Логические часы

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Распределенная обработка информации»
Сибирский государственный университет телекоммуникаций и информатики
Осенний семестр, 2019

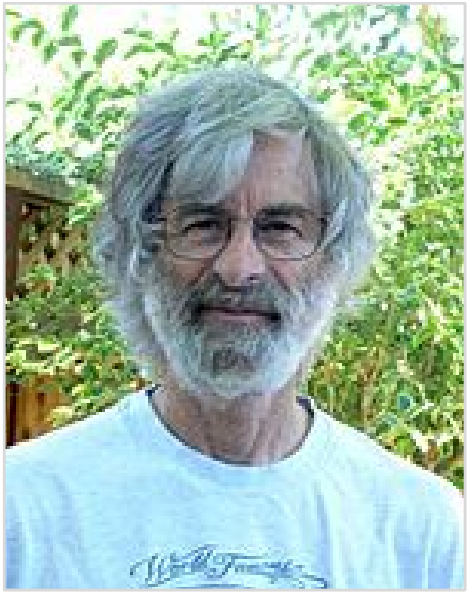
Синхронизация показаний локальных часов

- В распределенных системах невозможно идеально синхронизировать показания локальных часов процессов [Lamport, 1978] // http://www.lamport.org

- **Проблема**
  - ❑ В разных процессах произошло два события
  - ❑ Какое из событий наступило первым?
  - ❑ Как упорядочить все события системы (установить отношение порядка)?

Leslie Lamport

- **Alma mater:** MIT (BSc), Brandeis University (PhD)
- **Current:** Microsoft Research
- **Results**
  - ❑ Paxos algorithm for consensus
  - ❑ Sequential consistency
  - ❑ Lamport's bakery algorithm
  - ❑ Byzantine fault tolerance
  - ❑ Lamport signature
  - ❑ Temporal logic of actions
  - ❑ LaTeX
  - ❑ …



http://www.lamport.org

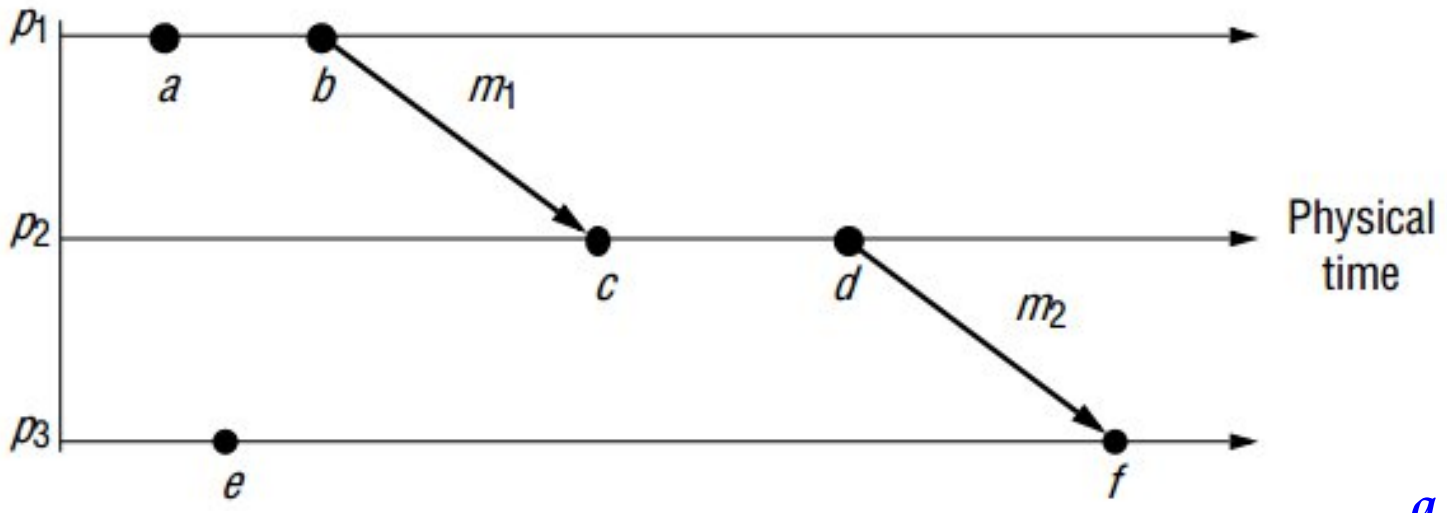
ACM Turing Award 2013

(Outstanding contributions in computer science)

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency

Логические часы Лампорта (Logical clock)

- Л. Лампорт в 1978 г. предложил идею логических часов, которые позволяют определить для событий **отношение «произошло-перед» (happened-before):**
  - HB1: События одного процесса упорядочены: $e \rightarrow_i e'$, тогда $e \rightarrow e'$
  - HB2: Приём сообщения происходит после его отправления: $\text{send}_i(m) \rightarrow \text{recv}_j(m)$
  - HB3: Отношение транзитивно: $e \rightarrow e'$ и $e' \rightarrow e''$, тогда $e \rightarrow e''$



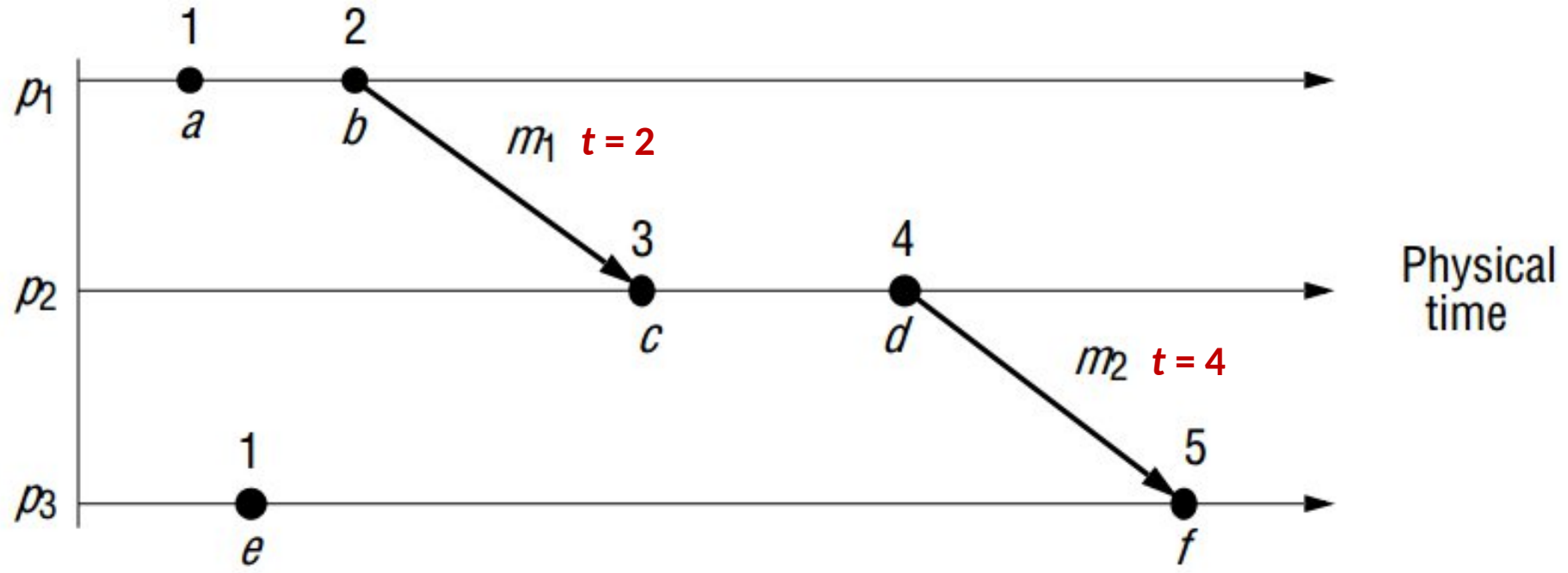
- $a \rightarrow b$
- $b \rightarrow c$
- $c \rightarrow d$
- $d \rightarrow f$
- $e \rightarrow f$

$a \rightarrow e, e \rightarrow a$ => $e \parallel a$ - concurrent

Логические часы Лампорта (Logical clock, 1978)

- **Позволяют упорядочить события в распределенной системе (установить отношение частичного порядка «произошло-перед»)**
- Каждый процесс p_i поддерживает свои *логические часы* L_i – монотонно увеличивающийся счетчик (в начальный момент времени $L_i = 0$)
- Каждому событию e процесса p_i присваивается временная метка $L_i(e)$ – Lamport timestamp
- LC1: L_i увеличивается на 1 перед каждым событием процесса p_i
- LC2:
  a) Когда процесс p_i отправляет сообщение m он снабжает его текущим показанием своих локальных часов $t = L_i$
  b) Когда процесс p_j получает сообщение (m, t) он корректирует показание своих локальных часов $L_j = \max(L_j, t)$ и применяет LC1 перед назначением метки для recv

Логические часы Лампорта (Logical clock)

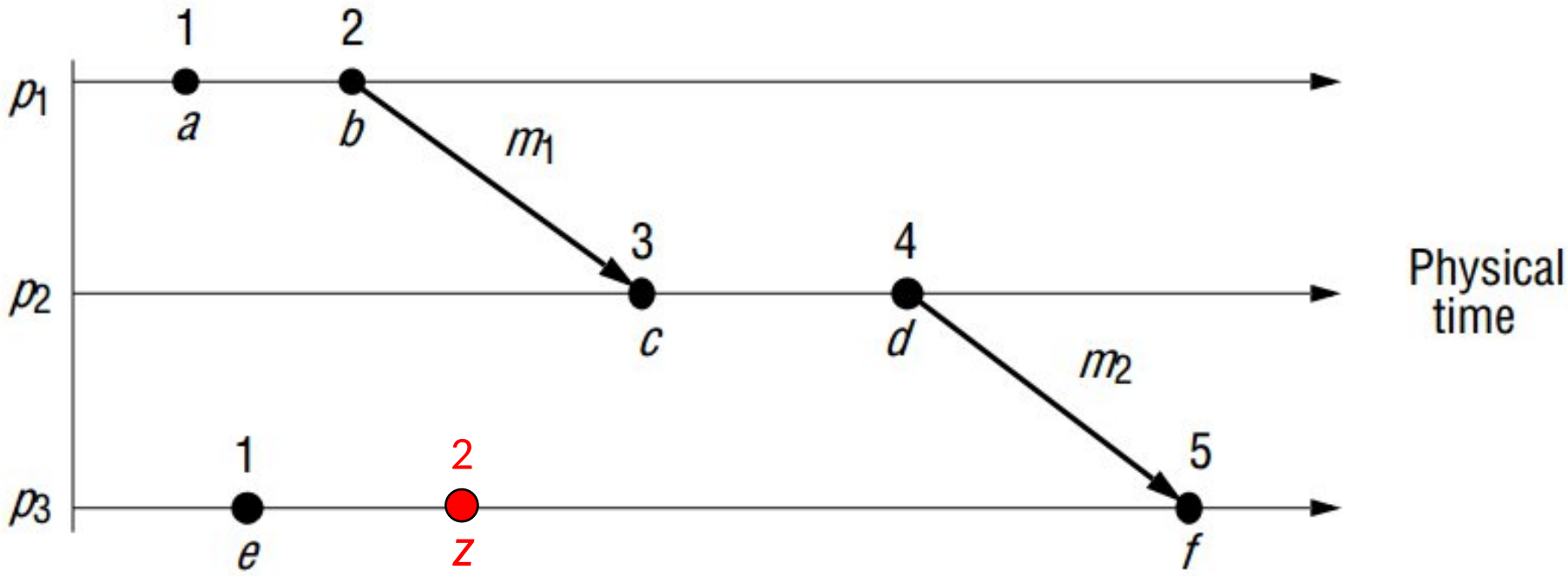


$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

$$L(e) < L(e') \Rightarrow e \rightarrow e'$$

$L(e) < L(b)$, но $e \parallel b$

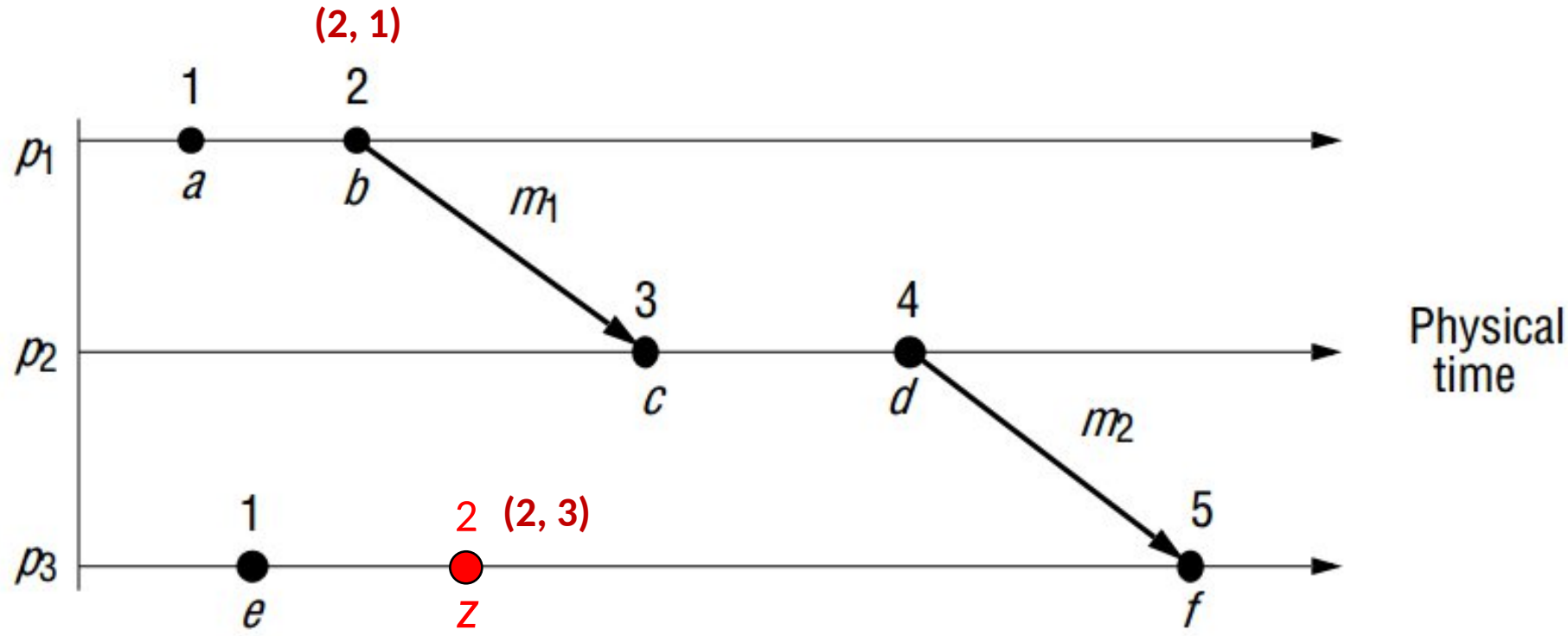
Логические часы Лампорта (Logical clock)



$L(b) = L(z)$

Какое событие произошло раньше b или z?

Логические часы Лампорта (Logical clock)



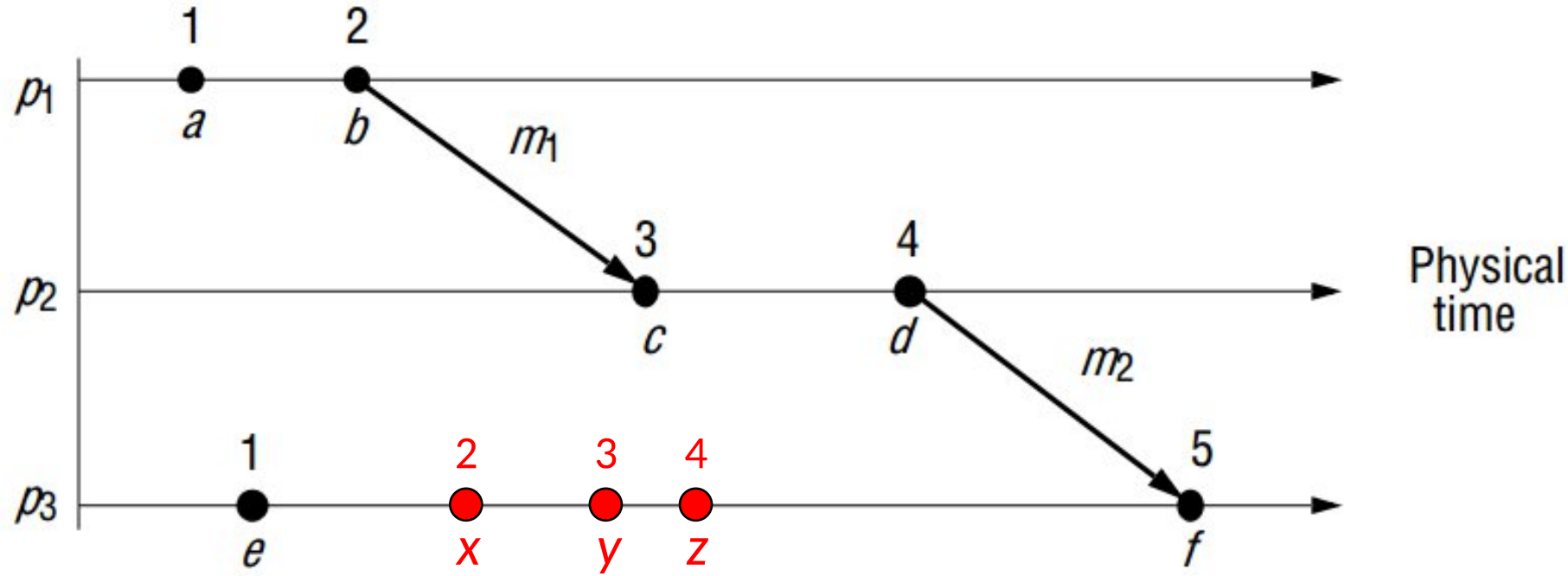
Решение Лампорта

Метка каждого события процесса p_i: (L_i, i)

$(L_i, i) < (L_j, j) \leftrightarrow L_i \leq L_j$ и $i < j$

$b \rightarrow z$

Логические часы Лампорта (Logical clock)



Из того что L(e) < L(e') не следует, что e -> e'

Логические часы Лампорта (Logical clock)

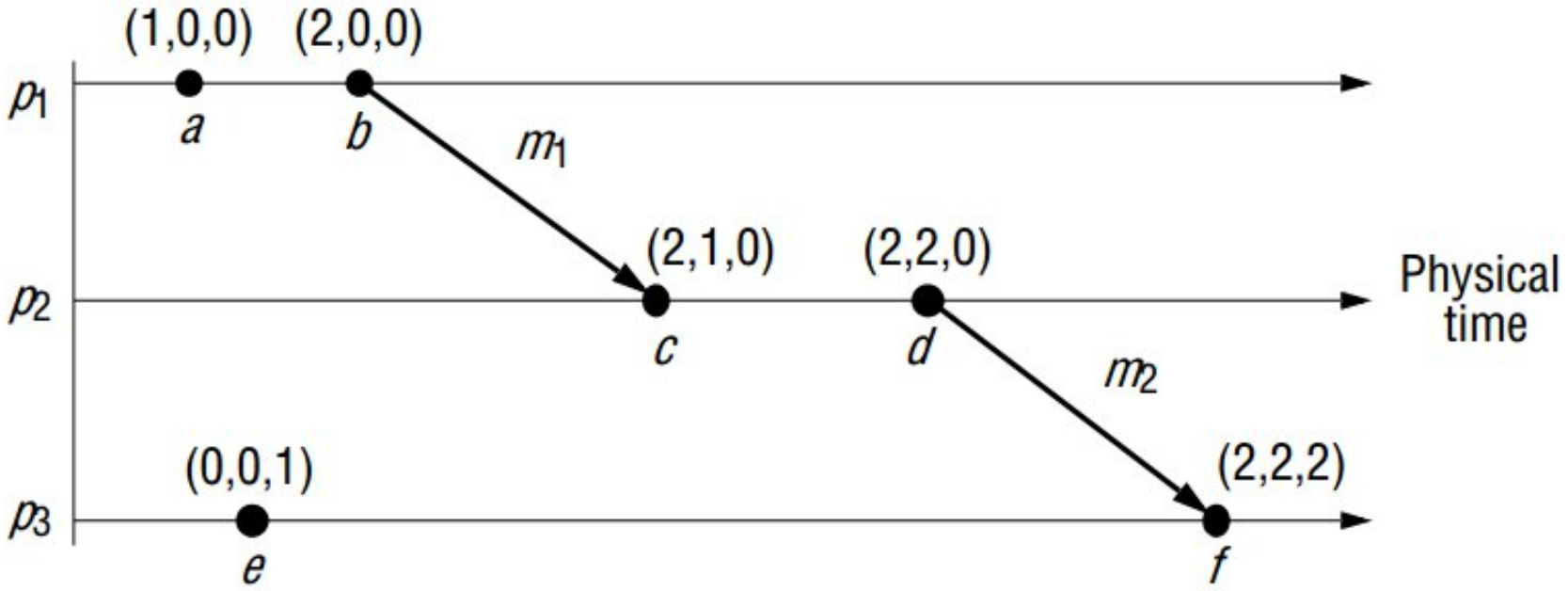
- **Пример**
- Процесс A отправляет запрос в базу данных на запись
- Процесс A отправляем запрос на чтение процессу B
- Процесс B получил запрос от A и отправляет свой запрос на чтение в базу
- База получила запросы от A и B одновременно, какой запрос предшествует какому?

Векторные часы (Mattern, 1989), (Fidge, 1991)

- **Векторные логические часы (vector clocks) – вектор *n* целых чисел**
  - Каждый процесс p_i поддерживает свои логические часы $V_i[1{:}n]$ – вектор длины n
  - VC1: В начальный момент времени $V_i[j] = 0$, $i, j = 1, 2, \ldots, n$
  - VC2: Перед каждым событием процесса p_i : $V_i[i] = V_i[i] + 1$
  - VC3: Когда процесс p_i отправляет сообщение m он снабжает его текущим показанием своих локальных часов $t = V_i[1{:}n]$
  - VC4: Когда процесс p_i получает сообщение $(m, t[1{:}n])$ он корректирует показание своих локальных часов (берётся поэлементный максимум):

$$V_i[j] = \max(V_i[j], t[j]), \quad j = 1, 2, \ldots, n$$

Векторные часы (Mattern, 1989), (Fidge, 1991)



$V = V' \ \textit{iff} \ V[j] = V'[j] \ \text{for} \ j = 1, 2 \ldots, N$

$V \leq V' \ \textit{iff} \ V[j] \leq V'[j] \ \text{for} \ j = 1, 2 \ldots, N$

$V < V' \ \textit{iff} \ V \leq V' \wedge V \neq V'$

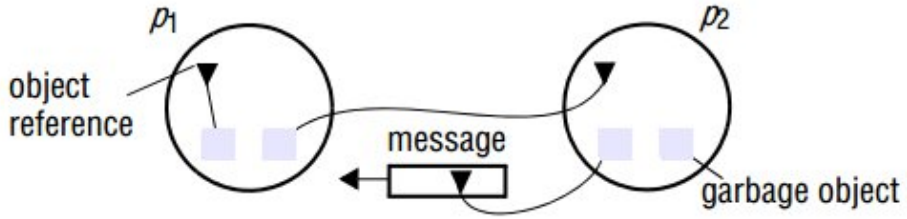
$V(e) < V(e') \Rightarrow e \longrightarrow e'$

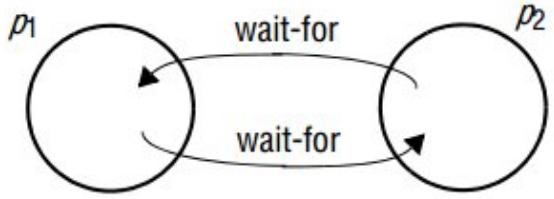
$V(a) < V(f) \Rightarrow a \longrightarrow f$

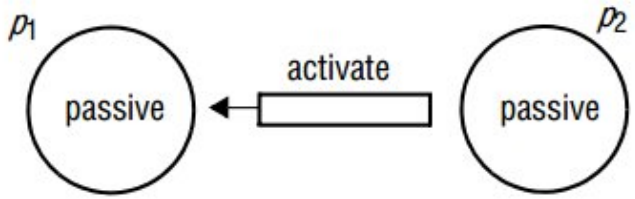
Глобальное состояние (Global state)

- **Как определить в каком состоянии находится распределенная система в заданный момент времени?**

- **Примеры**
  - ❑ Сборка мусора (Garbage collection): подсчет числа ссылок на заданный объект
  - ❑ Определение взаимной блокировки (Deadlock): обнаружение цикла в графе отношения “wait-for”
  - ❑ Определение завершения распределенного алгоритма
  - ❑ Отладка распределенного приложения







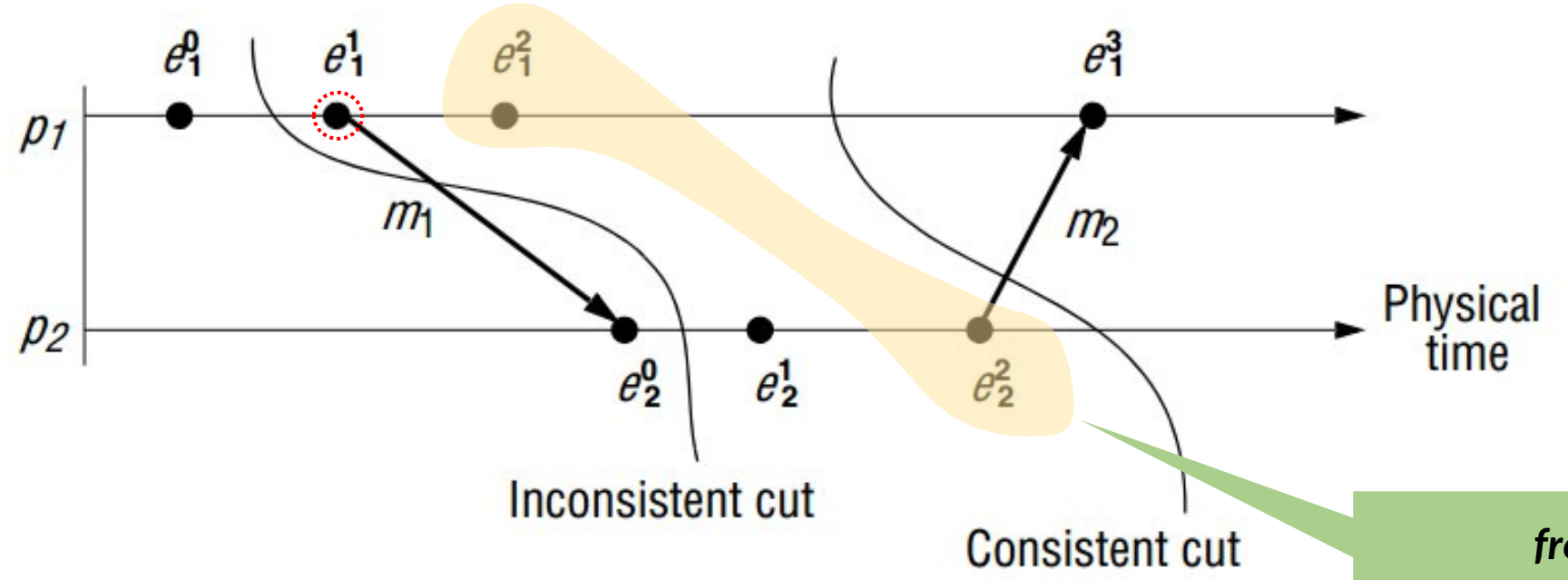
Глобальное состояние (Global state)

- **Глобальное состояние $S = (S_1, S_2, ..., S_n)$** – это совокупность состояний процессов распределенной системы

- **Присутствуют глобальные часы**
  - ❑ При наличии глобальных часов процессы договариваются о моменте времени *T* и достигнув его сохраняют свои состояния

- **Глобальных часов нет**
  - ❑ Можно ли корректно “собрать” глобальное состояние из локальных состояний процессов в отсутствии глобальных часов?

Глобальное состояние (Global state)



- Обозначим через $h_i^k = (e_i^0, e_i^1, \ldots, e_i^k)$ историю k событий процесса p_i
- ***Сечение*** (срез, cut) $C = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_n^{cn}$ – подмножество глобальной истории событий
- ***Согласование сечение*** (consistent cut) – это сечение, в котором для любого события *e* содержится событие предшествующее ему
- ***Согласованное глобальное состояние*** (consistent global state) – это глобальное состояние соответствующее согласованному сечению

Построение моментального снимка (Snapshot)

- **Моментальный снимок** (Snapshot) – это согласованное глобальное состояние системы, включающее:
  - ❑ состояния *n* процессов
  - ❑ состояния каналов связи между процессами

- На самом деле состояния могли не иметь место одновременно, но соответствуют согласованному срезу

Алгоритм Chandy-Lamport (1985)

- K. Mani Chandy, Leslie Lamport. **Distributed Snapshots: Determining Global States of Distributed Systems** // 1985, http://research.microsoft.com/users/lamport/pubs/chandy.pdf
- **Имеется** *n* процессов
- Процессы и каналы связи между ними абсолютно надежны
- Каналы односторонние и доставляют сообщения в FIFO-порядке
- Граф из процессов и каналов между ними связный

Алгоритм Chandy-Lamport (1985)

- K. Mani Chandy, Leslie Lamport. **Distributed Snapshots: Determining Global States of Distributed Systems** // 1985, http://research.microsoft.com/users/lamport/pubs/chandy.pdf
- **Требуется** построить согласованное глобальное состояние процессов (snapshot)
- Любой процесс может инициировать построение глобального снимка в любой момент времени
- Процессы могут продолжать свое выполнение и обмениваться сообщениями во время построения снимка

Алгоритм Chandy-Lamport (1985)

- Каждый процесс имеет исходящие (outgoing) и входящие (incomming) каналы
- Каждый процесс записывает свои состояния и сообщения, присылаемые ему
- Если процесс p_i отправил сообщение m процессу p_j, но процесс p_j его еще не получил, то m учитывается как состояние канала между процессами
- Алгоритм использует *сообщение-маркер (marker)*:
  - ❑ Информирует получателя о необходимости сохранить свое состояние
  - ❑ Запускает или заканчивает процедуру записи входящих сообщений

Алгоритм Chandy-Lamport Snapshot

Marker receiving rule for process p_i

On receipt of a *marker* message at p_i over channel c:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

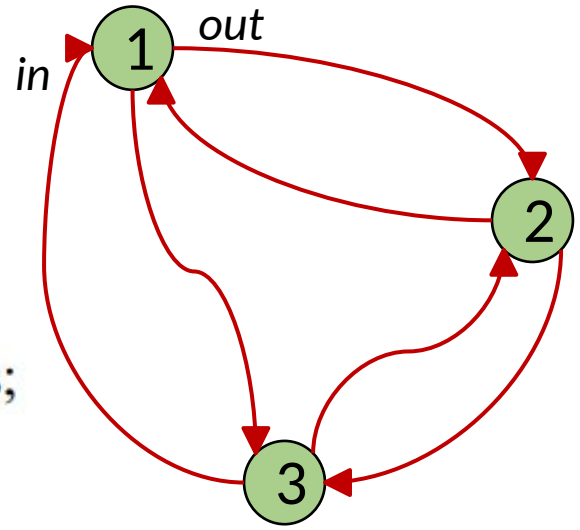
p_i records the state of c as the set of messages it has received over c since it saved its state.

end if

Marker sending rule for process p_i

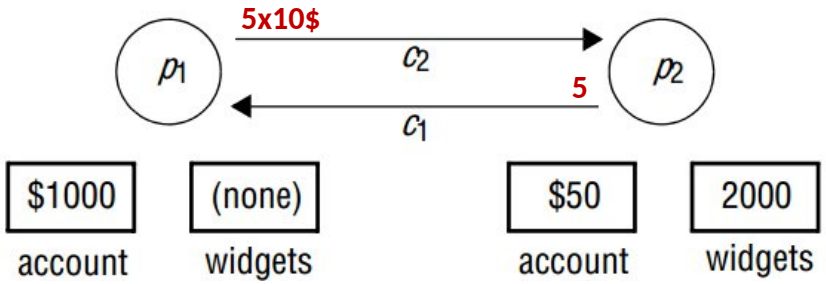
After p_i has recorded its state, for each outgoing channel c:

p_i sends one marker message over c (before it sends any other message over c).



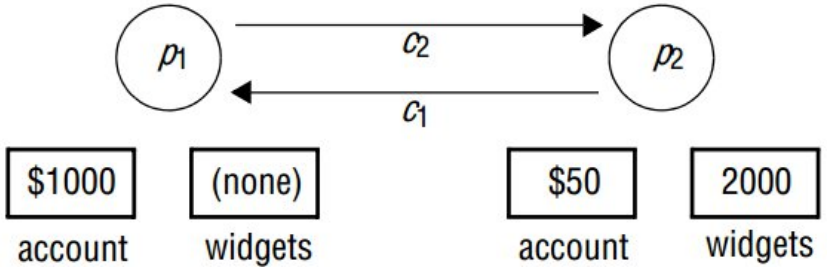
Любой процесс запускает алгоритм и действует по правилу “receiving rule” (принял маркер по несуществующему каналу)

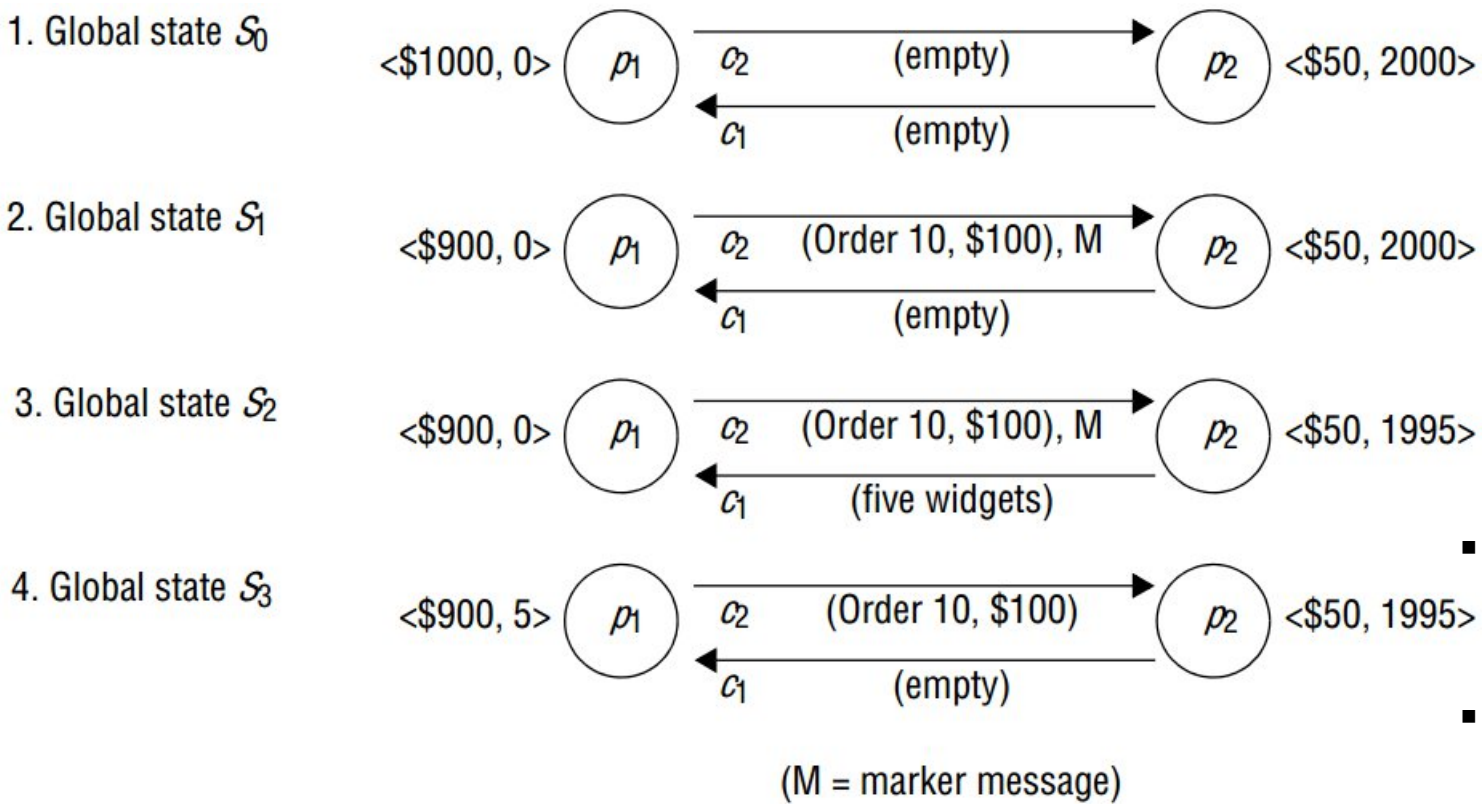
Алгоритм Chandy-Lamport



- Два процесса ведут торги
- Процесс p1 отправил запрос p2 на 5 предметов по 10$
- Процесс p2 отправил 5 предметов p1

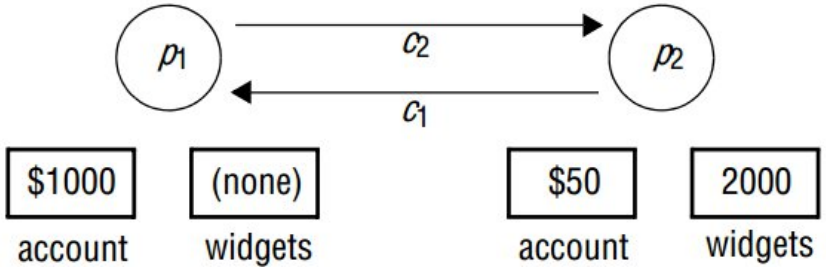
Алгоритм Chandy-Lamport

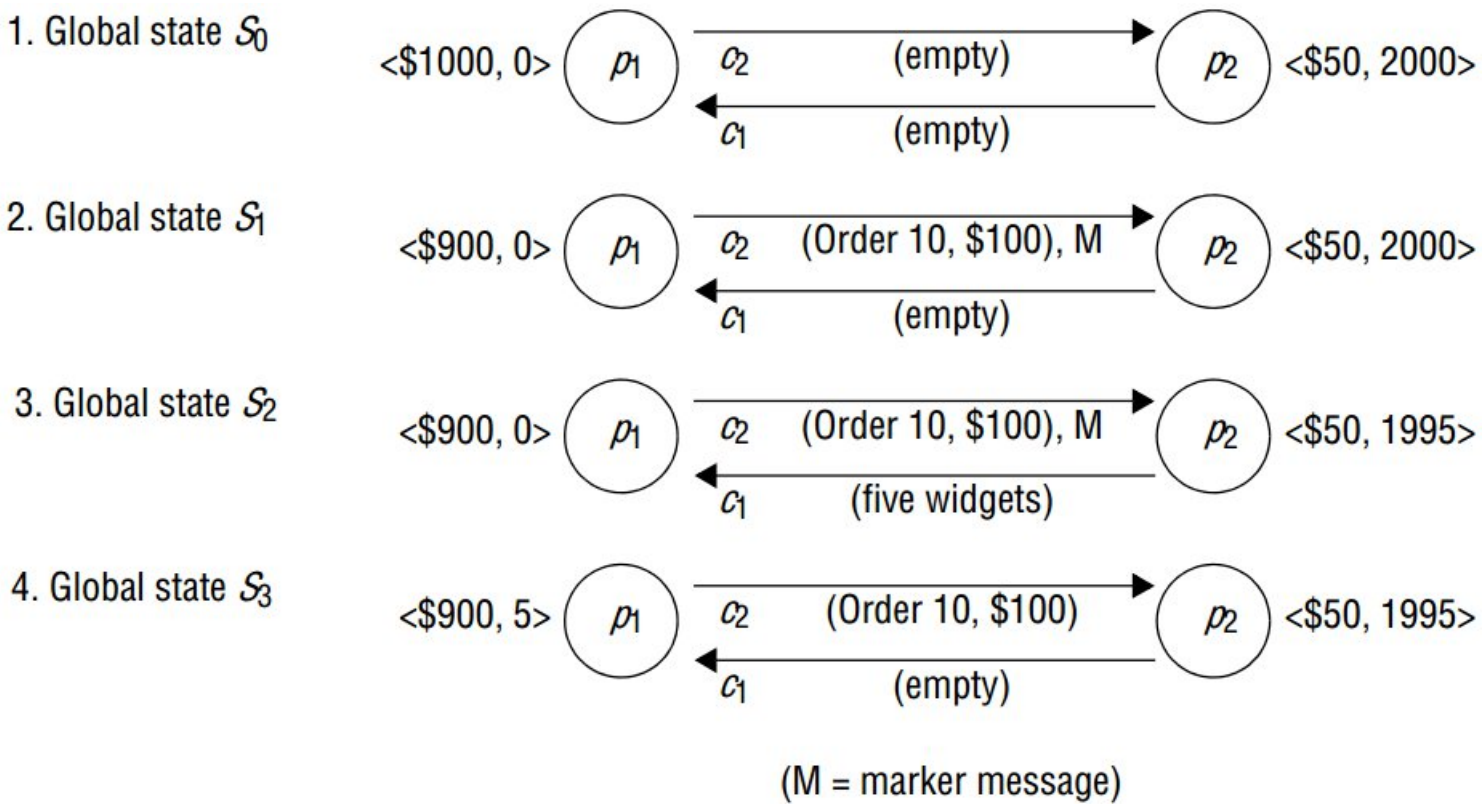




- **Процесс p1 запустил создание снимка:**
  - ❑ записал свое состояние S_0 = <$1000, 0>
  - ❑ начал записывать входящие сообщения
  - ❑ отправил маркер через *c*2,
  - ❑ отправил заказ на 100$ => S_1
- Процесс 2 отправил в ответ 5 предметов => S_2
- Процесс 1 получил <5 предметов>
  Процесс 2 получил маркер, сохранил свое состояние <50$, 1995>, отправил маркер через c_1
- Процесс 1 получил маркер от 2 и сохранил сообщение <5 предметов> => S_3

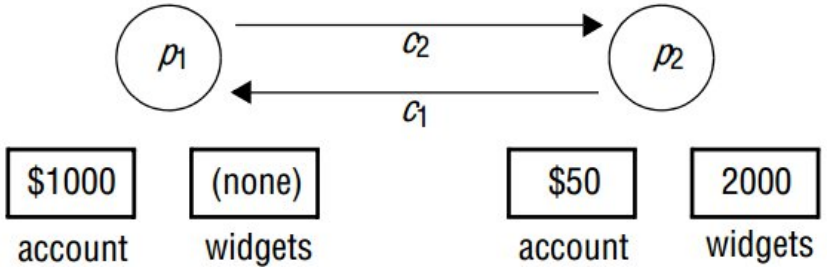
Алгоритм Chandy-Lamport

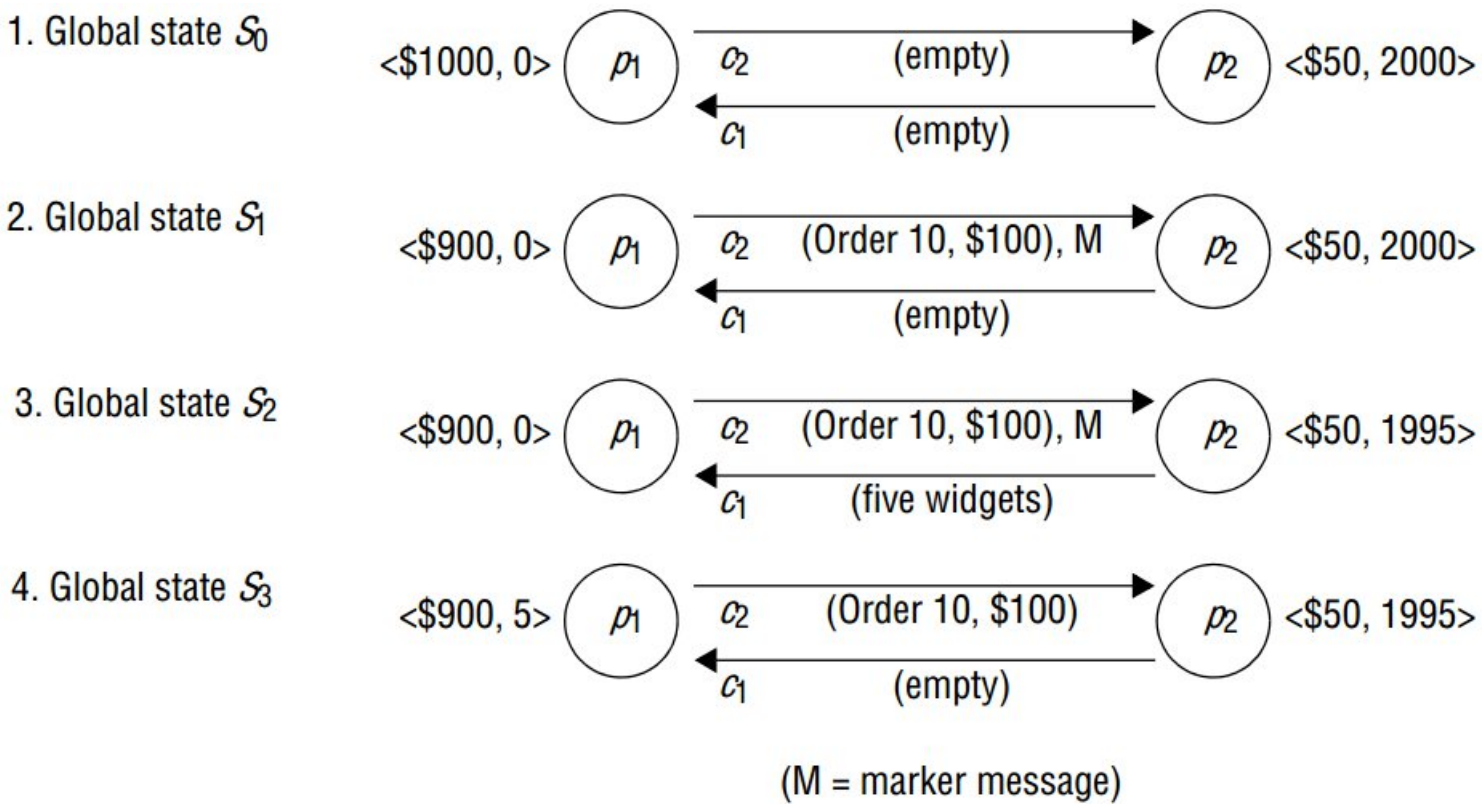




- Процесс p1 запустил создание снимка:
  - ❑ записал свое состояние S_0 = <$1000, 0>
  - ❑ начал записывать входящие сообщения
  - ❑ отправил маркер через *c*2,
  - ❑ отправил заказ на 100$ => S_1
- **Процесс p2 получил запрос от p1**
  - ❑ отправил в ответ 5 предметов через c1 => S_2
  - ❑ получил маркер через c2,
    сохранил свое состояние <$50, 1995> и c2 = <>
  - ❑ отправил маркер через c1

Алгоритм Chandy-Lamport





- Процесс p1 запустил создание снимка:
  - ❑ записал свое состояние S_0 = <$1000, 0>
  - ❑ начал записывать входящие сообщения
  - ❑ отправил маркер через *c*2,
  - ❑ отправил заказ на 100$ => S_1
- Процесс p2 получил запрос от p1
  - ❑ отправил в ответ 5 предметов через c1 => S_2
  - ❑ получил маркер через c2,
    сохранил свое состояние <$50, 1995> и c2 = <>
  - ❑ отправил маркер через c1
- **Процесс p1 получил маркер от p2**
  - ❑ записал состояние канала c1 = <5 widgets>

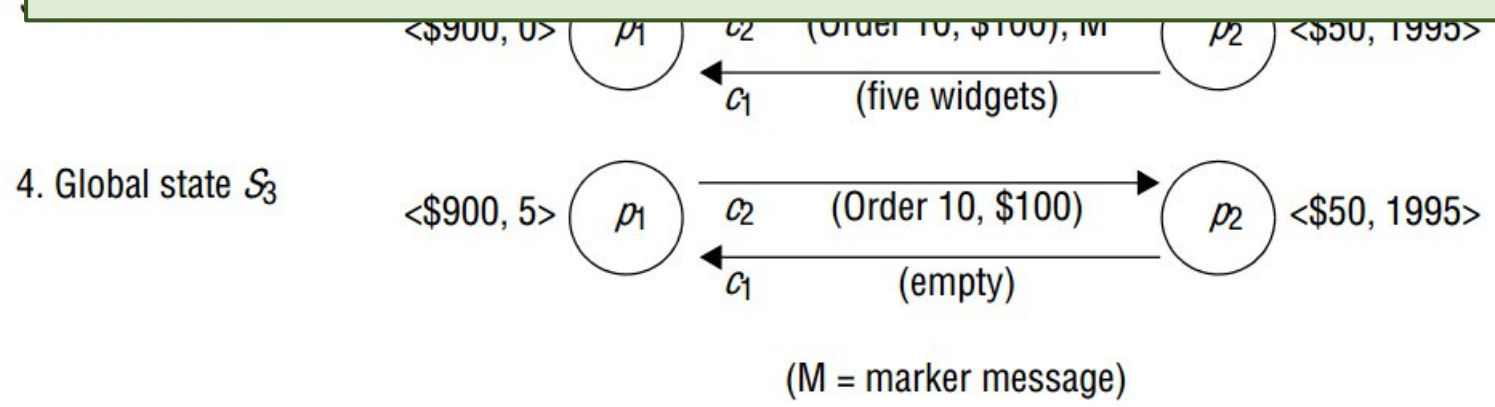
p1 <$1000, 0>, p2 <$50, 1995>, c1 = <5>, c2 = <>

Алгоритм Chandy-Lamport

Snapshot

p_1 = <$1000, 0>; p_2 = <$50, 1995>; c_1 = <5 предметов>; c_2 = <>

Как собрать в одном процессе сохранённые состояния всех?
(All-to-one broadcast, gather)



- получил маркер через c2,
  сохранил свое состояние <$50, 1995> и c2 = <>
- отправил маркер через c1

- **Процесс p1 получил маркер от p2**
  - записал состояние канала c1 = <5 widgets>

p1 <$1000, 0>, p2 <$50, 1995>, c1 = <5>, c2 = <>

Взаимное исключение (distributed mutual exclusion)

- Процессы распределенной системы работают с общим ресурсом, доступ к которому должен быть защищен критической секцией

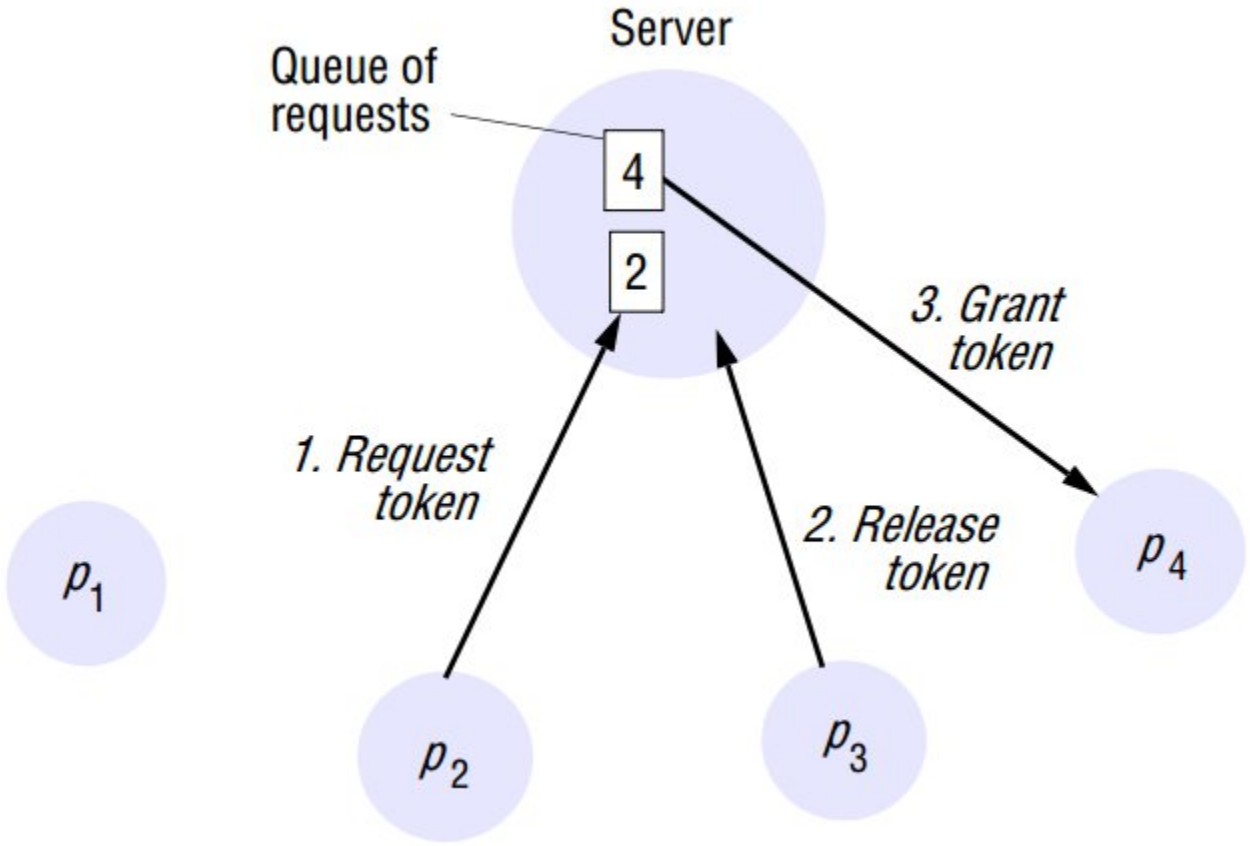
  EnterCriticalSection()
  Process()
  LeaveCriticalSection()

- **Пример**
  - ❑ Доступ к среде в протоколах Ethernet, IEEE 802.11 (WiFi) in ad-hoc mode

Взаимное исключение (Distributed Mutual Exclusion)

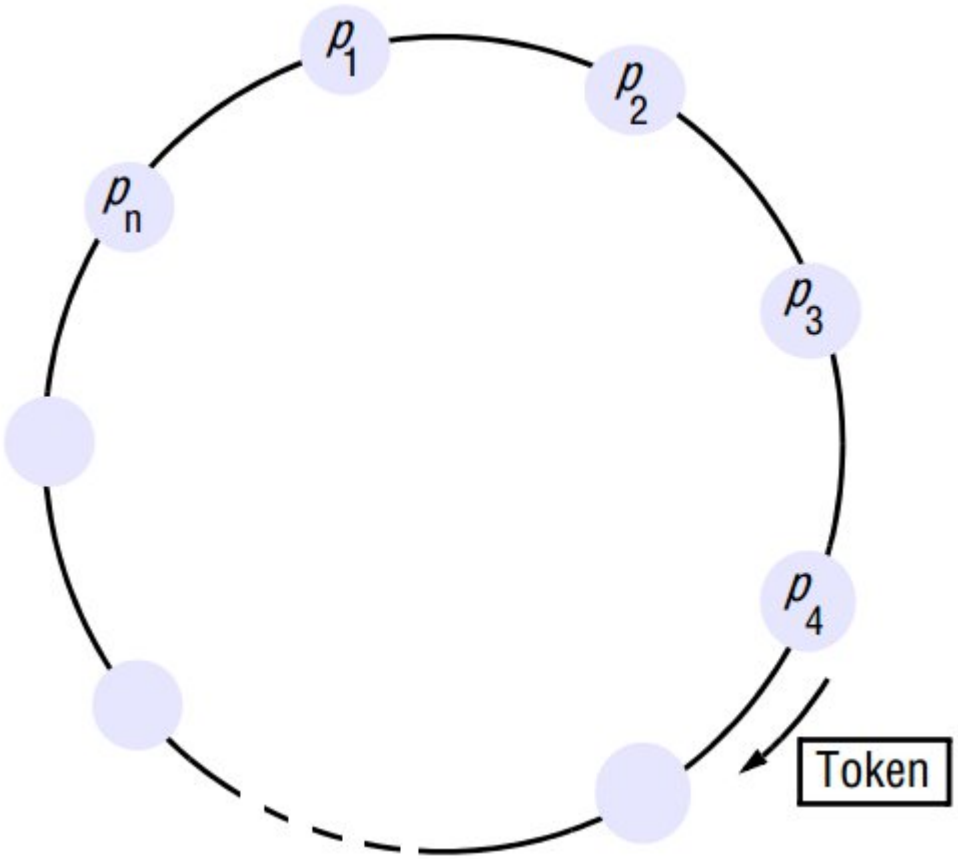
- **Требования к алгоритму**
  - Безопасность – в критической секции всегда находится не более одного процесса
  - Живучесть – запросы на вход и выход из критической секции успешно завершаются за конечное время
  - Справедливость – запросы на вход в критическую секцию выполняются в соответствии с отношением “happened-before”
- **Производительность**
  - Количество сообщений, отправляемых при входе/выходе в CS
  - Задержка процесса при входе/выходе в CS
  - Задержка между выходом одного процесса и входом следующего

Алгоритм с глобальным координатором (central server)



- Процессы обращаются к *центральному серверу* за разрешением войти в критическую секцию
- Сервер поддерживает очередь запросов: операции получить токен и вернуть токен

Кольцевой алгоритм (ring)



- Процессы логически организованы в кольцо
- По кольцу передается маркер, разрешающий вход в критическую секцию
- Если вход в критическую секцию не требуется, то маркер сразу передается дальше по кольцу

Алгоритм Ricart-Agrawala (multicast and logical clocks, 1981)

On initialization
    state := RELEASED;

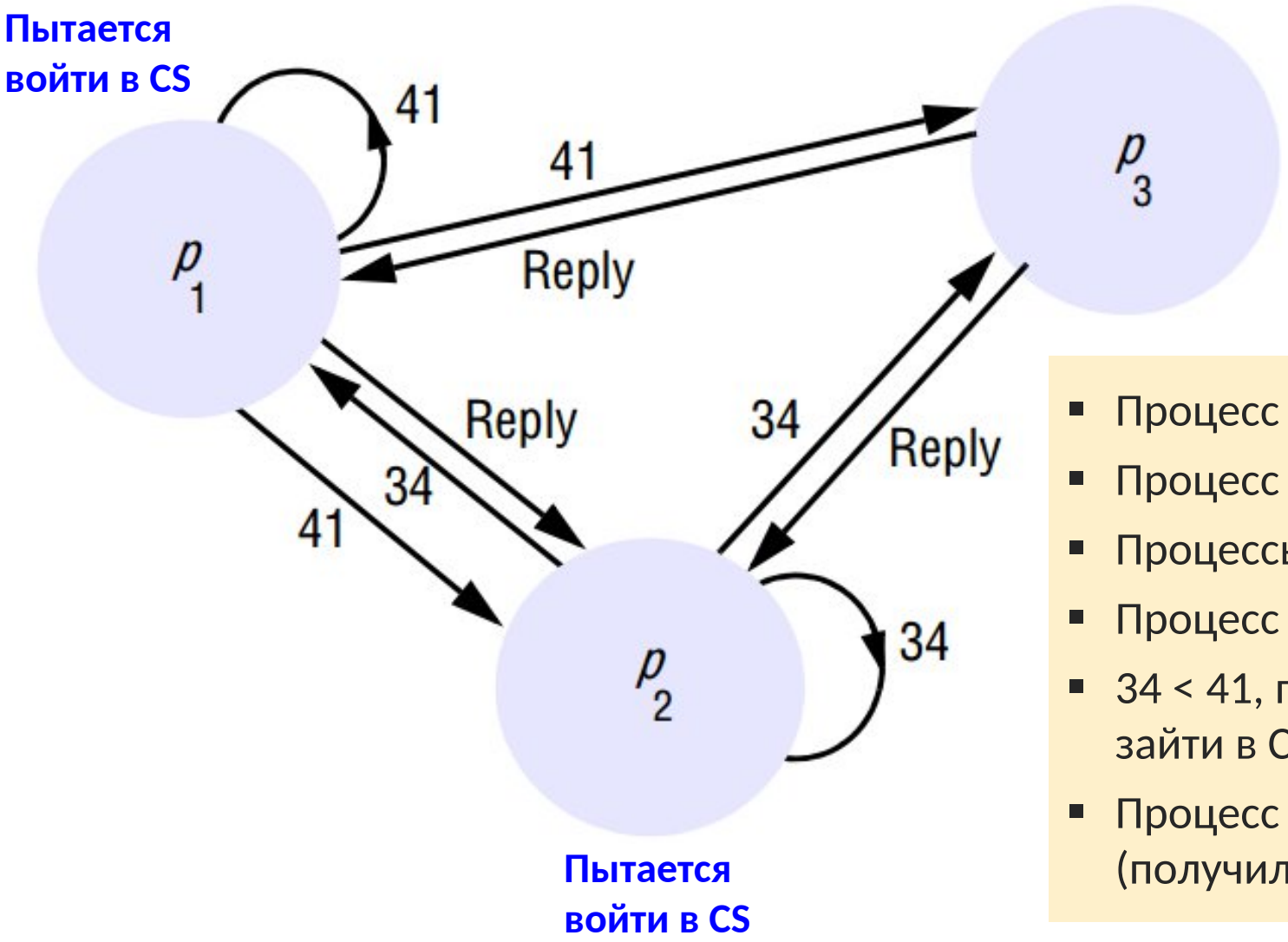
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Request processing deferred here
    Wait until (number of replies received = $(N-1)$); **// Ждем пока не ответят остальные процессы**
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD *or* (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Процесс, желающий войти в CS рассылает широковещательное сообщение (multicast) и ожидает разрешения от всех процессов

Алгоритм Ricart-Agrawala (1981)



- Процесс 1 Lamport’s clock = 41
- Процесс 2 Lamport’s clock = 34
- Процессы 1 и 2 рассылают запрос на вход в CS
- Процесс 3 разрешает всем зайти в CS
- 34 < 41, процесс 1 разрешает процессу 2 зайти в CS – процесс 2 в CS, выходит из CS
- Процесс 1 входит в CS (получил разрешение от 2)

Взаимное исключение (Distributed Mutual Exclusion)

- Ricart-Agrawala algorithm (an improvement over Lamport's algorithm)
- Lamport's Bakery Algorithm
- Raymond's Algorithm
- Maekawa's Algorithm
- Suzuki-Kasami's Algorithm
- Naimi-Trehel's Algorithm
- ...

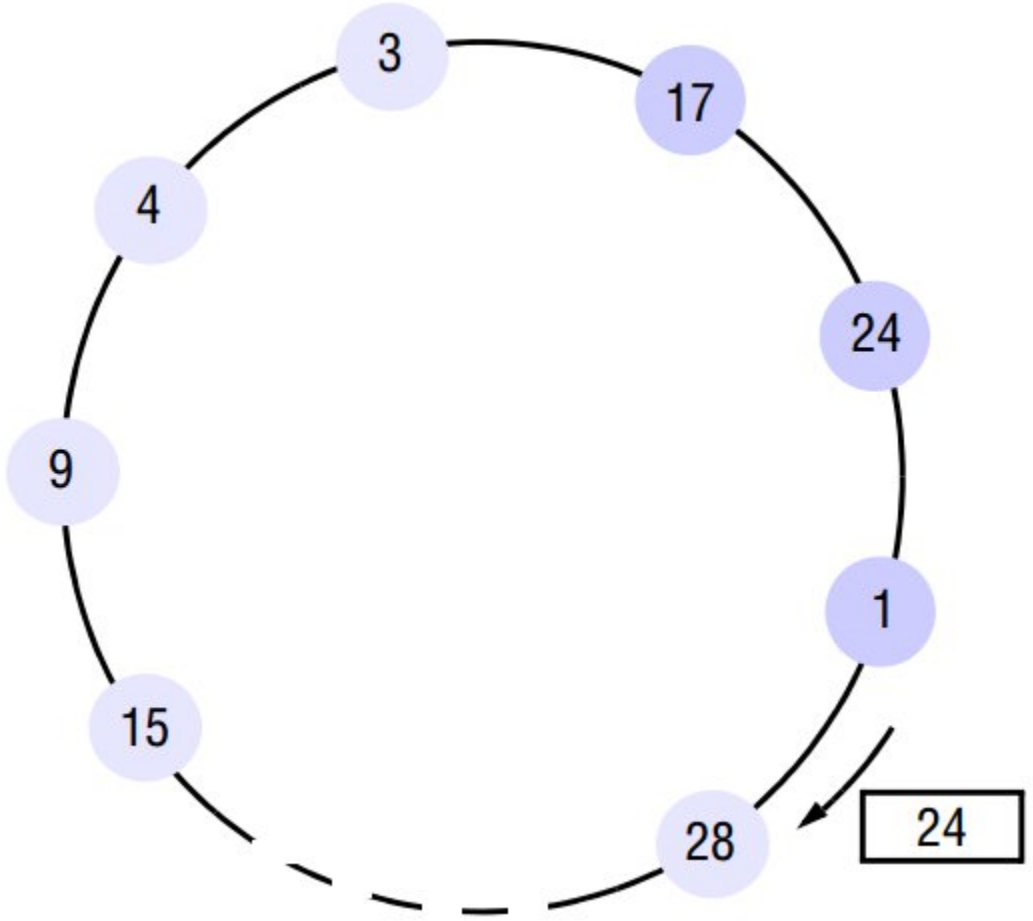
Выборы (Elections, leader election)

- **Необходимо выбрать (elect) один процесс из *n* для реализации заданной роли в распределенной системе**
  - ❑ Все процессы должны прийти к одному решению
  - ❑ Выборы могут быть инициированы любым процессом в любое время
  - ❑ *n* процессов могут одновременно инициировать *n* выборов
- **Например**, выбрать процесс для синхронизации часов, процесс для логирования, управления доступом к критической секции, ...
- Считаем, что выбранный процесс должен иметь наибольший «идентификатор»

Требования к алгоритму выборов

- Каждый процесс p_i имеет переменную $elected_i$
  - В начале участия в выборах устанавливается в *elected* = NULL
- Безопасность
  - Переменная *elected* у участвующего в выборах процесса может принимать значения NULL и P, где P – выбранный процесс с наибольшим идентификатором
- Живучесть
  - Каждый процесс в конечном итоге принимает участие в выборах и устанавливает значение elected != NULL или отказывает (в случае сбоя)
- Производительность
  - Количество сообщений
  - Время выборов

Кольцевой алгоритм выборов (Chang and Roberts, 1979)

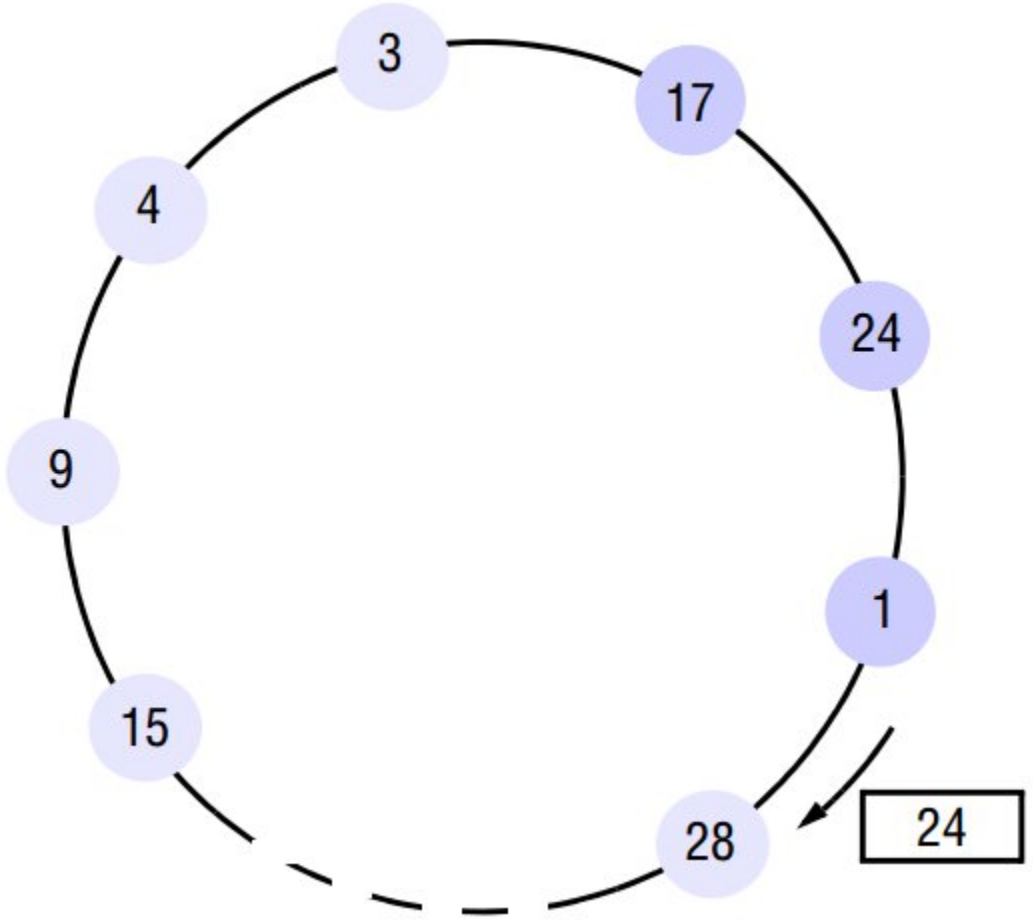


Выборы начаты процессом 17

- Процесс, желающий участвовать в выборах переводит себя в состояние “участник” и отправляет соседу (по часовой стрелке) свой id
- Когда процесс *i* получает сообщение, он проверяет:
  - ❑ Если он “участник”, алгоритм завершается
  - ❑ Выбирается максимум из принятого id и id текущего процесса: id = max(id_{recv}, id_i)
  - ❑ Вычисленное значение id передается дальше по кольцу, а текущий процесс становится “участником”

AllReduce(id, MAX)

Кольцевой алгоритм выборов (Chang and Roberts, 1979)

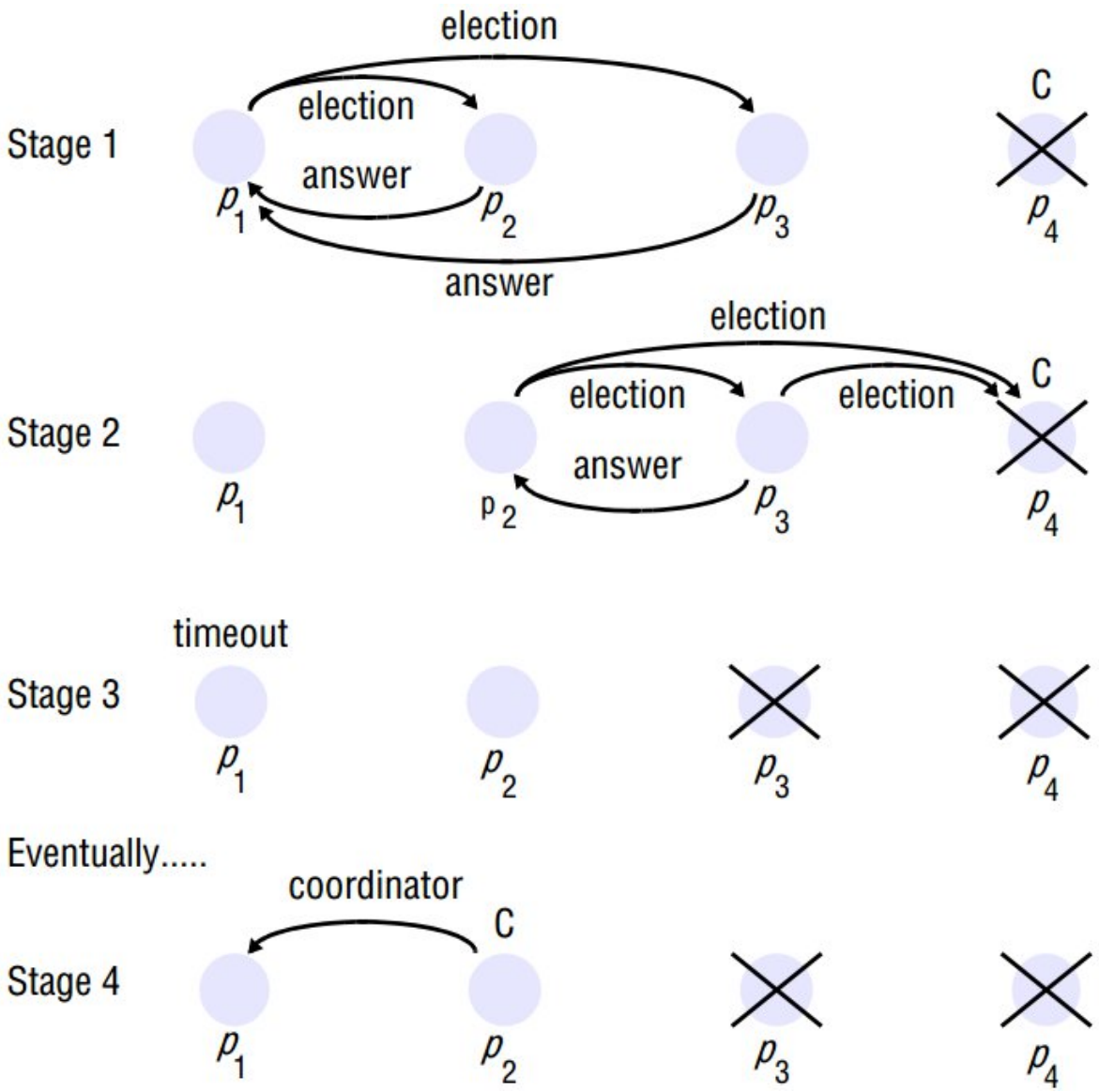


Выборы начаты процессом 17

Выбрать среди *n* процессов наименее загруженный

- Идентификатор процесса *i* = <1 / *load*, *i*> – номер процесса используется в сравнении, если загрузка процессов совпадает

Bully algorithm (Garcia-Molina, 1982)

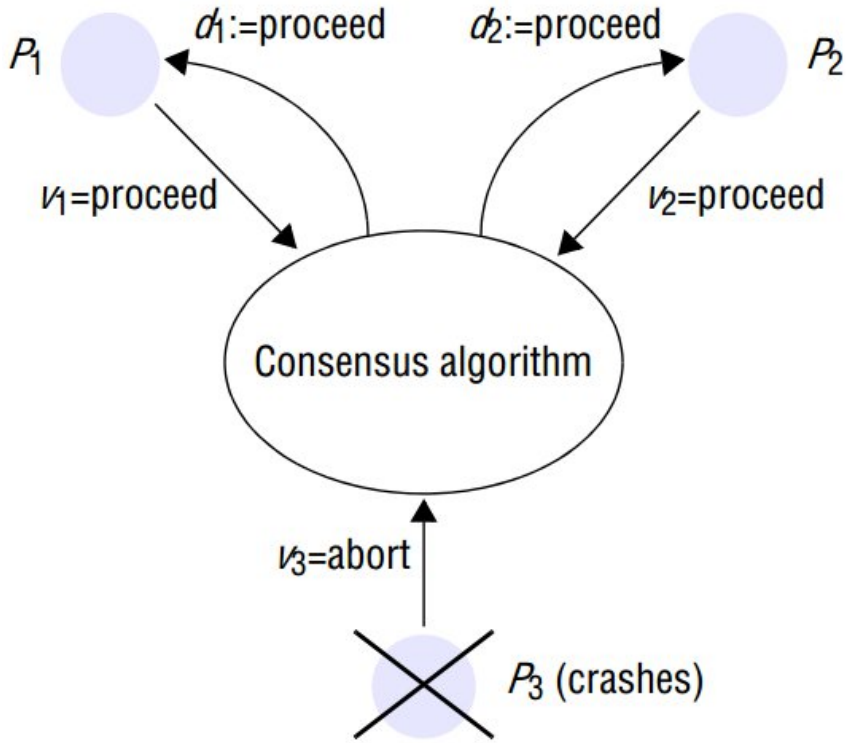


- Синхронная распределенная система
- Допускается отказ процессов, но не каналов
- Процесс отказал, если ответ не пришел в течении времени $T = T_{RTT} + T_{MessageProcess}$
- Все процессы знают, какой процесс имеет максимальный id
- В худшем случае $O(n^2)$ сообщений

Консенсус (Consensus)

- **Консенсус** – способ принятия решений при отсутствии принципиальных возражений у большинства заинтересованных лиц, принятие решения на основе общего согласия без проведения голосования
- (Pease et al. 1980, Lamport et al. 1982)

- **Для достижения консенсуса**, каждый процесс p_i начинает работу в состоянии “undecided” и предлагает значение v_i из множества D
- Процессы договариваются, обмениваясь значениями
- Каждый процесс p_i устанавливает значение переменной d_i и переходит в состояние “decided”
- **Все процессы должны перейти в состояние “decided” и установить одинаковые значения d_i**



Консенсус в надежной системе

- Каждый процесс p_i широковещательно отправляет свое значение v_i всем процессам
- Каждый процесс собрал вектор значений: v_1, v_2, ..., v_n
- Все процессы вычисляют значение функции $v = majority(v_1, v_2, ..., v_n)$, которая выбирает по некоторому критерию одно значение v
- Все процессы выберут одно и то же значение v, так как все вычисляют одну и ту же функцию

Алгоритмы достижения консенсуса

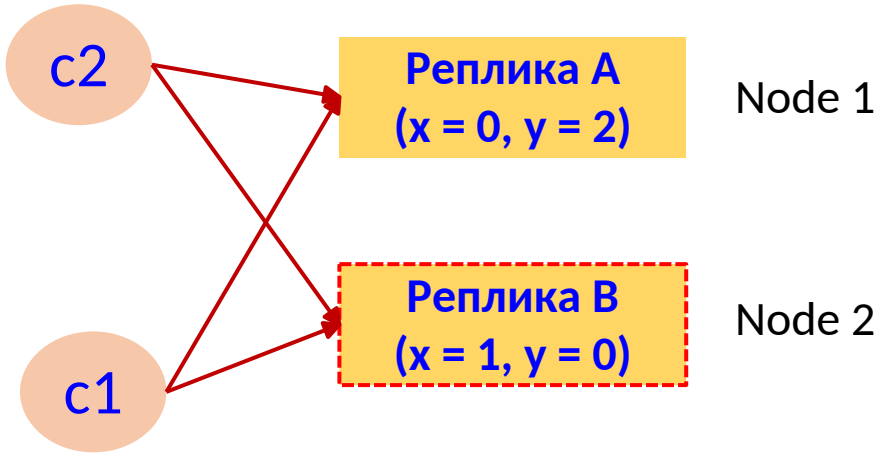
- Chandra-Toueg consensus algorithm
- Randomized consensus
- Raft consensus algorithm
- Семейство протоколов Paxos (L. Lamport, 1989, 1998) – достижение консенсуса при ненадежных процессорах

Реализации

- **Семейство протоколов Paxos**
  - ❑ Paxos Made Simple (L. Lamport, 2001)
  - ❑ Google: Chubby (BigTable), Spanner
  - ❑ IBM SAN Volume Controller
  - ❑ Microsoft Autopilot cluster management service
  - ❑ Ceph
- **Virtual synchrony**
  - ❑ A History of the Virtual Synchrony Replication Model (Birman, 2010)
  - ❑ Isis Toolkit, Corosync Cluster Engine, Appia, Jgroups
- **ZooKeeper Atomic Broadcast (Zab)**

Репликация (replication)

- Клиент обращается к локальному менеджеру реплик (RM)
- Если локальный RM не доступен, то обращается ко второму

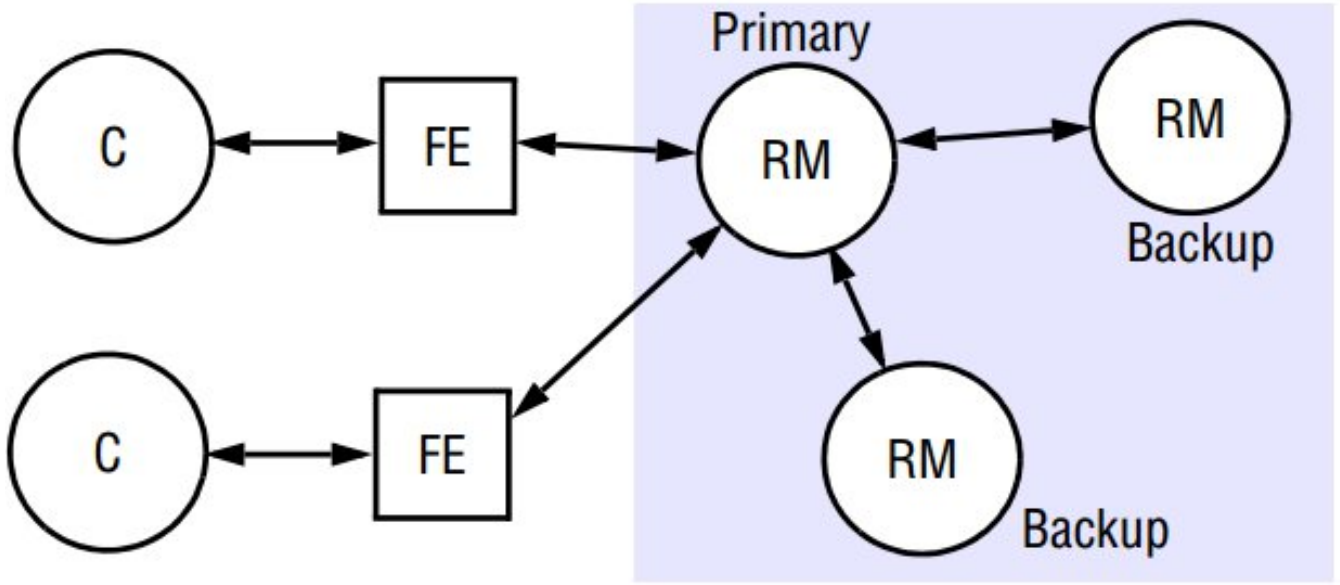


Client 1:	Client 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$ – *B fault*	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

B отказал, A не обновлена

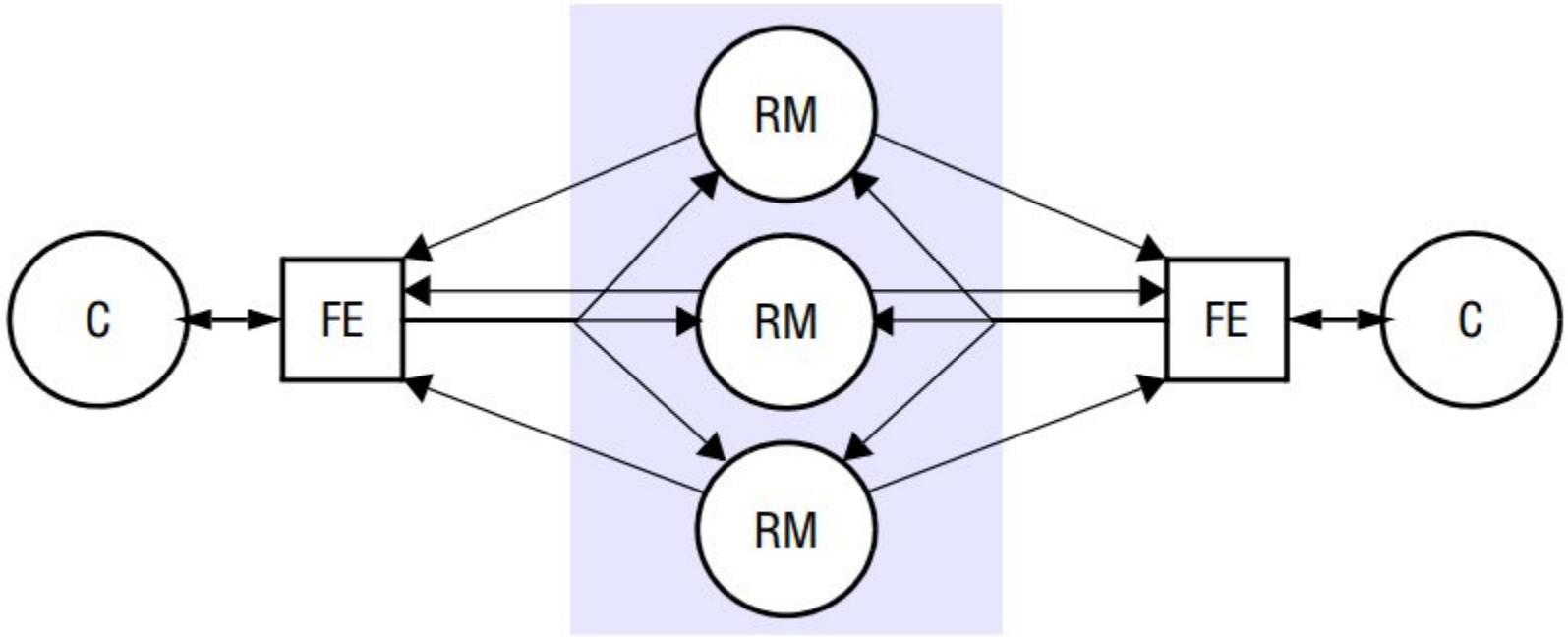
Client 1:	Client 2:
$setBalance_B(x, 1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

Пассивная модель репликации (primary-backup)



- Одна реплика активна (primary)
- Остальные реплики пассивные (backups, slaves)
- Активная реплика отправляет в подчиненные обновления данных
- В случае отказа активной реплики, одна из подчиненных становится главной

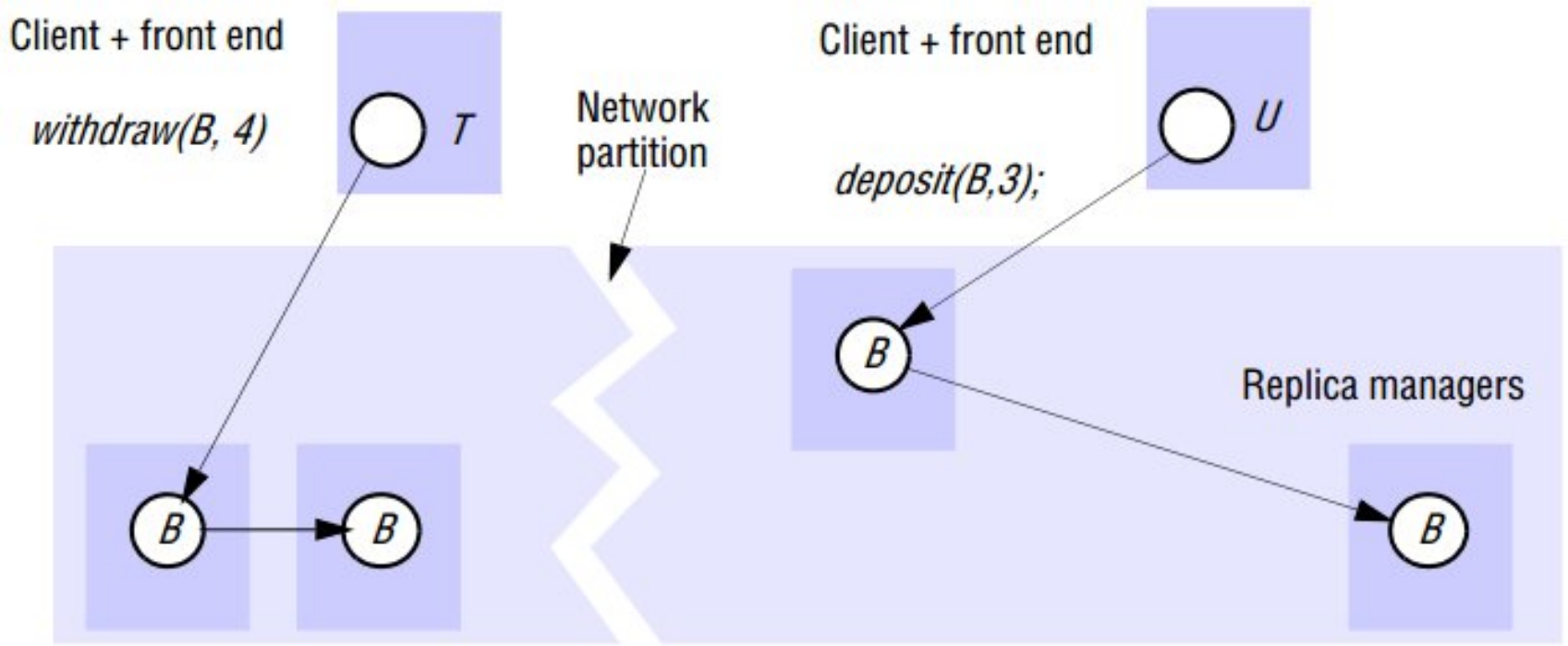
Активная модель репликации (Active replication)



- Пограничные узлы (FE) передают запросы всем репликам (RM)
- Все менеджеры реплик обрабатывают запросы одинаково

Разделение сети (Network partition)

- Возможна ситуация, когда между репликами будет потеряна связь (сетевое соединение)
- Несогласованность данных реплик разных частей



Теорема CAP

- **Теорема CAP** (**теорема Брюера, Brewer, 2000**) – эвристическое утверждение
- В распределенной системе невозможно одновременно обеспечить свойства
  - ❑ согласованности данных (Consistency)
  - ❑ доступности (Availability)
  - ❑ устойчивости к разделению (Partition tolerance) – отклик всегда корректный
- Из трех свойств одновременно можно обеспечить не более двух

http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Распределенные системы хранения данных

- Выбор реализуемых свойств на уровне архитектуры системы
  - ❑ CA: реляционные СУБД, LDAP
  - ❑ CP: Google BigTable, HBase
  - ❑ AP: веб-кэши, DNS, Amazon Dynamo
- Возможность выбора свойств пользователем на уровне отдельных операций
  - ❑ Apache Cassandra

Домашнее чтение

- Задача византийских генералов (Byzantine generals problem)
- Теорема CAP