Программирование распределенных систем

Роман Елизаров, 2016

elizarov@devexperts.com

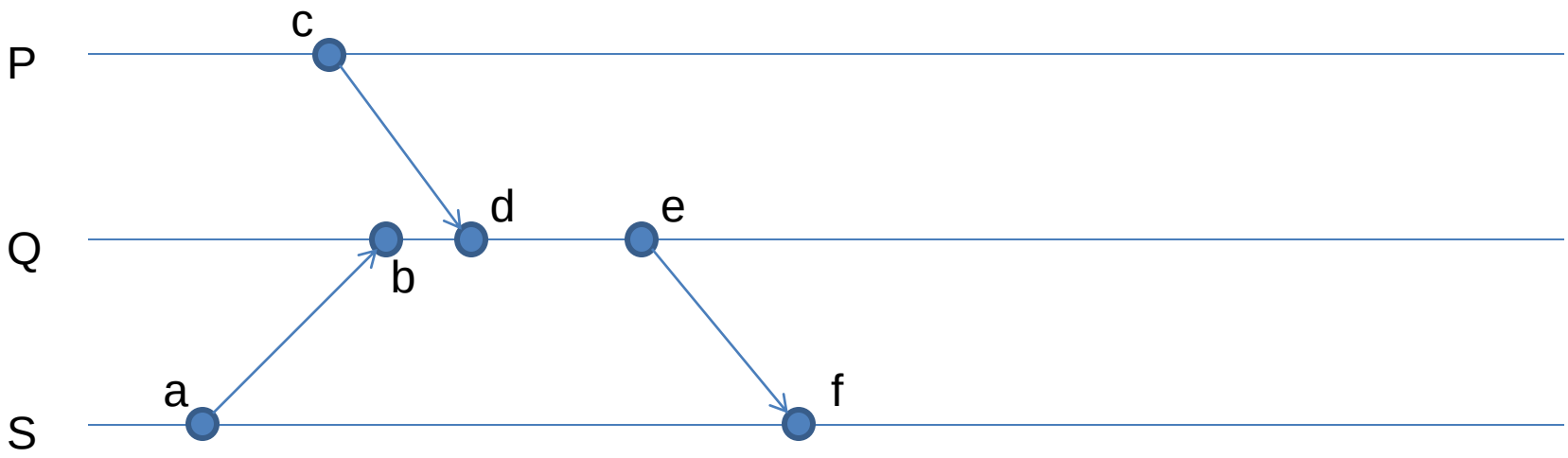
Сравнение: Concurrent Vs Distributed

Параллельная система	Распределенная система
SMP/NUMA система с общей памятью	Кластер из машин в сети
Вертикальная масштабируемость	Горизонтальная масштабируемость
Чтение/запись общей памяти	Посылка/получение сообщений
Централизованная	Возможно географически распределена
Обычно однородная	Часто гетерогенная
Взаимодействие всех со всеми	Возможно неполная *топология* связей
Определено общее время	Нет понятия общего времени
Определено состояние системы	Нет понятия общего состояния системы
Отказ всей системы в целом	Частичный отказ системы
Меньше надежность	Больше надежность
Проще программировать	Сложней программировать
Больше стоимость оборудования	Меньше стоимость оборудования

Модель

- Процессы $P, Q, R, \ldots \in \mathbf{P}$
- События $a,b,c,d,e,f,g,\ldots \in \mathbf{E}$, в процессах $proc(e) \in \mathbf{P}$
- Сообщения $m \in \mathbf{M}$, события посылки/приема $snd(m), rcv(m) \in \mathbf{E}$
- DEF: Отношение произошло-до между событиями ($\rightarrow$)
  - Такой минимальный *строгий частичный порядок* (транзитивное, антирефлексивное, антисимметричное отношение), что:
    - Если e и f произошли в одном процессе и e шло перед f (обозначаем как e<f), то $e \rightarrow f$
    - Если m сообщение, то $snd(m) \rightarrow rcv(m)$
  - То есть, это транзитивное замыкание порядка событий на процессе и посылки/приема сообщений

Графическая нотация



- Здесь стрелки задают посылки сообщений
- a и c не связаны отношением произошло-до
  - Такие события называются *параллельными*
- c произошло до f (из транзитивности)

Логические часы

- Для каждого события e определим число C(e) так, что:

$$\forall e, f \in \mathbf{E} : e \to f \Rightarrow C(e) < C(f)$$

- DEF: Такую функцию C будем называть логическими часами
- В обратную сторону отношение не верно и не может быть верно, ибо отношение порядка на числах задает полный порядок, а отношение произошло-до на событиях - частичный

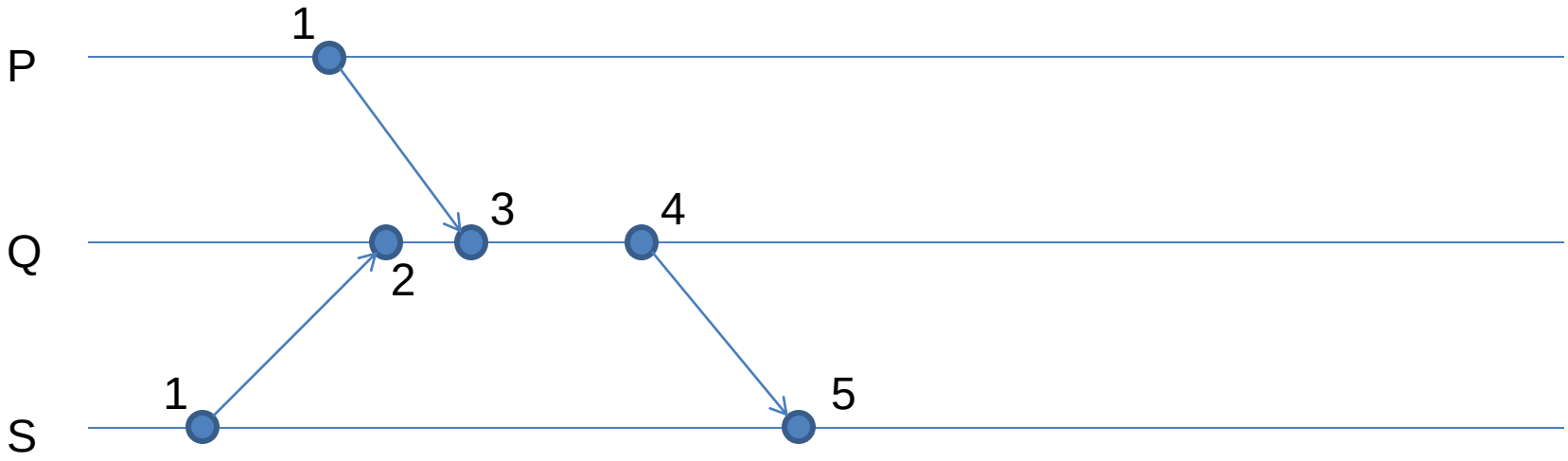
Логические часы Лампорта

- Время это целое число C в каждом процессе
- Алгоритм
  - Перед каждой посылкой сообщения процесс увеличивает часы на единицу

    C:= C+ 1
  - При посылке сообщение процесс посылает свое время C вместе с сообщением
  - При приеме сообщения делаем

    C := max(received_C, C) + 1

Логические часы Лампорта (2)



- Здесь указано время события (после увеличения переменной время и после операции max при приеме)
  - Время события не уникально!
  - Выполняется определение логических часов!

Векторные часы

- Для каждого события e определим вектор VC(e) так, что:

$$\forall e, f \in \mathbf{E} : e \to f \Leftrightarrow VC(e) < VC(f)$$

- Сравнение векторов происходит покомпонентно
- DEF: Такую функцию VC будем называть векторными часами

Алгоритм векторного времени

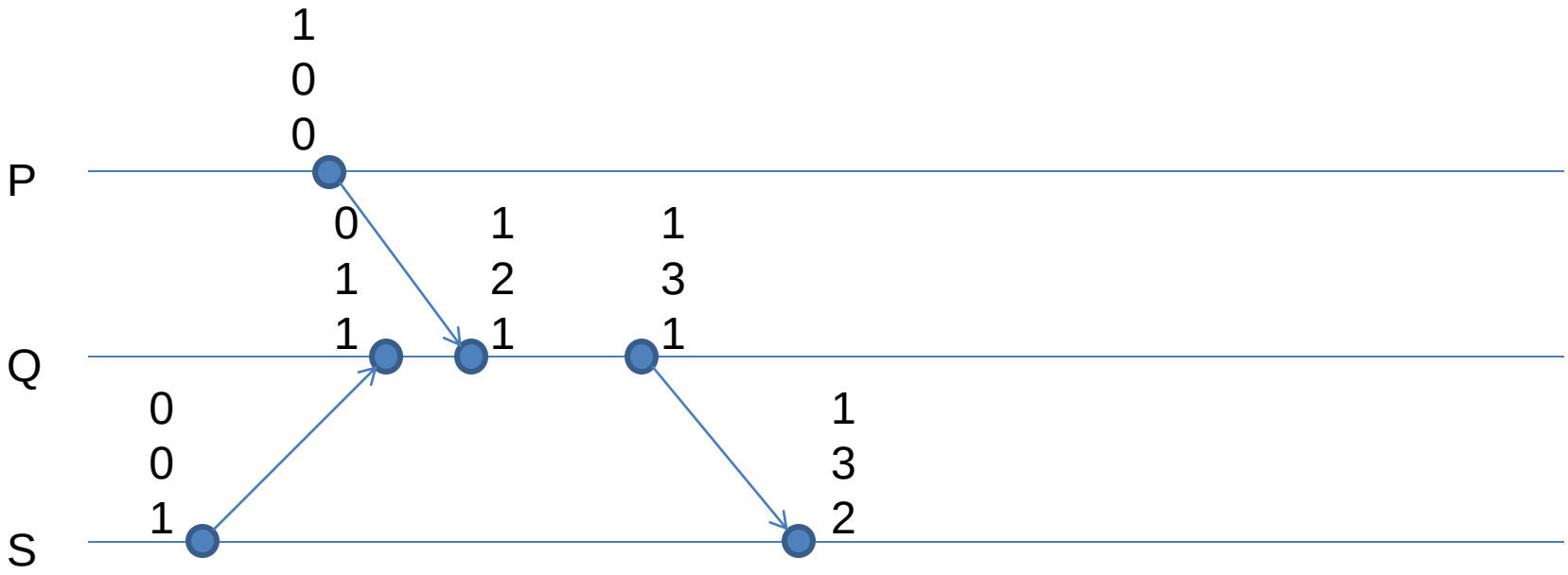
- Время это целое вектор VC в каждом процессе
  - Размер вектора это количество процессов
- Алгоритм
  - Перед каждой посылкой или приемом сообщения процесс увеличивает свой компонент в векторе времени на единицу

    $VC_i := VC_i + 1$
  - При посылке сообщение процесс посылает свой вектор времени вместе с сообщением
  - При приеме сообщение делаем покомпонентно

    VC:= max(received_VC, VC)
- ВАЖНО $\forall e, f \in E : proc(e) = P_i, proc(f) = P_j : e \to f \Leftrightarrow \begin{pmatrix} VC(e)_i \\ VC(e)_j \end{pmatrix} < \begin{pmatrix} VC(f)_i \\ VC(f)_j \end{pmatrix}$

DEVEXPERTS

Алгоритм векторного времени (2)



- Здесь указано время события (после его обработки)
  - Векторное время уникально для каждого события
  - Полностью передает отношение произошло-до

Другие часы

- Часы с прямой зависимостью (direct dependency)

$$\forall e, f \in \mathbf{E} : e \rightarrow_d f \Leftrightarrow VC_d(e) < VC_d(f)$$

где

$$e \rightarrow_d f \overset{def}{\Leftrightarrow} e < f \vee \exists m \in \mathbf{M} : e \leq snd(m) \,\&\, rcv(m) \leq f$$

  - Алгоритм это комбинация Лампорта и Векторного (храним вектор, посылаем одно число)
- Матричные часы (храним матрицу, посылаем матрицу)

Взаимное исключение в распределенных системах

- Критическая секция CS_i состоит из двух событий:
  - Enter(CS_i) вход в критическую секцию
  - Exit(CS_i) выход из критической секции
  - Здесь i это порядковый номер захода в критическую секцию
- Основное требование: ***взаимное исключение***
  - Два процесса не должны быть в критической сессии *одновременно*, то есть

$$Exit(CS_i) \rightarrow Enter(CS_{i+1})$$

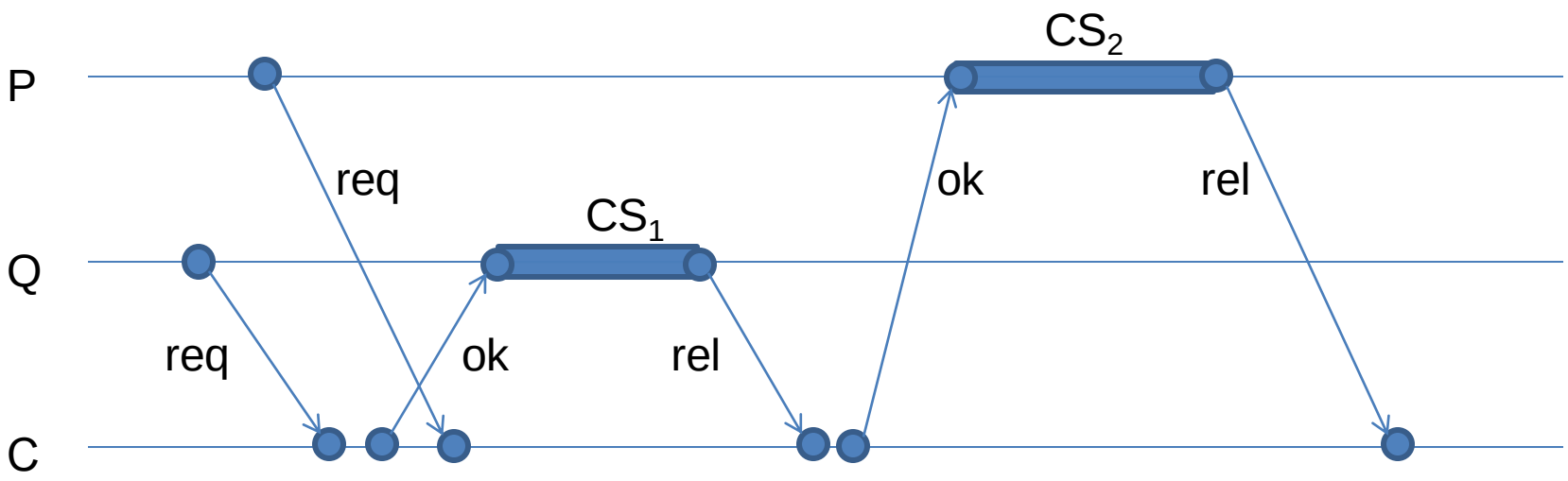
Взаимное исключение в распределенных системах (2)

- Так же как в системах с общей памятью нужны дополнительные требования ***прогресса***.
  - Минимальное требование прогресса заключается в том, что каждое желание процесса попасть в критическую секцию будет рано или поздно удовлетворено
  - Так же может быть гарантирован тот или иной уровень честности удовлетворения желания процессов о входе в критическую секцию

Централизованный алгоритм

- Выделенный координатор
- Три типа сообщений:
  - req[uest] (от запрашивающего процесса координатору)
  - ok (от координатора запрашивающему)
  - rel[ease] (после выхода из критической секции)
- Требует 3 сообщение на критическую секцию независимо от количества участвующих процессов
- Но не масштабируется из-за необходимости иметь выделенного координатора

Централизованный алгоритм (2)



Другие алгоритмы взаимного исключения

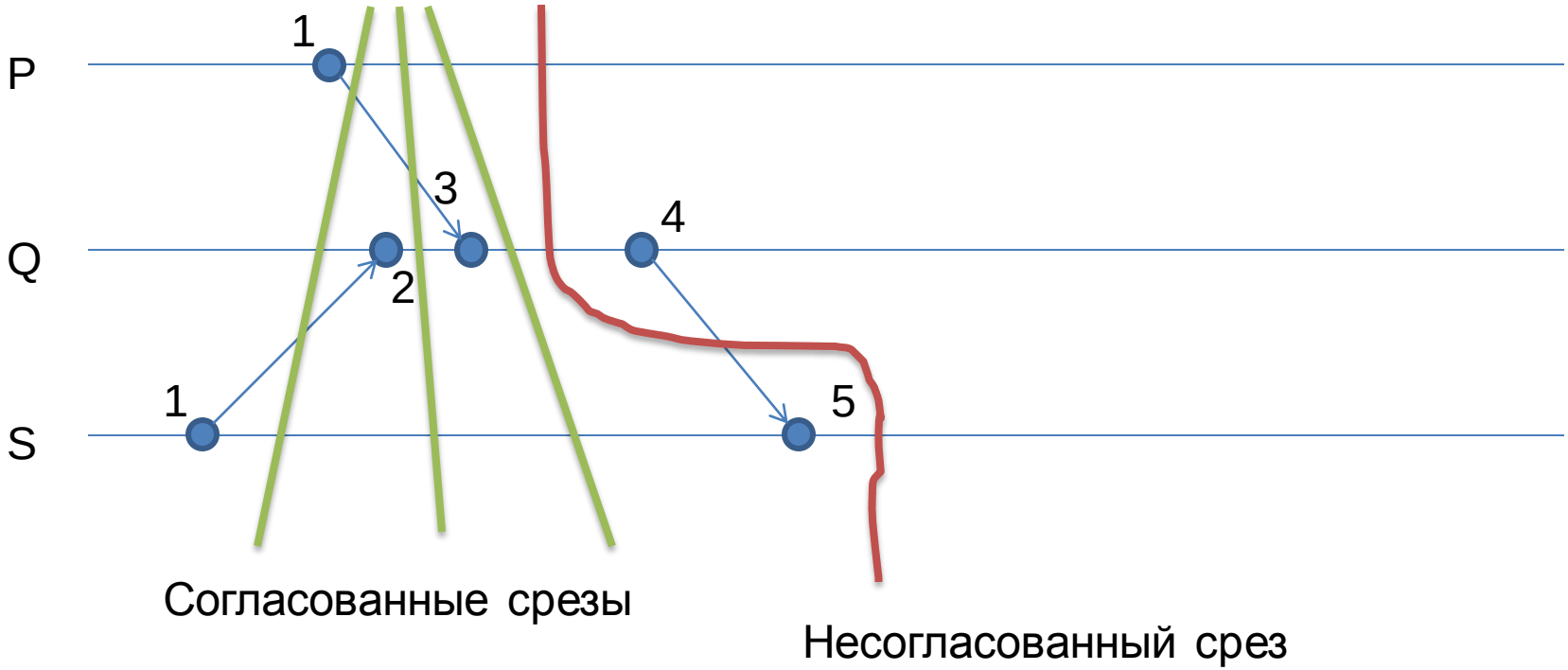
- Алгоритм Лампорта (3N-3 сообщения)
  - Работает только если между процессами сообщения идут FIFO
- Алгоритм Рикарда и Агравалы (2N-2 сообщения)
- Алгоритм обедающих философов (от 0 до 2N-2 сообщения)
- Алгоритм на основе токена
- Алгоритмы на основе кворума
  - DEF: Кворум $Q \subset 2^{\mathbf{P}} : \forall A, B \in Q : A \cap B \neq \emptyset$

    - Централизованный алгоритм как частный случай кворума
    - Простое большинство и взвешенное большинство
    - Рушащиеся стены

Согласованное глобальное состояние (срез)

- Если у распределенной системы нет «глобального состояния», то как запомнить её состояние на диске, чтобы можно было продолжить работу после восстановления с диска?
- DEF: Любое $G \subset \mathbf{E}$ называется срезом тогда и только тогда, когда
$$\forall f \in G, e \in \mathbf{E} : e < f \Rightarrow e \in G$$
- DEF: G является согласованным срезом тогда и только тогда, когда:
$$\forall f \in G, e \in \mathbf{E} : e \to f \Rightarrow e \in G$$
эквивалентное определение:
$$\nexists f \in G, e \in \mathbf{E} \setminus G : e \to f$$

Согласованное глобальное состояние



Алгоритм Чанди-Лампорта для согласованного запоминания глобального состояния системы

- Процесс инициатор:
  - Запоминает свое состояние
  - Посылает токен всем соседям
- При получении токена *первый раз* процесс:
  - Запоминает свое состояние
  - Посылает токен всем соседям
- Запомненные состояния образуют согласованный срез
  - Если сообщение между процессами идут FIFO.
- Чтобы восстановить работу системы из запомненного состояния (checkpoint) надо еще запоминать сообщения в пути

Запоминание сообщений

- Нужно запомнить все сообщения m такие что:

$$snd(m) \in G \,\&\, rcv(m) \in \mathbf{E} \setminus G$$

- Варианты реализации
  - Запоминание на стороне отправителя
  - Запоминание на стороне получателя

Глобальные свойства

- Стабильные предикаты
  - Берем согласованные срез; если предикат верен, значит будет верен и в дальнейшем (но построение согласованного среза «дорого» - $O(N^2)$ сообщений).
  - Пример: потеря токена, взаимная блокировка и т.п.
- Нестабильные предикаты
  - DEF: Локальный предикат – это предикат по состоянию одного процесса.
  - Если предикат это дизъюнкция локальных предикатов, то всё просто (в одном процессе «истина», то глобальная истинна).
  - А если конъюнкция? Как определить истинность нестабильного конъюнктивного предиката, если есть разные срезы?

Слабый конъюнктивный предикат

- Если предикат P имеет вид конъюнкции *локальных* предикатов над состоянием каждого процесса:

$$P = L_1 \,\&\, L_2 \,\&\, \ldots L_n$$

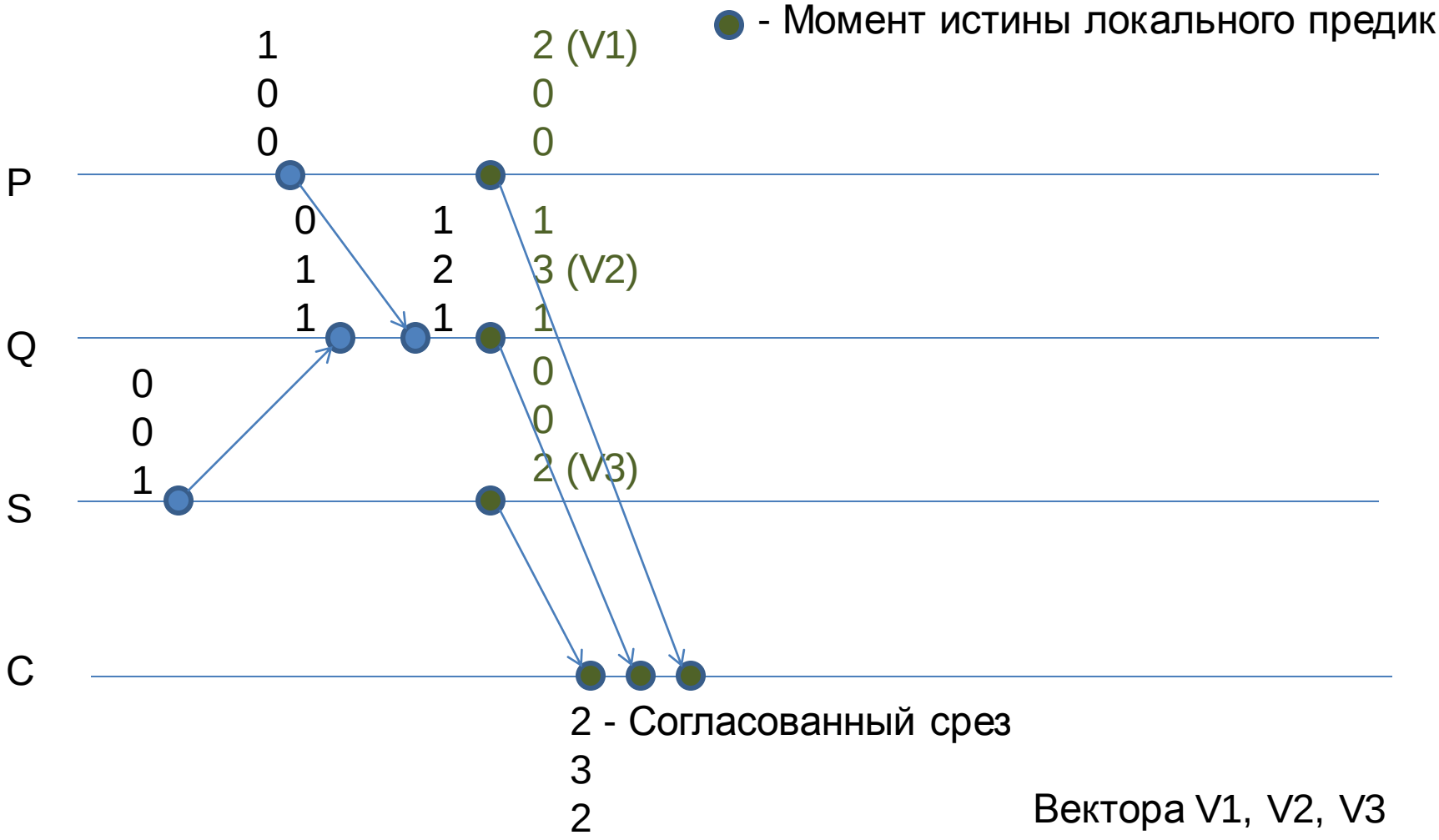
  - Пример предиката: «в системе нет координатора» (локальное условие «я не координатор»)
- DEF: Слабый конъюнктивный предикат истинен, если он истинен *на хотя бы одном согласованном срезе*.
- Сложные предикаты, являющиеся логической комбинаций локальных предикатов, всегда можно представить в нормальной дизъюнктивной форме и рассматривать как дизъюнкция слабых конъюнктивных предикатов

Слабый конъюнктивный предикат: централизованный алгоритм

- Каждый работающий процесс отслеживает свое векторное время VC.
- При наступлении истинности локального предиката L [увеличиваем свою компоненту вектора времени] и посылаем сообщение координатору C, [указывая векторное время, когда это произошло].
- В этом случае, любой срез можно однозначно задать вектором.
  - Координатор поддерживает в памяти *срез-кандидат* и очередь необработанных сообщений от каждого процесса.
  - Срез кандидат согласован тогда и только когда, когда все соответствующие вектора попарно несравнимы

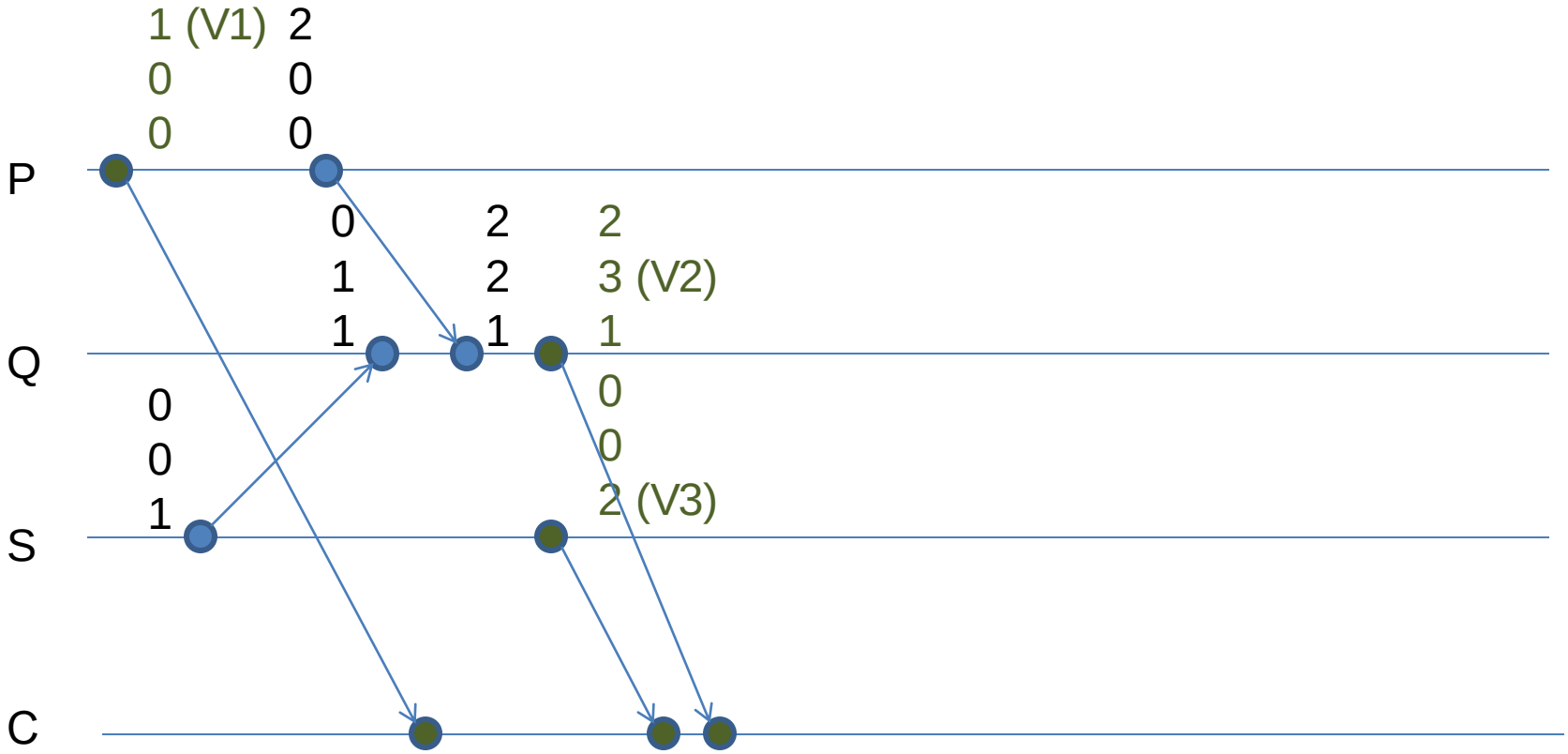
DEVEXPERTS

Согласованный срез-кандидат



Вектора V1, V2, V3
попарно несравнимы

Несогласованный срез-кандидат



1 – несогласованный срез-кандидат

3

2

Вектор V1 < V2

Слабый конъюнктивный предикат: централизованный алгоритм (2)

- Координатор хранит вектор среза-кандидата и флажок для каждой его компоненты:
  - Красный – этот элемент не может быть частью согласованного среза; Зеленый – наоборот
- Начальное состояние: всё по нулям, красное
- Обрабатываем приходящие сообщения только от красных процессов (сообщения от зеленых ставим в очередь)
  - Сравниваем пришедший вектор попарно с другими процессами (сравниваются только две соответствующие компоненты!),
    - Если новый вектор больше (нарушилась попарная несравнимость!), то делаем меньший процесс красным
    - После обработки сообщения от процесса его делаем зеленым
- Всё зеленое, значит нашли согласованный срез!

Слабый конъюнктивный предикат: распределенный алгоритм

- Каждый процесс имеет своего собственного координатора
- Процессы шлют сообщения (как раньше) своим координаторам, координаторы общаются между собой
  - Координаторы пересылают друг другу срезы-кандидаты и флажки (зеленый/красный)
- Красные координаторы обрабатывают сообщения от своих процессов (как раньше)
  - После обработки сообщения становятся зелеными
  - Если в процессе обработки они пометили красным другой процесс, то шлют сообщение его координатору

Специальные случаи стабильных предикатов: более эффективные алгоритмы

- Останов системы, например, при поиске кратчайшего пути
  - Обобщим для т.н. диффундирующих вычислений
- Взаимная блокировка при распределении ресурсов
  - Обобщим для т.н. локально-стабильных предикатов

Упорядочивание сообщений

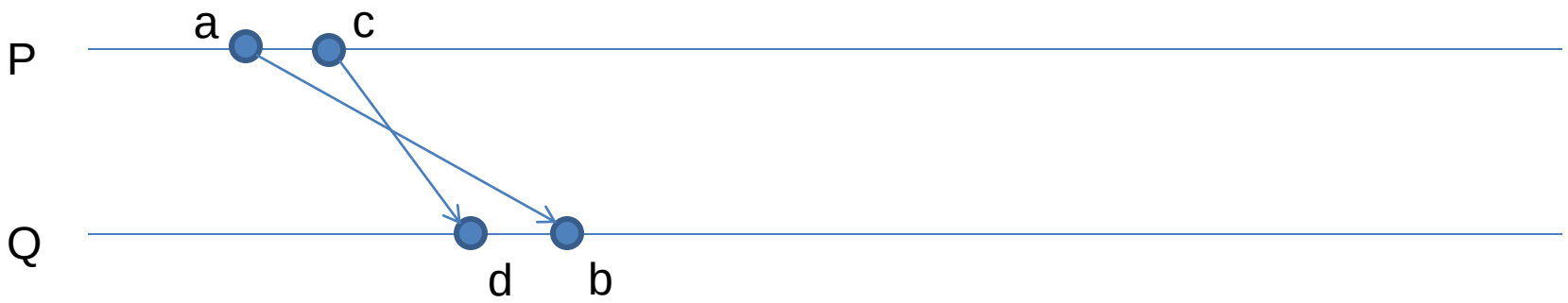
- Для обычных сообщений между парой процессов
  - Асинхронная передача (нет порядка)
  - FIFO
  - Причинно-согласованный порядок
  - Синхронный порядок
- Для multicast и broadcast сообщений
  - Общий порядок

FIFO порядок (First In First Out order)

- Не бывает пар сообщений m и n упорядоченных неверно!

$$\nexists m, n \in M : snd(m) < snd(n) \vee rcv(n) < rcv(m)$$

- Здесь “<“ это отношение порядка в одном процессе
- Пример нарушения FIFO с m=(a,b), n=(c,d):



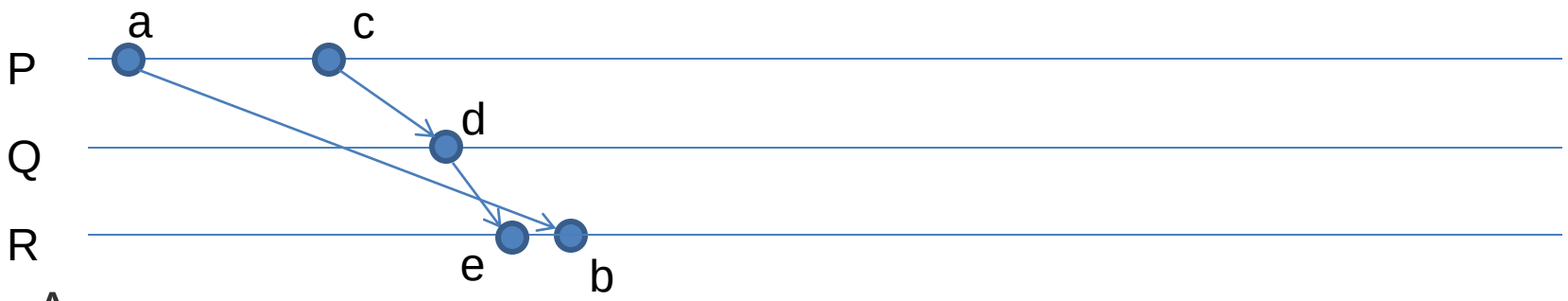
- Алгоритм FIFO основан на нумерации сообщений

Причинно-согласованный порядок (casually consistent order)

- Не бывает пар сообщений m и n упорядоченных неверно!

$$\nexists m, n \in M : snd(m) \rightarrow snd(n) \vee rcv(n) \rightarrow rcv(m)$$

- Здесь используется отношение «произошло до»
- Пример нарушения (FIFO не нарушен) с m=(a,b), n=(d,e):



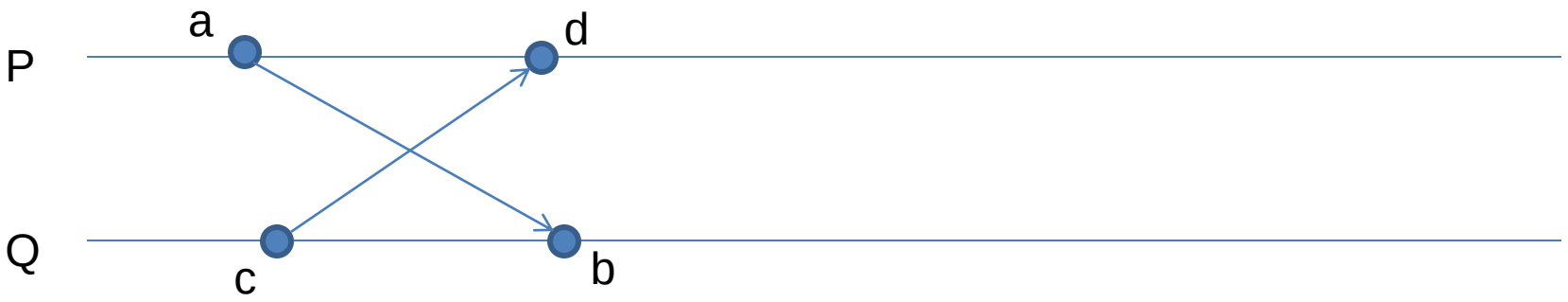
- Алгоритм для причинно-согласованного порядка основан на матричных часах

Синхронный порядок (synchronous order)

- Всем сообщениям можно сопоставить время $T(m)$ так что время событий $T\big(rcv(m)\big) = T\big(snd(m)\big) = T(m)$ и

$$\forall e, f \in E : e \longrightarrow f \Longrightarrow T(e) < T(f)$$

- Пример нарушения синхронного порядка (причинно-согласованность не нарушена)



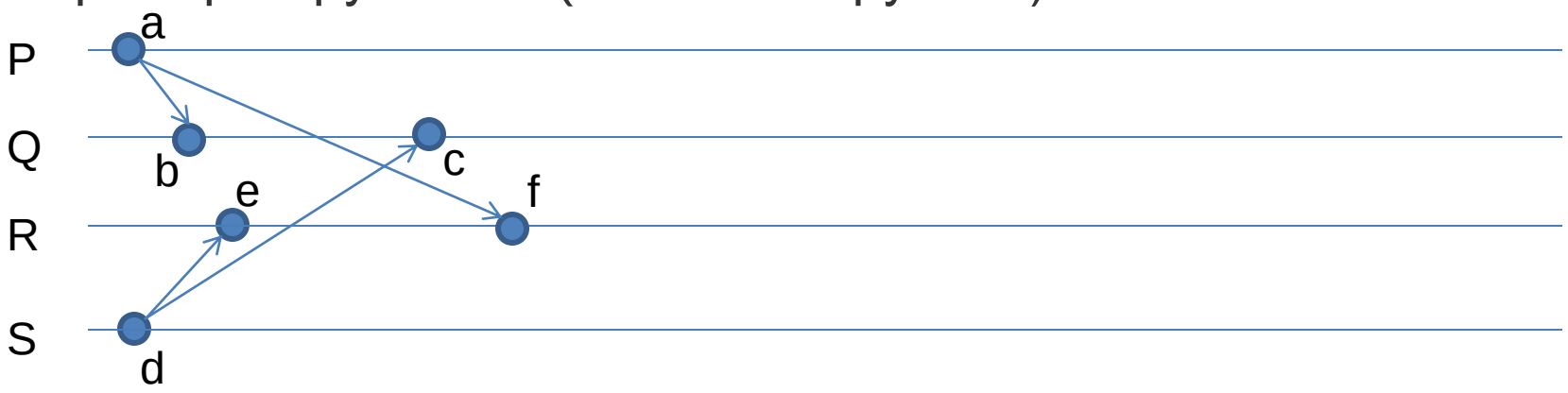
- Алгоритм основан на иерархии процессов

Общий порядок (total order)

- Только для случаем когда одно сообщение идет многим получателям (rcv с индексом процесса)

$$\nexists m,n \in M; p,q \in P : rcv_p(m) < rcv_p(n) \vee rcv_q(n) < rcv_q(m)$$

- Тривиально выполняется для всех unicast сообщений
- Пример нарушения (FIFO не нарушен!)



Алгоритмы общего порядка

- Алгоритм Лампорта
- Алгоритм Скина

DEVEXPERTS

Иерархия ошибок в распределенных системах

- Отказ одного или нескольких узлов (crash)
- Отказ одного или нескольких каналов (link failure)
- Ненадежная доставка сообщений (omission)
- Византийская ошибка (byzantine failure)

Синхронные и асинхронные системы

- **Синхронные системы**
  - Время передачи сообщений ограничено сверху
  - Можно разбить выполнение алгоритма на фазы
- **Асинхронные системы**
  - Время передачи сообщений не ограничено сверху
  - Но конечно, если система работает без ошибок

Консенсус в распределенной системе

- **Согласие** (agreement)
  - Все (не сбойные) процессы должны завершиться с одним и тем же решением
- **Нетривиальность** (non-triviality)
  - Должны быть варианты исполнения приводящие к разным решениям
- **Завершение** (termination)
  - Протокол должен завершиться за конечное время

Невозможность консенсуса в асинхронной системе с отказом узла (FLP)

- Результат Фишера-Линча-Патерсона (FLP), 1985 год
- Важные предпосылки
  - Система асинхронна (нет предела времени доставки сообщения)
  - (Один) узел может отказать
  - Консенсус надо достичь за конечное время
- **ТЕОРЕМА:** Невозможно достичь консенсуса N процессам
  - даже на множестве значений из двух элементов 0 и 1
- Доказательство от противного
  - Предположим что такой алгоритм существует и проанализируем возможные варианты его исполнения

FLP: Модель системы

- **Процесс** это некий автомат, который может делать
  - **receive():msg** чтобы ожидать получение сообщения
    - Нет возможности указать «время ожидания» (!)
  - **send(msg)** чтобы отправить сообщение
    - Отосланные сообщение не обязательно сразу обрабатываются
  - **decided(value)** когда принято решение
    - Решение процесс принимает один раз
    - Но имеет право продолжать выполняться (сообщать свое решение другим процессам)
- **Конфигурация** это
  - состояние всех процессов
  - все сообщения в пути (отправленные и не полученные)

FLP: Модель системы (2)

- **Начальная конфигурация** содержит начальные данные для каждого из процессов
  - Не обязательно один бит, а сколько угодно входных данных
    - Начальных конфигураций много (на каждый вариант входных данных)
  - И вообще каждый процесс может иметь свою программу
- **Шаг** от одной конфигурации до другой это
  - обработка какого-то сообщения процессом (*событие*)
  - внутренние действия этого процесса и посылка им от нуля до нескольких сообщений до тех пор, пока процесс не перейдет к ожиданию следующего сообщения.
    - Детерминировано (однозначно определяется) событием (!)

FLP: Модель системы (3)

- **Исполнение** это бесконечная цепочка шагов от начального состояния
  - ибо процессы продолжают выполняться и после принятия решения
- **Отказавший** процесс делает только конечное число шагов в процессе исполнения
  - И такой процесс от силы один
  - А каждый из остальных, не отказавших, процессов делает бесконечное число шагов
- Любое сообщение для не отказавшего процесса обрабатывается через конечное число шагов
  - Сообщения не теряются (!)

FLP: Базовые факты и определения

- Так как есть согласие, то все процессы пришедшие к решению имеют одно и то же решение (0 или 1)
  - Из-за возможности отказа одного процесса, даже если один процесс не делает шагов, то все остальные должны прийти к решению за конечное число шагов
- Конфигурация называется
  - i-**валентной**, если все цепочки шагов из неё приводят к решению i (0-валентные и 1-валентные конфигурации)
  - **бивалентной**, если есть так цепочки шагов приводящие к решению 0, так и цепочки шагов приводящие к решению 1
- Цепочки шагов с событиями на разных процессах коммутируют и приводят к одной и той же конфигурации если поменять их порядок выполнения

DEVEXPERTS

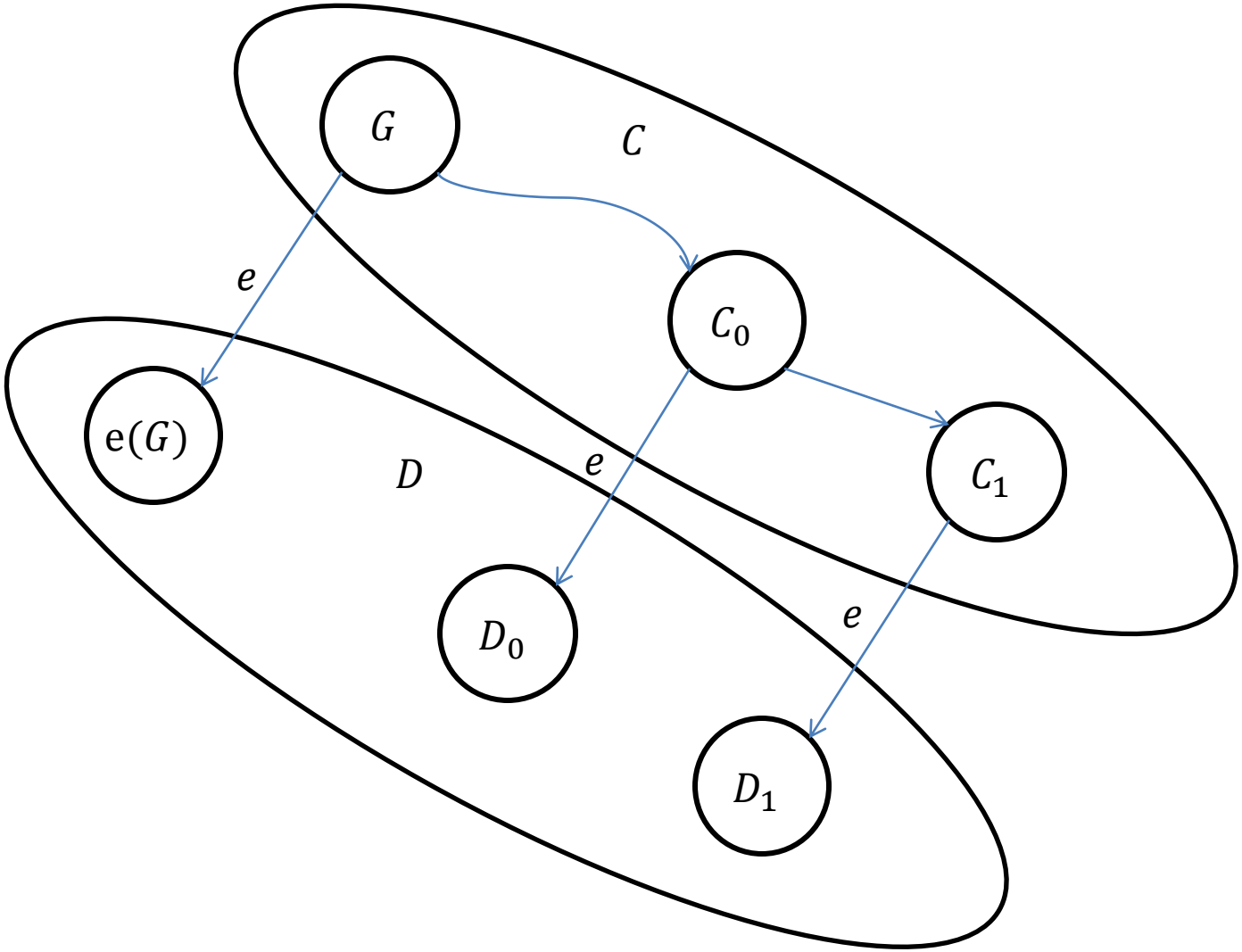
FLP Лемма 1: Существует (начальная) бивалентная конфигурация

- От противного. Если не существует (начальной) бивалентной конфигурации, значит все конфигурации одновалентны
  - Из нетривиальности есть как 0- так и 1- валентные
  - Значит найдем пару начальных конфигураций разной валентности, отличающихся начальным состоянием только одного процесса
  - Но этот процесс может отказать (не исполнятся) с самого начала, и тогда одна и та же цепочка шагов (других процессов) приводящая к решению возможна как в одной (0-валентной) конфигурации, так и в другой (1-валентной) конфигурации. Противоречие.

FLP Лемма 2: Для бивалентной конфигурации можно найти следующую за ней бивалентную

- Если G бивалентная конфигурация, и e это какое-то событие (процесс p и сообщение m) в этой конфигурации, то возьмем
  - C – множество конфигураций достижимых из G без e
  - D – множество конфигураций $D = e(C)$, то есть конфигураций где e это последнее событие
- Докажем что D содержит бивалентную конфигурацию
  - **Тем самым придем к противоречию с достижением консенсуса за конечное число шагов и докажем ТЕОРЕМУ**
  - По сути, мы воспользуемся асинхронностью: нет предела на время обработки сообщения, а значит любое сообщение можно отложить на любое конечное время
  - Докажем лемму 2 от противного. Предположим что D не содержит бивалентных конфигураций

FLP Лемма 2: Иллюстрация



FLP Лемма 2.1: i-валентные конфигурации

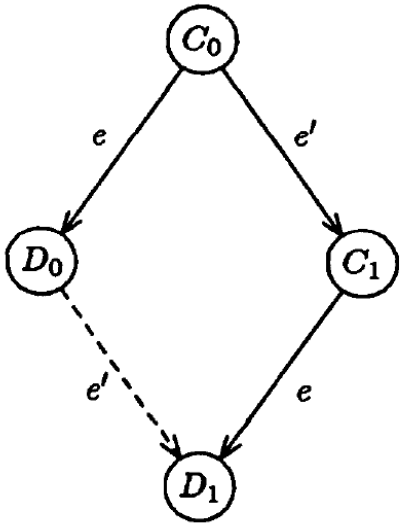
- Докажем что есть i-валентная конфигурация в D для любого i (0 или 1)
- Так как G бивалентная конфигурация, то по какой-то цепочке шагов из неё можно дойти до i-валентной E_i.
  - Если $E_i \in D$ то мы нашли искомую конфигурации
  - Если $E_i \in C$ то тогда $e(C) \in D$ искомая конфигурация
  - В противном случае e применялась в цепочке шагов для достижения E_i из G, а значит есть $F_i \in D$ (сразу после применения e) из которой доступна E_i по какой-то цепочке шагов
    - Но так как мы предположили, что в D нет бивалентных конфигураций, то $F_i \in D$ будет i-валентной

FLP Лемма 2.2: Соседние конфигурации

- Найдем такие соседние (отличающиеся одним шагом e') $C_0 \in C$ и $C_1 \in C$ что $D_0 = e(C_0) \in D$ является 0-валентной, а $D_1 = e(C_1) \in D$ является 1-валентной (например)
  - Пусть, не теряя общности, $C_0 = e'(C_1)$, где событие e' произошло на процессе p'
- Как это сделать:
  - Пусть, не теряя общности, $e(G) \in D$ является 0-валентной (если она 1-валентная, то симметрично)
    - Она соответствует пустой цепочке шагов из G.
  - Тогда по лемме 2.1 в D есть 1-валентная конфигурация $D_1 = e(C_1) \in D$
  - Будем убирать из цепочки шагов ведущей от G к C_1 по одному шагу с конца, пока не найдем искомую пару соседей C_0 и C_1

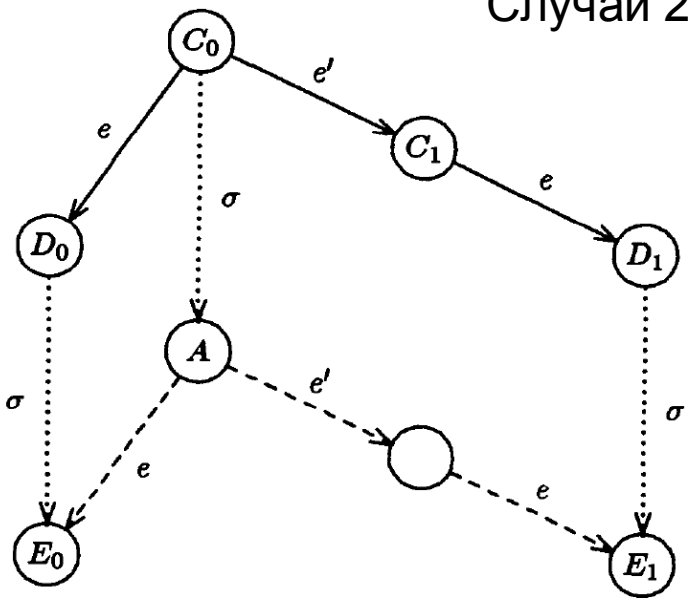
FLP: Разбор случаев

- Случай 1: Если $p \neq p'$ то e и e' коммутируют
  - Получается что D_1 должна быть одновременно 1- и 0- валентной. Противоречие
- Случай 2: Если $p = p'$ то рассмотрим цепочку шагов σ от состояния C_0 где процесс p отказал (не выполняется), а остальные пришли к решению
  - Тогда конфигурация $A = \sigma(C_0)$, с решением, должна быть 0- или 1-валентной
  - Но $E_0 = e(A) = \sigma(D_0)$ 0-валентая, а $E_1 = e\big(e'(A)\big) = \sigma(D_1)$ 1-валентная



Случай 1

Случай 2



FLP: Дополнительные наблюдения

- Результат FLP о невозможности консенсуса верен даже если процессу разрешено делать операцию «атомарной передачи» сообщения сразу несколько процессам
  - См. определение шага от одной конфигурации к другой в модели системы теоремы FLP
  - Однако, нет гарантии что все процессы обработают его
    - Один процесс может умереть и не получить
- Если есть гарантия получения сообщения всеми процессами (или ни одним), то такая операция называется Terminating Reliable Broadcast (TRB)
  - Имея TRB можно тривиально на его основе написать алгоритм консенсуса

Применения консенсуса

- Выбор лидера
  - Каждый процесс предлагает себя. Консенсус определяет лидера для последующего распределенного алгоритма
- Terminating Reliable Broadcast
  - Надо прийти к консенсусу о том, надо ли обрабатывать полученное сообщение
  - Таким образом, задача TRB эквивалентна задаче консенсуса

Консенсус в асинхронной сети

- Невозможность консенсуса при наличии ошибок (отказов) доказывается только для **детерминированных** алгоритмов в **асинхронной** сети
- Можно придти к консенсусу
  - Если сделать сеть синхронной (ограничить время доставки сообщений)
  - Если сделать алгоритм недетерминированным (случайным)
  - Если ослабить требования при которых в алгоритме обязан быть прогресс (т.е. он обязан завершаться)

Алгоритмы консенсуса в синхронных сетях

- При отказе f узлов делаем алгоритм с f+1 фазой
- Ненадежная доставка – проблема двух генералов
- При византийской ошибке
  - Решение возможно, только если N > 3f
  - 2-х фазный алгоритм решения при N=4, f=1
    - Обобщается на общий алгоритм с f+1 фазами
  - Доказательство невозможности при N=3, f=1

Недетерминированные алгоритмы консенсуса

- Разрешаем процессам кидать монетку
- Требуем достижения консенсуса с вероятностью 1
- Порядок исполнения операций в системе выбирает противник
  - Сильный противник знает всё
  - Слабый противник (более реалистично) как-то ограничен
- Алгоритм Бен-Ора (1983) для бинарного консенсуса работает даже при сильном противнике при любом $f < N/2$
  - Но ожидаемое время достижения консенсуса $O(2^N)$

Алгоритм Бен-Ора (Ben-Or)

- Множество раундов, по две фазы в раунде
  - На каждой фазе процесс ждет получения N-f сообщений
- Фаза 1: «**Предпочтение**» рассылает текущее предпочтение посылая всем сообщение (1, k, p), где k это номер раунда
  - Если больше N/2 голосов за одно значение, то **ратифицирует** его
- Фаза 2: «**Ратификация**» шлет (2, k, v) чтобы ратифицировать значение или (2, k, ?) если не набрал нужное число голосов
  - Получив хотя бы одну ратификацию (свою или от другого) на следующий раунд меняет предпочтение на полученное v
  - Получив больше f ратификаций процесс **принимает решение** v (**decide**), но продолжает исполняться
  - Не получив ратификации, меняет предпочтение для следующего раунда на случайное

Ben-Or: Корректность

- **Лемма 1**: В одном раунде разные процессы не могут **ратифицировать** разные значения
  - Очевидно из-за необходимость набрать больше N/2 сообщений (1, k, v) для ратификации значения v
- **Лемма 2**: Если в раунде k процесс **принял решение** v, то k+1 раунд все процессы начнут с предпочтением v
  - Чтобы принять решение процесс получил минимум f+1 сообщений вида (2, k, v)
  - Чтобы начать раунд с другим предпочтением процесс должен был получить N-f сообщений вида (2, k, ?)
    - Сообщений с другим v быть не может по Лемме 1, а при приеме хотя бы одного (2, k, v) он бы взял предпочтение v
    - Но (f+1) + (N-f) > N. Противоречие.

Ben-Or: Заключение

- Чтобы остановить алгоритм, а не исполнять его бесконечно, надо высылать третий тип сообщения «**Решение**» как только решение принято
  - И при его получении принимать решение и останавливаться
- Система асинхронная (!). Сообщения не обязаны приходить в каждый процесс «раунд за раундом».
  - Но так как на каждом раунде ждем N-f сообщений, то фактически раунды получаются почти синхронными
- Даже если противник, который решает как приходят сообщения (он выбирает цепочку шагов в исполнении), *сильной*, то есть знает всё состояние системы
  - То все-равно вероятность завершения алгоритма за конечное число шагов равна единице

Replicated State Machine (RSM)

- Имеем некое состояние (значение счета в банке, состояние персонажа в игре и т.п.)
  - И его надо быстро обновлять (т.е. не можем позволить себе хранить его на диске)
  - И его надо защитить от сбоев конкретного узла, где оно хранится
- **РЕШЕНИЕ**: Держим несколько копий этого состояния на разных узлах для надежности
  - Если операции детерминировано влияют на состояние, то можем независимо примять все операции к разным репликами
  - Но встает проблема поддержания одинакового состояния
    - Если операции не коммутируют между собой
  - То есть, при одновременной попытке выполнить несколько операций надо прийти к консенсусу какая операция будет первой

Алгоритм Paxos

- Лампорт (1989)
- Первый алгоритм практического асинхронного консенсуса
  - Каждый процесс выбирает одно значение из множества предложенных (более сильное требование, чем нетривиальность)
  - Гарантирует **согласие** при любых отказах и произвольных задержках сообщений
  - Но не гарантирует завершение за конечное время
    - По теореме FLP он и не может этого гарантировать
    - Однако, при определенных практических реалиях (когда отказы происходят не часто) консенсус достигается за конечное число шагов

Асинхронный консенсус для практического применения: Задача

- Для начала решим задачу однократного консенсуса
  - Задача выбора i-ой операции над RSM
- Имеем множество **предлагающих** (proposer) процессов
  - Это процессы пытающиеся выполнить операция над RSM и предлагающие свою операцию в качестве следующей
- Проще всего, когда принимает решение один **принимающий** (acceptor) процесс
  - Тогда нет проблемы прийти к консенсусу
    - Сообщение пришедшее к принимающему первым и принято
  - Но в случае отказа принимающего процесса в системе не будет прогресса
  - Поэтому нужно несколько принимающих решение процессов

Асинхронный консенсус для практического применения (2)

- Нужна возможность узнать о принятом решение некому множеству **узнающих** (learner) процессов
  - Принимающие процессы не обязаны быть теми же самыми, которые хранят копию состояния RSM, они могут лишь принимать решение о порядке операций, сообщая о нем множеству узнающих процессов, которые обновляют у себя состояние RSM
- Таким образом, в общем случае, имеем три роли:
  - **Предлагающие** (proposers) предлагают значения
  - **Принимающие** (acceptors) принимают решения
  - **Узнающие** (learners) узнают о решениях
- На практике роли могут произвольно совмещаться при необходимости

Кворум

- Хотим алгоритм, который корректно работает на основе **кворума** множества принимающих процессов
  - Множество принимающих процессов и используемый кворум (что считается кворумом) заранее выбрано и зафиксировано
- Можно использовать любой кворум
  - простое большинство, например (2 процесса из 3-х это кворум)
- Для кворума не нужны все процессы
  - Поэтому отказ одного или нескольких процессов (в зависимости от используемого кворума) не остановит работу
- Предполагается что отказ временный и через некоторое время отказавший принимающий процесс продолжит работу
  - **Отказ это когда процесс «уснул надолго и не отвечает»**
  - Переконфигурирование алгоритма это отдельная задача

Лидер

- Среди принимающих надо выбрать **лидера**
  - Каждый предлагающий должен знать множество принимающих (фиксированное) и кто из них лидер
    - А лидер должен меняться, если старый лидер отказал
- **Лидер выполняет вспомогательную функцию в алгоритме**
  - Мы не можем гарантировать выбор одного лидера за конечное время (для этого надо прийти к консенсусу)
  - Но будем выбирать лидера за конечное время (тривиально)
    - Но тогда согласно теореме FLP не можем дать гарантии что будет согласие о том, кто является лидером
  - Поэтому может временно оказаться несколько лидеров
    - Разные принимающие будут иметь свое мнение о лидере
    - Но алгоритм Paxos будет гарантировать **согласие** и в этом случае, но без гарантий завершения, пока лидеров несколько

Paxos: основа алгоритма

- Для прихода к консенсусу алгоритм Paxos делает один или несколько раундов **голосования**
  - Раунд голосования инициируется лидером (предложения высылаются ему, а он их ставит на голосование)
  - Несколько раундов может случиться только если нет согласия о едином лидере или что-то отказало и надо его начать заново
  - Вся структура алгоритма нужна для обеспечения **согласия** несмотря на то, что несколько голосований может происходить одновременно
  - Каждое запущенное голосование имеет уникальный номер k
    - Как обычно, можно взять пару из номера процесса и локально-увеличиваемого счетчика
  - Лидер может заново инициировать голосование «по своему вопросу», если видит что прогресса нет, с другим номером k

Paxos: 1-ая фаза голосования

- Фаза 1a: «**Подготовка**» Лидер инициирует голосование и посылает сообщение (1a, k) **кворуму** принимающих, где k это глобально уникальный номер голосования
- Фаза 1b: «**Обещание**» Получив сообщение (1a, k) принимающий обещает не принимать предложение с меньшим номером и отвечает (1b, k, ack, k', v'), где (k', v') это информация о **принятом** предложении с максимальным номером k' < k (см. фазу 2b) где k' = 0 если ничего еще не принято; или сообщает что уже дал другое **обещание** для k'' > k и отвечает (1b, k'', nack), на что лидер повторит 1a заново, послав (1a, k'')

Paxos: 2-ая фаза голосования

- Фаза 2a: «**Запрос**» Лидер, получив обещания (1b, k, ack, k’, v’) от **кворума** принимающих, предлагает свое значение. Берет значение v’ для наибольшего k’ полученного от acceptor-ов. А если все k’ == 0, то предлагает свое значение v и посылает запрос (2a, k, v) **кворуму** принимающих
  - На 2-ой фазе можно использовать другой кворум
- Фаза 2b: «**Подтверждение**» Если принимающий получает запрос (2a, k, v) и он не давал обещание для k’ > k, то он **принимает** предложение (k, v) и посылает сообщение (2b, k, v) всем узнающим.
- Узнающий, получив сообщение (2b, k, v) от **кворума** принимающих узнает о том, что выбрано значение v.

Модификации Paxos

- Mutli Paxos – выполняя Paxos много раз подряд, можно делать Фазу 1 для всех копий сразу (она не зависит от v), и подтвердив факт наличия уникального лидера через это, делать для каждой копии алгоритма Paxos свою Фазу 2. Получаем задержку в 3 сообщения между proposer-ом и learner-ом.
- Fast Paxos – посылаем предложения acceptor-ам (в обход лидера). Получаем задержку в 2 сообщения между proposer-ом и learner-ом, если нет коллизий.
- Dynamic Paxos – меняем набор серверов во время работы.
- Cheap Paxos – активно используем только кворум из f+1 серверов и только при отказе, подключаем до f запасных.
- Stoppable Paxos, Byzantine Paxos, и т.п.

Транзакции в распределенный системах

- **Транзакция** это единица работы над множеством элементов, хранящихся в базе данных
- **ACID** свойства
  - Atomicity (атомарность) – все изменения или ничего
  - Consistency (согласованность) – перевод системы согласованное состояние в конце транзакции
  - Isolation (изолированность) – параллельные транзакции не должны влиять друг на друга, а выполняться как будто бы последовательно
  - Durability (надежность) – завершенные транзакции сохраняются даже в случае сбоев и перезапуска системы

Транзакции в распределенный системах

- 2PL (2-Phase Locking)
- 2PC (2-Phase Commit)