообщения импульса t + 1.**

Командующий останавливается на x_g.

Для помощника p:

Для каждого помощника q в $Broadcast_p(N-1, t-1)$ встречается решение.

$W_p[q]$:= решение в $Broadcast_p(N-1, t-1)$;

$y_p := major(W_p)$

АЛГОРИТМ 14.3 ВЕЩАНИЕ (N, Т) (ДЛЯ Т> 0).

Лемма 14.2 (Завершение) *Если Broadcast(N, t) начинается в импульсе 1, каждый процесс принимает решение в импульсе t+1.*

Доказательство. Так как протокол рекурсивен, его свойства доказываются с использованием рекурсии по t.

В алгоритме Broadcast(N, 0) (Алгоритм 14.2), каждый процесс принимает решение в импульсе 1.

В алгоритме Broadcast(N, t) помощники начинают рекурсивные обращения алгоритма, Broadcast(N-1, t-1), в импульсе 2. Если алгоритм начат в импульсе 1, он принимает решение в импульсе t (это - гипотеза индукции), следовательно если он начат в импульсе 2, все вложенные вызовы принимают решение в импульсе t + 1. В одном том же импульсе принимается решение в Broadcast(N, t). □

Чтобы доказывать зависимость (также индукцией) предполагается, что командующий корректен, следовательно все t сбойных процесса находятся среди N-1 помощников. Так как t < (N - l) /3 не всегда выполняется, простую индукцию использовать нельзя, и мы рассуждаем, используя фактическое число неисправностей, обозначенное f.

Лемма 14.3 (Зависимость) *Если командующий корректен, если имеется f сбойных процессов, и если N > 2f+t, то все корректные процессы останавливаются на входе командующего.*

Доказательство. В алгоритме Broadcast(N, 0) если командующий корректен, все корректные процессы, останавливаются на значении входа генерала.

Теперь предположим, что лемма справедлива для Broadcast(N-1, t-1). Так как командующий корректен, он посылает свой вход всем помощникам в импульсе 1, так что каждый корректный помощник q выбирает $x_q = x_g$. Теперь N > 2f + t означает (N - 1) > 2f + (t - 1), поэтому гипотеза индукции применяется к вложенным вызовам, даже если теперь все f сбойных процесса находятся среди помощников. Таким образом, для корректных помощников p и q, решение p в Broadcast(N-1, t-1) равняется x_q, то есть, x_g. Но, поскольку строгое большинство помощников корректно (N > 2f + t), процесс p завершится с W_p, в котором большинство значений равняется x_g. Следовательно, применение *major* к p выдает нужное значение x_g. □

Лемма 14.4 (Соглашение) *Все корректные процессы останавливается на одном и том же значении.*

Доказательство. Так как зависимость означает соглашение в выполнениях, в которых командующий является корректным, мы теперь сконцентрируемся на случае, когда командующий сбойный. Но тогда самое большее t-l помощников сбойные, что означает, что вложенные вызовы функционируют в пределах своих способностей восстановления!

Действительно, t < N/3 означает t - 1 < (N - 1) / 3, следовательно, вложенные вызовы удовлетворяют соглашению. Таким образом, все *корректные* помощники остановятся на одном и том же значении x_q для *каждого* помощника q во вложенном вызове $Broadcast_q(N-1, t-1)$. Таким образом, каждый корректный помощник вычисляет точно такой же вектор W в импульсе t + 1, что означает, что применение *major* дает тот же самый результат в каждом корректном процессе. □

Теорема 14.5 *Протокол Broadcast(N, t) (Алгоритм 14.2/14.3) - t-Византийско-устойчивый протокол вещания при t < N/3.*

Доказательство. Завершение было показано в Лемме 14.2, зависимость в Лемме 14.3, и соглашение в Лемме 14.4. □

Протокол *Broadcast* принимает решение в (t + 1)-ом импульсе, что является оптимальным; см. Подраздел 14.2.6. К сожалению, его сложность по сообщениям экспоненциальная; см. Упражнение 14.1.

14.1.3 Полиномиальный Алгоритм Вещания

В этом разделе мы представляем Византийский алгоритм вещания Долева и других [DFF+82], который использует только полиномиальное число сообщений и бит. Временная сложность выше, чем у предыдущего протокола; алгоритм

требует 2t+3 импульса для достижения решения. В следующем описании будет предполагаться, что N = 3t + 1, и позже будет обсужден случай N > 3t + 1.

Алгоритм использует два порога, L = t + 1 и H = 2t + 1. Эти числа выбираются так, что (1) каждое множество из L процессов содержит по крайней мере один корректный процесс, (2) каждое множество из H процессов содержит по крайней мере L корректных процессов, и (3) имеется по крайней мере H корректных процессов. Обратите внимание, что предположение $N \geq 3t+1$ необходимо и достаточно для выбора L и H, удовлетворяющих этим трем свойствам.

Алгоритм обменивается сообщениями типа <**bm**, v>, где v или значение 1, или имя процесса (bm обозначает "broadcast message", "вещать сообщение".) Процесс p содержит двухмерную булеву таблицу R, где $R_p[q,v]$ истинен тогда и только тогда, когда p получил сообщение <**bm**, v> от процесса q. Первоначально все элементы таблицы ложны, и мы полагаем, что таблица обновляется в фазе получения каждого импульса (это не показано в Алгоритме 14.4). Заметьте, что R_p монотонна в импульсах, то есть, если $R_p[q,v]$ становится истинным в некотором импульсе, он остается истиной в более поздних импульсах. Кроме того, так как только корректные процессы "выкрикивают" сообщения, для корректных p, q, и r в конце каждого импульса имеем: $R_p[r,v] = R_q[r,v]$.

В отличие от протокола *Broadcast* предыдущего подраздела, протокол Долева и других является асимметричным в значениях 0 и 1. Решение 0 - значение по умолчанию и выбирается, если в обмене было недостаточно много сообщений. Если командующий имеет вход 1, он будет "выкрикивать" сообщения <**bm**, 1>, и получение достаточно большого количества отраженных сообщений, типа <**bm**, q>, заставляет процесс принять решение 1.

В алгоритме уместны три типа действия: *инициирование*, *поддержка* и *подтверждение*.

(1) *Поддержка*. Процесс p *поддерживает* процесс q в импульсе i, если в более ранних импульсах p получил достаточно доказательств, что q послал сообщения <**bm**, 1>; если дело обстоит так, p пошлет <**bm**, q> сообщения в импульсе i. Процесс p *прямо поддерживает* q, если p получил сообщение <**bm**, 1> от q. Процесс p *косвенно поддерживает* q, если p получил сообщение <**bm**, q> по крайней мере от L процессов. Множество процессов S_p, поддерживаемых p, определяется неявно из R_p

$$DS_p = \{q : R_p[q,1]\} \quad (*\text{прямая}*)$$

$$IS_p = \{q : \#\{r : R_p[r,q]\} \geq L\} \quad (*\text{косвенная}*)$$

$$S_p = DS_p \cup IS_p$$

Порог для становления косвенным поддерживающим означает, что если корректный процесс поддерживает процесс q, то q послал по крайней мере одно сообщение <**bm**, 1>. Действительно, предположим, что некоторый корректный процесс поддерживает q, пусть i - первый импульс, в котором это происходит. Так как косвенная поддержка q требует получения по крайней мере одного сообщения <**bm**, q> от корректного процесса в более раннем импульсе, первая поддержка корректным процес-

сом процесса q - прямая. Прямая поддержка корректным процессом означает, что этот процесс получил сообщение <**bm**, 1> от q.

(2) *Подтверждение*. Процесс p *подтверждает* процесс q после получения сообщения <**bm**, q> от H процессов, то есть,

$$C_p = \{q: \#\{r: R_p[r,q]\} \geq H\}$$

Выбор порогов означает, что, если корректный процесс p подтверждает q, то все корректные процессы подтверждают q самое большее одним импульсом позже. Действительно, предположим, что p подтверждает q после импульса i. Процесс p получил сообщения <**bm**, q> от H процессов, включая (выбором порогов) по крайней мере L корректных поддерживающих для q. Корректные поддерживающие для q посылают сообщение <**bm**, q> всем процессам, что означает, что в импульсе i все корректные процессы получают по крайней мере L сообщений <**bm**, q> и поддерживают q в импульсе i + 1. Таким образом, в импульсе i + 1 все корректные процессы посылают <**bm**, q>, и поскольку число корректных процессов по крайней мере H, каждый корректный процесс получает достаточную поддержку, чтобы подтвердить q.

(3) **Инициирование**. Процесс p *инициирует*, когда у него есть достаточно доказательств того, что окончательное значения решения будет 1. После инициирования, процесс p посылает сообщения <**bm**, 1>. Инициирование может быть вызвано тремя типами доказательств, а именно (1) p - командующий и $x_p = 1$, (2) p получает <**bm**, 1> от командующего в импульсе 1, или (3) p подтвердил достаточно много *помощников* в конце последнего импульса. Последняя возможность в частности требует некоторого внимания, так как число подтвержденных помощников, которое является "достаточным" увеличивается в течение выполнения, и подтвержденный командующий не идет в счет для этого правила. В первых трех импульсах L помощников должны быть подтверждены, чтобы инициировать, но начиная с импульса 4 порог увеличивается через каждые два импульса. Таким образом, инициирование согласно правилу (3) требует, чтобы к концу импульса i, $Th(i) = L + \max(0, \lfloor i/2 \rfloor - 1)$ помощников было подтверждено. Запись C_p^L в алгоритме обозначает множество подтвержденных помощников, то есть, $C_p \setminus \{g\}$. Инициирование процессом p представляется булевой переменной ini_p.

Если корректный помощник r инициирует в конце импульса i, все корректные процессы подтверждают r в конце импульса i + 2. Действительно, r “выкрикивает” <**bm**, 1> в импульсе i + 1, поэтому все корректные процессы (прямо) поддерживают r в импульсе i + 2, так что каждый процесс получает по крайней мере H сообщений <**bm**, q> в этом импульсе.

Алгоритм продолжается в течение 2t + 3 импульсов; если процесс p подтвердил по крайней мере H процессов (здесь считает командующий) к концу того им-

пульса, p принимает решение 1, иначе p принимает решение 0. См. Алгоритм 14.4.

---

var $R_p[\ldots,\ldots]$: boolean **init** false

ini_p : boolean **init if** $p = g \wedge x_p = 1$ **then** true **else** false;

Импульс i: (* Фаза посылки *)

if ini_p **then** shout<**bm**, 1>;

forall $q \in S_p$ **do** shout<**bm**, q>;

получить все сообщения импульса i;

(*обновление состояния*)

if i = 1 **and** $R_p[g,1]$ **then** $ini_p := true$;

if $\#C_p^L \geq Th(i)$ **then** $ini_p := true$;

if i = 2t + 3 **then** (*принять решение*)

if $\#C_p \geq H$ **then** $y_p := 1$ else $y_p := 0$

АЛГОРИТМ 14.4 ПРОТОКОЛ С НАДЕЖНЫМ ВЕЩАНИЕМ.

Командующий, являющийся единственным процессом, который может самостоятельно предписать инициирования (в других процессах) занимает мощное положение в алгоритме. Легко заметить, что, если командующий корректно инициирует, начинается лавина сообщений, которая заставляет все корректные процессы подтвердить H процессов и остановиться на значении 1. К тому же если он не инициирует, то нет "критической массы" сообщений, которая ведет к инициированию любого корректного процесса.

Лемма 14.6 *Алгоритм 14.4 удовлетворяет завершению (и одновременности) и зависимости.*

Доказательство. Из алгоритма видно, что все корректные процессы принимают решение в конце импульса 2t + 3, что показывает завершение и одновременность. Чтобы показать зависимость, мы предположим, что командующий корректен.

Если командующий корректен и имеет вход 1, он “выкрикивает” сообщение <**bm**, 1> в импульсе 1, заставляя каждый корректный процесс q инициировать. Следовательно, каждый корректный процесс q “выкрикивает” <**bm**, 1> в импульсе 2, так что к концу импульса 2 каждый корректный процесс p поддерживает все другие корректные процессы. Это означает, что в импульсе 3 каждый корректный p “выкрикивает” <**bm**, q> для каждого корректного q, так что в конце импульса 3 каждый корректный процесс получает <**bm**, q> от каждого другого корректного процесса, заставляя его подтвердить q. Таким образом с конца раунда 3 каждый корректный процесс подтвердил H процессов, что означает, что окончательное решение будет 1. (Командующий поддерживается и

подтверждается всеми корректными процессами одним импульсом ранее других процессов.)

Если командующий корректен и имеет вход 0, он не “выкрикивает” <**bm**, 1> в импульсе 1, чего не делает и никакой другой корректный процесс. Предположим, что никакой корректный процесс не инициировал в импульсах с 1 по i-1; тогда никакой корректный процесс не посылает <**bm**, 1> в импульсе i. В конце импульса i никакой корректный процесс не поддерживает и не подтверждает никакого корректного процесса, так как, как мы видели ранее, это подразумевает, что последний процесс послал сообщение <**bm**, 1>. Следовательно, никакой корректный процесс не инициирует в конце импульса i. Отсюда следует, что никакой корректный процесс не инициирует вообще. Это означает, что никакой корректный процесс никогда не подтверждает корректный процесс, так что никакой корректный процесс не подтверждает более t процессов, и решение в конечном импульсе - 0. □

Мы продолжим доказательством соглашения, и будем предполагать в следующих леммах, что командующий сбойный. Достаточная "критическая масса" сообщений, ведущая неизбежно к 1-решению, создается инициированием L корректных процессов, когда имеется по крайней мере четыре импульса.

Лемма 14.7 *Если L корректных процессов инициируют к концу импульса i, i < 2t, то все корректные процессы останавливаются на значении 1.*

Доказательство. Пусть i - первый импульс, в конце которого по крайней мере L корректных процессов инициируют, и пусть A обозначает множество корректных процессов, которые инициировали в конце импульса i. Все процессы в A - помощники, так как командующий сбойный. В конце импульса i + 2 все корректные процессы подтвердили помощников в A; мы покажем, что в это время все корректные процессы инициируют.

Случай i = 1: Все корректные процессы подтвердили помощников из A к концу импульса 3, и инициируют, так как $\# A \geq L = Th(3)$.

Случай $i \geq 2$: По крайней мере один процесс, допустим r, из A инициировал в импульсе i, так как он подтвердил Th(i) помощников (инициирование с помощью получения <**bm**, 1> от командующего возможно только в импульсе 1). Эти Th(i) помощников подтверждены всеми корректными процессами в конце импульса i + l, но r не принадлежит к этим Th(i) подтвержденным помощникам, так как r впервые посылает сообщения <**bm**, 1> в импульсе i + 1. Однако, все корректные процессы подтвердят r к концу импульса i + 2; таким образом они подтвердили по крайней мере Th(i) + 1 помощников в конце раунда i + 2. В заключение, так как Th(i+2)= Th(i)+1, все корректные процессы инициируют.

Теперь, поскольку все корректные процессы инициируют к концу раунда i + 2, они подтверждены (всеми корректными процессами) в конце раунда i + 4, следовательно все корректные процессы подтвердили по крайней мере H помощников. Поскольку предполагалось, что i < 2t, $i + 4 \leq 2t + 3$, так что все корректные процессы останавливаются на значении 1. □

Для любого корректного процесса, чтобы остановиться на 1, необходима "лавина", состоящая из инициирования по крайней мере L на корректных процессов. Действительно, 1-решение требует подтверждения по крайней мере H процессов, включая L корректных, что эти корректные процессы инициировали. Вопрос в том может ли Византийский заговор откладывать начало лавины достаточно долго, чтобы вызвать 1-решение в некоторых корректных процессах, без навязывания его в общем согласно Лемме 14.7. Конечно, ответ - нет, так как имеется ограничение на то, как долго заговор может откладывать лавину, и число импульсов, 2t + 3, выбрано точно так, чтобы предотвратить это. Причина - возрастающий порог для требуемого числа подтвержденных процессов; в более поздних импульсах он становится настолько высок, что уже стало необходимо L инициированных корректных помощников, чтобы инициировать следующий корректный процесс.

Лемма 14.8 *Предположим, что по крайней мере L корректных процессов инициируют в течение алгоритма, и пусть i - первый импульс, в конце которого инициируют L корректных процессов. Тогда i < 2t.*

Доказательство. Чтобы инициировать в импульсе 2t или выше, корректный процесс должен подтвердить по крайней мере Th(2t) = L + (t-1) помощников. Так как командующий сбойный, то есть самое большее t-1 сбойных помощников, поэтому по крайней мере L корректных помощников должны были быть подтверждены, что показывает, что уже, в более раннем импульсе, L корректных процессов, должно быть, инициировали. □

Теорема 14.9 *Алгоритм вещания Долева и других (Алгоритм 14.4) - t-Византийско-устойчивый протокол вещания.*

Доказательство. Завершение (и одновременность также) и зависимость показаны в Лемме 14.6. Чтобы показать соглашение, предположим, что имеется корректный процесс, который останавливается на значении 1. Мы заметили, что это означает, что по крайней мере L корректных процессов инициировали. По Лемме 14.8, впервые это случалось в импульсе i < 2t. Но тогда по Лемме 14.7, все корректные процессы останавливаются на значении 1. □

Чтобы облегчить представление алгоритма, было сделано предположение, что процессы повторяют в каждом раунде сообщения, которые они посылали в более ранних раундах. Поскольку корректные процессы записывают сообщения, полученные в более ранних раундах, это не нужно, поэтому достаточно послать каждое сообщение только. Таким образом, каждый корректный процесс посылает каждое из N+1 возможных сообщений другому процессу самое большее один раз, что ограничивает сложность по сообщениям величиной $\Theta(N^3)$. Так как имеется только N+1 различных сообщений, каждое сообщения должно содержать только O (log N) бит.

Если число процессов превышает 3t + 1, для выполнения алгоритма выбирается совокупность 3t активных помощников. (Выбор выполняется статически, например, выбирая 3t процессов, чьи имена следуют за g в порядке имен процессов. Командующий и активные помощники сообщают пассивным помощникам о своем решении, и пассивные помощники останавливаются на значении, которое они получают от более t процессов. Сложность по сообщениям этого послойного подхода - $\Theta(t^3 + t \cdot N)$, и разрядная сложность - $\Theta(t^3 \log t + tN)$.

14.2 Протоколы с Установлением Подлинности

Злонамеренное поведение, рассматриваемое до сих пор включало неправильную пересылку информации в дополнение к посылке неправильной информации о собственном состоянии процесса. К счастью, это чрезвычайно злонамеренное поведение Византийских процессов может быть ограничено с помощью криптографических средств, которые сделали бы Теорему 14.1 недействительной. Действительно, в сценариях, используемых в ее доказательстве, сбойные процессы посылали бы сообщения как в сценарии 1, получив только сообщения сценария 0.

В этом разделе предполагается наличие средства для цифровой *подписи* и *установления подлинности* сообщений. Процесс p, посылающий сообщение M, добавляет к этому сообщению некоторую дополнительную информацию $S_p(M)$, которая называется *цифровой подписью* p для сообщения M. В отличие от рукописных подписей, цифровая подпись зависит от M, что делает бесполезным копирование подписи в другие сообщения. Схема подписи удовлетворяет следующим свойствам.

(1) Если p корректен, только p может правдоподобно вычислить $S_p(M)$. Это вычисление - *подпись* сообщения M.

(2) Каждый процесс может эффективно проверять (имея p, M и S) $S = S_p(M)$. Эта проверка - *установление подлинности* сообщения M.

Схемы подписи основаны на *частных* и *общих* ключах. Первое предположение не исключает того, что Византийские процессы могут открыть свои секретные ключи друг другу, что позволяет одному Византийскому процессу подделать подпись другого. Предполагается, что только корректные процессы хранят свои частные ключи в секрете.

Мы изучим реализацию схем подписи в Подразделах с 14.2.2 по 14.2.5. В следующем подразделе, сообщение <**msg**>, подписанное процессом p, то есть, пара, содержащая <**msg**> и $S_p(< msg >)$, обозначается <**msg**> : p.

14.2.1 Протокол Высокой Степени Восстановления

Эффективный Византийский алгоритм вещания, использующий полиномиально много сообщений и t+1 импульсов, был предложен Долевом и Стронгом [DS83]. Установление подлинности, используемое в этом протоколе, разрешает неограниченную способность восстановления. Мы заметим, тем не менее, что могут отказать не более N процессов (из N), и N процессов отказывают, все требова-

ния примитивно удовлетворяются; следовательно пусть t < N. Их протокол основан на более раннем, предложенном Лампортом, Шостаком и Пизом [LSP82], который является экспоненциальным по числу сообщений. Мы представляем сначала последний протокол.

В импульсе 1 командующий "выкрикивает" сообщение <**value**, x_g> : g, содержащее свой (подписанный) вход.

В импульсах со 2 по t + 1 процессы подписывают и пересылают сообщения, которые они получили в предыдущем импульсе; следовательно, сообщение, которым обмениваются в импульсе i, содержит i подписей. Сообщение <**value**, v> : g : p_2 : ... : p_i называется *действительным* (имеющим силу) для получающего процесса p, если справедливо следующее.

(1) Все i подписей корректны.

(2) i подписей от i различных процессов.

(3) p не встречается в списке подписей.

В течение алгоритма, процесс p содержит множество W_p значений, содержащихся в действительных сообщениях, полученных p; первоначально это множество пусто, и значение каждого действительного сообщения вставляется в него.

Сообщения, пересылаемые в импульсе i - в точности те действительные сообщения, полученные в предыдущем импульсе. В конце импульса t + 1, процесс p принимает решение основанное на W_p. Если W_p состоит из одиночного элемента {v}, p принимает решение v, иначе p принимает решение значения по умолчанию (например, 0). Чтобы сэкономить на числе сообщений, p пересылает сообщение <**value**, v> : g : p_2 : ... : p_i : p только процессам, не встречающимся в списке g, p_2, ..., p_i. Эта модификация не имеет никакого влияния на поведение алгоритма, так как для процессов в списке сообщение не действительно.

Теорема 14.10 *Алгоритм Лампорт, Шостака и Пиза - корректный Византийский алгоритм вещания при t < N, использующий t + 1 импульс.*

Доказательство. Все процессы принимают решение в импульсе t + 1, что подразумевает и завершение и одновременность алгоритма.

Если командующий корректен и имеет вход v, все процессы получают его сообщение <**value**, x_g>: g в импульсе 1, так что все корректные процессы включают v в W. Никакое другое значение не вставляется в W, так как никакое другое значение никогда не подписывается командующим. Следовательно, в импульсе t + 1 все процессы имеют W = {v} и останавливаются на v, что означает зависимость.

Чтобы показать соглашение, мы получим, что для корректных процессов p и q, $W_p = W_q$ в конце импульса t + 1. Предположим, $v \in W_p$ в конце импульса t + 1, и пусть i - импульс, в котором p вставил v в Wp, по получении сообщения <**value,** v> : g : p_2 : ... : p_i.

Случай 1: Если q встречается в g, p_2, ..., p_i, то q сам видел значение v и вставил его в W_q.

Случай 2: Если q не встречается в последовательности g, p_2, ..., p_i и $i \leq t$, то p пересылает сообщение <**value**, v>: g : p_2 : ... : p_i : p процессу q в импульсе i + i, так что q утверждает (придает силу) v самое позднее в импульсе i + 1.

Случай 3: Если q не встречается в последовательности g, p_2, ..., p_i, и i = t + 1, заметьте, что сообщение, полученное p, было подписано t + 1 последовательными процессами, включая по крайней мере один корректный процесс. Этот процесс переслал сообщение всем другим процессам, включая q, так что q видит v.

Так как $W_p = W_q$ к концу импульса t + 1, p и q принимают одинаковое решение. □

Завершить алгоритм ранее импульса t + 1 невозможно. Во всех импульсах до t, корректный процесс мог бы получать сообщения, созданные и пересланные только сбойными процессами, и не посланные другим корректным процессам, что могло бы вести к противоречивым решениям.

Промежуточный результат предыдущего алгоритма, а именно соглашение о множестве значений среди всех корректных процессов, более сильное, чем необходимо для достижения соглашения об одиночном значении; Это было замечено Долевом и Стронгом [DS83], которые предложили более эффективную модификацию. Фактически достаточно, что в конце импульса t + 1, или (a) для каждого корректного p множество W_p - *один и тот же* одиночный элемент, или (b) ни для какого корректного p множество W_p не является одиночным элементом. В первом случае все процессы принимают решение v, в последнем случае они все принимают решение 0 (или, если желательно изменить алгоритм таким образом, они принимают решение "командующий сбойный").

Алгоритмом Долева и Стронга достигается более слабое требование на множества W. Вместо того, чтобы передавать каждое действительное сообщение, процесс p пересылает самое большее два сообщения, а именно одно сообщение с первым и одно сообщение со вторым значением, принятым p. Полное описание алгоритма оставлено читателю.

Теорема 14.11 *Алгоритм Долева и Стронга, описанный выше - протокол Византийского-вещания, использующий t + 1 импульс и самое большее* $2N^2$ *сообщений.*

Доказательство. Завершение и одновременность доказываются, как в предыдущем протоколе, так как каждый корректный процесс принимает решение в конце импульса t + 1. Зависимость выводится так же, как в предыдущем протоколе. Если g правильно “выкрикивает” v в первом импульсе, все корректные процессы принимают v в этом импульсе, и никакое другое значение никогда не принимается; следовательно, все корректные процессы останавливаются на v. Заявленная сложность по сообщениям следует из факта, что каждый (корректный) процесс “выкрикивает” самое большее два сообщения.

Чтобы показать соглашение, мы покажем, что для корректных процессов p и q, W_p и W_q удовлетворяют в конце импульса t + 1 следующему.

(1) Если $W_p = \{v\}$, то $v \in W_q$.

(2) Если $\#W_p > 1$, то $\#W_q > 1$.

Для (1): Предположим, что p принял значение v после получения сообщения <**value**, v>: g : p_2 : ... : p_i в импульсе i, и рассуждаем как в доказательстве Теоремы 14.10:

Случай 1: Если q встречается среди g, p_2, ..., p_i, q точно принял v.

Случай 2: Если q не встречается среди g, p_2, ..., p_i, и $i \le t$, $i \le t$, то p пересылает значение процессу q, который примет его в этом случае.

Случай 3: Если q не встречается и i = t + 1, по крайней мере один из процессов, который подписал сообщение, допустим r, корректен. Процесс r также переслал значение v процессу q, что значит, что v находится в W_q.

Для (2): Предположим, что $\#W_p > 1$ в конце алгоритма, и пусть w - второе значение, принятое p. Снова подобным рассуждением можно показать, что $w \in W_q$, что означает, что $\#W_q > 1$. (Равенство W_p и W_q не может быть получено, так как процесс p не будет пересылать свое третье принятое значение или более поздние.)

Доказав (1) и (2), предположим, что корректный процесс p останавливается на $v \in W_p$, то есть $W_p = \{v\}$. Затем, из (1), v содержится во всех W_q для корректного q, но, следовательно, W_q не шире одиночного элемента {v}; иначе W_p - не одиночный элемент, из (2). Следовательно, каждый корректный процесс q также останавливается на v. Далее, предположим, что корректный процесс p останавливается на значении по умолчанию, так как W_p - не одиночный элемент. Если W_p пуст, каждый корректный q имеет пустой W_q по (1) и если $\#W_p > 1$, то $\#W_q > 1$ по (2); следовательно, q также останавливается на значении по умолчанию.

□

Долев и Стронг в дальнейшем улучшили алгоритм и получили алгоритм, который решает проблему Византийского-вещания за то же самое число импульсов и всего за O(Nt) сообщений.

14.2.2 Реализация Цифровых Подписей

Так как подпись p $S_p(M)$ должна представлять собой достаточное доказательство того, что p - создатель сообщения, подпись должна состоять из некоторой формы информации, которая

(1) Может быть эффективно вычислена процессом p (подписана);

(2) Не может быть эффективно вычислена любым другим процессом, отличным от p (подделана).

Мы должны немедленно отметить, что, для большинства схем подписи, использующихся сегодня, второе требование не доказано до такой степени, что показана экспоненциальная трудность проблемы подделки. Обычно, проблема подделки, как показывают, связана (или иногда эквивалентна) с некоторой вычислительной проблемой, которая изучалась в течение длительного времени в отсутствие знания о ее полиномиальной разрешимости. Например, подделывание подписей в схеме Фиата-Шамира требует разлагать на множители большие целых числа; так как последняя задача (предположительно) в вычислительном отношении очень сложна, первая, должно быть, также сложна в вычислительном отношении.

Были предложены схемы подписи, основанные на различных, как предполагается, трудных проблемах, типа вычисления дискретного логарифма, разложения на множители больших чисел, проблемы рюкзака. Требования (1) и (2) подразумевают, что процесс p должен иметь вычислительное "преимущество" над другими процессами; это преимущество - некоторая секретная информация во владении p, *секретный* (или частный) ключ p. Таким образом, вычисление $S_p(M)$ эффективно, когда секретный ключ известен, но (предположительно) трудно без этой информации. Ясно, что если p удается хранить свой ключ в тайне, то только p может с легкостью вычислять $S_p(M)$.

Все процессы должны уметь проверять подписи, то есть, имея сообщение M и подпись S, должно быть возможно эффективно проверить, что S действительно был вычислен из M с помощью секретного ключа p. Эта проверка требует, чтобы была раскрыта некоторая информация о секретном ключе p; эта информация - *общий ключ* p. Общий ключ должен позволять проверку подписи, но нужно, чтобы его быть невозможно или по крайней мере в вычислительном отношении трудно использовать для вычисления секретного ключа p или подделки подписей.

Наиболее успешные схемы подписи, предложенные до настоящего времени, основаны на вычислениях из теории чисел в арифметических кольцах по модулю больших чисел. Базисные арифметические операции добавления, умножения, и возведения в степень могут выполняться в этих кольцах за полиномиальное время от длины модуля (в битах). Деление возможно, если знаменатель и модуль взаимно просты (то есть, не имеют общих простых делителей), и может также выполняться за полиномиальное время. Так как подписание и проверка требуют вычислений над сообщением, M интерпретируется как большое число.

14.2.3 Схема Подписи ЭльГамаля

Схема подписи ЭльГамаля [EIG85] основана на функции теории чисел под названием *дискретный логарифм*. Для большого простого числа P, группа по умножению по модулю P, обозначаемая Z_P^*, содержит P-1 элементов и чвлчется *циклической*. Последнее означает, что можно выбрать такой элемент $g \in Z_P^*$, что P-1 чисел

$$g^0 = 1, g^1, g^2, \ldots, g^{P-3}, g^{P-2}$$

все различны и следовательно, перечисляют все элементы Z_P^*. Такое g называется *генератором* Z_P^*, или также *ïåðâîîáðàçíûì êîðíåì* по модулю P. Генератор не уникален; обычно их много. Имея фиксированное P и генератор g, для каждого $x \in Z_P^*$ имеется *уникальное* целое число i по модулю P-1 такое, что $g^i = x$ (равенство в Z_P^*). Это *i* называется *дискретным логарифмом* (иногда *индексом*) x. В отличие от вышеупомянутых базисных арифметических операций, вычислять дискретный логарифм *не просто*. Это - хорошо-изученная проблема, для которой до настоящего времени не найдено эффективное общее решение, но также не была доказана ее труднорешаемость; см. [0dl84] для краткого обзора результатов.

Схема подписи ЭльГамаля [EIG85] основана на трудности вычисления дискретных логарифмов. Процессы совместно используют большое простое число P и ïåðâîîáðàçíûé êîðåíь g из Z_P^*. Процесс p выбирает в качестве своего секретного ключа число d, случайно между 1 и P - 2, и общий ключ p - число $e = g^d$; заметьте, что d - дискретный логарифм e. Подпись p можно эффективно вычислить, зная логарифм e, и следовательно она формирует неявное доказательство того, что подписывающий знает d.

Действительная подпись для сообщения M - пара (r, s), удовлетворяющая $g^M = e^r r^s$. Такую пару p легко найдет с помощью секретного ключа d. Процесс p выбирает произвольное число a, взаимно простое с P-1, и вычисляет

$$r = g^a \pmod{P}$$

и

$$s = (M - dr)a^{-1} \pmod{(P-1)}$$

Эти числа действительно удовлетворяют

$$e^r r^s = e^r (g^a)^{(M-dr)a^{-1}} = g^{dr} g^{M-dr} = g^M.$$

(Все равенства в Z_P^*.) Действительность подписи S = (r, s) для сообщения M легко проверить, проверяя на равенство $g^M = e^r r^s$.

Алгоритмы для дискретного логарифма. Так как секретный ключ p, d, равняется дискретному логарифму общего ключа, e, схема раскрыта, если можно эффективно вычислять дискретные логарифмы по модулю P. До настоящего времени, не известно эффективного алгоритма, чтобы делать это в общем случае или подделывать подписи любым другим способом.

Общий алгоритм для вычисления дискретных логарифмов был представлен Одлыжко [0dl84]. Его сложность имеет тот же порядок, как у хорощо известных алгоритмов для разложения на множители целых чисел, столь же больших как P. Алгоритм сначала вычисляет несколько таблиц, используя только P и g, и, во второй фазе, вычисляет логарифмы для данных чисел. Если Q самый большой простой множитель P-1, время для первой фазы и размер таблиц имеют порядок

Q; следовательно, желательно выбрать P такой, что P-1 имеет большой простой множитель. Вторая фаза, подсчет логарифмов, может выполняться в течении секунд даже на очень маломощных компьютерах. Следовательно, необходимо менять P и g достаточно часто, допустим каждый месяц, так, чтобы таблицы для конкретного P устаревали до их завершения.

Рандомизированное подписывание (подписывание с уравниванием вероятностей). Рандомизация в процедуре подписывания делает каждую из $\phi(P-1)$ различных подписей[1] для данного сообщения одинаково вероятным результатом процедуры подписывания. Таким образом, один и тот же документ, подписанный дважды, почти обязательно произведет две различных действительных подписи. Рандомизация необходима в процедуре подписывания; если p подписывает два сообщения, пользуясь одним и тем же значением a, из подписей можно вычислить секретный ключ p; см. Упражнение 14.6.

14.2.4 Схема Подписи RSA

Если n - большое число, произведение двух простых чисел P и Q, то очень трудно вычислить квадратный корень и корни более высоких порядков по модулю n, если не известно разложение на множители. Возможность вычисления квадратных корней можно использовать, чтобы найти множители n (см. Упражнение 14.7), что показывает, что вычисление квадратных корней является таким же трудным, как и разложение на множители.

В схеме подписи Ривеста, Шамира и Эйдлмана [RSA78], общий ключ p - большое число n, разложение которого на множители p знает, и показатель степени e. Подпись p для сообщения M - e-й корень M по модулю n, который легко проверить, пользуясь возведением в степень. Этот корень более высокого порядка находится p также с использованием возведения в степень; при генерации своего ключа p вычисляет число d такое, что $de = 1(\mathrm{mod}(\phi(n)))$, что означает, что $(M^d)^e = M$, то есть, M^d - e-й корень M. Секретный ключ p состоит только из номера d, то есть, p не должен запоминать разложение на множители n.

В схеме RSA, p показывает свой идентификатор вычислением корней по модулю n, что требует (неявного) знания разложения n на множители; предполагается, что только p знает его. В этой схеме каждый процесс использует свой модуль.

14.2.5 Схема Подписи Фиата-Шамира

Более тонкое использование трудности нахождения (квадратных) корней сделано в схеме Фиата и Шамира [FS86]. В RSA схеме процесс подписывает сообщение, показывая, что он способен вычислять корни по модулю своего общего ключа, а способность вычислять корни возможно требует знания разложения на множители. В схеме Фиата-Шамира процессы используют *общий* модуль n, разложение на множители которого известно только доверенному центру. Процес-

---

[1] ϕ - функция Эйлера “фи”; $\phi(n)$ - размер Z_n^*

су p даются квадратные корни некоторых специфических чисел (в зависимости от идентификатора p), и подпись p для M обеспечивает доказательство того, что подписывающий знает эти квадратные корни, но не раскрывая их.

Преимущество схемы Фиата-Шамира над RSA схемой - более низкая арифметическая сложность и отсутствие отдельного общего ключа для каждого процесса. Недостаток - потребность в доверенном источнике, который выдает секретные ключи. Как упоминалось прежде, схема использует большое целое число n, произведение двух больших простых чисел, известных только центру. Кроме того имеется односторонняя псевдо-случайная функция f, отображающая строки на Z_n^*; эта функция известна и может быть вычислена каждым процессом, но обратная функция не может быть вычислена.

Секретные и общие ключи. В качестве секретного ключа p даны квадратные корни с s_1 по s_k k чисел по модулю n, а именно $s_j = \sqrt{v_j^{-1}}$, где $v_j = f(p, j)$. v_j можно считать общими ключами p, но поскольку они могут быть вычислены из идентификатора p, их не нужно хранить. Чтобы избежать технических неудобств, мы предположим, что эти k чисел - квадратичные остатк по модулю n. Квадратные корни могут быть вычислены центром, который знает множители n.

Подписавание сообщений: первая попытка. Подпись p неявно доказывает, что подписывающий знает корни v_j, то есть, может выдать число s такой, что $s^2 v_j = 1$. Такое число - s_j, но посылка самого s_j раскрыла бы секретный ключ; чтобы избежать раскрытия ключа, схема использует следующую идею. Процесс p выбирает произвольное число r и вычисляет $x = r^2$. Теперь p - единственный процесс, который может выдать число y, удовлетворяющее $y^2 v_j = x$, а именно, $y = rs_j$. Таким образом, p может показывать свое знание s_j без их раскрытия, посылая пару (x, y), удовлетворяющую $y^2 v_j = x$. Так как p не посылает число r, вычислить s_j из этой пары не возможно без вычисления квадратного корня.

Но имеются две проблемы с подписями, состоящими из таких пар. Во-первых, любой может произвести такую пару, жульничая следующим образом: *сначала* выбрать y, и впоследствии вычислить $x = y^2 v$. Во-вторых, подпись не зависит от сообщения, так что процесс, получивший подписанное сообщение от p, может скопировать подпись на любое поддельное сообщение. Трудный вопрос этой схемы подписи - сделать так, чтобы p показывал знание корня *произведения из подмножества* v_j, где подмножество зависит от сообщения и произвольного числа. Шифрование сообщения и произвольного числа с помощью f не дает подделывающему сначала выбрать y. Чтобы подписать сообщение M, p действует следующим образом.

(1) P выбирает произвольное число r и вычисляет $x = r^2$.

(2) P вычисляет f (M, x), назовем первые k бит с e_1 по e_k.

(3) P вычисляет $y = r\prod_{e_j=1} s_j$.

Подпись $S_p(M)$ состоит из кортежа (e_1,..., e_k , y).

Чтобы проверить подпись (e_1,..., e_k , y) процесса p для сообщения M, надо действовать следующим образом.

(1) Вычислить v_j и $z = y^2\prod_{e_j=1} v_j$

(2) Вычислить f (M, z) и проверить, что первые k бит - e_1 по e_k .

Если подпись истинна, значение z, вычисленное на первом шаге проверки равняется значению x, используемому в подписывании и следовательно первые k бит f (M, z) равны e_1 ... e_k .

Подделка и заключительное решение. Мы рассмотрим теперь стратегию подделывающего для получения подписи согласно вышеупомянутой схеме без знания s_j .

(1) Выбрать k произвольных бит e_1 ... e_k .

(2) Выбрать произвольное число y и вычислить $x = y^2\prod_{e_j=1} v_j$

(3) Вычислить f (M, x) и посмотреть, равняются ли первые k бит значениям e_1 ... e_k , выбранным ранее. Если это так, то (e_1,..., e_k , y) - подделанная подпись для сообщения M.

Поскольку вероятность равенства в шаге (3) может быть принята 2^{-k} , подделка становится успешной после ожидаемого числа 2^k испытаний.

При k = 72 и принятым временем 10^{-9} секунд для опробования одного выбора e_j , ожидаемое время подделки (с этой стратегией) - $2^{72}\cdot 10^{-9}$ секунд или 1,5 миллиона лет, что делает схему вполне безопасной. Однако, каждый процесс должен хранить k корней, и если k должен быть ограничен из-за ограничений пространства, ожидаемое время подделки 2^k может быть неудовлетворительно. Мы покажем сейчас как изменить схему, чтобы использовать k корней, и получать ожидаемое время подделки 2^{kt} для выбранного целого числа t. Идея состоит в том, чтобы использовать первые kt бит f-результата, чтобы определить t подмножеств от v_j , и заставить p показывать свое знание t этих произведений. Чтобы подписать сообщение M, p действует следующим образом.

(1) p выбирает произвольные r_1,..., r_t и вычисляет $x_i = r_i^2$.

(2) p вычисляет f (M, x_1, ..., x_t); назовем первые kt бит e_{ij} . ($1 \le i \le t$ и $1 \le j \le k$).

(3) p вычисляет $y_i = r_i\prod_{e_{ij}=1} s_j$ для $1 \le i \le t$. Подпись $S_p(M)$ состоит из (e_{11} , ..., e_{tk} , y_1, ..., y_t).

Чтобы проверить подпись (e_{11}, ..., e_{tk}, y_1, ..., y_t) процесса p для сообщения M, надо действуовать следующим образом.

(1) Вычислить v_j и $z_i = y_i^2 \prod_{e_{ij}=1} v_j$.

(2) Вычислить f (M, z_1, ..., z_t), и проверить, что первые kt бит - e_{11}, ..., e_{tk}.

Подделывающий, пытающийся произвести действительную подпись с той же самой стратегией, что приведена выше, теперь имеет вероятность успеха в третьем шаге 2^{-kt}, что означает ожидаемое число испытаний 2^{kt}. Фиат и Шамир показывают в своей работе, что если разложение n на множители не оказывается простым, то существенно лучшего алгоритма подделки не существует, следовательно схему можно сделать произвольно безопасной, выбирая k и t достаточно большими.

14.2.6 Резюме и Обсуждение

В этом и предыдущем разделе было показано, что в синхронных системах существуют детерминированные решения для проблемы Византийского вещания. Максимальная способность восстановления таких решений - t < N/3, если не используется проверка подлинности (Раздел 14.1), и неограничена, если она используется (этот раздел). Во всех решениях, представленных здесь, синхронность была смоделирована с помощью (довольно сильных) предположений модели импульса; отказоустойчивая реализация модели импульса обсуждается в Разделе 14.3.

Проблема расстрельной команды. В дополнение к принятию модели импульса, второе предположение, лежащее в основе всех решений, представленных до сих пор - то, что импульс, в котором вещание начинается, известен всем процессам (и пронумерован 1 для удобства). Если априори дело обстоит не так, возникает проблема старта алгоритма одновременно, после того, как один или более процессов (спонтанно) инициируют запрос просьбу о выполнении алгоритма вещания. Запрос может исходить от командующего (после вычисления результата, который должен быть объявлен всем процессам) или от помощников (понимающих, что им всем нужна информация, хранящаяся у командующего). Эта проблема изучается в литературе как проблема *расстрельной команды*. В этой проблеме, один или более процессов *инициируют* (запрос), но не обязательно в одном и том же импульсе, и процессы могут *стрелять*. Требования:

(1) **Обоснованность**. Никакой корректный процесс не стреляет, если никакой процесс не инициировал.

(2) **Одновременность**. Если любой корректный процесс стреляет, тогда все корректные процессы стреляют в том же самом импульсе.

(3) **Завершение**. Если корректный процесс инициирует, то все корректные процессы стреляют в течение конечного числа импульсов.

Действительно, имея решение для проблемы расстрельной команды, не нужно заранее соглашаться о первом импульсе вещания; процессы, запрашивающие вещание, инициируют алгоритм расстрельной команды, и вещание начинается в

импульсе следующем за стрельбой. Методы, используемые в решениях проблемы Византийского-вещания и проблемы расстрельной команды, могут быть объединены, чтобы получить протоколы, более эффективные по времени, которые решают проблему вещания непосредственно в отсутствии априорного соглашения о первом импульсе.

Временная сложность и рано останавливающиеся протоколы. В этой главе мы представили протоколы, использующие t + 1 или 2t + 3 импульсов, или раундов связи. Фишером и Линчем [FL82] показано, что t + 1 раундов связи - оптимальное число для t-устойчивых протоколов согласия, и результат был расширен, чтобы охватить протоколы с установлением подлинности, Долевом и Стронгом [DS83].

Тонкий момент в этих доказательствах - то, что в использованных сценариях процесс должен отказывать в каждом из импульсов с 1 по t, поэтому нижние границы в худшем случае - число фактических сбоев в течение выполнения. Так как в большинстве выполнений фактическое число сбоев намного ниже способности восстановления, изучалось существование протоколов, которые могут достигать соглашения ранее в тех выполнении, которые имеют маленькое число сбоев. Протоколы вещания с таким свойством называются *рано останавливающимися*. Долев, Райсчук и Стронг [DRS82] продемонстрировали нижнюю границу в f + 2 раунда для любого протокола в выполнении с f сбоями. Обсуждение нескольких рано останавливающихся протоколов вещания и согласия есть в [BGP92].

Рано останавливающиеся протоколы принимают решение в течение нескольких импульсов после того, как корректные процессы заключают, что был импульс без новых сбоев. Однако, нельзя гарантировать, что все корректные процессы достигают этого заключения в одном и том же импульсе. (Если, конечно, они не достигают его в импульсе t + 1; так как самое большее t процессов отказывают, среди первых t + 1 имеется один раунд, в котором никакого нового сбоя не происходит.) Как следствие, рано останавливающиеся протоколы не удовлетворяет одновременности. Коун и Дворк показали [CD91], что, чтобы достигнуть одновременности, в выполнении, где никаких сбоев не происходит, необходимо также t + 1 раунд, даже для рандомизированных протоколов и в (очень слабый) модели аварий. Это означает, что протоколам с установлением подлинности также нужно t + 1 импульсов для одновременного соглашения.

Поблемы решения и согласованная непротиворечивость. При использовании протокола вещания в качестве подпрограммы, фактически все проблемы решения для синхронных систем могут быть решены достижением *согласованной непротиворечивости*, то есть соглашения о множестве входов. В проблеме согласованной непротиворечивости, процессы принимают решение о *векторе* входов, с одним элементов для каждого процесса в системе. Формально, требования таковы:

(1) **Завершение**. Каждый корректный процесс p останавливается на векторе V_p с одним элементом для каждого процесса.

(2) **Соглашение**. Векторы решения корректных процессов равны.

(3) **Зависимость**. Если q корректен, то для корректного p, $V_p[q] = x_q$.

Согласованной непротиворечивости можно достичь множественными вещаниями: каждый процесс вещает свой вход, и процесс p помещает свое решение в вещании q в $V_p[q]$. Завершение, соглашение, и зависимость непосредственно наследуются от соответствующих свойств алгоритма вещания.

Так как каждый корректный процесс вычисляет один и тот же вектор (соглашение), большинство проблем решения легко решается с помощью детерминированной функции на векторе решения (что непосредственно гарантирует соглашение). Согласие, например, решается с помощью извлечения значения, имеющего большинство, из решающего вектора. Выбор решается извлечением самого маленького уникального идентификатора в векторе (остерегайтесь; избранный процесс может быть сбойным).

14.3 Синхронизация Часов

В предыдущих разделах было показано, что (когда рассматриваются детерминированные алгоритмы) синхронные системы имеют более высокую способность восстановления, чем асинхронные. Это было сделано для идеализированной модели синхронности, где процессы функционируют в импульсах. Более высокая способность восстановления модели импульса означает, что не возможно детерминированно устойчиво синхронизировать полностью асинхронные сети. В этом разделе будет показано, что устойчивая реализация модели импульса возможна в модели асинхронных сетей ограниченных задержек (ABD (asynchronous bounded-delay) сети - АСОЗ).

Модель АСОЗ характеризуется наличием локальных часов и верхней границей на задержку сообщений. В описании и анализе алгоритмов мы используем *кадр реального времени* (real-time frame), который является назначением времени наступления $t \in R$ каждому событию. Согласно релятивистской физике, нет стандартного или предпочтительного способа сделать это назначение; в дальнейшем будем предполагать, что физически значимое назначение было выбрано. Кадр реального времени не поддается наблюдению для процессов в системе, но процессы могут косвенно отслеживать время, используя свои *часы*, значения которых связаны с реальным временем. Часы процесса p обозначаются C_p и он может читать и записывать в них (запись в часы необходима для синхронизации). Значение часов непрерывно изменяется во времени, если часы не назначены; мы пишем $C_p(t) = T$, чтобы обозначить, что в момент реального времени t часы содержат T.

Заглавные буквы (C, T) используются для времени часов, а строчные буквы (c, t) - для реального времени. Часы могут использоваться для управления наступлением событий, как в выражении

when $C_p = T$ **then** send message

rоторое вызывает посылку сообщения во время $C_p^{-1}(T)$. Функция C_p^{-1} обозначается c_p.

Значение идеальных часов увеличивается на Δ за Δ единиц времени, то есть, оно удовлетворяет $C(T+\Delta) = C(T) + \Delta$. Синхронизированные идеальные часы никогда не нуждаются в корректировке, но, к сожалению, они всего лишь (полезная) математическая абстракция. Часы, используемые в распределенных системах, испытывают *отклонение*, ограниченное маленькой известной константой ρ (обычно порядка 10^{-5} или 10^{-6}). Отклонение часов С ρ-*ограничено*, если, для t_1 и t_2, таких, что между t_1 и t_2 не происходит присваивания С,

$$(t_2 - t_1)(1+\rho)^{-1} \le C(t_2) - C(t_1) \le (t_2 - t_1)(1+\rho) \qquad (14.1)$$

Различные часы в распределенных системах не показывают одно и то же время часов в любой заданный момент реального времени, то есть, $C_p(t) = C_q(t)$ не обязательно справедливо. Часы δ-*синхронизированы* в момент реального времени t, если $\left|C_p(t) - C_q(t)\right| \le \delta$, и δ-*синхронизированы* момент часового времени T, если $\left|C_p(T) - C_q(T)\right| \le \delta$. Мы считаем эти понятия эквивалентными; см. Упражнение 14.8. Цель алгоритмов синхронизации часов состоит в том, чтобы достичь и поддерживать глобальную δ-синхронизацию, то есть, δ-синхронизацию между каждой парой часов. Параметр δ - *точность* синхронизации.

Задержка сообщений ограничена снизу $\delta_{\min}$ и сверху $\delta_{\max}$, где $0 \le \delta_{\min} \le \delta_{\max}$; формально, если сообщение посылается в реальное время σ и получается в реальное время τ, то

$$\delta_{\min} \le \tau - \sigma \le \delta_{\max} \qquad (14.2)$$

Так как выбор кадра реального времени свободный, предположения (14.1) и (14.2) относятся к временному кадру так же, как и к часам и системе связи.

14.3.1 Чтение Удаленных Часов

В этом подразделе будет изучена степень точности, с которой процесс p может настраивать свои идеальные часы на идеальные часы надежного сервера s. У детерминированного протокола самая лучшая доступная точность - $\frac{1}{2}(\delta_{\max} - \delta_{\min})$, и эта точность может быть получена для простого протокола, который обменивается только одним сообщением. Вероятностные протоколы могут достигать произвольной точности, но сложность по сообщениям зависит от желательной точности и распределения времен доставки сообщений.

Теорема 14.12 *Существует детерминированный протокол для синхронизирования C_p с C_s с точностью $\frac{1}{2}(\delta_{\max} - \delta_{\min})$, который обменивается одним сообщением. Никакой детерминированный протокол не достигает более высокой точности.*

Доказательство. Мы сначала представим простой протокол и докажем, что он достигает точности, заявленной в теореме. Чтобы синхронизировать C_p, сервер

посылает одно сообщение, <**time**, C_s>. Когда p получает <**time**, T>, он корректирует часы на T + $\frac{1}{2}(\delta_{\max} + \delta_{\min})$.

Чтобы доказать заявленную точность, назовем реальные времена посылки и получения сообщения <**time**, T> σ и τ соответственно; теперь $T = C_s(\sigma)$. Так как часы идеальны, $C_s(\tau) = T + (\tau - \sigma)$. Во время τ, p корректирует часы, чтобы на них было $C_p(\tau) = T + \frac{1}{2}(\delta_{\max} + \delta_{\min})$, поэтому $C_s(\tau) - C_p(\tau) = (\tau - \sigma) - \frac{1}{2}(\delta_{\max} + \delta_{\min})$. Теперь $\delta_{\min} \le (\tau - \sigma) \le \delta_{\max}$ означает $\left|C_s(\tau) - C_p(\tau)\right| \le \frac{1}{2}(\delta_{\max} - \delta_{\min})$.

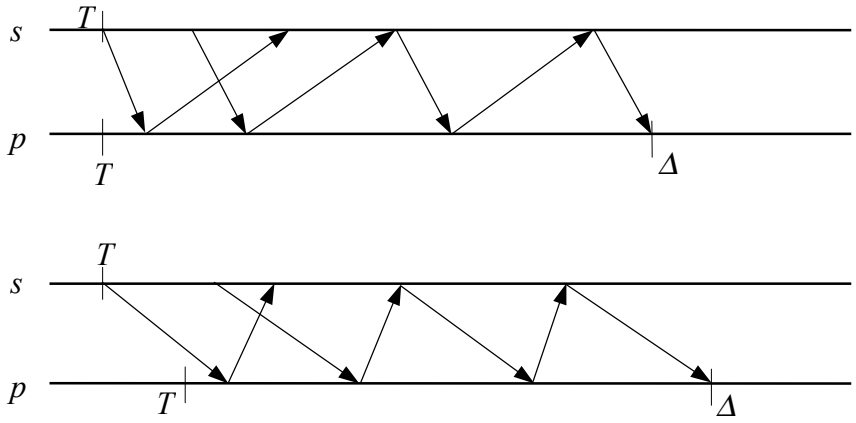


РИСУНОК 14.5 СЦЕНАРИИ ДЛЯ ДЕТЕРМИНИРОВАННОГО ПРОТОКОЛА.

Чтобы показать нижнюю границу для точности, пусть дан детерминированный протокол; в этом протоколе p и s обмениваются некоторыми сообщениями, после которых p корректирует свои часы. Рассматриваются два сценария для протокола, как изображено на Рисунке 14.5. В первом сценарии, часы равны до выполнения, все сообщения из s в p доставляются после $\delta_{\min}$, и все сообщения из p в s доставляются после $\delta_{\max}$. Если корректировка в этом сценарии - Δ_1, то часы p в точности на Δ_1 опережают C_s после синхронизации.

Во втором сценарии C_p до выполнения отстает от C_s на $\delta_{\max} - \delta_{\min}$, все сообщения из p в s доставляются после $\delta_{\min}$, и все сообщения из s в p доставляются после $\delta_{\max}$. Назвав корректировку в этом сценарии Δ_2, мы видим, что часы p после синхронизации отстают от C_s в точности на $\delta_{\max} - \delta_{\min} - \Delta_2$.

Однако, ни p ни s не наблюдают различия между сценариями, так как неопределенность в задержке сообщения скрывает различие; следовательно $\Delta_1 = \Delta_2$. Это означает, что точность самого худшего случая

$$\min_{\Delta} \max(|\Delta|, \left|\delta_{\max} - \delta_{\min} - \Delta\right|).$$

Этот минимум равняется (и случается при $\Delta = \frac{1}{2}(\delta_{\max} - \delta_{\min})$). □

Если два процесса p и q синхронизируют свои часы с сервером с этой точностью, достигается глобальная $\delta_{\max} - \delta_{\min}$-синхронизация, который достаточно для большинства прикладных программ.

Лучшая точность достижима у вероятностного протокола синхронизации, предложенного Кристианом [Cri89]. Для этого протокола принимается, что задержка сообщения - стохастическая переменная, распределенная согласно (не обязательно известной) функции $F:[\delta_{\min}, \delta_{\max}] \to [0,1]$. Вероятность прибытия любого сообщения *в течение x* - F(x), и задержки различных сообщений независимы. Произвольная точность достижима только если нижняя граница $\delta_{\min}$ плотна, то есть, для всех x> $\delta_{\min}$, F (x) > 0.

Протокол прост; p просит, чтобы s послал время, и s немедленно отвечает сообщением <**time**, T>. Процесс p измеряет время I между посылкой запроса и получения ответа; справедливо $I \geq 2\delta_{\min}$. Задержка сообщения ответа - по крайней мере $\delta_{\min}$ и самое большее $I - \delta_{\min}$, и следовательно отличается самое большее на $\frac{1}{2}(I - 2\delta_{\min})$ от $\frac{1}{2}I$. Таким образом, p может установить свои часы в $T + \frac{1}{2}I$, и достигает точности $\frac{1}{2}(I - 2\delta_{\min})$. Если желательная точность - ε, p посылает новый запрос если $\frac{1}{2}(I - 2\delta_{\min}) > \varepsilon$, в противном случае завершается.

Лемма 14.13 *Вероятностный протокол синхронизации часов достигает точности* ε *с ожидаемым числом сообщений самое большее* $F(\delta_{\min} + \varepsilon)^{-2}$.

Доказательство. Вероятность того, что запрос p прибывает в течение $\delta_{\min} + \varepsilon$ - $F(\delta_{\min} + \varepsilon)$ и такова же вероятность того, что ответ прибывает внутри в течение $\delta_{\min} + \varepsilon$. Следовательно, вероятность того, что p получает ответ в течение $2\delta_{\min} + 2\varepsilon$ - по крайней мере $F(\delta_{\min} + \varepsilon)^2$, что означает границу в $F(\delta_{\min} + \varepsilon)^{-2}$ на ожидаемое число испытаний до успешного обмена сообщениями. □

Временная сложность протокола уменьшается, если p посылает новый запрос когда никакого ответа не было получено $2\delta_{\min} + 2\varepsilon$ после посылки запроса. Ожидаемое время тогда не зависит от ожидаемой или максимальной задержки, а именно $2(\delta_{\min} + \varepsilon)F(\delta_{\min} + \varepsilon)^{-2}$, и протокол устойчив против потери сообщений. (Нужно использовать номера в порядке следования, чтобы отличить устаревшие ответы.)

14.3.2 Распределенная Синхронизация Часов

Этот раздел представляет t-Византийско-устойчивый (при t < N/3) распределенный алгоритм синхронизации часов Махани и Шнайдера [MS85]. Долев, Халперн и Стронг [DHS84] показали, что никакая синхронизация не возможна при $t \geq N/3$, если не используется установление подлинности.

Ядро алгоритма синхронизации - протокол, который достигает *неточного соглашения* о средних значениях часов. Процессы корректируют свои часы и достигают высокой степени синхронизации. Из-за отклонения через некоторое

время точность ухудшается, что влечет за собой новый раунд синхронизации после некоторого интервала. Предположим, что в реальное время t_0 часы δ_0-синхронизированы; тогда до времени $t_0 + R$ часы $(\delta_0 + 2\rho R)$-синхронизированы. Таким образом, если желательная точность - δ, и раунд синхронизации достигает точности δ_0, раунды повторяются каждые $(\delta - \delta_0) / 2\rho$ единиц времени. Так как время, допустим S, для выполнения раунда синхронизации обычно очень мало по сравнению с R, то оправдано упрощающее предположение о том, что в течение синхронизации отклонением можно пренебречь, то есть, часы являются идеальными.

Неточное соглашение: алгоритм с быстрой сходимостью. В проблеме неточного соглашения, используемой Махани и Шнайдером [MS85] для синхронизации часов, процесс p имеет действительное входное значение x_p, где для корректных p и q $|x_p - x_q| \le \delta$. Выход процесса p - действительное значение y_p, и точность выхода определяется как $\max_{p,q} |y_p - y_q|$; цель алгоритма состоит в том, чтобы достичь очень малого значения точности.

---

var x_p, y_p, $esti_p$: real; (*Вход, выход, оценщик V *)
V_p, A_p : multiset of real;

begin (*фаза сбора входов*)
$V_p := \varnothing$;
forall $q \in$ P **do** send <**ask**> to q
wait $2\delta_{\max}$; (*Обработать сообщения <**ask**> и <**val**, x>*)
while $\#V_p < N$ **do** insert (V_p, ∞);
(*Теперь вычислить приемлемые значения*)
$A_p := \{x \in V_p : \#\{y \in V_p : |y - x| \le \delta\} \ge N - t\}$;
$esti_p := estimator(A_p)$;
while $\#A_p < N$ **do** insert(A_p, $esti_p$);
$y_p := (\sum A_p) / N$
end

После получения <**ask**> от q:
send<**val**, x_p> to q

После получения <**val**, x> от q:
if такое сообщение не было получено от q прежде
then insert(V_p, x)

АЛГОРИТМ 14.6 АЛГОРИТМ С БЫСТРОЙ СХОДИМОСТЬЮ.

Алгоритм с быстрой сходимостью, предложенный Махани и Шнайдером дан как Алгоритм 14.6. Для конечного множества $A \subset R$, определим две функции *intvl*(A)=[min(A), max(A)] и *width*(A) = max(A) - min(A). Алгоритм имеет фазу сбора входов и фазу вычисления. В первой фазе процесс p просит каждый дру-

гой процесс послать свой вход (“выкрикивая” сообщение <**ask**>) и ждет $2\delta_{\max}$ единиц времени. После этого времени, p получил все входы от корректных процессов, также как и ответы от подмножества сбойных процессов. Эти ответы заполняются (бессмысленными) значениями ∞ для процессов, которые не ответили.

Затем процесс применяет к полученным значениям фильтр, который гарантированно пропускает все значения корректных процессов и только те сбойные значения, которые достаточно близки к правильным значениям. Поскольку корректные значения отличаются только на δ и имеется по крайней мере N-t корректных значений, каждое корректное значение имеет по крайней мере N-t значения, которые отличаются самое большее на δ от него; A_p сохраняет полученные значения с этим свойством.

Затем вычисляется выход с помощью усреднения значений - все отклоненные значения заменяются оценкой, вычисленной применением детерминированной функции *estimator* к оставшимся значениям. Эта функция удовлетворяет $estimator(A) \in intvl(A)$, но в остальном произвольна; она может быть минимумом, максимумом, средним, или $\frac{1}{2}(\max(A) + \min(A))$.

Теорема 14.14 *Алгоритм с быстрой сходимостью достигает точности* $2t\delta / N$.

Доказательство. Пусть v_{pr} - значение, включаемое в V_p для процесса r, когда p превышает лимит времени (то есть, v_{pr} - или x_r или ∞), и a_{pr} - значение в A_p для процесса r, когда p вычисляет y_p (То есть a_{pr} - или v_{pr} или $esti_p$). Точность будет ограничена разделением суммирования в вычислении решения на суммирование над корректными процессами (C) и некорректными процессами (B). Для корректных p и q, разность $\left|a_{pr} - a_{qr}\right|$ ограничена 0, если $r \in C$, и 2δ если $r \in B$.

Первая граница следует из того, что, так как если p и r - корректные процессы, то $a_{pr} = x_r$. Действительно, так как r отвечает на сообщение p <**ask**> быстро, $v_{pr} = x_r$. Точно так же $v_{pr'} = x_{r'}$ для всех корректных r', и предположение о входе означает, что значение r переживает фильтрацию процессом p, следовательно Учреждение, несущее основную ответственность $a_{pr} = v_{pr}$.

Вторая граница справедлива, так как для корректных p и q, $width(A_p \cup A_q) \le 2\delta$, когда p и q вычисляют свои решения. Так как добавленные оценки находятся между принятыми значениями, достаточно рассмотреть максимальное различие между значениями a_p и a_q, которые прошли фильтры p и q соответственно. Имеется по крайней мере N-t процессов r, для которых $\left|v_{pr} - a_p\right| \le \delta$, и по крайней мере N-t процессов r, для которых $\left|v_{qr} - a_p\right| \le \delta$. Это означает, что имеется *корректный* r такой, что $\left|v_{pr} - a_p\right| \le \delta$ и $\left|v_{qr} - a_p\right| \le \delta$; но так как r корректен, $v_{pr} = v_{qr}$, следовательно $\left|a_p - a_q\right| \le 2\delta$.

Отсюда следует, что для корректных p и q,

$$\begin{aligned}
\left|y_p - y_q\right| &= \left|(\sum A_p)/N - (\sum A_q)/N\right| \\
&= \frac{1}{N}\cdot\left|\left(\sum_{r\in C} a_{pr} + \sum_{r\in B} a_{pr}\right) - \left(\sum_{r\in C} a_{qr} + \sum_{r\in B} a_{qr}\right)\right| \\
&= \frac{1}{N}\cdot\left|\left(\sum_{r\in C} a_{pr} - \sum_{r\in C} a_{qr}\right) + \left(\sum_{r\in B} a_{pr} - \sum_{r\in B} a_{qr}\right)\right| \\
&\le \frac{1}{N}\cdot\left[\left(\sum_{r\in C}\left|a_{pr} - a_{qr}\right|\right) + \left(\sum_{r\in B}\left|a_{pr} - a_{qr}\right|\right)\right] \\
&\le \frac{1}{N}\cdot\left[(0) + \left(\sum_{r\in B} 2\delta\right)\right] \le 2t\delta / N.
\end{aligned}$$

□

Произвольная точность может быть достигнута повторением алгоритма; после i итераций, точность становится $(\frac{2}{3})^i\delta$. Точность даже лучше, если меньшая доля (чем треть) процессов сбойная; в выводе точности t можно понимать как фактическое число сбойных процессов. Выходную точность алгоритма (самую худшую) нельзя улучшить подходящим выбором функции *estimator*; действительно, Византийский процесс r может навязать p любое значение $a_{pr} \in intvl(A_p)$, просто посылая p это значение. Функция может быть выбрана соответственно, чтобы достигнуть хорошей средней точности, когда известно что-нибудь о наиболее вероятном поведении сбойных процессов.

Синхронизация Часов. Чтобы синхронизировать часы, используется алгоритм с быстрой сходимостью, чтобы достигнуть неточного соглашения по новому значению часов. Предполагается, что часы первоначально δ – синхронизированы. Алгоритм должен быть адаптирован, так как

(1) Задержка сообщения точно не известна, так что процесс не может знать точное значение другого процесса; и

(2) При выполнении алгоритма идет время, так что часы не имеют постоянных значения, а увеличиваются со временем.

Чтобы компенсировать неизвестную задержку, процесс добавляет $\frac{1}{2}(\delta_{\max} + \delta_{\min})$ к получаемым значениям часов (как в детерминированном протоколе Теоремы 14.12), вводя дополнительное слагаемое $\delta_{\max} - \delta_{\min}$ в выходную точность. Чтобы представлять полученное значение как значение часов, а не как константу, p хранит разность полученного значения часов (плюс $\frac{1}{2}(\delta_{\max} + \delta_{\min})$) и собственного как Δ_{pr}. Во время t, приближение часов r процессом p - $C_p(t) + \Delta_{pr}$. Измененный алгоритм дан как Алгоритм 14.7.

---

var C_p, Δ_p, $esti_p$: real; (*Вход, адаптация, оценщик V *)

D_p, A_p : multiset of real;

begin (*фаза сбора входов*)

$D_p := \varnothing$;

forall $q \in$ P **do** send <**ask**> to q

wait $2\delta_{\max}$; (*Обработать сообщения <**ask**> и <**val**, x>*)

while $\#D_p < N$ **do** insert (D_p, ∞);

(*Теперь вычислить приемлемые значения*)

$A_p := \{x \in D_p : \#\{y \in D_p : |y - x| \le \delta + (\delta_{\max} - \delta_{\min})\} \ge N - t\}$;

$esti_p := estimator(A_p)$;

while $\# A_p < N$ **do** insert(A_p, $esti_p$);

$\Delta_p := (\sum A_p) / N$;

$C_p := C_p + \Delta_p$

end

После получения <**ask**> от q:

send<**val**, C_p > to q

После получения <**val**, C> от q:

if такое сообщение не было получено от q прежде

then $insert(D_p, (C + \frac{1}{2}(\delta_{\max} - \delta_{\min})) - C_p)$

АЛГОРИТМ 14.7 БЫСТРАЯ СХОДИМОСТЬ ЧАСОВ.

Заметьте, что в Алгоритме 14.7 фильтр имеет более широкую грань, а именно $\delta + (\delta_{\max} - \delta_{\min})$, чем в Алгоритме 14.6, где грань δ. Более широкая грань компенсирует неизвестную задержку сообщения, и порог возникает из следующего утверждения. Пусть d_{pr} обозначает значение, которое p вставил в D_p для процесса r после первой фазы p (сравните со значением v_{pr} в предыдущем алгоритме).

Утверждение 14.15 *Для корректных p, q, и r, после лимита времени p выполняется* $|d_{pr} - d_{qr}| \le \delta + (\delta_{\max} - \delta_{\min})$.

Доказательство. Передача сообщения <**val**, C> от q до p осуществляет детерминированный алгоритм чтения часов из Теоремы 14.12. Когда p получает это сообщение, $\left|C_q - \left[C + \frac{1}{2}(\delta_{\max} + \delta_{\min})\right]\right|$ ограничено $\frac{1}{2}(\delta_{\max} - \delta_{\min})$, поэтому d_{pq} отличается самое большее на $\frac{1}{2}(\delta_{\max} + \delta_{\min})$ от $C_q - C_p)$. Точно так же d_{pr} отличается